# BEA WebLogic Server™

## and BEA WebLogic Express™

## Programming WebLogic JTA

BEA WebLogic Server Version 6.1
Document Date: June 24, 2002

## Copyright



## Restricted Rights Legend



## Trademarks or Service Marks



**Programming WebLogic JTA**

| Part Number | Document Date | Software Version |
|---|---|---|
| N/A | June 24, 2002 | BEA WebLogic Server 6.1 |

# Contents

## About This Document



## 1. Introducing Transactions

# About This Document

This document explains how to use transactions in EJB and RMI applications that run in the BEA WebLogic Server™ environment.

This document is organized as follows:

- Chapter 1, “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 2, “Configuring and Managing Transactions,” describes how to administer transactions in the WebLogic Server environment.
- Chapter 3, “Transaction Service,” describes the WebLogic Server Transaction Service.
- Chapter 4, “Java Transaction API and BEA WebLogic Extensions,” provides a brief overview of the Java Transaction API (JTA).
- Chapter 5, “Transactions in EJB Applications,” describes how to implement transactions in EJB applications.
- Chapter 6, “Transactions in RMI Applications,” describes how to implement transactions in RMI applications.
- Chapter 7, “Using Third-Party JDBC XA Drivers with WebLogic Server,” describes how to configure and use third-party XA drivers in transactions.
- Chapter 8, “WebLogic Server XA Resource Provider Requirements,” describes the requirements for XA resources that participate in distributed transactions in WebLogic Server.
- Chapter 9, “Troubleshooting Transactions,” describes how to perform troubleshooting tasks for applications using JTA.

# 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.

# e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the WebLogic Server Product Documentation page at http://e-docs.bea.com/wls/docs60.

# How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

# Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version your are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

# Documentation Conventions

The following documentation conventions are used throughout this document.

| Convention | Usage |
|---|---|
| Ctrl+Tab | Keys you press simultaneously. |
| *italics* | Emphasis and book titles. |

| Convention | Usage |
|---|---|
| `monospace text` | Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.<br>*Examples*:<br>`import java.util.Enumeration;`<br>`chmod u+w *`<br>`config/examples/applications`<br>`.java`<br>`config.xml`<br>`float` |
| *`monospace italic text`* | Variables in code.<br>*Example*:<br>`String CustomerName;` |
| UPPERCASE TEXT | Device names, environment variables, and logical operators.<br>*Examples*:<br>LPT1<br>BEA_HOME<br>OR |
| `{ }` | A set of choices in a syntax line. |
| `[ ]` | Optional items in a syntax line. *Example*:<br>`java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]` |
| `\|` | Separates mutually exclusive choices in a syntax line. *Example*:<br>`java weblogic.deploy [list\|deploy\|undeploy\|update] password {application} {source}` |
| `...` | Indicates one of the following in a command line:<br>■ An argument can be repeated several times in the command line.<br>■ The statement omits additional optional arguments.<br>■ You can enter additional parameters, values, or other information |

| Convention | Usage |
| --- | --- |
| .<br>.<br>. | Indicates the omission of items from a code example or from a syntax line. |

# 1 Introducing Transactions

The following sections provide an overview of WebLogic transactions in WebLogic Server applications:

- ACID Properties of Transactions
- Supported Programming Model
- Supported API Models
- Distributed Transactions and the Two-Phase Commit Protocol
- Support for Business Transactions
- When to Use Transactions
- When Not to Use Transactions
- What Happens During a Transaction
- Transactions Sample Code

## ACID Properties of Transactions

One of the most fundamental features of the WebLogic Server system is transaction management. Transactions are a means to guarantee that database transactions 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 permanent; 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 1.0.1, 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.

For example, consider an Internet-based online shopping cart application. Users of the client application browse through an online catalog and make multiple purchase selections. When the users are done choosing all the items they want to buy, they proceed to check out and enter their credit card information to make the purchase. If the credit card check fails, the shopping application needs a way to cancel all the pending purchase selections in the shopping cart, or roll back any purchase transactions made during the conversation.

- 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

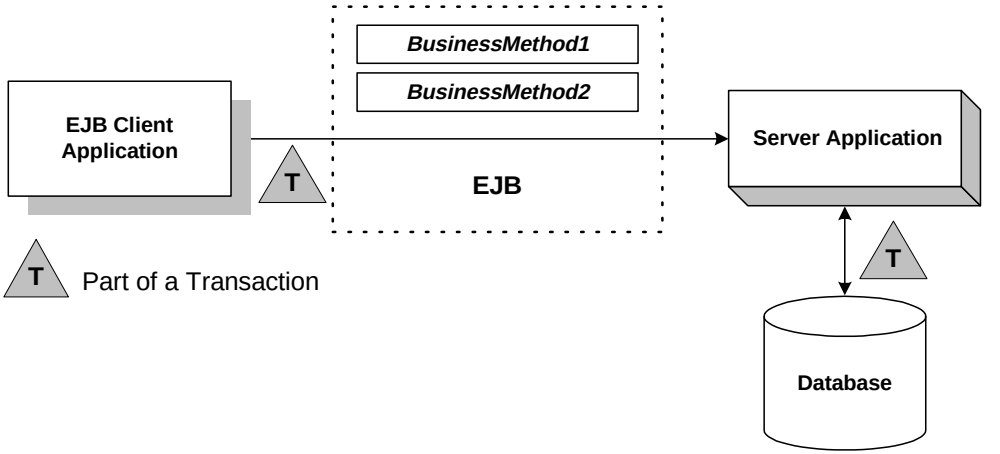
This topic includes the following sections:

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

## Introducing Transactions in WebLogic Server EJB Applications

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

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



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

- In **container-managed transactions**, the WebLogic Server EJB container manages the transaction demarcation. Transaction attributes in the EJB deployment descriptor determine how the WebLogic Server EJB container handles transactions with each method invocation. For more information about the deployment descriptor, see *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 Javadoc.

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 16.7.2 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).

You can control transaction timeouts by setting the `trans-timeout-seconds` element using the Administration Console.

## 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.

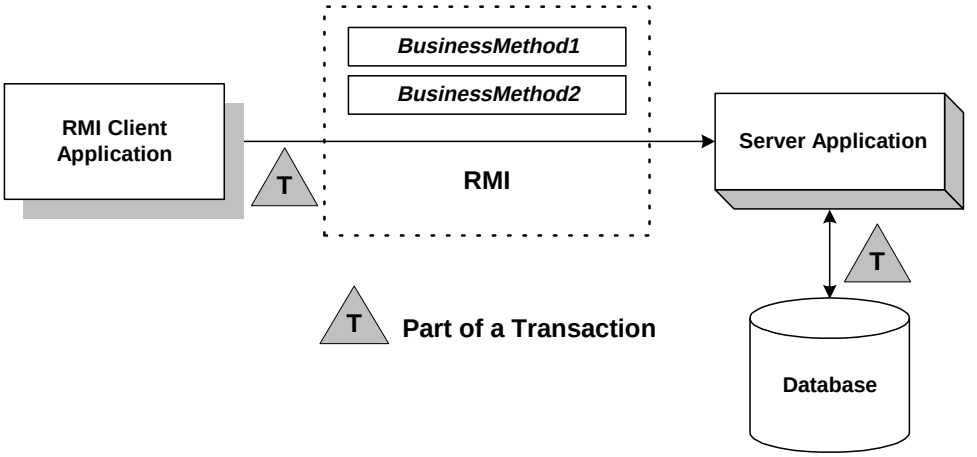
- If no exceptions occur, the client application commits the current transaction using the `UserTransaction.commit` method. This method ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

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.

# Introducing Transactions in WebLogic Server RMI Applications

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

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




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

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

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

2. The client application begins a transaction using the `UserTransaction.begin` method, and issues a request to the server application. All operations on the server application execute within the scope of a transaction.

   - If a call to any of these operations raises an exception (either explicitly or 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 6, “Transactions in RMI Applications.”

# Transactions Sample Code

This topic includes the following sections:

- Transactions Sample EJB Code
- Transactions Sample RMI Code

# Transactions Sample EJB Code

This topic 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:** 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.

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.

## Importing Packages

Listing 1-1 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 1-1 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.*;
```

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

## Using JNDI to Return an Object Reference

Listing 1-2 shows using JNDI to look up an object reference.

**Listing 1-2 Performing a JNDI Lookup**

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

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

// Parameters for the WebLogic Server.
// Substitute the corect 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 1-3 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 1-3 Starting a Transaction**

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

## Completing a Transaction

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

- If an exception was thrown during any of the database operations, the application calls the `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 1-4 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 6, "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 1-5 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 1-5 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 1-6 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 1-6 Performing a JNDI Lookup**

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

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

// Parameters for the WebLogic Server.
// Substitute the corect 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 1-7 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 1-7 Starting a Transaction**

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

## Completing a Transaction

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

- If an exception was thrown, the application calls the `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 1-8 Completing a Transaction**

```
tx.commit();

// or:

tx.rollback();
```

# 2 Configuring and Managing Transactions

The following sections provides an overview of commonly performed administration tasks related to transactions. For general information on JTA configuration tasks, see Managing Transactions in the *Administration Guide*. For information on specific configuration attributes and procedures, see the JTA topic in the Administration Console Online Help.

- Configuring Transactions
- Monitoring Transactions
- Adding a Transactional Resource Manager

## Configuring Transactions

The Administration Console provides the interface used to configure features of WebLogic Server, including WebLogic JTA. To invoke the Administration Console, refer to the procedures described in Configuring WebLogic Servers and Clusters. 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.

# Monitoring Transactions

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.

## 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. For more information on setting server logging attributes, see the Server topic in the Administration Console Online Help. For information on using logging in troubleshooting and debugging, see "Transaction Log" in Chapter 9, "Troubleshooting Transactions."

## Statistics

WebLogic Server keeps statistics on transactions organized by server, resource, and transaction name. For more information on viewing statistics, see the JTA topic in the Administration Console Online Help. For information on using statistics in troubleshooting and debugging, see "Transaction Statistics" in Chapter 9, "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. For more information on monitoring transactions, see the Server topic in

the Administration Console Online Help. For more information on using monitoring data in troubleshooting, see "Transaction Monitoring," in Chapter 9, "Troubleshooting Transactions."

# Adding a Transactional Resource Manager

A transactional resource manager provides access to a collection of information and processes. Transaction-aware JDBC drivers are common resource managers. When adding a JDBC driver, you must configure driver properties for proper operation with JTA. See Managing Transactions in the *Administration Guide* for JDBC configuration guidelines.

*CHAPTER*

# 3 Transaction Service

The following sections provide information that programmers need to write transactional applications for the WebLogic Server system:

- About the Transaction Service
- Capabilities and Limitations
- Transaction Service in EJB Applications
- Transaction Service in RMI Applications

## 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 the JTA, see the Java Transaction API (JTA) Specification 1.0.1, 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

The following 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. The `Transaction` object returned from JNDI supports both the `UserTransaction` and the `TransactionManager` interfaces.

## 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.

# 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, and between clustered servers within a domain. Transactions between multiple domains are not supported.

# 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 "Introducing 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 "Introducing Transactions in WebLogic Server RMI Applications," and "Transactions Sample RMI Code" in the "Introducing Transactions" section.

# 4 Java Transaction API and BEA WebLogic Extensions

The following sections provide a brief overview of the Java Transaction API (JTA) and extensions to the API provided by BEA Systems.

- JTA API Overview
- BEA WebLogic Extensions to JTA

## 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 1.0.1) 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 both within and between clusters.

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`
  - `getTransaction`
  - `forceResume` and `forceSuspend`

- `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`
  - `getStatusAsString`
  - `getMillisSinceBegin`
  - `getTimeToLiveMillis`

- `weblogic.transaction.TxHelper`

  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`

For a detailed description of the WebLogic extensions to the `javax.transaction` and `javax.transaction.xa` interfaces, see the `weblogic.transaction` package description.

# 5 Transactions in EJB Applications

The following sections describe the behavior and use of transactions in EJB applications:

- Before You Begin
- General Guidelines
- Transaction Attributes
- Participating in a Transaction
- Transaction Semantics
- Session Synchronization
- Synchronization During Transactions
- Setting Transaction Timeouts
- Handling Exceptions in EJB Transactions

This topic 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 1, "Introducing Transactions," particularly the following topics:

- Introducing 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 EJB*.

# 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.
- WebLogic Server supports any JTA-compliant XA resource. For information on the XA resource driver supplied with WebLogic Server, see "Transactions and the WebLogic jDriver for Oracle" in *Installing and Using WebLogic jDriver for Oracle* at http://e-docs.bea.com/wls/docs61/oracle/trxjdbcx.html.

- 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 "The WebLogic Server EJB Container" in *Programming WebLogic Server Enterprise Java Beans* at http://e-docs.bea.com/wls/docs61/ejb/EJB_environment.html.

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

# Transaction Attributes

This topic includes the following sections:

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

## About Transaction Attributes for EJBs

Transaction attributes determine how transactions are managed in EJB applications. For each EJB, the transaction attribute specifies whether transactions are demarcated by the WebLogic Server EJB container (container-managed transactions) or by the EJB itself (bean-managed transactions). The setting of the `transaction-type` element in the deployment descriptor determines whether an EJB is container-managed or bean-managed. See Chapter 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 "WebLogic Server 6.1 Properties" in *Programming WebLogic Server Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/reference.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 transaction attribute, trans-timeout-seconds, is based on BEA WebLogic JTA extensions. 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 trans-`timeout-seconds` configuration parameter. The default timeout value is 30 seconds.

For more information on transaction configuration parameters, see Chapter 2, "Configuring and Managing Transactions," in this guide and Managing Transactions in the *Administration Guide*.

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:

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

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 5-1 describes the transaction semantics for stateful session beans in container-managed transactions.

**Table 5-1 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 |

**Table 5-1 Transaction Semantics for Stateful Session Beans in Container-Managed Transactions (Continued)**

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

## Transaction Semantics for Stateless Session Beans

Table 5-2 describes the transaction semantics for stateless session beans in container-managed transactions.

**Table 5-2 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 5-3 describes the transaction semantics for entity beans in container-managed transactions.

**Table 5-3 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 5-4 describes the transaction semantics for stateful session beans in bean-managed transactions.

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

**Table 5-5 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 timout before you `begin()` the transaction. Setting a timeout does not affect the current transaction. This is different from earlier versions of WebLogic Server, in which timeouts affected the current transaction.

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 deplopyment 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`).

# 6 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 Introducing Transactions, particularly the following topics:

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

For more information about RMI applications, see *Programming WebLogic RMI and RMI/IIOP*.

## 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. For more information about callback objects, see *Programming WebLogic RMI and RMI/IIOP*.
- 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."

# 7 Using Third-Party JDBC XA Drivers with WebLogic Server

The following sections describe how to use JDBC XA drivers in WebLogic Server transactions:



## Overview of Third-Party XA Drivers

This section provides an overview to using third-party JDBC two-tier drivers with WebLogic Server in distributed transactions. These drivers provide connectivity between WebLogic Server 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 6.1:

**Table 7-1 Two-Tier JDBC/XA Drivers**

| Driver/Database Version | Comments |
|---|---|
| **Type 2 XA Drivers (native .dll)** | |
| IBM DB2<br>■ Version 7.2<br>■ Platform: NT | See "Using DB2 7.2/XA Driver" on page 7-13. |
| **Type 4 XA Drivers (all-Java)** | |
| Oracle Thin Driver XA<br>■ Driver version 8.1.7<br>■ Database version 8.1.7 | See "Using Oracle Thin 8.1.7/XA Driver" on page 7-5. |
| Sybase jConnect/XA<br>■ Version 5.2.1<br>■ Adaptive Server Enterprise 12.0 | See "Using Sybase jConnect 5.2.1/XA Driver" on page 7-8. |
| Cloudscape<br>■ Version 3.5.1 | See "Software Requirements for the Cloudscape 3.5.1/XA Driver" on page 7-11. |

# Third-Party Driver Configuration and Performance Requirements

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

**Note:** You may need to set additional connection pool properties when using third-party drivers not listed here. See Additional XA Connection Pool Properties in the *Administration Guide* at `http://e-docs.bea.com/wls/docs61/adminguide/jdbc.html#addxaprops`.

# Using Oracle Thin 8.1.7/XA Driver

The following sections provide information for using the Type 4 Oracle Thin 8.1.7/XA Driver with WebLogic Server 6.1.

## Software Requirements for the Oracle Thin 8.1.7/XA Driver

The Oracle Thin 8.1.7/XA Driver requires the following:

- JDK 1.2.x
- Oracle 8.1.7 server in order to have XA functionality (limitation does not apply for non-XA usage)

## Known Oracle Thin 8.1.7/XA Issues

These are the known issues and BEA workarounds::

**Table 7-2 Oracle Thin Driver Known Issues and Workarounds**

| Description | Oracle Bug | Comments/Workarounds for WebLogic Server 6.1 |
|---|---|---|
| ORA-01002 - Fetch out of sequence exception. Iterating result set after XAResource.end(TMSUSPEND) and XAResource.start(TMRESUME) results in ORA-01002 | — | As a workaround, set the statement fetch size to be at least the result set size. This implies that the Oracle Thin 8.1.7 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. |
| XAResource.end(TMSUSPEND) followed by XAResource.end(TMSUCCESS) gives XAER_RMERR. | 1527725 | WebLogic Server has provided an internal workaround for this bug. |

**Table 7-2 Oracle Thin Driver Known Issues and Workarounds**

| Description | Oracle Bug | Comments/Workarounds for WebLogic Server 6.1 |
|---|---|---|
| Driver hangs or gives `XAER_RMERR` for multi-threaded XA usage. | 1569235 | WebLogic Server has provided an internal workaround for this bug. |
| 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.<br>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. |
| XAResource.recover repeatedly returns the same set of in-doubt Xids irrespective of the input flag. According to the XA spec, the Transaction Manager should initially call XAResource.recover with TMSTARTRSCAN and then call XAResource.recover with TMNOFLAGS repeatedly until no Xids are returned. This Oracle bug could lead to infinite recursion and subsequent running out of Oracle cursors with error "ORA-01000: maximum open cursors exceeded." | — | Weblogic Server provides an internal workaround for this issue. |

Set the Environment for the Oracle Thin 8.1.7/XA Driver

Set the environment as follows:

Enable the database server for XA

- Log on to sqlplus as system user, e.g. `sqlplus sys/CHANGE_ON_INSTALL@<DATABASE ALIAS NAME>`
- Execute the sql: grant select on `DBA_PENDING_TRANSACTIONS` to `public`

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 crash, recover for the Oracle resource will fail with `XAER_RMERR`.

## Oracle Thin 8.1.7/XA Driver Configuration Properties

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

**Oracle Thin 8.1.7/XA Driver: Connection Pool Configuration**

| Property Name | Property Value |
|---|---|
| Name | `jtaXAPool` |
| Targets | `myserver,server1` |
| URL | `jdbc:oracle:thin:@baybridge:1521:bay817` |
| DriverClassname | `oracle.jdbc.xa.client.OracleXADataSource` |
| Initial Capacity | `1` |
| MaxCapacity | `20` |
| CapacityIncrement | `2` |
| Properties | `user=scott;password=tiger` |

The following table contains sample code for configuring a TxDataSource:

**Table 7-3 Oracle Thin 8.1.7/XA Driver: TxDataSource Configuration**

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

# Using Sybase jConnect 5.2.1/XA Driver

The following sections provide important configuration information and performance issues when using the Sybase jConnect Driver 5.2.1/XA Driver.

## Known Sybase jConnect 5.2.1/XA Issues

These are the known issues and BEA workarounds:

**Table 7-4 Sybase jConnect 5.2.1 Known Issues and Workarounds**

| Description | Sybase Bug | Comments/Workarounds for WebLogic Server 6.1 |
|---|---|---|
| When calling setAutoCommit(true)the following exception is thrown:<br>`java.sql.SQLException: JZ0S3: The inherited method setAuto-Commit(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:<br>Set the connection pool property `XAEndOnlyOnce="true"`.<br>Vendor fix has been requested. |

## Set the Environment for the Sybase jConnect/XA Driver

Follow these instructions to setup your environment:

- set `CLASSPATH=.;%SYBASE_INSTALL_DIR%\jCONNECT-5_2\classes\jconn2.jar`

  where `SYBASE_INSTALL_DIR` is the directory where you installed the Sybase driver.

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

- Run `sp_configure "enable xact coordination",1.`
- Run grant role `dtm_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

## Connection Pools for the Sybase jConnect 5.2.1/XA Driver

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

**Table 7-5 Sybase jConnect 5.2.1/XA Driver: Sample Connection Pool Configuration**

| Property Name | Property Value |
|---|---|
| Name | `jtaXAPool` |
| Targets | `myserver,server1` |
| DriverClassname | `com.sybase.jdbc2.jdbc.SybXADataSource` |
| Properties | `User=`*`dbuser`*`;`<br>`DatabaseName=`*`dbname`*`;`<br>`ServerName=`*`server_name_or_IP_address`*`;`<br>`PortNumber=`*`serverPortNumber`*`;`<br>`NetworkProtocol=Tds;`<br>`resourceManagerName=`*`Lrm_name_in_xa_config`*`;`<br>`resourceManagerType=2` |
| Password | *`dbpassword`* |
| Initial Capacity | `1` |

**Table 7-5 Sybase jConnect 5.2.1/XA Driver: Sample Connection Pool Configuration**

| Property Name | Property Value |
|---|---|
| MaxCapacity | `10` |
| CapacityIncrement | `1` |

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

**Note:** You must also add `KeepXAConnTillTxComplete="true"` to the connection pool tag in the `config.xml` file. See Additional XA Connection Pool Properties in the *Administration Guide* at `http://e-docs.bea.com/wls/docs61/adminguide/jdbc.html#addxaprops`.

The following table contains sample code for configuring a TxDataSource:

**Table 7-6 Sybase jConnect 5.2.1/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 7-7 Sybase jConnect 5.2.1/XA Driver: Java Client Connection Properties**

| Property Name | Property Value |
|---|---|
| `ds.setPassword` | `<password>` |
| `ds.setUser` | `<username>` |

**Table 7-7 Sybase jConnect 5.2.1/XA Driver: Java Client Connection Properties**

| Property Name | Property Value |
|---|---|
| `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` | `4100` |

# Using Cloudscape 3.5.1/XA Driver

The following sections provide information for using the Type 2 Cloudscape 3.5.1/XA Driver with WebLogic Server 6.1.

## Software Requirements for the Cloudscape 3.5.1/XA Driver

The Cloudscape 3.5.1/XA Driver supports JDK 1.3 RC1. For more information, see http://www.cloudscape.com/support/techinfo.jsp.

## Known Cloudscape 3.5.1/XA Driver Issues

The following table contains known issues:

**Table 7-8 Cloudscape 3.5.1/XA Driver Known Issues**

| Description | Cloudscape Enhancement Request | Comments/Workarounds for WebLogic Server 6.1 |
|---|---|---|
| No known issues. | | . |

## Set the Environment for the Cloudscape 3.5.1/XA Driver

Set the following environment variables (assuming NT syntax):

- set `CLOUDSCAPE_INSTALL=<directory where Cloudscape is installed>`
- set `CLASSPATH=.;%cloudscape_install%\lib\cloudscape.jar; %cloudscape_install%\lib\cloudsync.jar; %cloudscape_install%\lib\client.jar;%cloudscape_install%\lib\ tools.jar;c:\weblogic\dev\src\3rdparty\weblogicaux.jar`

**Note:** Note that the `weblogicaux.jar` is for the javax classes only.

## Cloudscape 3.5.1/XA Driver Configuration Properties

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

**Table 7-9 Cloudscape 3.5.1/XA Driver: Connection Pool Configuration**

| Property Name | Property Value |
|---|---|
| Name | `jtaXAPool` |
| Targets | `myserver,server1` |
| DriverClassname | `COM.cloudscape.core.XaDataSource` |
| Initial Capacity | `1` |
| Max Capacity | `10` |
| Capacity Increment | `2` |
| Properties | `databaseName=CloudscapeDB; createDatabase=create` |
| Supports Local Transaction | True |

The following table contains sample code for configuring a TxDataSource:

**Table 7-10 Cloudscape 3.5.1/XA Driver: TxDataSource Configuration**

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

# Using DB2 7.2/XA Driver

The following sections describe how to set your environment to use the Type2 DB2 7.2/XA Driver with WebLogic Server 6.1.

## 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

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.

## DB2 7.2/XA Driver Configuration Properties

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

**Table 7-11 DB2 7.2/XA Driver: Connection Pool Configuration**

| Property Name | Property Value |
|---|---|
| Name | `jtaXAPool` |
| Targets | `server1` |
| DriverClassname | `COM.ibm.db2.jdbc.DB2XADataSource` |
| Initial Capacity | `1` |
| MaxCapacity | `10` |
| CapacityIncrement | `2` |
| Properties | `user=db2admin;`<br>`password=db2admin;`<br>`DatabaseName=NEWDEMO` |

**Note:** You must also add `KeepXAConnTillTxComplete="true"` to the connection pool tag in the `config.xml` file. See Additional XA Connection Pool Properties in the *Administration Guide* at `http://e-docs.bea.com/wls/docs61/adminguide/jdbc.html#addxaprops`.

The following table contains sample code for configuring a TxDataSource:

**Table 7-12 DB2 7.2/XA Driver: TxDataSource Configuration**

| Property Name | Property Value |
|---|---|
| Name | `jtaXADS` |
| Targets | `server1` |

**Table 7-12 DB2 7.2/XA Driver: TxDataSource Configuration**

| Property Name | Property Value |
|---|---|
| JNDIName | `jtaXADS` |
| PoolName | `jtaXAPool` |

# 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`.

*CHAPTER*

# 8 WebLogic Server XA Resource Provider Requirements

BEA WebLogic Server supports the Java Transaction API (JTA) and includes a Transaction Manager that coordinates distributed transactions with any XA-compliant resource. The following sections describe the requirements for XA resources to be able to participate in distributed transactions in WebLogic Server. This information is written for third-party application integrators.

For information on JTA, see the Java Transaction API (JTA) Specification version 1.0.1, published by Sun Microsystems, Inc.

- Overview of XA Resource Provider Requirements
- Registering with the Transaction Manager
- XAResource Enlistment and Delistment
- Optional weblogic.transaction.XAResource Interface

# Overview of XA Resource Provider Requirements

An XA resource provider must support the JTA `XAResource` interface with no thread affinity limitations to be able to participate in distributed transactions in WebLogic Server.

For non-JDBC resources, the resource provider also needs to do the following:

- Register the XA resource with the WebLogic Server transaction manager on startup
- Optionally enlist and delist the XA resource with the WebLogic Server transaction manager before and after resource usage
- Optionally implement the `weblogic.transaction.XAResource` interface

# Registering with the Transaction Manager

The JTA specification does not define how to bootstrap an XA resource into a server's transaction manager. WebLogic Server defines a registration API for this purpose.

XA resource providers must perform the following steps to register their `XAResource` implementation with the local transaction manager on startup:

1. Obtain the JTA transaction manager using JNDI or the `TxHelper` interface. The following code shows how to use the `TxHelper` interface to obtain the current transaction manager:

   ```
   import javax.transaction.xa.XAResource;
   import javax.transaction.TransactionManager;
   import weblogic.transaction.TxHelper;

   TransactionManager tm = TxHelper.getTransactionManager();
   ```

2. Register the XA resource with the JTA transaction manager

```
XAResource res = ... // Resource provider's implementation of XAResource
```

```
tm.registerStaticResource(name, res); // Static enlistment resource
tm.registerDynamicResource(name, res2); // Dynamic enlistment resource
```

The resource provider supplies its name and implementation using either the static or dynamic registration method. See "XAResource Enlistment and Delistment" for information on static and dynamic enlistment.

The resource name determines the transaction branch. If the resource supports different instances, each resource instance should use a different name. This name is also the resource name that is used in administration.

Note that it is important that resource providers register the XA resource with the transaction manager before enlisting the XA resource for any transactional work.

3. Unregister the XA resource with the JTA transaction manager

   ```
   XAResource res = … // Resource provider's implementation of
   XAResource
   ```

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

   The resource provider associated with this name on the current server is unregistered. If there are any transactions outstanding for this resource, unregistering a resource might result in rolled back transactions or transaction branch abandonment.

# XAResource Enlistment and Delistment

In the JTA architecture, the application server plays the role of transactional resource manager, including enlisting and delisting resources implicitly with the transaction manager when necessary.

WebLogic Server supports two modes of XA resources enlistment: static and dynamic.

# Static Enlistment and Delistment

If the XA resources are registered using static enlistment, WebLogic Server plays the role of application server and performs enlistment and delistment implicitly for the XA resource provider.

In particular, WebLogic Server enlists resources on transaction begin and resume. Note that a transaction is implicitly resumed upon the start of method calls for bean-managed transaction beans that have transactions previously associated with the beans, and also when an outgoing call to another server returns.

Similarly, WebLogic Server delists resources on transaction suspend, commit and rollback. Note that a transaction is implicitly suspended when making calls to another server, and also when method calls return to the client. The delist flag used is either obtained from the `getDelistFlag()` method if the resource provider supports it, or is `TMSUSPEND` if the resource provider does not support the `getDelistFlag()` method. Please refer to "Optional weblogic.transaction.XAResource Interface" for more details about the `getDelistFlag()` method.

# Dynamic Enlistment and Delistment

XA resource providers can also perform enlistment and delistment themselves by registering as using dynamic enlistment. In this case, the XA resource provider itself plays the role of JTA application server partially in performing enlistment and delistment. The advantage of dynamic enlistment is that resource provider can optionally perform lazy enlistment to avoid enlisting resources unnecessarily.

To dynamically enlist the XA resource, the XA resource provider does the following:

```
import javax.transaction.Transaction;
// Obtain the current transaction via JNDI or TxHelper
Transaction tx = TxHelper.getTransaction();
tx.enlistResource(res);
```

Resource enlistment and delistment are potentially expensive operations. As an optimization, the WebLogic Server transaction manager ignores duplicate enlistments of the same resource in the same thread.

To dynamically delist the XA resource, the XA resource provider obtains the transaction as above, and calls the delistResource method, supplying the delist flag as well.

```
tx.delistResource(res, flag);
```

Note that for resources that are registered as dynamically enlisted, the enlistment step is essential. However, the delistment step is optional.

WebLogic Server transaction manager performs delayed delistment. That is, the transaction manager will actually call the `end()` method on XA resource on transaction suspend and completion. The transaction manager obtains the delist flag in the following order:

- If the resource provider has previously delisted the XA resource, then use the delist flag that the resource provider previously supplied.
- If the resource provider supports `getDelistFlag()` method, then obtain the delist flag by calling the method. See "Optional weblogic.transaction.XAResource Interface" for details.
- If the resource provider supports neither of the above, conservatively use `TMSUSPEND`.

# Optional weblogic.transaction.XAResource Interface

For the case of static delistment or omitted dynamic delistment, the WebLogic Server JTA transaction manager conservatively delists the XA resources with `TMSUSPEND` across method invocations to preserve potentially opened cursors. However, for some resources, it may hold up more internal resources than delisting with `TMSUCCESS`. XA resource provider can override the default delist flag used by WebLogic Server by supporting the optional `weblogic.transaction.XAResource` interface.

The `weblogic.transaction.XAResource` interface supports the following methods:

```
package weblogic.transaction;
```

```
public interface XAResource extends
javax.transaction.xa.XAResource {
  int getDelistFlag();
}
```

If the resource provider supports it, the WebLogic Server transaction manager calls the `getDelistFlag()` method to obtain the delist flag to be used to delist it at transaction suspend and method end.

# 9 Troubleshooting Transactions

The following sections describe troubleshooting tools and tasks for use in determining why transactions fail and deciding what actions to take to correct the problem.

- Overview of Troubleshooting Transactions
- Troubleshooting Tools
- Debugging Tips
- Handling Heuristic Completions
- Transaction System Recovery

## Overview of Troubleshooting Transactions

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:

- Exceptions
- Transaction identifier
- Transaction naming and properties
- Transaction status
- Transaction statistics
- Transaction monitoring
- Transaction logging

## 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

You can monitor transactions in progress using the WebLogic Console. In addition to displaying statistics, as described in "Transaction Statistics," you can display the following:

- transactions by name, including rollback and time active information
- transactions by resource, including statistics on total, committed, and rolled back transactions
- all active transactions, including information on status, servers, resources, properties, and the transaction identifier

# Transaction Log

Each server has a transaction log which records information about the propagation of a transaction through the system. The transaction log is written to persistent storage and assists the server in recovering from system crashes and network failures. You cannot directly access the transaction log; the file is in a binary format.

The transaction log consists of multiple files. Each file is subject to garbage collection; 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.

Transaction log files are uniquely named using a pathname prefix, the server name, a four-digit numeric suffix, and a file extension. Specify a value for the `TransactionLogFilePrefix` server attribute using the WebLogic Console to set the pathname prefix. The `TransactionLogFilePrefix` attribute should be set so that transaction log files are created on a highly available file system, for example, on a RAID device.

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

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

Similarly, on a Windows NT system, 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 one or more 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, `commit()` throws `SystemException`, and the transaction manager places a message in the system error log. No transactions are committed until more space is available.

When migrating a server to another machine, move the transaction log files as well, keeping all the log files for a server together.

### 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
```

# Debugging Tips

Use the naming properties of transactions to isolate and identify problem transactions.

Debugging transactions may require fault isolation to be performed between WebLogic JTA and the participating resources.

# Handling Heuristic Completions

An **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 transaction 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.

# Transaction System Recovery

The WebLogic Server transaction manager is designed to recover from system crashes with minimal user intervention. A prepared transaction is not left unresolved in the resource manager without either a commit or rollback action from the transaction manager, even after multiple crashes.

You can choose to abandon transactions after a specified amount of time. Using the `AbandonTimeoutSeconds` attribute, you can set the maximum time, in seconds, that a transaction coordinator will persist in attempting to complete a transaction. 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.

The transaction manager has the following responsibilities after a system crash:

- Maintain consistency across resources

  If a transaction is committed before a crash, and `XAResource.recover()` returns the transaction ID, the transaction manager consistently calls `XAResource.commit()`. If a transaction is not committed before a crash, and `XAResource.recover()` returns its transaction ID, the transaction manager consistently calls `XAResource.rollback()`. In other words, a transaction manager crash by itself cannot cause a mixed heuristic completion where some branches are committed and some are rolled back.

- Resolve prepared transactions

  Once the transaction manager has prepared any transaction with a resource manager, it must call `XAResource.recover()` during crash recovery for that resource manager and eventually resolve (by calling the `commit()`, `rollback()`, or `forget()` method) all transaction IDs returned by `recover()`.

- Persist in achieving transaction resolution

  If a resource manager crashes, the transaction manager 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 and can be limited by setting the `AbandonTimeoutSeconds` configuration attribute.

- Report heuristic completions

  If the resource manager reports a heuristic commit or heuristic rollback, this is recorded in the server log by the transaction manager, and `forget()` called 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 in resolving heuristic completions.

# A 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

Transactions that are demarcated and coordinated by an external Transaction Manager via the Two Phase Commit Protocol across multiple resource managers. Also known as global transactions.

### global transactions

See distributed transactions.

### 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

## F



## G



## H



## I



## J



## L



## M



## N



## O



## P



## R

## S



## T



## U

## X