BEA WebLogic Server™

Programming WebLogic JTA

Version 9.0 BETA
Revised: December 15, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Introduction and Roadmap

2. Introducing Transactions

5. Transaction Service

8. Transactions in RMI Applications

9. Using Third-Party JDBC XA Drivers with WebLogic Server

10. Coordinating XAResources with the WebLogic Server Transaction Manager

11. Participating in Transactions Managed by a Third-Party Transaction Manager

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming WebLogic JTA*.

Document Scope and 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™ 2, Enterprise Edition (J2EE) programming, and transaction processing concepts.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.

- Chapter 2, "Introducing Transactions," introduces transactions in EJB and RMI applications running in the WebLogic Server environment. This chapter also describes distributed transactions and the two-phase commit protocol for enterprise applications.
- Chapter 3, "Configuring Transactions," describes how to administer transactions in the WebLogic Server environment.
- Chapter 4, "Managing Transactions," provides information on administration tasks used to manage transactions.
- Chapter 5, "Transaction Service," describes the WebLogic Server Transaction Service.
- Chapter 6, "Java Transaction API and BEA WebLogic Extensions," provides a brief overview of the Java Transaction API (JTA).
- Chapter 7, "Transactions in EJB Applications," describes how to implement transactions in EJB applications.
- Chapter 8, "Transactions in RMI Applications," describes how to implement transactions in RMI applications.
- Chapter 9, "Using Third-Party JDBC XA Drivers with WebLogic Server," describes how to configure and use third-party XA drivers in transactions.
- Chapter 10, "Coordinating XAResources with the WebLogic Server Transaction Manager," describes how to configure third-party systems to participate in transactions coordinated by the WebLogic Server transaction manager.
- Chapter 11, "Participating in Transactions Managed by a Third-Party Transaction Manager," describes the process for configuring and participating in foreign-managed transactions.
- Chapter 12, "Troubleshooting Transactions," describes how to perform troubleshooting tasks for applications using JTA.

Related Documentation

This document contains JTA-specific design and development information.

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

- *Developing W*ebLogic Server Applications is a guide to developing WebLogic Server applications.

- *Deploying WebLogic Server Applications* is the primary source of information about deploying WebLogic Server applications.

Samples and Tutorials

In addition to this document, BEA Systems provides a variety of code samples and tutorials for developing transactional applications. The examples and tutorials illustrate WebLogic Server in action, and provide practical instructions on how to perform key JTA development tasks.

BEA recommends that you run some or all of the JTA examples.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample J2EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and J2EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

MedRec includes a service tier comprised primarily of Enterprise Java Beans (EJBs) that work together to process requests from web applications, web services, and workflow applications, and future client applications. The application includes message-driven, stateless session, stateful session, and entity EJBs.

CHAPTER 2

Introducing Transactions

This section discusses the following topics:

Overview of Transactions in WebLogic Server Applications

This section includes the following sections:

ACID Properties of Transactions

One of the most fundamental features 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.

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 of the operations fails, the entire set of operations is rolled back.

Supported Programming Model

WebLogic Server supports transactions in the Sun Microsystems, Inc., Java™ 2, Enterprise Edition (J2EE) programming model. WebLogic Server provides full support for transactions in Java applications that use Enterprise JavaBeans, in compliance with the Enterprise JavaBeans Specification 2.0, published by Sun Microsystems, Inc. WebLogic Server also supports the Java Transaction API (JTA) Specification 1.0.1a, also published by Sun Microsystems, Inc.

Supported API Models

WebLogic Server supports the Sun Microsystems, Inc. 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 about JTA, see the following sources:

- The `javax.transaction` and `javax.transaction.xa` package APIs.
- The Java Transaction API specification, published by Sun Microsystems, Inc.

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 of them 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.
- 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 can 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, need to be coordinated 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

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.

- Within the scope of a single client invocation on an object, the object performs multiple edits to data in a database. If one of the 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, can 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 needs to make 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.

When Not to Use Transactions

Transactions are not always appropriate. For example, if a series of transactions take a long time, implement them with a series of distributed transactions. Here is an example of an incorrect use of transactions.

- The client application needs to make invocations on several objects, which may involve write operations to one or more databases. If any one invocation is unsuccessful, any state that is written (either in memory or, more typically, to a database) must be rolled back.

  For example, consider a travel agent application. The client application needs to arrange for a journey to a distant location; for example, from Strasbourg, France, to Alice Springs, Australia. Such a journey would inevitably require multiple individual flight reservations. The client application works by reserving each individual segment of the journey in sequential order; for example, Strasbourg to Paris, Paris to New York, New York to Los Angeles. However, if any individual flight reservation cannot be made, the client application needs a way to cancel all the flight reservations made up to that point.

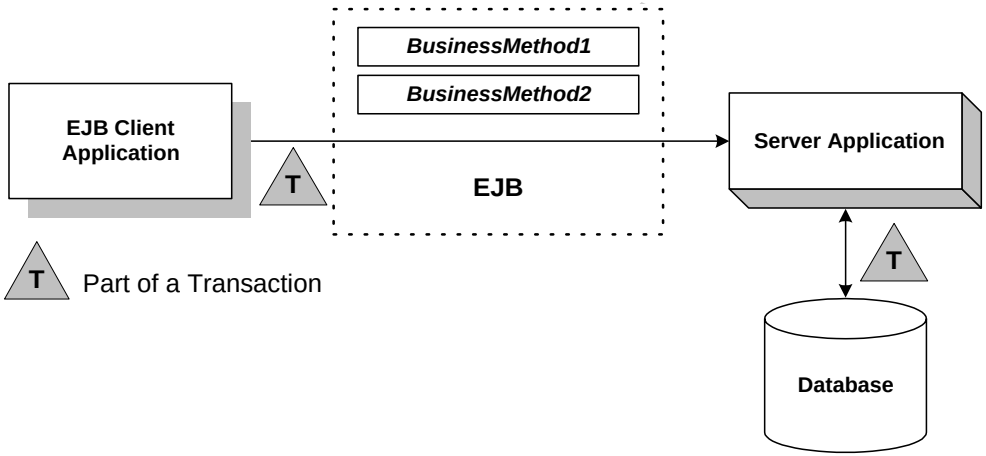
What Happens During a Transaction

This topic includes the following sections:

Transactions in WebLogic Server EJB Applications

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

Figure 2-1 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 *Programming WebLogic EJB*.
- 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 the `UserTransaction` object, see the WebLogic Server Javadoc at `http://e-docs.bea.com/wls/docs90/javadocs/weblogic/transaction/UserTransaction.html`.

The sequence of transaction events differs between container-managed and 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 `Never`. For a detailed description of these settings, see Section 17.6.2 in the Enterprise JavaBeans Specification 2.0, published by Sun Microsystems, Inc.
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.
   - In another example, if the `trans-attribute` setting is `Mandatory`, the EJB container invokes the method within the existing transaction context. If the client called without a transaction context, the EJB container throws the `javax.transaction.TransactionRequiredException` exception.
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 the end of the method execution and before the result is sent to the client, the EJB container completes the transaction, either by committing the transaction or rolling it back (if the `EJBContext.setRollbackOnly` method was called).

Bean-managed Transactions

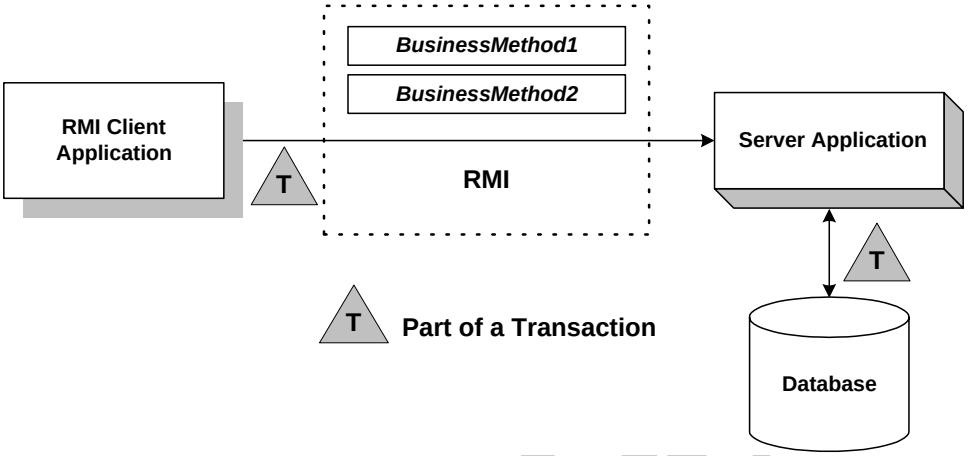
For EJB applications with bean-managed transaction demarcations, a basic transaction works in the following way:

1. In the EJB's deployment descriptor, the Bean Provider or Application Assembler specifies the transaction type (`transaction-type` element) for container-managed demarcation (`Bean`).
2. The client application uses JNDI to obtain an object reference to the `UserTransaction` object for the WebLogic Server domain.
3. The client application begins a transaction using the `UserTransaction.begin` method, and issues a request to the EJB through the EJB container. All operations on the EJB execute within the scope of a transaction.
   - If a call to any of these operations raises an exception (either explicitly or as a result of a communication failure), the exception can be caught and the transaction can be rolled back using the `UserTransaction.rollback` method.
   - 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.
4. The `UserTransaction.commit` method causes the EJB container to call the transaction manager to complete the transaction.
5. The transaction manager is responsible for coordinating with the resource managers to update any databases.

Transactions in WebLogic Server RMI Applications

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

Figure 2-2 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 as a result of a communication failure), the exception can be caught and the transaction can be rolled back using the `UserTransaction.rollback` method.

   – 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.

For more information, see Chapter 8, "Transactions in RMI Applications."

Transactions Sample Code

This section includes the following sections:

- 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:

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

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`).

Notes: In a global transaction, 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.

These code fragments do not derive from any of the sample applications that ship with WebLogic Server. They merely illustrate the use of the `UserTransaction` object within an EJB application.

Importing Packages

Listing 2-1 shows importing the necessary packages for transactions, including:

- `javax.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.

Listing 2-1 Importing Packages

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

Using JNDI to Return an Object Reference

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

Listing 2-2 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("javax.transaction.UserTransaction");
```

Starting a Transaction

Listing 2-3 shows starting a transaction by getting a `UserTransaction` object and calling the `javax.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.

Listing 2-3 Starting a Transaction

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

Completing a Transaction

Listing 2-4 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 `javax.transaction.UserTransaction.rollback()` method.
- If no exception was thrown, the application calls the `javax.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.

Listing 2-4 Completing a Transaction

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

```
// or:

tx.rollback();
```

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:

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

The code fragments demonstrate using the `UserTransaction` object for RMI transactions. For guidelines on using transactions in RMI applications, see Chapter 8, "Transactions in RMI Applications."

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

Importing Packages

Listing 2-5 shows importing the necessary packages, including the following packages used to handle transactions:

- `javax.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.

Listing 2-5 Importing Packages

```
import javax.naming.*;
import java.rmi.*;
import javax.transaction.UserTransaction;
import javax.transaction.SystemException;
```

```
import javax.transaction.HeuristicMixedException
import javax.transaction.HeuristicRollbackException
import javax.transaction.NotSupportedException
import javax.transaction.RollbackException
import javax.transaction.IllegalStateException
import javax.transaction.SecurityException
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

Listing 2-6 shows searching the JNDI tree to return an object reference to the `UserTransaction` object for the appropriate WebLogic Server domain.

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.

Listing 2-6 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("javax.transaction.UserTransaction");
```

Starting a Transaction

Listing 2-7 shows starting a transaction by calling the `javax.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.

Listing 2-7 Starting a Transaction

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

Completing a Transaction

Listing 2-8 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 `javax.transaction.UserTransaction.rollback()` method if an exception was thrown during any of the database operations.
- If no exception was thrown, the application calls the `javax.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 the transaction manager to complete the transaction. The transaction is committed only if all of the participants in the transaction agree to commit.

Listing 2-8 Completing a Transaction

```
tx.commit();

// or:

tx.rollback();
```

Configuring Transactions

The following sections provide configuration tasks related to transactions:

- Overview of Transaction Configuration
- Configuring JTA
- Configuring Domains for Inter-Domain Transactions

Overview of Transaction Configuration

The Administration Console provides the interface used to configure 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 J2EE components that can participate in transactions, such as EJBs, JDBC, and JMS.

Configuring JTA

Once you configure WebLogic JTA and any transaction participants, the system can perform transactions using the JTA API and the WebLogic JTA extensions. Configuration settings for JTA (transactions) are applicable at the domain level. This means that configuration attribute

settings apply to all servers within a domain. Monitoring and logging tasks for JTA are performed at the server level.

You can configure any transaction attributes before starting the server (static configuration) or, with one exception, while the server is running (dynamic configuration). The `TransactionLogFilePrefix` attribute must be set before starting the server.

The following sections provide information on how to configure transactions and JTA using the WebLogic Administration console:

- "How to Configure JTA in the Administration Console" on page 3-2
- "Additional Attributes for Managing Transactions" on page 3-2

How to Configure JTA in the Administration Console

Use the following steps to configure JTA in the WebLogic Administration Console:

1. In the left pane of the Administration Console, select the domain node (right below the word "console"). The Configuration tab for the domain is displayed by default.
2. Click the JTA tab.
3. Enter values in the Timeout Seconds, Abandon Timeout Seconds, Before Completion Iteration Limit, Max Transactions, Max Unique Name Statistics, and Checkpoint Interval Seconds attribute fields or accept the default values as assigned.
4. Enable or disable the Forget Heuristics attribute as desired.
5. Click Apply to save any changes you made.
6. Ensure that the `TransactionLogFilePrefix` attribute is set appropriately when you configure the server.

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 configuration attributes listed in Table 3-1:

Table 3-1 XA Resource Health Monitoring Configuration Attributes

Attribute	MBean	Definition
`EnableResourceHealthMonitoring`	`weblogic.management.configuration.JDBCConnectionPoolMBean`	Enables or disables resource health monitoring for the JDBC connection pool. This attribute only applies to connection pools that use an XA JDBC driver for database connections. It is ignored if a non-XA JDBC driver is used. If set to `true`, resource health monitoring is enabled. If an XA resource fails to respond to an XA call within the period specified in the `MaxXACallMillis` attribute, WebLogic Server marks the connection pool as unhealthy and blocks any further calls to the resource. If set to `false`, the feature is disabled. Default: `true`
`MaxXACallMillis`	`weblogic.management.configuration.JTAMBean`	Sets the maximum allowed duration (in milliseconds) of XA calls to XA resources. This setting applies to the entire domain. Default: `120000`
`MaxResourceUnavailableMillis`	`weblogic.management.configuration.JTAMBean`	The maximum duration (in milliseconds) that an XA resource is marked as unhealthy. After this duration, the XA resource is declared available again, even if the resource is not explicitly re-registered with the transaction manager. This setting applies to the entire domain. Default: `1800000`
`MaxResourceRequestOnServer`	`weblogic.management.configuration.JTAMBean`	Maximum number of concurrent requests to resources allowed for each server in the domain. Default: `50` Minimum: `10` Maximum: `java.lang.Integer.MAX_VALUE`

You set these attributes directly in the config.xml file when the domain is inactive. These attributes are not available in the Administration Console. The following example shows an excerpt of a configuration file with these attributes:

```
...
   <JTA
    MaxUniqueNameStatistics="5"
    TimeoutSeconds="300"
    RecoveryThresholdMillis="150000"
    MaxResourceUnavailableMillis="900000"
    MaxResourceRequestOnServer="60"
    MaxXACallMillis="180000"
   />
 <JDBCConnectionPool
  Name="XAPool"
  Targets="myserver"
  DriverName="weblogic.qa.tests.transaction.
    testsupport.jdbc.XADataSource"
  InitialCapacity="1"
  MaxCapacity="10"
  CapacityIncrement="2"
  RefreshMinutes="5"
  TestTableName="dual"
  EnableResourceHealthMonitoring="true"
  Properties="user=scott;password=tiger;server=dbserver1"
 />
 <JDBCTxDataSource
  Name="XADataSource"
  Targets="myserver"
  JNDIName="weblogic.jdbc.XADS"
  PoolName="XAPool"
 />
...
```

Configuring Domains for Inter-Domain Transactions

For a transaction manager to manage distributed transactions, the transaction manager must be able to communicate with all participating servers to prepare and then commit or rollback the transactions. This applies to cases when your WebLogic domain acts as the transaction manager or a transaction participant (resource) in a distributed transaction. The following sections describe how to configure your domain to enable inter-domain transactions.

The following sections provide information on how to configure domains for inter-domain transactions:

Limitations for Inter-Domain Transactions

Please note the following limitations for inter-domain transactions:

- You cannot manually resolve incomplete transactions on resources from a WebLogic Server domain from WebLogic Server version 7.0 or earlier.
- The domains and all participating resources must have unique names. That is, you cannot have a JDBC connection pool, a server, or a domain with the same name as an object in another domain or the domain itself.
- Only one data source with *both* of the following attribute conditions can participate in a global transaction, regardless the domain in which the data source is configured:
  - Emulate Two-Phase Commit is selected (`EnableTwoPhaseCommit=true` in `config.xml`).
  - The Pool Name attribute in the data source specifies a connection pool that uses a non-XA driver to create database connections.

Note: BEA recommends that you use an XA driver instead of a non-XA driver (with Emulate Two-Phase Commit) in global transactions. There are risks involved with using a non-XA driver in a global transaction.

Inter-Domain Transactions for WebLogic Server 8.x and 7.x Domains

To manage or participate in transactions that span multiple WebLogic Server 8.x and 7.x domains (that is, all participating domains run on WebLogic Server 7.x or 8.x, or a combination of 7.x and 8.x), you must set a security credential for all domains to the same value. To set the credential value, follow these steps *for each participating domain*:

1. Start the Administration Console for one of the participating domains.
2. In the left pane, click the Security node to display Security settings for the domain.
3. Click the Configuration tab (if necessary), then the Advanced tab.
4. Clear the Enable Generated Credential check box and click Apply.
5. In the Credential field, enter the new credential, then re-enter it in the Confirm field. Enter the *same* credential for each domain and click Apply. If WebLogic Server 6.x domains will participate in distributed transactions, use the `system` password from the WebLogic Server 6.x domain.
6. Restart the administration server. You may also need to restart all managed servers.
7. Repeat steps 1 through 6 for each domain that participates in inter-domain transactions. In step 5, enter the *same* credential for each domain.

Inter-Domain Transactions Between WebLogic Server 7.x/8.x and WebLogic Server 6.x Domains

To manage transactions that use servers in both WebLogic Server 7.x or 8.x and WebLogic Server 6.x domains, you must do the following:

In all participating WebLogic Server 6.x domains:

- Change the password for the `system` user to the same value in all participating domains on the Security→Users tab in the Administration Console. See Changing the System Password at `http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html#cnfgsec003`.

In all participating WebLogic Server 7.x and 8.x domains:

- Set a security credential for all domains to the same value on the Domain→Security→Advanced tab. The credential must match the `system` password in all

participating WebLogic Server 6.x domains. For instructions, see “Inter-Domain Transactions for WebLogic Server 8.x and 7.x Domains” on page 3-6.

CHAPTER 4

Managing Transactions

The following sections provide information on administration tasks used to manage transactions:

Overview of Transaction Management

You can monitor transactions on a server using the logging, statistics, and monitoring facilities. Use the Administration Console to configure these features and to display the resulting output.

You use the Administration Console to access tools for configuring the WebLogic Server features, including the Java Transaction API (JTA). The transaction configuration process involves specifying values for attributes. These attributes define various aspects of the transaction environment:

- Transaction timeouts and limits
- Transaction Manager behavior

- Transaction log file prefix

Before configuring your transaction environment, you should be familiar with the J2EE components that can participate in transactions, such as EJBs, JDBC, and JMS.

- EJBs (Enterprise JavaBeans) use JTA for transaction support. Several deployment descriptors relate to transaction handling. For more information about programming with EJBs and JTA, see *Programming WebLogic Enterprise JavaBeans*.
- JDBC (Java Database Connectivity) provides standard interfaces for accessing relational database systems from Java. JTA provides transaction support on connections retrieved using a JDBC driver and transaction data source. For more information about programming with JDBC and JTA, see *Programming WebLogic JDBC*.
- JMS (Java Messaging Service) uses JTA to support transactions across multiple data resources. WebLogic JMS is an XA-compliant resource manager. For more information about programming with JMS and JTA, see *Programming WebLogic JMS*.

Logging

The transaction log consists of multiple files. Each file is named using a prefix indicating the location in the file system, as defined by the `TransactionLogFilePrefix` attribute, the server name, a unique numeric suffix, and a file extension. The `TransactionLogFilePrefix` attribute is set for each server in a domain. The overall amount of space consumed by the transaction log is limited only by the file system's available disk space.

Note: 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 will throw an exception. In that case, you must reconfigure your application to work around the buffer size.

WebLogic Server keeps statistics on transactions organized by server, resource, and transaction name. For information on using statistics in troubleshooting and debugging, see "Transaction Statistics" in Chapter 12, "Troubleshooting Transactions."

Monitoring

You can monitor transactions in progress using the Administration Console. You can display information for transactions by name, transactions by resource, or all active transactions.

Monitoring Transactions

In the Administration Console, you can monitor transactions for each server in the domain. Transaction statistics are displayed for a specific server, not the entire domain, even though transaction settings apply to the entire domain.

The following sections provide information on monitoring transactions:

Viewing Transaction Statistics for a Server

To view transaction statistics for a server, follow these steps:

1. In the Administration Console, click the server node in the left pane and select a server.
2. In the right pane, select the Monitoring tab and then the JTA tab. Totals for transaction statistics are displayed.
3. Optionally, click the text links at the bottom of the page to view transaction details by server resource or by transaction name, or for all active transactions on the server.

Viewing Transaction Statistics for Named Transactions

To view aggregate statistics about named transactions coordinated by a server, follow these steps:

1. In the Administration Console, click the server node in the left pane and select a server.
2. In the right pane, select the Monitoring tab and then the JTA tab.
3. Click the Monitor All Transactions by Name text link at the bottom of the page.

Viewing Transaction Statistics for Server Resources

To view aggregate statistics for each transactional resource accessed on a server, follow these steps:

1. In the Administration Console, click the server node in the left pane and select a server.
2. In the right pane, select the Monitoring tab and then the JTA tab.
3. Click the Monitor All Transactions by Resource text link at the bottom of the page.

Viewing Current (Inflight) Transactions for a Server

To view aggregate statistics for each transactional resource accessed on a server, follow these steps:

1. In the Administration Console, click the Servers node in the left pane and select a server.
2. In the right pane, select the Monitoring tab and then the JTA tab.
3. Click the Monitor All Inflight Transactions text link at the bottom of the page.

Manually Resolving Current (Inflight) Transactions

In some cases, a transaction may not complete normally due to system or network failures. In such situations there may be locks held on behalf of the pending transaction that are inhibiting the progress of other 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. You can also manually resolve "stuck" transactions.

To manually resolve a transaction, you view current (inflight) transactions for a server from the Server→Monitoring→JTA tab (see "Viewing Current (Inflight) Transactions for a Server" on page 4-4) 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. Table 4-1 lists the transaction states and manual resolution options for each transaction state.

Table 4-1 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

Table 4-1 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 javax.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 javax.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 javax.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

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.

Manual Commit and Rollback Options

To manually resolve a transaction, you can choose from the following options. Options are restricted as described in Table 4-1.

- **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.
- **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.

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 Administration 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 `javax.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.

To Manually Resolve a Transaction

1. In the Administration Console, click the Servers node in the left pane and select a server.
2. In the right pane, select the Monitoring tab and then the JTA tab.
3. Click the Monitor All Inflight Transactions text link at the bottom of the page.
4. Click a Transaction Id to view details about the transaction.
5. Choose one of the following options: (options are restricted by transaction status; see Table 4-1)
   - Click Force Local Rollback to roll back the transaction on resources on the current server.
   - Click Force Global Rollback to roll back the transaction on all resources that participate in the transaction.
   - Click Force Local Commit to commit the transaction on resources on the current server.
   - Click Force Global Commit to commit the transaction on all resources that participate in the transaction.

See "Manually Resolving Current (Inflight) Transactions" on page 4-4 for more information about manually resolving transactions.

Transaction Log Files

The following sections provide information on how to use transaction log files:

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 when recovering from system crashes or network failures. You cannot directly view the transaction log—the file is in a binary format.

The transaction log consists of multiple files. Each file is subject to garbage collection by the transaction manager. That is, when none of the records in a transaction log file are needed, the system deletes the file and returns the disk space to the file system. In addition, the system creates a new transaction log file if the previous log file becomes too large or a checkpoint occurs.

Caution: Do not manually delete transaction log files. Deleting transaction log files may cause inconsistencies in your data.

Transaction log files are uniquely named using a pathname prefix, the server name, a four-digit numeric suffix, and a file extension. The pathname prefix determines the storage location for the file. You can specify a value for the `TransactionLogFilePrefix` server attribute using the WebLogic Administration Console. The default `TransactionLogFilePrefix` is the server's working directory.

You should set the `TransactionLogFilePrefix` so that transaction log files are created on a highly available file system, for example, on a RAID device. 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 "Preparing to Migrate the Transaction Recovery Service" on page 4-17 for more information.

On a UNIX system with a server name of `websvr` and with the `TransactionLogFilePrefix` set to `/usr7/applog1/`, you might see the following log files:

```
/usr7/applog1/websvr0000.tlog
/usr7/applog1/websvr0001.tlog
/usr7/applog1/websvr0002.tlog
```

Similarly, on a Windows system with the `TransactionLogFilePrefix` set to `C:\weblogic\logA\`, you might see the following log files:

```
C:\weblogic\logA\websvr0000.tlog
C:\weblogic\logA\websvr0001.tlog
C:\weblogic\logA\websvr0002.tlog
```

If you notice a large number of transaction log files on your system, this may be an indication of multiple long-running transactions that have not completed. This can be caused by resource manager failures or transactions with especially large timeout values.

If the file system containing the transaction log 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.

Note: 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 will throw an exception. In that case, you must reconfigure your application to work around the buffer size.

When migrating a server to another machine, move the transaction log files as well, keeping all the log files for a server together. See "Moving a Server to Another Machine" on page 4-12 for more information.

Setting the Transaction Log File Location (Prefix)

To set the prefix for the transaction log files, which determines the location of the transaction log files, follow these steps:

1. In the Administration Console, click the server node in the left pane and select a server.
2. In the right pane, select the Logging tab and then the JTA tab.
3. Enter a transaction log file prefix (storage location for transaction logs) then click Apply to save the attribute setting. The new transaction log file prefix takes effect after you restart the server.

   The default transaction log file prefix is the server's working directory. You can specify a relative path from the server's working directory or an absolute path to another storage location.

Setting the Transaction Log File Write Policy

You can select a transaction log file write policy to change the way WebLogic Server writes transaction log file entries. You can select either of the following options:

- Cache-Flush—(the default) Flushes operating system and on-disk caches after each entry to the transaction log. Transactions cannot commit until the commit record is written to stable storage.
- Direct-Write—Forces the operating system to write transaction log entries directly to disk with each write. This option is available on Windows, Solaris and HP-UX platforms.

Warning: On Windows, the Direct-Write transaction log file write policy may leave transaction data in the on-disk cache without immediately writing it to disk. This is not transactionally safe because a power failure can cause loss of on-disk cache data. To prevent cache data loss when using the Direct-Write transaction log file write

policy on Windows, disable all write caching for the disk (enabled by default) or use a battery backup for the system.

The transaction log file write policy can affect transaction performance. You should test these options with your system to see which performs better. Direct-Write typically performs as well or better than Cache-Flush, depending on operating system and OS parameter settings, and is available on Windows, HP-UX, and Solaris. Windows systems optimize serial writes to disk such that subsequent writes to a file get faster after the first write to the file. Transaction log file entries are written serially, so this could improve performance. On some UNIX systems, the Cache-Flush option will flush all cached disk writes, not only those for the transaction log file, which could degrade transaction performance.

To set the transaction log file write policy, follow these steps:

1. In the Administration Console, click the server node in the left pane and select a server.
2. In the right pane, select the Logging tab and then the JTA tab.
3. Select a Transaction Log File Write Policy: Cache-Flush (the default) or Direct Write.
4. Click Apply to save the attribute setting. The new transaction log file write policy takes effect after you restart the server.

Heuristic Log Files

When importing transactions from a foreign transaction manager into WebLogic Server, the WebLogic Server transaction manager acts as an XA resource coordinated by the foreign transaction manager. In rare catastrophic situations, such as after the transaction abandon timeout expires or if the XA resources participating in the WebLogic Server imported transaction throw heuristic exceptions, the WebLogic Server transaction manager will make a heuristic decision. That is, the WebLogic Server transaction manager will decide to commit or roll back the transaction without input from the foreign transaction manager. If the WebLogic Server transaction manager makes a heuristic decision, it stores the information of the heuristic decision in the heuristic log files until the foreign transaction manager tells it to forget the transaction.

Heuristic log files are stored with transaction log files and look similar to transaction log files with `.heur` before the `.tlog` extension. They use the following format:

```
<TLOG_file_prefix>\<server_name><4-digit number>.heur.tlog
```

On a UNIX system with a server name of `websvr`, you might see the following heuristic log files:

```
/usr7/applog1/websvr0000.heur.tlog
/usr7/applog1/websvr0001.heur.tlog
/usr7/applog1/websvr0002.heur.tlog
```

Similarly, on a Windows system, you might see the following heuristic log files:

```
C:\weblogic\logA\websvr0000.heur.tlog
C:\weblogic\logA\websvr0001.heur.tlog
C:\weblogic\logA\websvr0002.heur.tlog
```

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. 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 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.

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 will continue to try to complete the transaction until all resource managers indicate that the transaction is completed. Using the `AbandonTimeoutSeconds` attribute, you can set the maximum time, in seconds, that a transaction manager will persist 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 will roll back the transaction to release any locks held on behalf of the abandoned transaction and will write an heuristic error to the server log.

You may want to review the following releated information:

- For instructions on how to set the `AbandonTimeoutSeconds` attribute, see "Configuring JTA" on page 3-1.
- For information about manually resolving a transaction, see "Manually Resolving Current (Inflight) Transactions" on page 4-4.
- For more information about the two-phase commit process, see Distributed Transactions and the Two-Phase Commit Protocol in *Programming WebLogic JTA*.

Moving a Server to Another Machine

When an application server is moved to another machine, it must be able to locate the transaction log files on the new disk. For this reason, BEA recommends moving the transaction log files to the new machine before starting the server on the new machine. By doing so, you can ensure that recovery runs properly. When you start WebLogic Server on the new system, the server reads the transaction log files to recover pending transactions, if any. If the pathname is different on the new machine, update the `TransactionLogFilePrefix` attribute with the new path before

starting the server. For instructions on how to change the `TransactionLogFilePrefix`, see "Setting the Transaction Log File Location (Prefix)" on page 4-9.

Transaction Recovery After a Server Fails

The WebLogic Server transaction manager is designed to recover from system crashes with minimal user intervention. The transaction manager 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, WebLogic Server provides the Transaction Recovery Service, which automatically attempts to recover transactions on system startup. The Transaction Recovery Service owns the transaction log for a server. On startup, the Transaction Recovery Service parses all log files for incomplete transactions and completes them as described in "Transaction Recovery Service Actions After a Crash" on page 4-14.

Because the Transaction Recovery Service is designed to gracefully handle transaction recovery after a crash, BEA 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 transaction log files) to another system (machine) to recover transactions. See "Recovering Transactions for a Failed Non-Clustered Server" on page 4-15 for more information. For a server in a cluster, you can manually *migrate* 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 Administration Console or the WebLogic command line interface.

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

For more information about migrating the Transaction Recovery Service, see "Recovering Transactions for a Failed Clustered Server" on page 4-15. For more information about clusters, see *Using WebLogic Server Clusters* at http://e-docs.bea.com/wls/docs90/cluster/index.html.

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

- "Transaction Recovery Service Actions After a Crash" on page 4-14
- "Recovering Transactions for a Failed Non-Clustered Server" on page 4-15

- "Recovering Transactions for a Failed Clustered Server" on page 4-15

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" on page 4-11 for more information.

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" on page 4-12 for more information.

Recovering Transactions for a Failed Non-Clustered Server

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

1. Move (or make available) all transaction log files from the failed server to a new server.
2. Set the `TransactionLogFilePrefix` attribute with the path to the transaction log files. For instructions, see "Setting the Transaction Log File Location (Prefix)" on page 4-9.
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" on page 4-14.

When moving transaction logs after a server failure, make all transaction log files available on the new machine before starting the server there. You can accomplish this by storing transaction log files on a dual-ported disk available to both machines. As in the case of a planned migration, update the `TransactionLogFilePrefix` attribute with the new path before starting the server if the pathname is different on the new machine. Ensure that all transaction log files are available on the new machine before the server is started 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.

Note: The Transaction Recovery Service is designed to gracefully handle transaction recovery after a crash. BEA 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 crashes, you can manually migrate the Transaction Recovery Service from the crashed server to another server in the same cluster using the Administration Console or the command line interface. 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 files from the failed server for incomplete transactions and completes them as described in "Transaction Recovery Service Actions After a Crash" on page 4-14.

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 (including transaction log files) for the failed server so the failed server can reclaim it upon restart.

For instructions to migrate the Transaction Recovery Service using the Administration Console, see "Migrating the Transaction Recovery Service to a Server in the Same Cluster" on page 4-17.

A server can perform transaction recovery for more than one failed server. 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 will continue the transaction recovery. You can also manually migrate the Transaction Recovery Service back to the original failed server using the Administration Console or the command line interface. See "Manually Migrating the Transaction Recovery Service Back to the Original Server" on page 4-19 for more information.

When a backup server completes transaction recovery for a server, it releases ownership of the Transaction Recovery Service (and transaction logs) 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 will then complete 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 will 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 Administration Console or the command line interface.

Limitations of Migrating the Transaction Recovery Service

When 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.

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 files from the failed server. Therefore, you must store transaction log files on persistent storage available to both (or more) servers. BEA recommends that you store transaction log files on a Storage Area Network (SAN) device or a dual-ported disk. Do not use an NFS file system to store transaction log files. Because of the caching scheme in NFS, transaction log files on disk may not always be current. Using transaction log files stored on an NFS device for recovery may cause data corruption.

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.

Migrating the Transaction Recovery Service to a Server in the Same Cluster

To migrate the Transaction Recovery Service from a failed server in a cluster, follow these steps:

1. Make sure the failed or failing server is not running:
   a. Click the Servers node in the left pane in the Administration Console to expand it.
   b. Right-click the failed server and select Start/Stop this server.
   c. In the right pane of the Administration Console, click Graceful shutdown of this server. If that action fails, retry these steps and select Force shutdown of this server.
2. In the left pane in the Administration Console, select the failed server from which you want to migrate the Transaction Recovery Service. A dialog displays in the right pane with tabs for configuring the server.
3. Select the Control tab, then select the JTA Migration tab. The JTA Migration tab includes the following:

- Cluster—The name of the cluster to which the server belongs.
- Current Server—The server that currently owns the Transaction Recovery Service for the selected server. (The selected server name is displayed at the top of the page.)
- Destination Server—A list of servers in the cluster. This list shows all servers in the cluster or a list of servers that you specify. See "Constraining the Servers to Which the Transaction Recovery Service can Migrate" on page 4-18.

4. In Destination Server, select the server that you want to recover transactions for the failed server.
5. Click Migrate and follow any additional instructions in the right pane.

Note: The Transaction Recovery Service is designed to gracefully handle transaction recovery after a crash. BEA recommends that you attempt to restart a crashed server and allow the Transaction Recovery Service to handle incomplete transactions, rather than migrate the Transaction Recovery Service to another server.

Constraining the 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 files for a server. You can limit the list of destination servers available on the Server—Control—Migrate JTA tab in the Administration Console by following these instructions:

1. In the Administration Console, click the Servers node in the left pane to expand it.
2. Select the server for which you want to specify Transaction Recovery Service backup servers. A dialog displays in the right pane with tabs for configuring the server.
3. Select the Control tab, then select the JTA Migration Config tab.
4. In the Available list of Constrained Candidate Servers, select the servers you want to use as a Transaction Recovery Service backup and click the right arrow to move the servers to the Chosen list.

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 Administration Console enforces this rule.

5. Click Apply to save your changes.

Viewing Current Owner of the Transaction Recovery Service

When you migrate the Transaction Recovery Service to another server in the cluster, the backup server takes ownership of the Transaction Recovery Service until it completes all incomplete transactions. After which, it releases ownership of the Transaction Recovery Service and the original server can reclaim it. You can see the current owner on the Server→Control→Migrate JTA tab in the Administration Console. Follow these instructions:

1. In the Administration Console, click the Servers node in the left pane to expand it.
2. Select the server for which you want to see the owner of the Transaction Recovery Service. A dialog displays in the right pane with tabs for configuring the server.
3. Select the Control tab. If necessary, select the JTA Migration tab. On the JTA Migration tab, the Current Server indicates the current owner of the Transaction Recovery Service.

Manually Migrating the Transaction Recovery Service Back 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 will 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:

- A backup server will continue to recover incomplete transactions after you restart it. You will not need to manually migrate the Transaction Recovery Service back to the original server if the backup server completes the transaction recovery.
- If you restart the original server while the backup server is recovering transactions, the backup server will gracefully release ownership of the Transaction Recovery Service. You do not need to stop the backup server. See "Recovering Transactions for a Failed Clustered Server" on page 4-15.

1. Make sure the backup server is not running. To do this, click the Servers node in the left pane in the Administration Console to expand it, then right-click the backup server and select Stop this server. Follow any additional instructions.
2. In the Administration Console, click the Servers node in the left pane to expand it.

3. Select the failed server that you want to migrate the Transaction Recovery Service back to. A dialog displays in the right pane with tabs for configuring the server.
4. Select the Control tab. If necessary, select the JTA Migration tab.
5. In Destination Server, select the original server.
6. Click Migrate and follow any additional instructions in the right pane.

CHAPTER 5

Transaction Service

This section provides information that programmers need to write transactional applications for the WebLogic Server system.

This section discusses the following topics:

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 Enterprise JavaBeans Specification 2.0, published by Sun Microsystems, Inc.

For EJB and RMI applications, WebLogic Server also provides the `javax.transaction` and `javax.transaction.xa` packages, from Sun Microsystems, Inc., which implements the Java Transaction API (JTA) for Java applications. For more information about JTA, see the Java Transaction API (JTA) Specification 1.0.1a, published by Sun Microsystems, Inc. For more information about the `UserTransaction` object that applications use to demarcate transaction boundaries, see the WebLogic Server Javadoc.

Capabilities and Limitations

This section includes the following sections:

These sections describe the capabilities and limitations of the Transaction Service that supports EJB and RMI applications:

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` will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. The Transaction Service *provides* checked transaction behavior that is equivalent to that provided by the request/response interprocess 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 via 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.

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 will return 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 will throw an exception. In that case, you must reconfigure your application to work around the buffer size.

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, you must configure a common credential for all participating 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:

- **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 the online Javadoc.

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

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" in the "Introducing Transactions" section.

CHAPTER 6

Java Transaction API and BEA WebLogic Extensions

This section provides a brief overview of the Java Transaction API (JTA) and extensions to the API provided by BEA Systems.

This section discusses the following topics:

JTA API Overview

WebLogic Server supports the `javax.transaction` package and the `javax.transaction.xa` package, from Sun Microsystems, Inc., which implement the Java Transaction API (JTA) for Java applications. For more information about JTA, see the Java Transaction API (JTA) Specification (version 1.0.1a) published by Sun Microsystems, Inc. For a detailed description of the `javax.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, `javax.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, `javax.transaction.TransactionManager`. This interface is used by an EJB container as part of a container-managed transaction and uses the `javax.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, `javax.transaction.Status` and `javax.transaction.Synchronization`. These interfaces are accessed only by the transaction manager and cannot be used as part of an applications program.
- 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`). These interfaces are accessed only by the transaction manager and cannot be used as part of an applications program.

BEA 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.

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

- Client-initiated transactions—the JTA transaction manager interface (`javax.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.

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

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

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

- `weblogic.transaction.TransactionManager` (extends `javax.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

- `weblogic.transaction.Transaction` (extends `javax.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 allows 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 allows 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 one-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 `javax.transaction` and `javax.transaction.xa` interfaces, see the weblogic.transaction package description.

CHAPTER 7

Transactions in EJB Applications

This section includes the following topics:

This section describes how to integrate transactions in Enterprise JavaBeans (EJBs) applications that run under BEA WebLogic Server.

Before You Begin

Before you begin, you should read Chapter 2, "Introducing Transactions," particularly the following topics:

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

This document describes the BEA WebLogic Server implementation of transactions in Enterprise JavaBeans. The information in this document supplements the Enterprise JavaBeans Specification 2.0, published by Sun Microsystems, Inc.

Note: Before proceeding with the rest of this chapter, you should be familiar with the contents of the EJB Specification 2.0 document, particularly the concepts and material presented in Chapter 16, "Support for Transactions."

For information about implementing Enterprise JavaBeans in WebLogic Server applications, see Programming WebLogic Enterprise JavaBeans at `http://e-docs.bea.com/wls/docs90/ejb/index.html`.

General Guidelines

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 2.0 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. For more information, see *Programming WebLogic Server Enterprise Java Beans* at `http://e-docs.bea.com/wls/docs90/ejb/index.html`.

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

Transaction Attributes

This section includes the following sections:

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 16, "Support for Transactions," and Chapter 21, "Deployment Descriptor," in the EJB Specification 2.0, 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 16.4 in the EJB Specification 2.0.

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 *Programming WebLogic Server Enterprise JavaBeans* at `http://e-docs.bea.com/wls/docs90/ejb/index.html`.

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 16.7.2 in the EJB Specification 2.0.

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 javax.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 javax.ejb.EJBContext interface, where invocations are restricted by rules specified in Sections 16.4.4.2 and 16.4.4.3 of the EJB Specification 2.0.

Transaction Attributes for Bean-Managed Transactions

For bean-managed transactions, the bean specifies transaction demarcations using methods in the javax.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 nonterminated transaction.

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

Participating in a Transaction

When the EJB Specification 2.0 uses the phrase “participating in a transaction,” BEA 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

This topic contains the following sections:

The EJB Specification 2.0 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 `javax.transaction.UserTransaction` interface. EJB applications must be designed with these semantics in mind.

Transaction Semantics for Container-Managed Transactions

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

Transaction Semantics for Stateful Session Beans

Table 7-1 describes the transaction semantics for stateful session beans in container-managed transactions.

Table 7-1 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

Table 7-1 Transaction Semantics for Stateful Session Beans in Container-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
`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 7-2 describes the transaction semantics for stateless session beans in container-managed transactions.

Table 7-2 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 7-3 describes the transaction semantics for entity beans in container-managed transactions.

Table 7-3 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
`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

Table 7-4 describes the transaction semantics for stateful session beans in bean-managed transactions.

Table 7-4 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
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 7-5 describes the transaction semantics for stateless session beans in bean-managed transactions.

Table 7-5 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 `javax.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 `javax.ejb.SessionSynchronization` interface, see Section 6.5.3 in the EJB Specification 2.0.

Synchronization During Transactions

If a bean implements `SessionSynchronization`, the WebLogic Server EJB container will typically make the following callbacks to the bean during transaction commit time:

- `afterBegin()`
- `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 already 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. For more information, see the *Administration Guide*.

  The Bean Provider should configure the trans-timeout-seconds attribute in the weblogic-ejb-jar.xml deployment descriptor.

- **Bean-managed transactions**. An application calls the `UserTransaction.setTransactionTimeout` method.

Handling Exceptions in EJB Transactions

WebLogic Server EJB applications need to catch and handle specific exceptions thrown during transactions. For detailed information about handling exceptions, see Chapter 17, "Exception Handling," in the EJB Specification 2.0 published by Sun Microsystems, Inc.

For more information about how exceptions are thrown by business methods in EJB transactions, see the following tables in Section 17.3: Table 12 (for container-managed transactions) and Table 13 (for bean-managed transactions).

For a client's view of exceptions, see Section 17.4, particularly Section 12.4.1 (application exceptions), Section 17.4.2 (`java.rmi.RemoteException`), Section 17.4.2.1 (`javax.transaction.TransactionRolledBackException`), and Section 17.4.2.2 (`javax.transaction.TransactionRequiredException`).

CHAPTER 8

Transactions in RMI Applications

The following sections provide guidelines and additional references for using transactions in RMI applications that run under BEA WebLogic Server:

- Before You Begin
- General Guidelines

Before You Begin

Before you begin, read Chapter 2, "Introducing Transactions," particularly the following topics:

- "Transactions in WebLogic Server RMI Applications" on page 2-8
- "Transactions Sample RMI Code" on page 2-13

For more information about RMI applications, see *Programming WebLogic RMI over IIOP* at `http://e-docs.bea.com/wls/docs90/rmi_iiop/index.html`.

General Guidelines

The following general guidelines apply when 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 (`weblogic.rmic`) using the `-nontransactional` flag.

- 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."

CHAPTER 9

Using Third-Party JDBC XA Drivers with WebLogic Server

This section discusses the following topics:

Overview of Third-Party XA Drivers

This section provides an overview of using third-party JDBC two-tier 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.

Table of Third-Party XA Drivers

The following table summarizes known functionality of these third-party JDBC/XA drivers when used with WebLogic Server:

Table 9-1 Two-Tier JDBC/XA Drivers

Driver/Database Version	Comments
Oracle Thin Driver XA	See "Using Oracle Thin/XA Driver" on page 9-2.

Table 9-1 Two-Tier JDBC/XA Drivers

Driver/Database Version	Comments
IBM DB2 Type 2	See "Using the IBM DB2 Type 2 XA JDBC Driver" on page 9-7.
Sybase jConnect/XA • Version 5.5 • Adaptive Server Enterprise 12.0	See "Using Sybase jConnect 5.5/XA Driver" on page 9-8.

Third-Party Driver Configuration and Performance Requirements

Here are requirements and guidelines for using specific third-party XA drivers with WebLogic Server.

Using Oracle Thin/XA Driver

WebLogic Server ships with the Oracle Thin Driver version 10g (10.1.0.2.0) preconfigured and ready to use. If you want to update the driver or use a different version, see Using the Oracle Thin Driver in *Programming WebLogic JDBC* at `http://e-docs.bea.com/wls/docs90/jdbc/thirdparty.html#update_thin`.

The following sections provide information for using the Oracle Thin/XA Driver with WebLogic Server.

Software Requirements for the Oracle Thin/XA Driver

The Oracle Thin/XA Driver requires the following:

- Java 2 SDK 1.4.x or later. WebLogic Server requires a Java 2 SDK 1.4.*X* (and ships with SDK 1.4.1_*XX*).

  Note: The Oracle 10g and 9.2 Thin driver (`ojdbc14.jar`) is the only versions of the driver supported for use with a Java 2 SDK 1.4.*X*.

- Oracle server configured for XA functionality (limitation does not apply for non-XA usage).

Known Oracle Thin Driver Issues

Table 9-2 lists known issues and workarounds for the Oracle Thin driver. See the Oracle Web site for the most up-to-date information about these issues.

Table 9-2 Oracle Thin Driver Known Issues and Workarounds

Description	Oracle Bug	Comments/Workarounds for WebLogic Server
When using the 9.2.0.3 or earlier version of the Oracle Thin driver, after restarting WebLogic Server, you may see an XAER_PROTO error or an intermittent hang.	2717235	This situation occurs because on server restart, WebLogic Server calls `XA.recover()` to recover any pending transactions. With the 9.2.0.3 or earlier version of the Oracle Thin driver, the Oracle DBMS opens a local transaction to complete the transaction recovery work, but the local transaction is never closed. When the connection used to recover transactions is returned to the connection pool and is then reused by an application, the local transaction is still present. With the first operation on the connection, an XAER_PROTO error is thrown. (If `TestConnsOnReserve` is set to `true`, the connection test is the first operation on the connection.) WebLogic Server then attempts to unregister the connection with the resource and waits a fixed amount of time for all transaction work on the resource to complete. This may appear as a hang. Oracle has provided a patch for this bug. You can download the patch from the Oracle Metalink Web site at http://metalink.oracle.com. Refer to the Oracle bug number 2717235. This issue is fixed in version 9.2.0.4 and 10G.

Table 9-2 Oracle Thin Driver Known Issues and Workarounds

Description	Oracle Bug	Comments/Workarounds for WebLogic Server
The 9.2.0.1 and 9.2.0.2 versions of the Oracle Thin driver do not allow you to work with a BLOB in tables that also contain a long raw. When you retrieve a BLOB from the table and call `blob.length()`, you will get a SQL protocol violation.	2696397	This issue is fixed in version 9.2.0.3 and 10G.
When using the Oracle 9.2.0.1 or 9.2.0.2 Thin driver, you will get a null pointer exception when you run addBatch with setNull with a data conversion. For example, the following will fail with the Oracle 9.2.0 Thin driver: 1. `pstmt.setNull(1, java.sql.Types.REAL)` 2. `pstmt.addBatch()` 3. `pstmt.setNull(1, java.sql.Types.VARCHAR)`		This issue is fixed in version 9.2.0.3 and 10G.
The 9.2.0.1 and 9.2.0.2 versions of the Oracle Thin driver do not allow you to work with a CLOB in tables that also contain a long. When you retrieve a CLOB from the table and call `clob.length()`, you will get a SQL protocol violation.		Workaround: In this scenario, you can read the LONG column before calling `clob.length()`. This issue is fixed in version 9.2.0.3 and 10G.
The 9.2.0.1 and 9.2.0.2.0 versions of the Oracle Thin driver do not allow you to use "`alter session set NLS_DATE_FORMAT='YYYY-MM-DD HH24:MI:SS'`" to change the default Oracle timestamp format. Previous versions did allow this.	2632931	TAR number 2677656.995. Fixed in 9.2.0.2.1. Oracle has provided a patch for this bug. You can download the patch from the Oracle Metalink Web site at http://metalink.oracle.com. Refer to the Oracle bug number 2632931.
ORA-01453 - SET TRANSACTION must be first statement of transaction		When using the Oracle Thin/XA driver, you cannot change the transaction isolation level for a transaction. Transactions use the default transaction isolation as set for the database.

Table 9-2 Oracle Thin Driver Known Issues and Workarounds

Description	Oracle Bug	Comments/Workarounds for WebLogic Server
ORA-01002 - Fetch out of sequence exception. Iterating result set after XAResource.end(TMSUSPEND) and XAResource.start(TMRESUME) results in ORA-01002 This also occurs when an external client gets a result set using a pooled connection in WebLogic Server that uses the Oracle Thin driver. When the result set is sent to the client, the current transaction is suspended.	—	As a workaround, set the statement fetch size to be at least the result set size. This implies that the Oracle Thin Driver cannot be used on the client side or that the bean cannot keep result sets open across method invocations, unless this workaround is used. This is an Oracle limitation that Oracle does not intend to fix.
Does not support update with no global transaction. If there is no global transaction when an update is attempted, Oracle will start a local transaction implicitly to perform the update, and subsequent reuse of the same XA connection for global transaction will result in XAER_RMERR. Moreover, if application attempts to commit the local transaction via either setting auto commit to true or calling Connection.commit() explicitly, Oracle XA driver returns “SQLException: Use explicit XA call.”	—	Applications should always ensure that there is a valid global transaction context when using the XA driver for update. That is, ensure that bean methods have transaction attributes Required, RequiresNew, or Mandatory.

Set the Environment for the Oracle Thin/XA Driver

Configure WebLogic Server

See "Using the Oracle Thin Driver" in *Programming WebLogic JDBC* at `http://e-docs.bea.com/wls/docs90/jdbc/thirdparty.html#oracle_thin`.

Enable XA on the Database Server

To prepare the database for XA, perform these steps:

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

3. Grant the following permissions:
   - `grant select on v$xatrans$ to public (or <user>);`
   - `grant select on pending_trans$ to public;`
   - `grant select on dba_2pc_pending to public;`
   - `grant select on dba_pending_transactions to public;`
   - (when using the Oracle Thin driver 10.1.0.3 or later)
     `grant execute on dbms_system to <user>;`

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 will fail with `XAER_RMERR`. Crash recovery is a standard operation for an XA resource.

Oracle Thin/XA Driver Configuration Properties

The following table contains sample code for configuring a Connection Pool:

Oracle Thin/XA Driver: Connection Pool Configuration

Property Name	Property Value
Name	`jtaXAPool`
Targets	`myserver,server1`
URL	`jdbc:oracle:thin:@`*`serverName`*`:`*`port(typically 1521 on Windows)`*`:`*`sid`*
DriverClassname	`oracle.jdbc.xa.client.OracleXADataSource`
Initial Capacity	`1`
MaxCapacity	`20`
CapacityIncrement	`2`
Properties	`user=scott;password=tiger`

The following table contains sample attributes for configuring a TxDataSource. To create a TxDataSource from the Administration Console, select Honor Global Transactions when creating a data source.

Table 9-3 Oracle Thin/XA Driver: TxDataSource Configuration

Property Name	Property Value
Name	`jtaXADS`
Targets	`myserver,server1`
JNDIName	`jtaXADS`
PoolName	`jtaXAPool`

Using the IBM DB2 Type 2 XA JDBC Driver

The following sections describe how to set your environment to use the Type2 DB2 7.2/XA Driver with WebLogic Server.

For installation instructions and connection pool configuration instructions, see "Installing and Using the IBM DB2 Type 2 JDBC Driver" in *Programming WebLogic JDBC* at `http://e-docs.bea.com/wls/docs90/jdbc/thirdparty.html#db2`.

Set the Environment for the DB2 7.2/XA Driver

Set your environment as follows:

- Execute the batch file `usejdbc2.bat` located in the `<db2>/java12` directory to extract the correct version of the `db2java.zip` file and move it to the proper location. This enables the JDBC2.0 features of the driver. Make sure that no DB2 processes are running before executing this batch file.
- Include `<db2>/java/db2java.zip` in the `CLASSPATH` environment variable.
- Include `<db2>/bin` in `PATH` environment variable.

Where `<db2>` represents the directory in which the DB2 server is installed.

Limitation and Restrictions using DB2 as an XAResource

1. In case of multiple connection-pooled configurations, each connection pool should have separate database instance.

2. A transaction cannot be initiated with a resource that is already associated with a suspended transaction. In this case, a `javax.transaction.InvalidTransactionException` (attempt to resume an inactive transaction) is thrown. If in between `suspend` and `resume`, an intermediate transaction enlists the same resource as used in the suspended transaction, a `javax.transaction.invalidtransation` exception is thrown. If a different resource is used inside the intermediate transaction, it works fine.

Using Sybase jConnect 5.5/XA Driver

The following sections provide important configuration information and performance issues when using the Sybase jConnect Driver 5.5/XA Driver.

Known Sybase jConnect 5.5/XA Issues

These are the known issues and BEA workarounds:

Table 9-4 Sybase jConnect 5.5 Known Issues and Workarounds

Description	Sybase Bug	Comments/Workarounds for WebLogic Server
When calling setAutoCommit(true) the following exception is thrown: `java.sql.SQLException: JZ0S3: The inherited method setAutoCommit(true) cannot be used in this subclass.`	10726192	No workaround. Vendor fix required.
When driver used in distributed transactions, calling `XAResource.end(TMSUSPEND)` followed by `XAResource.end(TMSUCCESS)` results in `XAER_RMERR`.	10727617	WebLogic Server has provided an internal workaround for this bug: Set the connection pool property `XAEndOnlyOnce="true"`. Vendor fix has been requested.

Set Up the Sybase Server for XA Support

Follow these instructions to set up the environment on your database server:

- Install license for Distributed Transaction Management.
- Run s`p_configure "enable DTM",1` to enable transactions.

- Run `sp_configure "enable xact coordination",1.`
- Run grant role `dtm_tm_role` to `<USER_NAME>.`
- Copy the sample `xa_config` file from the `SYBASE_INSTALL\OCS-12_0\sample\xa-dtm` subdirectory up three levels to `SYBASE_INSTALL`, where `SYBASE_INSTALL` is the directory of your Sybase server installation. For example:

  ```
  $ SYBASE_INSTALL\xa_config
  ```

- Edit the `xa_config file`. In the first [`xa`] section, modify the sample server name to reflect the correct server name.

To prevent deadlocks when running transactions, enable row level lock by default:

- Run `sp_configure "lock scheme",0,` datarows

Note: Both the `jConnect.jar` and `jconn2.jar` files are included in the *`WL_HOME`*`\server\lib` folder and are referenced in the `weblogic.jar` manifest file. When you start WebLogic Server, the drivers are loaded automatically and are ready to use with WebLogic Server. To use these drivers with the WebLogic utilities or with other applications, you must include the path to these files in your `CLASSPATH`.

Notes About XA and Sybase Adaptive Server

Correct support for XA connections is available in the Sybase Adaptive Server Enterprise 12.0 and later versions only. XA connections with WebLogic Server are not supported on Sybase Adaptive Server 11.5 and 11.9.

Execution Threads and Transactions in Sybase Adaptive Server

Prior to Adaptive Server version 12.0, all resources of a transaction were privately owned by a single task on the server. The server could not share a transaction with any task other than the one that initiated the transaction. Adaptive Server version 12.x includes support for the suspend and join semantics used by XA-compliant transaction managers (such as WebLogic Server). Transactions can be shared among different execution threads, or may not be associated with an execution thread (detached).

Setting the Timeout for Detached Transactions

On the Sybase server, you can set the `dtm detach timeout period`, which sets the amount of time (in minutes) that a distributed transaction branch can remain in the detached state (without an associated execution thread). After this period, the DBMS automatically rolls back the

transaction. The `dtm detach timeout period` applies to all transactions on the database server. It cannot be set for each transaction.

For example, to automatically rollback transactions after being detached for 10 minutes, use the following command:

```
sp_configure 'dtm detach timeout period', 10
```

You should set the `dtm detach timeout period` higher than the transaction timeout to prevent the database server from rolling back the transaction before the transaction times out in WebLogic Server.

For more information about the `dtm detach timeout period`, see the Sybase documentation.

Transaction Behavior on Sybase Adaptive Server

If a global transaction is started on the Sybase server, but is not completed, the outcome of the transaction varies depending on the transaction state before the transaction is abandoned:

- If the client is terminated before the `xa.end` call, the transaction is rolled back.
- If the client is terminated after the `xa.end` call, the transaction remains on the database server (and holds all relevant locks).
- If an application calls `xa.start` but has not called `xa.end` and the application terminates unexpectedly, the database server immediately rolls back the transaction and frees locks held by the transaction.
- If an application calls `xa.start` and `xa.end` and the application terminates unexpectedly, the database server rolls back the transaction and frees locks held by the transaction *after the* `dtm detach timeout period` *has elapsed*. See "Setting the Timeout for Detached Transactions" on page 9-9.
- If an application calls `xa.start` and `xa.end`, and then the transaction is prepared, if the application terminates unexpectedly, the transaction will persist so that it can be properly recovered. The Transaction Manager must call rollback or commit to complete the transaction.

Connection Pools for the Sybase jConnect 5.5/XA Driver

The following table contains sample code for configuring a Connection Pool:

Table 9-5 Sybase jConnect 5.5/XA Driver: Sample Connection Pool Configuration

Property Name	Property Value
Name	`jtaXAPool`
Targets	`myserver,server1`
DriverClassname	`com.sybase.jdbc2.jdbc.SybXADataSource`
Properties	User=*dbuser*; DatabaseName=*dbname*; ServerName=*server_name_or_IP_address*; PortNumber=*serverPortNumber*; NetworkProtocol=Tds; resourceManagerName=*Lrm_name_in_xa_config*; resourceManagerType=2
Initial Capacity	`1`
MaxCapacity	`10`
CapacityIncrement	`1`
Supports Local Transaction	True

Where *`Lrm_name`* refers to the Logical Resource Manager name.

The following table contains sample code for configuring a TxDataSource:

Table 9-6 Sybase jConnect 5.5/XA Driver: TxDataSource Configuration

Property Name	Property Value
Name	`jtaXADS`
Targets	`server1`
JNDIName	`jtaXADS`
PoolName	`jtaXAPool`

Configuration Properties for Java Client

Set the following configuration properties when running a Java client.

Table 9-7 Sybase jConnect 5.5/XA Driver: Java Client Connection Properties

Property Name	Property Value
ds.setPassword	<*password*>
ds.setUser	<*username*>
ds.setNetworkProtocol	Tds
ds.setDatabaseName	<*database-name*>
ds.setResourceManagerName	<*Lrm name in xa_config file*>
ds.setResourceManagerType	2
ds.setServerName	<*machine host name*>
ds.setPortNumber	*port* (Typically 4100)

Other Third-Party XA Drivers

To use other third-party 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.

CHAPTER 10

Coordinating XAResources with the WebLogic Server Transaction Manager

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 is done 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 Chapter 11, "Participating in Transactions Managed by a Third-Party Transaction Manager." Note that WebLogic Server IIOP, WebLogic Tuxedo Connector (WTC) gateway, and BEA Java Adapter for Mainframe (JAM) gateway internally use the same mechanism described in these chapters to import and export transactions in WebLogic Server.

The following sections describe how to configure third-party systems to participate in transactions coordinated by the WebLogic Server transaction manager:

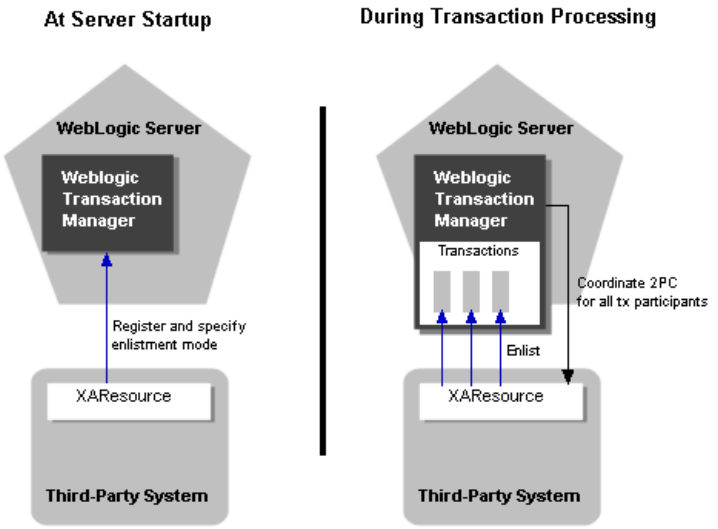
Overview of Coordinating Distributed Transactions with Foreign XAResources

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 J2EE Javadocs at `http://java.sun.com/j2ee/sdk_1.3/techdocs/api/index.html`.

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

Figure 10-1 shows the process for third-party systems to participate in transactions coordinated by the WebLogic Server transaction manager.

Figure 10-1 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" on page 10-6. For more information about registering XAResource objects with the WebLogic Server transaction manager, see "Registering an XAResource to Participate in Transactions" on page 10-3.

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 (i.e., 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 needs to enlist an XAResource instance with the JTA `javax.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" on page 10-6.

- Bootstrap the `XAResource` in the event that the WebLogic Server transaction manager must perform crash recovery. (The JTA Specification does not define a standard API to do so; refer to JTA 1.0.1 Specification Section 3.4.8 for details).

  The JTA 1.0.1 specification section 3.4.9 suggests that the transaction manager is responsible for assigning the branch qualifiers. However, for recovery to work properly, the same transaction branch qualifier needs to be supplied both at normal processing and upon crash recovery. Therefore, registration with the WebLogic Server transaction manager is required to support crash recovery because the transaction branch qualifier is specified during registration.

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

  – It reads its transaction log 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 already 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 will use 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();
```

   or

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

   or

```
TransactionManager tm =
(TransactionManager)initCtx.lookup("javax.transaction.TransactionManage
r");
```

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" on page 10-6 for a detailed discussion of the different enlistment modes. Note that when you register the XAResource, you specify the enlistment mode that will be used subsequently, 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. Make sure 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
- 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 XAResource to Participate in Transactions" on page 10-3.

`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 already 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. See the *WebLogic Server Javadocs* for more details.

Standard Enlistment

With standard enlistment mode, you need to enlist the XAResource instance only once with the Transaction object. Also, it is possible to enlist more than one XAResource instance 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 will be 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:

```
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 is 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

Note: Consider the following before using the static enlistment mode:

- Static enlistment mode eliminates the need 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 will either use 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 will receive 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 make sure 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 needs to 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.

The WebLogic Server transaction manager checkpoints a new XAResource in its transaction log when it 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 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 make sure 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 already 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 already

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 will cause 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.

Note: It is possible that transactions that have already been completed successfully are re-committed upon server restart. This happens because of a WebLogic Server transaction log optimization: the WebLogic Server transaction manager never deletes individual log records from the transaction log file, but waits until all the transactions of a particular log file are completed successfully and deletes the whole file. As a result, upon server restart, some of the transactions read from a particular log file may have already completed successfully.

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 will declare it dead. Any further requests to the XAResource are shunned, and an `XAER_RMFAIL XAException` is thrown.

The two minute interval can be configured via the `maxXACallMillis` JTAMBean attribute. It is not exposed through the Administration Console. 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

`javax.transaction.xa.XAResource`) and register it with the transaction manager. The transaction manager will call the `detectUnavailable` method of the XAResource when it is about to declare it unavailable. If the XAResource returns `true`, then it will not be declared unavailable. If the XAResource is indeed unavailable, it can use this opportunity to perform cleanup and re-registration with the transaction manager. See the *WebLogic Server Javadocs* for `weblogic.transaction.XAResource` for more information.

J2EE Connector Architecture Resource Adapter

Besides registering with the WebLogic Server transaction manager directly, you can also implement the J2EE Connector Architecture resource adapter interfaces. When you deploy the resource adapter, the WebLogic Server J2EE container will register the resource manager's XAResource with the WebLogic Server transaction manager automatically.

For more information, see *Programming WebLogic J2EE Connectors*.

Implementation Tips

The following sections provide tips for exporting and importing transactions with the WebLogic Server transaction manager:

- "Sharing the WebLogic Server Transaction Log" on page 10-12
- "Transaction global properties" on page 10-13
- "TxHelper.createXid" on page 10-14

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 will only remove 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. See the *WebLogic Server Javadocs* for the `weblogic.transaction.TransactionLogger` interface for more details and the usage of the `TransactionLogger` interface.

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 the *WebLogic Server Javadocs* for the `weblogic.transaction.TxHelper` class for more information.

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 the *WebLogic Server Javadocs* for the `weblogic.transaction.TxHelper` class for more information.

FAQs

- XAResource's Xid has a branch qualifier, but not the transaction manager's transaction.

WebLogic Server JTA transaction objects do not have branch qualifiers (i.e., `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

Refer to the JTA specification 1.0.1B 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 `http://java.sun.com/products/jta/`.

CHAPTER 11

Participating in Transactions Managed by a Third-Party Transaction Manager

WebLogic Server can participate in distributed transactions coordinated by third-party systems (referred to as foreign transaction managers). The WebLogic Server processing is done 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.

The following sections describe the process for configuring and participating in foreign-managed transactions:

Overview of Participating in Foreign-Managed Transactions

The WebLogic Server transaction manager exposes a `javax.transaction.xa.XAResource` implementation via 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.

When importing a transaction from a foreign transaction manager into the WebLogic Server transaction manager, you must register the WebLogic Server interposed transaction manager 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 11-1 shows the interaction between the two transaction managers and the client-side gateway.

Figure 11-1 Importing Transactions into WebLogic Server Using a Client-Side Gateway

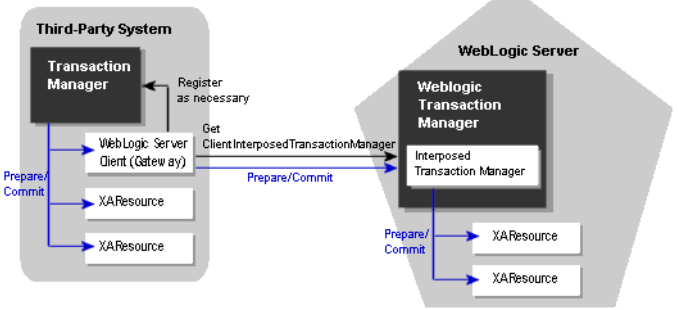
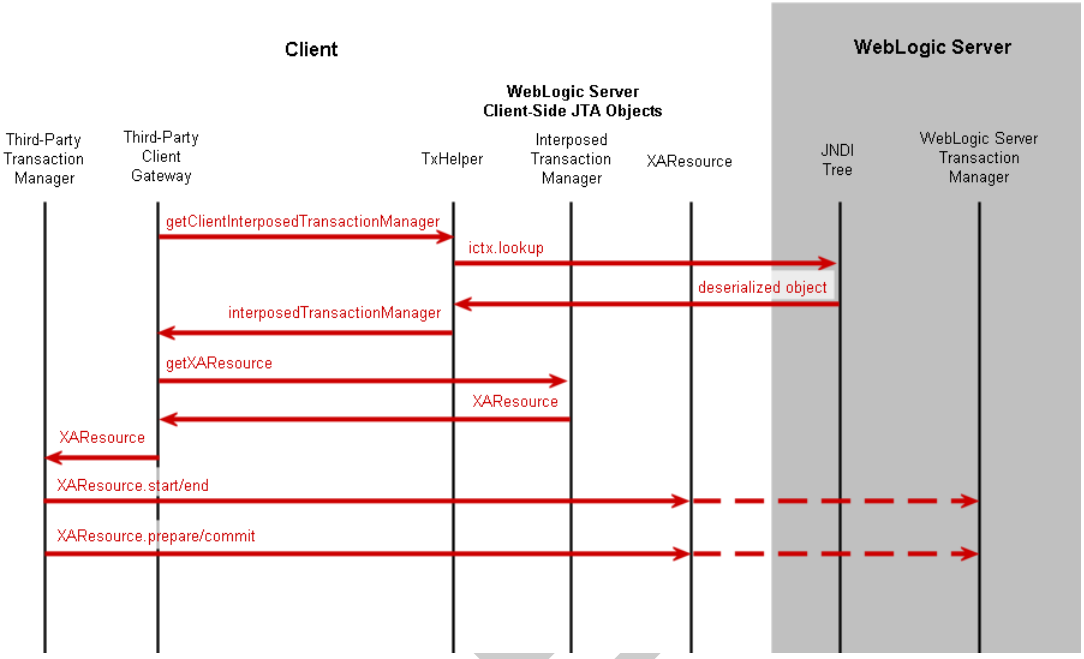


Figure 11-2 shows the flow of interactions between a foreign transaction manager, WebLogic Server client-side JTA objects, and the WebLogic Server transaction manager.

Figure 11-2 State Diagram Illustrating Steps to Import a Transaction Using the Client Interposed Transaction Manager



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

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 needs to contact the same WebLogic Server server to obtain the list of in-doubt transactions that were previously imported into WebLogic Server.

See the *WebLogic Server Javadocs* for the `weblogic.transaction.TxHelper` class for more information.

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();
```

Limitations of the Client Interposed Transaction Manager

Note the following limitations when importing transactions using a client-side gateway:

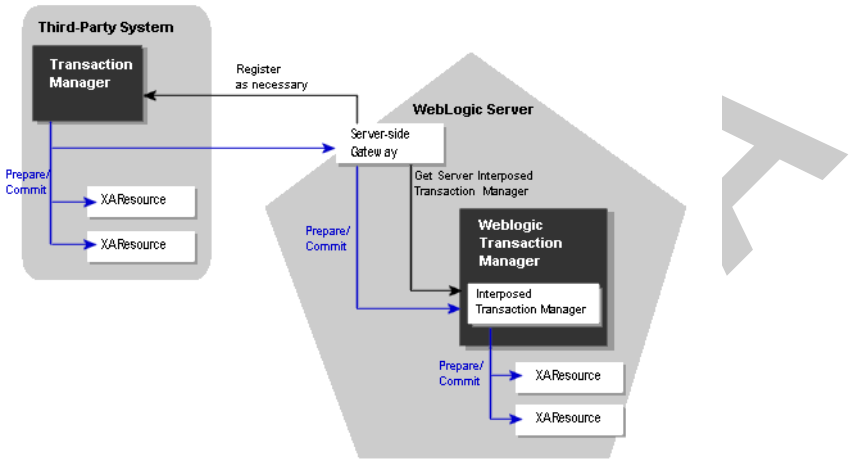
- You cannot use the `TxHelper.getServerInterposedTransactionManager()` method in client-side gateways.
- You can only use *one* WebLogic Server client interposed transaction manager at a time. Do not use more than one client interposed transaction manager (connecting to different WebLogic Server servers) to import transactions at the same time. (See "Transaction Processing for Imported Transactions" on page 11-7 for more information about this limitation and how transactions are processed with the WebLogic Server interposed transaction manager.)

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 manger. The WebLogic Server transaction manager then acts as a subordinate transaction manager and coordinates the transaction participants within WebLogic Server. Figure 11-3 shows the interaction between the two transaction managers and the server-side gateway.

Figure 11-3 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
- Get the XAResource from the 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 the *WebLogic Server Javadocs* for the `weblogic.transaction.TxHelper` class for more information.

After you get the interposed transaction manager, you must get the XAResource. See "Get the XAResource from the Interposed Transaction Manager" on page 11-5.

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 will cause performance issues.
- You can only use *one* WebLogic Server server interposed transaction manager at a time. Do not use more than one server interposed transaction manager (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

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 will 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 already been imported previously on the same WebLogic Server server, WebLogic Server will associate 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);
```

Note that the WebLogic Server transaction manager ignores the flag.

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) will throw an `XAException` with `XAER_RMERR`. With a server-side gateway, attempting to call `xar.start` on a client or server interposed transaction manager will also throw a `XAException` with `XAER_RMERR` if there is already 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 will be 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 will be in a separate branch from the original WebLogic Server transaction.
- A foreign transaction manager should make sure 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 and overwriting of TLOG files.

Commit Processing for 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 (e.g., 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 section 3.4 of the JTA 1.0.1b specification:

- 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 more than one WebLogic Server participants for the imported transaction, the transaction manager logs a prepare record even if the `XAResource.commit` is a one-phase commit.

The WebLogic Server transaction manager logs a heuristic record for the imported transaction if there is heuristic outcome for `XAResource.commit` or `rollback`. The heuristic log records are stored in heuristic log files (separate from the transaction log files). The heuristic log files have extensions `heur.tlog`.

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.

JCA Resource Adapter

The current JCA specification version does not have provisions for importing transactions. The JCA 1.5 specification will have APIs for importing transactions into the J2EE container. When the WebLogic Server JCA container supports the JCA 1.5 specification, resource providers can then also import transactions via JCA APIs.

Troubleshooting Transactions

This section describes troubleshooting tools and tasks for use in determining why transactions fail and deciding what actions to take to correct the problem.

This section discusses the following topics:

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 the following aids to transaction troubleshooting:

- "Transaction Monitoring and Logging" on page 12-3

Exceptions

WebLogic JTA supports all standard JTA exceptions. For more information about standard JTA exceptions, see the Javadoc for the `javax.transaction` and `javax.transaction.xa` package APIs.

In addition to the standard JTA exceptions, WebLogic Server provides the class `weblogic.transaction.RollbackException`. This class extends `javax.transaction.RollbackException` and preserves the original reason for a rollback. 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.

Transaction Identifier

The Transaction Service assigns a transaction identifier (`Xid`) to each transaction. This ID can be used to 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 the `weblogic.transaction.Transaction` Javadoc.

Transaction Name and Properties

WebLogic JTA provides extensions to `javax.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 the `weblogic.transaction.Transaction` Javadoc.

Transaction Status

The Java Transaction API provides transaction status codes using the `javax.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 `javax.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 the Administration Console Online Help.

Transaction Monitoring and Logging

The Administration Console allows you to monitor transactions and to specify the transaction log file prefix. Monitoring and logging tasks are performed at the server level. Transaction statistics are displayed for a specific server and each server has a transaction log file.

CHAPTER 13

UsingLLR Enabled Data Sources

See "Understanding the Logging Last Resource Transaction Option" in *Configuring and Managing WebLogic JDBC*.

Glossary of Terms

local transaction

Transactions that are local to a single resource manager only; for example a transaction that relates to only one database.

distributed transaction

Global transaction involving multiple servers and one or more resources. In a distributed transaction environment, a client application may send requests to several servers resulting in resource updates at multiple resource managers. To complete a transaction, the transaction manager for each participant (client, servers, and resource managers) must be polled to coordinate the commit process for each participant within its domain.

global transactions

Transaction managed by an external transaction manager (such as WebLogic Server) that can include multiple servers or multiple resources as participants. The transaction is coordinated as an atomic unit of work: All participants either commit or rollback the entire transaction.

transaction branches

Each resource manager's internal unit of work in support of a global transaction is part of exactly one transaction branch. Each Global Transaction Identifier (GTRID or XID) that the transaction manager gives to the resource manager identifies both a distributed transaction and a specific branch.

heuristic decision

An heuristic decision (or heuristic completion) occurs when a resource makes a unilateral decision during the completion stage of a distributed transaction to commit or rollback updates. This can leave distributed data in an indeterminate state. Network failures or transaction timeouts are possible causes for a heuristic decision.

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.

Index

S

T

U

X