BEA WebLogic Enterprise

Using Transactions

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Using Transactions

Document Edition	Date	Software Version
5.0	December 1999	BEA WebLogic Enterprise 5.0

Contents

About This Document

This document explains how to use transactions in CORBA, EJB, and RMI applications that run on WebLogic Enterprise (WLE).

This document covers the following topics:

- Chapter 1, “Introducing Transactions,” introduces transactions in CORBA, EJB, and RMI applications running in the WLE environment.
- Chapter 2, “Transaction Service,” describes the WLE Transaction Service.
- Chapter 3, “Transactions in CORBA Server Applications,” describes how to implement transactions in CORBA server applications.
- Chapter 4, “Transactions in CORBA Client Applications,” describes how to implement transactions in CORBA client applications.
- Chapter 5, “Transactions Sample CORBA Java JDBC Application,” describes a Java CORBA sample application that implements transactions.
- Chapter 6, “Transactions Sample CORBA C++ XA Application,” describes a C++ CORBA sample application that implements transactions.
- Chapter 7, “Transactions Sample CORBA Java/C++ XA Application,” describes a Java CORBA application that implements distributed transactions.
- Chapter 8, “Transactions in EJB Applications,” describes how to implement transactions in EJB applications.
- Chapter 9, “Transactions in RMI Applications,” describes how to implement transactions in RMI applications.
- Chapter 10, “Transactions and the WLE JDBC/XA Driver,” describes how to use the WLE JDBC/XA driver in conjunctions with distributed transactions in WLE Java applications.

- Chapter 11, “Administering Transactions,” describes how to administer transactions in the WLE environment.

What You Need to Know

This document is intended primarily for application developers who are interested in building transactional C++ and Java applications that run in the WLE environment. It assumes a familiarity with the WLE platform, C++ or Java programming, and transaction processing concepts.

e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the “e-docs” Product Documentation page at *http://e-docs.beasys.com*.

How to Print the Document

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

A PDF version of this document is available on the WebLogic Enterprise 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 Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at *http://www.adobe.com/*.

Related Information

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA TUXEDO, distributed object computing, transaction processing, C++ programming, and Java programming, see the *WLE Bibliography* in the WebLogic Enterprise online documentation.

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.0 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSupport at *www.beasys.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	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*

Convention	**Item**
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Introducing Transactions

This topic includes the following sections:

- Overview of Transactions in WLE Applications
- When to Use Transactions
- What Happens During a Transaction
- Transactions Sample Code

Overview of Transactions in WLE Applications

This topic includes the following sections:

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

ACID Properties of Transactions

One of the most fundamental features of the WebLogic Enterprise (WLE) 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 (atomicity, consistency, isolation, and durability) of a high-performance transaction. WLE 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 Models

WLE supports transactions in two different programming models:

- The Object Management Group’s Common Object Request Broker (CORBA) in C++ and Java applications, in compliance with the *The Common Object Request Broker: Architecture and Specification*. Revision 2.2, February 1998.

- Sun Microsystem’s Java 2 Platform, Enterprise Edition (J2EE). WLE provides full support for transactions in Java applications that use Enterprise JavaBeans, in compliance with Sun Microsystem’s *Enterprise JavaBeans 1.1 Specification* (Public Release 2 dated October 18, 1999). WLE also supports Sun Microsystem’s *Java Transaction API (JTA)* specification version 1.0.1.

Supported API Models

WLE supports two transaction API models:

- CORBAservices Object Transaction Service (OTS) and the Java Transaction Service (JTS).

  WLE provides a C++ interface to the OTS and a Java interface to the OTS and the JTS. The JTS is the Sun Microsystems, Inc. Java interface for transaction services, and is based on the OTS. The OTS and the JTS are accessed through the `org.omg.CosTransactions.Current` environmental object. For information about using the `TransactionCurrent` environmental object, see the *C++ Bootstrap Object Programming Reference* or the *Java Bootstrap Object Programming Reference*.

- The Sun Microsystems, Inc. Java Transaction API (JTA), which is used by:

  - CORBA applications within BEA’s TP Framework.
  - Enterprise JavaBean (EJB) applications within the WLE EJB Container.
  - Remote Method Invocation (RMI) applications within the WLE infrastructure.

  Only the application-level demarcation interface (`javax.transaction.UserTransaction`) is supported. For information about JTA, see the following sources:

  - The `javax.transaction` package description in the *WLE Javadoc*.
  - The Java Transaction API specification, published by Sun Microsystems, Inc. and available from the Sun Microsystems, Inc. Web site (*www.sun.com*).

Support for Business Transactions

OTS, JTS, and JTA each provide the following support for your business transactions:

- Creates a global transaction identifier when a client application initiates a transaction.
- Works with the WLE 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. Almost all 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.

Distributed Transactions and the Two-Phase Commit Protocol

WLE 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. The *two-phase commit protocol (2PC)* is a method of coordinating a single transaction across one 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.

When to Use Transactions

Transactions are appropriate in the situations described in the following list. Each situation describes a transaction model supported by the WLE system.

- 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.

- 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.

  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 CORBA, EJB, or RMI invocations.)

  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.

What Happens During a Transaction

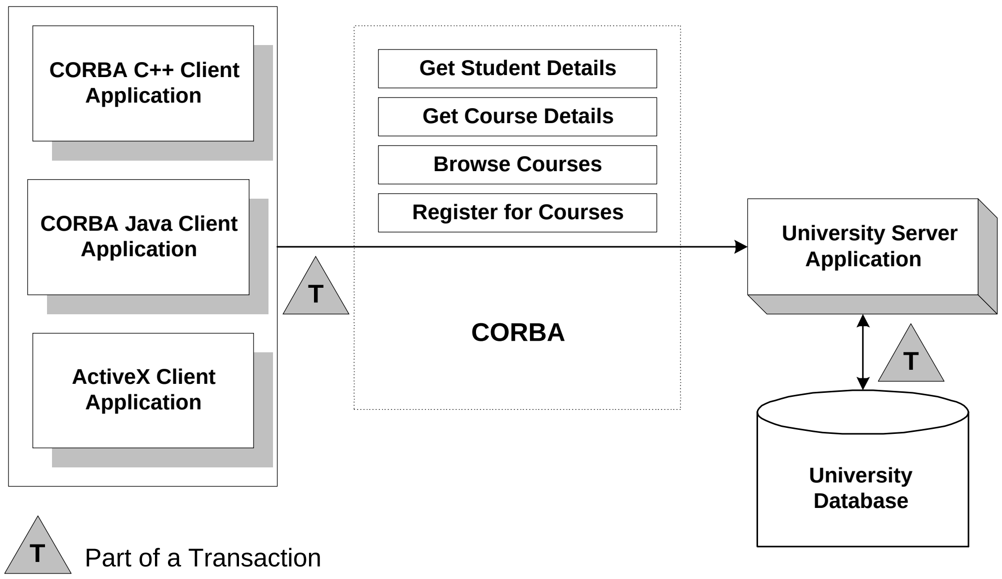
This topic includes the following sections:

- Transactions in WLE CORBA Applications
- Transactions in WLE EJB Applications
- Transactions in WLE RMI Applications

Transactions in WLE CORBA Applications

Figure 1-1 illustrates how transactions work in a WLE CORBA application.

Figure 1-1 How Transactions Work in a WLE CORBA Application



For CORBA applications, a basic transaction works in the following way:

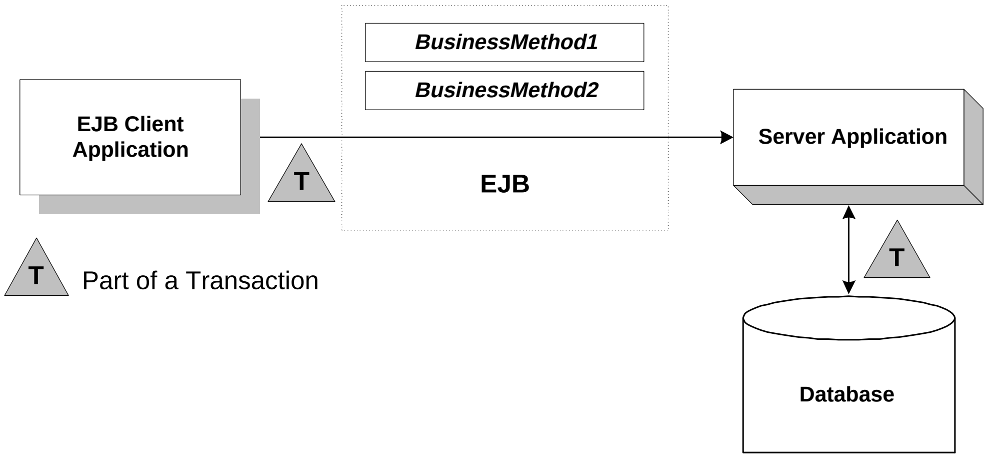
1. The client application uses the `Bootstrap` object to return an object reference to the `TransactionCurrent` object for the WLE domain.
2. A client application begins a transaction using the `Tobj::TransactionCurrent::begin()` operation, and issues a request to the CORBA interface through the TP Framework. All operations on the CORBA interface 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.
   - If no exceptions occur, the client application commits the current transaction using the `Tobj::TransactionCurrent::commit()` operation. This operation 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 `Tobj::TransactionCurrent:commit()` operation causes the TP Framework to call the Transaction Manager to complete the transaction.

4. The Transaction Manager is responsible for coordinating with the Resource Managers to update the database.

Transactions in WLE EJB Applications

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

Figure 1-2 How Transactions Work in a WLE EJB Application



WLE supports two types of transactions in WLE EJB applications:

- In *container-managed transactions*, the WLE EJB Container manages the transaction demarcation. Transaction attributes in the EJB deployment descriptor determine how the WLE EJB Container handles transactions with each method invocation. For more information about the deployment descriptor, see the *WebLogic EJB Extensions Reference*.
- 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 “UserTransaction API” on page 2-24.

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 11.6.2 in Sun Microsystem's *Enterprise JavaBeans Specification 1.1* (Public Release 2 dated October 18, 1999).

3. Optionally, in the EJB's deployment descriptor, the Bean Provider or Application Assembler specifies the `trans-attribute` for one or more methods.

4. When a client application invokes a method in the EJB, the EJB Container checks the `trans-attribute` setting in the deployment descriptor for that method. If no setting is specified for the method, the EJB uses the default `trans-attribute` setting for that EJB.

5. The EJB Container takes the appropriate action depending on the applicable `trans-attribute` setting.

   - For example, if the `trans-attribute` setting is `Required`, the EJB Container invokes the method within the existing transaction context or, if the client called without a transaction context, the EJB Container begins a new transaction before executing the method.

   - In another example, if the `trans-attribute` setting is `Mandatory`, the EJB Container invokes the method within the existing transaction context. If the client called without a transaction context, the EJB Container throws the `javax.transaction.TransactionRequiredException` exception.

6. During invocation of the business method, if it is determined that a rollback is required, the business method calls the `EJBContext.setRollbackOnly` method, which notifies the EJB Container that the transaction is to be rolled back at the end of the method invocation.

Note: Calling the `EJBContext.SetRollbackOnly` method is allowed only for methods that have a meaningful transaction context.

7. At the end of the method execution and before the result is sent to the client, the EJB Container completes the transaction, either by committing the transaction or rolling it back (if the `EJBContext.SetRollbackOnly` method was called).

Bean-Managed Transactions

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

1. In the EJB's deployment descriptor, the Bean Provider or Application Assembler specifies the transaction type (`transaction-type` element) for container-managed demarcation (`Bean`).

2. The client application uses JNDI to obtain an object reference to the `UserTransaction` object for the WLE domain.

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

   - If a call to any of these operations raises an exception (either explicitly or as a result of a communication failure), the exception can be caught and the transaction can be rolled back using the `UserTransaction.rollback` method.

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

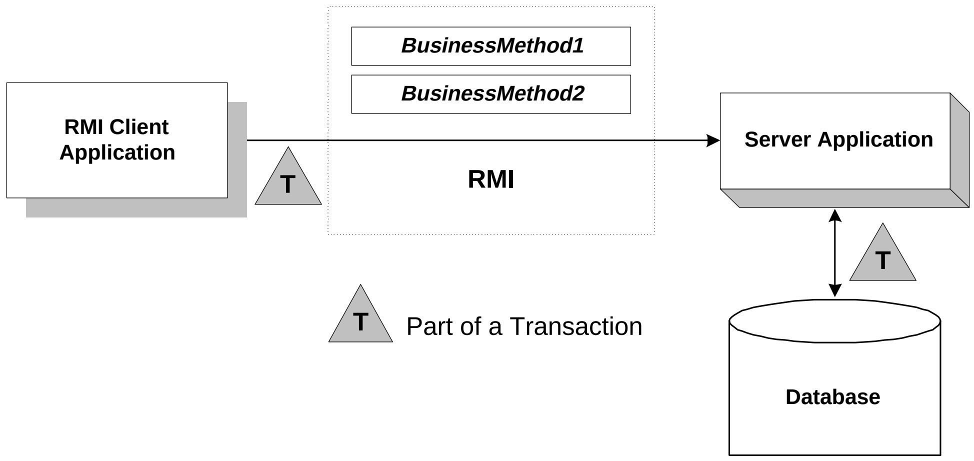
4. The `UserTransaction.commit` method causes the EJB Container to call the Transaction Manager to complete the transaction.

5. The Transaction Manager is responsible for coordinating with the Resource Managers to update any databases.

Transactions in WLE RMI Applications

Figure 1-3 illustrates how transactions work in a WLE RMI application.

Figure 1-3 How Transactions Work in a WLE 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 WLE 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 WLE 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 WLE 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.

Transactions Sample Code

This topic includes the following sections:

- Transactions Sample CORBA Application
- Transactions Sample EJB Code
- Transactions Sample RMI Code

Transactions Sample CORBA Application

In the Transactions sample CORBA application, the operation of registering for courses is executed within the scope of a transaction. The transaction model used in the Transactions sample application is a combination of the conversational model and the model in which a single client invocation makes multiple individual operations on a database.

Workflow for the Transactions Sample Application

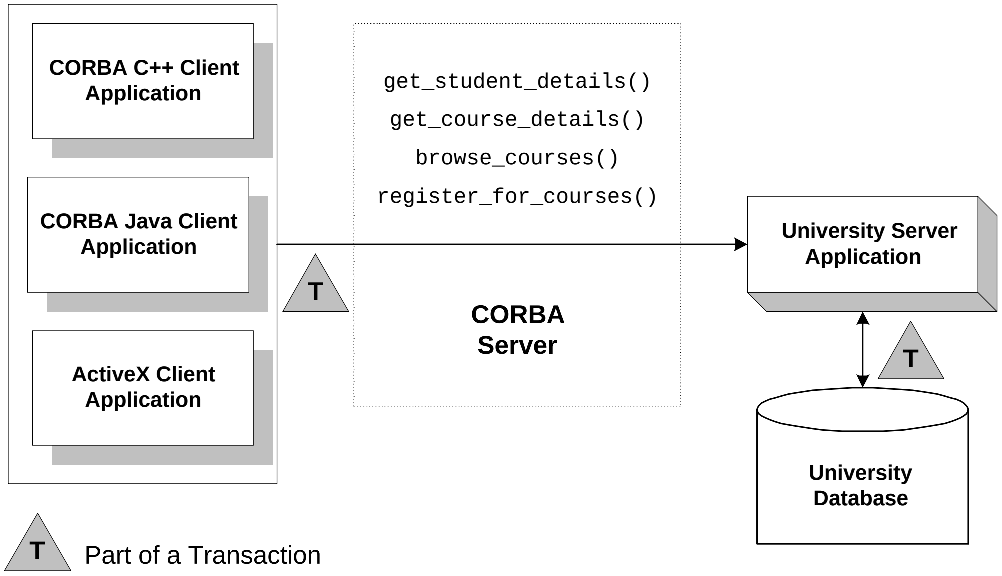
The Transactions sample application works in the following way:

1. Students submit a list of courses for which they want to be registered.
2. For each course in the list, the server application checks whether:
   - The course is in the database.
   - The student is already registered for a course.
   - The student exceeds the maximum number of credits the student can take.

3. One of the following occurs:
   - If the course meets all the criteria, the server application registers the student for the course.
   - If the course is not in the database or if the student is already registered for the course, the server application adds the course to a list of courses for which the student could not be registered. After processing all the registration requests, the server application returns the list of courses for which registration failed. The client application can then choose to either commit the transaction (thereby registering the student for the courses for which registration request succeeded) or to roll back the transaction (thus, not registering the student for any of the courses).
   - If the student exceeds the maximum number of credits the student can take, the server application returns a `TooManyCredits` user exception to the client application. The client application provides a brief message explaining that the request was rejected. The client application then rolls back the transaction.

Figure 1-4 illustrates how the Transactions sample application works.

Figure 1-4 Transactions Sample Application



The Transactions sample application shows two ways in which a transaction can be rolled back:

- **Nonfatal**. If the registration for a course fails because the course is not in the database, or because the student is already registered for the course, the server application returns the numbers of those courses to the client application. The decision to roll back the transaction lies with the user of the client application (and the Transaction client application code rolls back the transaction automatically in this case).
- **Fatal**. If the registration for a course fails because the student exceeds the maximum number of credits he or she can take, the server application generates a CORBA exception and returns it to the client. The decision to roll back the transaction also lies with the client application.

  Thus, the Transactions sample application also shows how to implement user-defined CORBA exceptions. For example, if the student tries to register for a course that would exceed the maximum number of courses for which the student can register, the server application returns the `TooManyCredits` exception. When the client application receives this exception, the client application rolls back the transaction automatically.

Note: For information about how transactions are implemented in Java WLE applications, see Chapter 7, "Transactions Sample CORBA Java/C++ XA Application."

Development Steps

This topic describes the following development steps for writing a WLE application that contains transaction processing code:

- Step 1: Writing the OMG IDL
- Step 2: Defining Transaction Policies for the Interfaces
- Step 3: Writing the Server Application
- Step 4: Writing the Client Application
- Step 5: Creating a Configuration File

The Transactions sample application is used to demonstrate these development steps. The source files for the Transactions sample application are located in the `\samples\corba\university` directory of the WLE software. For information about building and running the Transactions sample application, see Chapter 6, "Transactions Sample CORBA C++ XA Application."

The XA Bankapp sample application demonstrates how to use transactions in Java WLE applications. The source files for the XA Bankapp sample application are located in the `\samples\corba\bankapp_java\XA` directory of the WLE software. For information about building and running the XA Bankapp sample application, see Chapter 7, "Transactions Sample CORBA Java/C++ XA Application."

Step 1: Writing the OMG IDL

You need to specify interfaces involved in transactions in Object Management Group (OMG) Interface Definition Language (IDL) just as you would any other CORBA interface. You must also specify any user exceptions that might occur from using the interface.

For the Transactions sample application, you would define in OMG IDL the `Registrar` interface and the `register_for_courses()` operation. The `register_for_courses()` operation has a parameter, `NotRegisteredList`, which returns to the client application the list of courses for which registration failed. If the value of `NotRegisteredList` is empty, then the client application commits the transaction. You also need to define the `TooManyCredits` user exception.

Listing 1-1 includes the OMG IDL for the Transactions sample application.

Listing 1-1 OMG IDL for the Transactions Sample Application

```
#pragma prefix "beasys.com"
module UniversityT

{
       typedef unsigned long CourseNumber;
       typedef sequence<CourseNumber> CourseNumberList;

       struct CourseSynopsis
       {
              CourseNumber   course_number;
              string         title;
       };
       typedef sequence<CourseSynopsis> CourseSynopsisList;
```

```
        interface CourseSynopsisEnumerator
        {
        //Returns a list of length 0 if there are no more entries
        CourseSynopsisList get_next_n(
               in  unsigned long number_to_get, // 0 = return all
               out unsigned long number_remaining
        );

        void destroy();
};
        typedef unsigned short Days;
        const Days MONDAY    =  1;
        const Days TUESDAY   =  2;
        const Days WEDNESDAY =  4;
        const Days THURSDAY  =  8;
        const Days FRIDAY    = 16;

//Classes restricted to same time block on all scheduled days,
//starting on the hour

struct ClassSchedule
{
        Days           class_days; // bitmask of days
        unsigned short start_hour; // whole hours in military time
        unsigned short duration;   // minutes
};

struct CourseDetails
{
        CourseNumber   course_number;
        double         cost;
        unsigned short number_of_credits;
        ClassSchedule  class_schedule;
        unsigned short number_of_seats;
        string         title;
        string         professor;
        string         description;
};
        typedef sequence<CourseDetails> CourseDetailsList;
        typedef unsigned long StudentId;

struct StudentDetails
{
        StudentId         student_id;
        string            name;
        CourseDetailsList registered_courses;
};
```

```
enum NotRegisteredReason
{
       AlreadyRegistered,
       NoSuchCourse
};

struct NotRegistered
{
       CourseNumber        course_number;
       NotRegisteredReason not_registered_reason;
};
       typedef sequence<NotRegistered> NotRegisteredList;

exception TooManyCredits
{
       unsigned short maximum_credits;
};

//The Registrar interface is the main interface that allows
//students to access the database.
interface Registrar
{
       CourseSynopsisList
       get_courses_synopsis(
              in string                     search_criteria,
              in unsigned long              number_to_get,
              out unsigned long             number_remaining,
              out CourseSynopsisEnumerator rest
);

       CourseDetailsList get_courses_details(in CourseNumberList
        courses);
       StudentDetails get_student_details(in StudentId student);
       NotRegisteredList register_for_courses(
              in StudentId        student,
              in CourseNumberList courses
       ) raises (
              TooManyCredits
       );

};

// The RegistrarFactory interface finds Registrar interfaces.

interface RegistrarFactory
{
       Registrar find_registrar(
       );
};
```

Step 2: Defining Transaction Policies for the Interfaces

Transaction policies are used on a per-interface basis. During design, it is decided which interfaces within a WLE application will handle transactions. Table 1-1 describes the CORBA transaction policies:

Table 1-1 CORBA Transaction Policies

Transaction Policy	Description
`always`	The interface must always be part of a transaction. If the interface is not part of a transaction, a transaction will be automatically started by the TP Framework.
`ignore`	The interface is not transactional. However, requests made to this interface within a scope of a transaction are allowed. The `AUTOTRAN` parameter, specified in the `UBBCONFIG` file for this interface, is ignored.
`never`	The interface is not transactional. Objects created for this interface can never be involved in a transaction. The WLE system generates an exception (`INVALID_TRANSACTION`) if an interface with this policy is involved in a transaction.
`optional`	The interface may be transactional. Objects can be involved in a transaction if the request is transactional. This transaction policy is the default.

During development, you decide which interfaces will execute in a transaction by assigning transaction policies in the following ways:

- For C++ server applications in CORBA, you specify transaction policies in the Implementation Configuration File (ICF). A template ICF file is created by the `genicf` command. For more information about the ICFs, see “Implementation Configuration File (ICF)” in the *CORBA C++ Programming Reference.*
- For Java server applications in CORBA, you specify transaction policies in the Server Description File, written in Extensible Markup Language (XML). For more information about Server Description files, see “Server Description File” in the *CORBA Java Programming Reference*.

In the Transactions sample application, the transaction policy of the `Registrar` interface is set to `always`.

Step 3: Writing the Server Application

When using transactions in server applications, you need to write methods that implement the interface's operations. In the Transactions sample application, you would write a method implementation for the `register_for_courses()` operation.

If your WLE application uses a database, you need to include in the server application code that opens and closes an XA Resource Manager. These operations are included in the `Server::initialize()` and `Server::release()` operations of the `Server` object. Listing 1-2 shows the portion of the code for the `Server` object in the Transactions sample application that opens and closes the XA Resource Manager.

Note: For a complete example of a C++ server application that implements transactions, see Chapter 6, "Transactions Sample CORBA C++ XA Application." For an example of a Java server application that implements transactions, see Chapter 7, "Transactions Sample CORBA Java/C++ XA Application."

Listing 1-2 C++ Server Object in Transactions Sample Application

```
CORBA::Boolean Server::initialize(int argc, char* argv[])
{
       TRACE_METHOD("Server::initialize");
       try {
              open_database();
              begin_transactional();
              register_fact();
              return CORBA_TRUE;
}
       catch (CORBA::Exception& e) {
              LOG("CORBA exception : " <<e);
       }
       catch (SamplesDBException& e) {
              LOG("Can't connect to database");
       }
       catch (...) {
              LOG("Unexpected database error : " <<e);
       }
       catch (...) {
              LOG("Unexpected exception");
       }
       cleanup();
       return CORBA_FALSE;
}
```

```
void Server::release()
{
       TRACE_METHOD(“Server::release”);
       cleanup();
}

static void cleanup()
{
       unregister_factory();
       end_transactional();
       close_database();
}
//Utilities to manage transaction resource manager

CORBA::Boolean s_became_transactional = CORBA_FALSE;
static void begin_transactional()
{
       TP::open_xa_rm();
       s_became_transactional = CORBA_TRUE;
}
static void end_transactional()
{
       if(!s_became_transactional){
       return//cleanup not necessary
}
try {
       TP::close_xa_rm ();
}
       catch (CORBA::Exception& e) {
              LOG(“CORBA Exception : “ << e);
       }
       catch (...) {
              LOG(“unexpected exception”);
       }

       s_became_transactional = CORBA_FALSE;
}
```

Step 4: Writing the Client Application

The client application needs code that performs the following tasks:

1. Obtains a reference to the `TransactionCurrent` object from the `Bootstrap` object.

2. Begins a transaction by invoking the `Tobj::TransactionCurrent::begin()` operation on the `TransactionCurrent` object.

3. Invokes operations on the object. In the Transactions sample application, the client application invokes the `register_for_courses()` operation on the `Registrar` object, passing a list of courses.

Listing 1-3 illustrates the portion of the CORBA C++ client applications in the Transactions sample application that illustrates the development steps for transactions.

For an example of a CORBA Java client application that uses transactions, see Chapter 7, “Transactions Sample CORBA Java/C++ XA Application.” For an example of using transactions in an ActiveX client application, see Chapter 4, “Transactions in CORBA Client Applications.”

Listing 1-3   Transactions Code for CORBA C++ Client Applications

```
CORBA::Object_var var_transaction_current_oref =
     Bootstrap.resolve_initial_references(“TransactionCurrent”);
CosTransactions::Current_var transaction_current_oref=
     CosTransactions::Current::_narrow(var_transaction_current_oref.in());
//Begin the transaction
var_transaction_current_oref->begin();
try {
//Perform the operation inside the transaction
     pointer_Registar_ref->register_for_courses(student_id, course_number_list);
     ...
//If operation executes with no errors, commit the transaction:
     CORBA::Boolean report_heuristics = CORBA_TRUE;
     var_transaction_current_ref->commit(report_heuristics);
    }
catch (...) {
//If the operation has problems executing, rollback the
//transaction. Then throw the original exception again.
//If the rollback fails,ignore the exception and throw the
//original exception again.
try {
     var_transaction_current_ref->rollback();
    }
catch (...) {
            TP::userlog("rollback failed");
            }
throw;
}
```

Step 5: Creating a Configuration File

You need to add the following information to the configuration file for a transactional WLE application:

- In the `GROUPS` section:
  - In the `OPENINFO` parameter, include the information needed to open the Resource Manager for the database. You obtain this information from the product documentation for your database. Note that the default version of the `com.beasys.Tobj.Server.initialize` method automatically opens the Resource Manager.
  - In the `CLOSEINFO` parameter, include the information needed to close the Resource Manager for the database. By default, the `CLOSEINFO` parameter is empty.
  - Specify the `TMSNAME` and `TMSCOUNT` parameters to associate the XA Resource Manager with a specified server group.
- In the `SERVERS` section, define a server group that includes both the server application that includes the interface and the server application that manages the database. This server group needs to be specified as transactional.
- Include the pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter. For more information about the transaction log, see Chapter 11, "Administering Transactions."

Listing 1-4 includes the portions of the configuration file that define this information for the Transactions sample application.

Listing 1-4 Configuration File for Transactions Sample Application

```
*RESOURCES
       IPCKEY    55432
       DOMAINID  university
       MASTER    SITE1
       MODEL     SHM
       LDBAL     N
       SECURITY  APP_PW

*MACHINES
       BLOTTO
       LMID = SITE1
       APPDIR = C:\TRANSACTION_SAMPLE
```

```
        TUXCONFIG=C:\TRANSACTION_SAMPLE\tuxconfig
        TLOGDEVICE=C:\APP_DIR\TLOG
        TLOGNAME=TLOG
        TUXDIR="C:\WLEdir"
        MAXWSCLIENTS=10

*GROUPS
        SYS_GRP
          LMID      = SITE1
          GRPNO     = 1
        ORA_GRP
          LMID      = SITE1
          GRPNO     = 2

        OPENINFO  = "ORACLE_XA:Oracle_XA+SqlNet=ORCL+Acc=P
        /scott/tiger+SesTm=100+LogDir=.+MaxCur=5"
        CLOSEINFO = ""
        TMSNAME   = "TMS_ORA"
        TMSCOUNT  = 2

*SERVERS
        DEFAULT:
        RESTART = Y
        MAXGEN  = 5

        TMSYSEVT
          SRVGRP  = SYS_GRP
          SRVID   = 1

        TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 2
          CLOPT   = "-A -- -N -M"

        TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 3
          CLOPT   = "-A -- -N"

        TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 4
          CLOPT   = "-A -- -F"

        TMIFRSVR
          SRVGRP  = SYS_GRP
          SRVID   = 5

        UNIVT_SERVER
          SRVGRP  = ORA_GRP
          SRVID   = 1
          RESTART = N

        ISL
          SRVGRP  = SYS_GRP
          SRVID   = 6
          CLOPT   = -A -- -n //MACHINENAME:2500

 *SERVICES
```

For information about the transaction log and defining parameters in the Configuration file, see Chapter 11, “Administering Transactions.”

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
- Initializing the UserTransaction Object
- 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 *bean-managed* transaction demarcation. The deployment descriptor for this bean specifies the transaction type (`transaction-type` element) for transaction demarcation (`Bean`).

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

Importing Packages

Listing 1-5 shows importing the necessary packages for transactions, including:

- `javax.transaction.UserTransaction`. For a list of methods associated with this object, see “UserTransaction Methods” on page 2-24.

- System exceptions. For a list of exceptions, see “Exceptions Thrown by UserTransaction Methods” on page 2-26.

Listing 1-5 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
```

Initializing the UserTransaction Object

Listing 1-6 shows initializing an instance of the `UserTransaction` object to null.

Listing 1-6 Initializing the `UserTransaction` Object

```
    UserTransaction tx = null;
```

Using JNDI to Return an Object Reference to the UserTransaction Object

Listing 1-7 shows searching the JNDI tree to return an object reference to the `UserTransaction` object for the appropriate WLE domain.

Listing 1-7 Performing a JDNI Lookup

```
try {
   Context ctx = getInitialContext();
   tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
```

Starting a Transaction

Listing 1-8 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 1-8 Starting a Transaction

```
tx.begin();
```

Completing a Transaction

Listing 1-9 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 the WLE 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 1-9 Completing a Transaction

```
if(gotException){
   try{
      tx.rollback();
      }catch(Exception e){}
   }
   elseif{
      tx.commit();
      }
```

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
- Initializing the UserTransaction Object
- Using JDNI 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.

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

Importing Packages

Listing 1-10 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 “UserTransaction Methods” on page 2-24.
- System exceptions. For a list of exceptions, see “Exceptions Thrown by UserTransaction Methods” on page 2-26.

Listing 1-10 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
```

Initializing the UserTransaction Object

Listing 1-11 shows initializing an instance of the `UserTransaction` object to null.

Listing 1-11 Initializing the `UserTransaction` Object

```
    UserTransaction tx = null;
```

Using JDNI to Return an Object Reference to the UserTransaction Object

Listing 1-12 shows searching the JNDI tree to return an object reference to the `UserTransaction` object for the appropriate WLE 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 WLE infrastructure does not perform any deactivation or activation.

Listing 1-12 Performing a JDNI Lookup

```
try {
   Context ctx = getInitialContext();
   tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
```

Starting a Transaction

Listing 1-13 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 1-13 Starting a Transaction

```
tx.begin();
```

Completing a Transaction

Listing 1-14 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 WLE 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 1-14 Completing a Transaction

```
if(gotException){
   try{
      tx.rollback();
      }catch(Exception e){}
   }
   elseif{
      tx.commit();
      }
```

CHAPTER

2 Transaction Service

This topic includes the following sections:

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

This topic provides the information that programmers need to write transactional applications for the WebLogic Enterprise (WLE) system. Before you begin, you should read Chapter 1, “Introducing Transactions.”

About the Transaction Service

WLE provides a Transaction Service that supports transactions in CORBA, EJB, and RMI applications. The Transaction Service provides:

- An implementation of the CORBAservices Object Transaction Service (OTS) that is described in Chapter 10 of the *CORBAservices: Common Object Services Specification*. This specification defines the interfaces for an object service that provides transactional functions.
- In the WLE EJB Container, an implementation of the transaction services described in Sun Microsystem's *Enterprise JavaBeans Specification 1.1* (Public Release 2 dated October 18, 1999).

For CORBA Java, EJB, and RMI applications, WLE also provides Sun Microsystems, Inc.'s `javax.transaction` package, which implements the Java Transaction API (JTA) for Java applications. For more information about the JTA, see Sun Microsystem's *Java Transaction API (JTA) Specification* (version1.0.1). For more information about the UserTransaction object that applications use to demarcate transaction boundaries, see "UserTransaction API" on page 2-24.

Capabilities and Limitations

This topic includes the following sections:

- Lightweight Clients with Delegated Commit
- Transaction Propagation (CORBA Only)
- Transaction Integrity
- Transaction Termination
- Flat Transactions
- Interoperability Between Remote Clients and the WLE Domain
- Intradomain and Interdomain Interoperability

- Network Interoperability
- Relationship of the Transaction Service to Transaction Processing
- Process Failure
- Multithreaded Transaction Client Support
- General Constraints

These sections describe the capabilities and limitations of the Transaction Service that supports CORBA and EJB 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. WLE 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 `TransactionCurrent` implementation that CORBA clients use, or the remote implementation of `UserTransaction` that EJB or RMI clients use, delegate the actual responsibility of transaction coordination to transaction manager on the server.

Transaction Propagation (CORBA Only)

For CORBA applications, the CORBAservices Object Transaction Service specification states that a client can choose to propagate a transaction context either implicitly or explicitly. WLE *provides* implicit propagation. Explicit propagation *is strongly discouraged*.

Note: For EJB and RMI applications, only implicit propagation is supported for clients.

Objects that are related to transaction contexts that are passed around using explicit transaction propagation *should not* be mixed with implicit transaction propagation APIs. It should be noted, however, that explicit propagation does not place any constraints on when transactional methods can be processed. There is no guarantee that all transactional methods will be completed before the transaction is committed.

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. If implicit transaction propagation is used, 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. For CORBA applications, for example, the Transaction Service performs `reply` checks, `commit` checks, and `resume` checks, as described in the *CORBAservices Object Transaction Service Specification.*

Unchecked transaction behavior relies completely on the application to provide transaction integrity. If explicit propagation is used, the Transaction Service *does not* provide checked transaction behavior and transaction integrity *is not* guaranteed.

Transaction Termination

WLE 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

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

Interoperability Between Remote Clients and the WLE Domain

WLE supports remote clients invoking methods on server objects in *different* WLE domains in the *same* transaction.

Remote clients with multiple connections to the same WLE domain *may* make invocations to server objects on these separate connections within the same transaction.

Intradomain and Interdomain Interoperability

For C++ (but not Java) applications, WLE supports native clients invoking methods on server objects in the WLE domain. In addition, WLE supports server objects invoking methods on other objects in the same or in different processes within the same WLE domain.

Network Interoperability

A client application can have only one active bootstrap object and `TransactionCurrent` object within a single domain. WLE does *not* support exporting or importing transactions to or from remote WLE domains.

However, transactions can encompass multiple domains in a serial fashion. For example, a server with a transaction active in Domain A can communicate with a server in Domain B within the context of that same transaction.

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 BEA TUXEDO ATMI servers.

  Servers using the WLE Transaction Service can make invocations on other BEA TUXEDO Application-to-Transaction Monitor Interface (ATMI) server processes in the same domain. WLE *does not* support the following:

  - Remote clients or native clients invoking ATMI services in the WLE domain.
  - ATMI services invoking objects.

- 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. WLE 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). WLE *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. WLE *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. WLE *does not* support interactions with ODMG transactions.

Process Failure

The Transaction Service monitors the participants in a transaction for failures and inactivity. The BEA TUXEDO system provides management tools for keeping the application running when failures occur. Because WLE is built upon the existing BEA TUXEDO transaction management system, it inherits the TUXEDO capabilities for keeping applications running.

Multithreaded Transaction Client Support

WLE supports multithreaded clients for non-transactional clients. For transactional clients, WLE supports only single-threaded client implementation. Clients cannot make transaction requests concurrently in multiple threads.

General Constraints

The following constraints apply to the Transaction Service:

- In WLE, 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.
- For CORBA applications, a server application object using transactions from the WLE Transaction Service library *requires* the TP Framework functionality. For more information about the TP Framework, see "TP Framework" in the *CORBA C++ Programming Reference*.
- For CORBA applications, a return from the `rollback` method on the `Current` object is asynchronous. Similarly, for EJB and RMI applications, a return from the `rollback` method on the `UserTransaction` object is asynchronous.

  As a result, the objects that were infected by (or participating in) the rolled back transaction get their states cleared by WLE *a little later*. Therefore, *no* other client can infect these objects with a different transaction until WLE clears the states of these objects. This condition exists for a very short amount of time and is typically not noticeable in a production application. A simple workaround for this race condition is to try the appropriate operation after a short (typically a 1-second) delay.
- In WLE, clients using third-party implementations of the CORBAservices Object Transaction Service (for CORBA applications) or the Java Transaction API (for Java applications) *are not* supported.
- In WLE CORBA applications, clients may not make one-way method invocations within the context of a transaction to server objects having the `NEVER`, `OPTIONAL`, or `ALWAYS` transaction policies.

No error or exception will be returned to the client because it is a one-way method invocation. However, the method on the server object will not be executed, and an appropriate error message will be written to the log. Clients may make one-way method invocations within the context of a transaction to server objects with the `IGNORE` transaction policy. In this case, the method on the server object will be executed, but not in the context of a transaction. For more information about the transaction policies, see "Server Description File" in the *CORBA Java Programming Reference* or "Implementation Configuration File (ICF)" in the *CORBA C++ Programming Reference*.

Transaction Service in CORBA Applications

This topic includes the following sections:

- Getting Initial References to the TransactionCurrent Object
- CORBA Transaction Service API
- CORBA Transaction Service API Extensions
- Notes on Using Transactions in WLE CORBA Applications

These sections describe how WLE implements the OTS, with particular emphasis on the portion of the CORBAservices Object Transaction Service that is described as implementation-specific. They describe the OTS application programming interface (API) that you use to begin or terminate transactions, suspend or resume transactions, and get information about transactions.

Getting Initial References to the TransactionCurrent Object

To access the Transaction Service API and the extension to the Transaction Service API as described later in this chapter, an application needs to complete the following operations:

1. Create a `Bootstrap` object. For more information about creating a `Bootstrap` object, see “C++ Bootstrap Object Programming Reference” in the *CORBA C++ Programming Reference*.

2. Invoke the `resolve_initial_reference("TransactionCurrent")` method on the `Bootstrap` object. The invocation returns a standard CORBA object pointer. For a description of this `Bootstrap` object method, see the *CORBA C++ Programming Reference*.

3. If an application requires only the Transaction Service APIs, it should issue an `org.omg.CosTransactions.Current.narrow()` (in Java) or `CosTransactions::Current::_narrow()` (in C++) on the object pointer returned from step 2 above.

   If an application requires the Transaction Service APIs with the extensions, it should issue a `com.beasys.Tobj.TransactionCurrent.narrow()` (in Java) or `Tobj::TransactionCurrent::_narrow()` (in C++) on the object pointer returned from step 2 above.

CORBA Transaction Service API

This topic includes the following sections:

- Data Types
- Exceptions
- Current Interface
- Control Interface
- TransactionalObject Interface

These sections describe the CORBA-based components of the `CosTransactions` modules that WLE implements to support the Transaction Service. For more information about these components, see Chapter 10 of the *CORBAservices: Common Object Services Specification*.

Data Types

Listing 2-1 shows the supported data types.

Listing 2-1 Data Types Supported by the Transaction Service

```
enum Status {
       StatusActive,
       StatusMarkedRollback,
       StatusPrepared,
       StatusCommitted,
       StatusRolledBack,
       StatusUnknown,
       StatusNoTransaction,
       StatusPreparing,
       StatusCommitting,
       StatusRollingBack
};

// This information comes from CORBAservices: Common Object
// Services Specification, p. 10-15. Revised Edition:
// March 31, 1995. Updated: March 1997. Used with permission by OMG.
```

Exceptions

Listing 2-2 shows the supported exceptions in IDL code.

Listing 2-2 Exceptions Supported by the Transaction Service

```
// Heuristic exceptions
exception HeuristicMixed {};
exception HeuristicHazard {};

// Other transaction-specific exceptions
exception SubtransactionsUnavailable {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
```

Table 2-1 describes the exceptions.

Note: This information comes from *CORBAservices: Common Object Services Specification*, pages 10-16, 19, 20. Revised Edition: March 31, 1995. Updated: March 1997. Used with permission by OMG.

Table 2-1 Exceptions Supported by the Transaction Service

Exception	Description
`HeuristicMixed`	A request raises this exception to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.
`HeuristicHazard`	A request raises this exception to report that a heuristic decision was made, that the disposition of all relevant updates is not known, and that for those updates whose disposition is known, either all have been committed or all have been rolled back. Therefore, the `HeuristicMixed` exception takes priority over the `HeuristicHazard` exception.
`SubtransactionsUnavailable`	This exception is raised for the `Current` interface `begin` method if the client already has an associated transaction.
`NoTransaction`	This exception is raised for the `Current` interface `rollback` and `rollback_only` methods if there is no transaction associated with the client thread.
`InvalidControl`	This exception is raised for the `Current` interface `resume` method if the parameter is not valid in the current execution environment.
`Unavailable`	This exception is raised for the `Control` interface `get_terminator` and `get_coordinator` methods if the `Control` interface cannot provide the requested object.

Current Interface

The `Current` interface defines methods that allow a client of the Transaction Service to explicitly manage the association between threads and transactions. The `Current` interface also defines methods that simplify the use of the Transaction Service for most applications. These methods can be used to begin and end transactions, to suspend and resume transactions, and to obtain information about the current transaction.

The `CosTransactions` module defines the `Current` interface (shown in Listing 2-3).

Listing 2-3 `Current` Interface idl

```
// Current transaction
interface Current : CORBA::Current {
       void begin()
              raises(SubtransactionsUnavailable);
       void commit(in boolean report_heuristics)
              raises(
                     NoTransaction,
                     HeuristicMixed,
                     HeuristicHazard
             );
       void rollback()
               raises(NoTransaction);
       void rollback_only()
               raises(NoTransaction);
       Status get_status();
       string get_transaction_name();
       void set_timeout(in unsigned long seconds);
       Control get_control();
       Control suspend();
       void resume(in Control which)
               raises(InvalidControl);

};

// This information comes from CORBAservices: Common Object
// Services Specification, p. 10-18. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
// OMG
```

Table 2-2 provides a description of the `Current` transaction methods.

Note: This information comes from *CORBAservices: Common Object Services Specification*, pages 10-18, 19, 20. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Table 2-2 `Current` Transaction Methods

Method	Description
`begin`	Creates a new transaction. The transaction context of the client thread is modified so that the thread is associated with the new transaction. If the client thread is currently associated with a transaction, the `SubtransactionsUnavailable` exception is raised. If the client thread cannot be placed in transaction mode due to an error while starting the transaction, the standard system exception `INVALID_TRANSACTION` is raised. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised.
`commit`	If there is no transaction associated with the client thread, the `NoTransaction` exception is raised. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. If the system decides to roll back the transaction, the standard exception `TRANSACTION_ROLLEDBACK` is raised and the thread's transaction context is set to `null`. A `HeuristicMixed` exception is raised to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back. A `HeuristicHazard` exception is raised to report that a heuristic decision was made, and that the disposition of all relevant updates is not known; for those updates whose disposition is known, either all have been committed or all have been rolled back. The `HeuristicMixed` exception takes priority over the `HeuristicHazard` exception. If a heuristic exception is raised or the operation completes normally, the thread's transaction exception context is set to `null`. If the operation completes normally, the thread's transaction context is set to `null`.

Table 2-2 `Current` Transaction Methods (Continued)

Method	Description
`rollback`	If there is no transaction associated with the client thread, the `NoTransaction` exception is raised. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. If the operation completes normally, the thread's transaction context is set to `null`.
`rollback_only`	If there is no transaction associated with the client thread, the `NoTransaction` exception is raised. Otherwise, the transaction associated with the client thread is modified so that the only possible outcome is to roll back the transaction.
`get_status`	If there is no transaction associated with the client thread, the `StatusNoTransaction` value is returned. Otherwise, this method returns the status of the transaction associated with the client thread.
`get_transaction_name`	If there is no transaction associated with the client thread, an empty string is returned. Otherwise, this method returns a printable string describing the transaction (specifically, the `XID` as specified by The Open Group). The returned string is intended to support debugging.

Table 2-2 `Current` Transaction Methods (Continued)

Method	Description
`set_timeout`	This method modifies a state variable associated with the target object that affects the time-out period associated with transactions created by subsequent invocations of the `begin` method. The initial transaction timeout value is 300 seconds. Calling `set_timeout()` with an argument value larger than zero specifies a new timeout value. Calling `set_timeout()` with a zero argument sets the timeout value back to the default of 300 seconds. After calling `set_timeout()`, transactions created by subsequent invocations of `begin` are subject to being rolled back if they do not complete before the specified number of seconds after their creation. **Note:** The initial transaction timeout value is 300 seconds. If a transaction is started via `AUTOTRAN` instead of the `begin` method, then the timeout value is determined by the `TRANTIME` value in the WLE configuration file. For more information, see Chapter 11, “Administering Transactions.”
`get_control`	If the client is not associated with a transaction, a `null` object reference is returned. Otherwise, a `Control` object is returned that represents the transaction context currently associated with the client thread. This object may be given to the `resume` method to reestablish this context.

Table 2-2 `Current` Transaction Methods (Continued)

Method	Description
`suspend`	If the client thread is not associated with a transaction, a null object reference is returned. If the associated transaction is in a state such that the only possible outcome of the transaction is to be rolled back, the standard system exception `TRANSACTION_ROLLEDBACK` is raised and the client thread becomes associated with no transaction. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. The caller's state with respect to the transaction is not changed. Otherwise, an object is returned that represents the transaction context currently associated with the client thread. The same client can subsequently give this object to the `resume` method to reestablish this context. In addition, the client thread becomes associated with no transaction. **Note:** As defined in The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998, the standard system exception `TRANSACTION_ROLLEDBACK` indicates that the transaction associated with the request has already been rolled back or has been marked to roll back. Thus, the requested method either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

Table 2-2 `Current` Transaction Methods (Continued)

Method	Description
`resume`	If the client thread is already associated with a transaction which is in a state such that the only possible outcome of the transaction is to be rolled back, the standard system exception `TRANSACTION_ROLLEDBACK` is raised and the client thread becomes associated with no transaction. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. If the system is unable to resume the global transaction because the caller is currently participating in work outside any global transaction with one or more Resource Managers, the standard system exception `INVALID_TRANSACTION` is raised. If the parameter is a `null` object reference, the client thread becomes associated with no transaction. If the parameter is valid in the current execution environment, the client thread becomes associated with that transaction (in place of any previous transaction). Otherwise, the `InvalidControl` exception is raised. **Note:** See `suspend` for a definition of the standard system exception `TRANSACTION_ROLLEDBACK`.

Control Interface

The `Control` interface allows a program to explicitly manage or propagate a transaction context. An object that supports the `Control` interface is implicitly associated with one specific transaction.

Listing 2-4 shows the `Control` interface, which is defined in the `CosTransactions` module.

Listing 2-4 `Control` Interface

```
interface Control {
       Terminator get_terminator()
              raises(Unavailable);
       Coordinator get_coordinator()
```

```
              raises(Unavailable);
};

// This information comes from CORBAservices: Common Object
// Services Specification, p. 10-21. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
// OMG.
```

The `Control` interface is used only in conjunction with the `suspend` and `resume` methods.

TransactionalObject Interface

The `org.omg.CosTransactions.TransactionalObject` interface (in Java) or `CosTransactions::TransactionalObject` (in C++) is used by an object to indicate that it is transactional. By supporting this interface, an object indicates that it wants the transaction context associated with the client thread to be propagated on requests to the object. *However, this interface is no longer needed*. For details on transaction policies that need to be set to infect objects with transactions, see "Server Description File" in the *CORBA Java Programming Reference* or "Implementation Configuration File (ICF)" in the *CORBA C++ Programming Reference.*

The `CosTransactions` module defines the `TransactionalObject` interface (shown in Listing 2-5). The `org.omg.CosTransactions.TransactionalObject` interface defines no methods. It is simply a marker.

Listing 2-5 `TransactionalObject` Interface

```
interface TransactionalObject {
};

// This information comes from CORBAservices: Common Object
// Services Specification, p. 10-30. Revised Edition:
// March 31, 1995. Updated: November 1997. Used with permission by
// OMG.
```

Other CORBAservices Object Transaction Service Interfaces

All other CORBAservices Object Transaction Service interfaces are *not* supported. Note that the `Current` interface described earlier is supported only if it has been obtained from the `Bootstrap` object. The `Control` interface described earlier is supported only if it has been obtained using the `get_control` and the `suspend` methods on the `Current` object.

CORBA Transaction Service API Extensions

This topic describes specific extensions to the CORBAservices Transaction Service API described earlier. The APIs in this topic enable an application to open or close an Open Group Resource Manager.

The following APIs help facilitate participation of Resource Managers in a distributed transaction by allowing their two-phase commit protocol to be controlled via The Open Group XA interface.

The following definitions and interfaces are defined in the `com.beasys.Tobj` module (in Java) or `Tobj` module (in C++).

Exception

The following exception is supported:

```
exception RMfailed {};
```

A request raises this exception to report that an attempt to open or close a Resource Manager failed.

TransactionCurrent Interface

This interface supports all the methods of the `Current` interface in the `CosTransactions` module and is described in “Java Bootstrap Object Programming Reference” in the *CORBA Java Programming Reference* or in “C++ Bootstrap Object Programming Reference” in the *CORBA C++ Programming Reference*. Additionally, this interface supports APIs to open and close the Resource Manager.

Listing 2-6 shows the `TransactionCurrent` interface, which is defined in the `Tobj` module.

Listing 2-6 `TransactionCurrent` Interface

```
Interface TransactionCurrent: CosTransactions::Current {
      void open_xa_rm()
             raises(RMfailed);
      void close_xa_rm()
             raises(Rmfailed);
}
```

Table 2-3 describes APIs that are specific to the Resource Manager. For more information about these APIs, see the *CORBA Java Programming Reference* or the *CORBA C++ Programming Reference*.

Table 2-3 Resource Manager APIs for the `Current` Interface

Method	Description
`open_xa_rm`	This method opens The Open Group Resource Manager to which this process is linked. A `RMfailed` exception is raised if there is a failure while opening the Resource Manager. Any attempts to invoke this method by remote clients or the native clients raises the standard system exception `NO_IMPLEMENT`.
`close_xa_rm`	This method closes The Open Group Resource Manager to which this process is linked. An `RMfailed` exception is raised if there is a failure while closing the Resource Manager. A `BAD_INV_ORDER` standard system exception is raised if the function was called in an improper context (for example, the caller is in transaction mode). Any attempts by the remote clients or the native clients to invoke this method raises the standard system exception `NO_IMPLEMENT`.

Notes on Using Transactions in WLE CORBA Applications

Consider the following guidelines when integrating transactions into your WLE CORBA client/server applications:

- Nested transactions are not permitted in the WLE system.

You cannot start a new transaction if an existing transaction is already active. (You may start a new transaction if you first suspend the existing one; however, the object that suspends the transaction is the only object that can subsequently resume the transaction.)

- The object that starts a transaction is the only entity that can end the transaction. (In a strict sense, the object can be the client application, the TP Framework, or an object managed by the server application.) An object that is invoked within the scope of a transaction may suspend and resume the transaction (and while the transaction is suspended, the object can start and end other transactions). However, you cannot end a transaction in an object unless you began the transaction there.
- WLE does not support concurrent transactions. Objects can be involved with only one transaction at one time. An object is involved in a transaction for the duration of the entire transaction, and is available to be involved in a different transaction only after the current transaction is completed.
- WLE does not queue requests to objects that are currently involved in a transaction. If a non-transactional client application attempts to invoke an operation on an object that is currently in a transaction, the client application receives the following error message:

  Java:

  `org.omg.CORBA.OBJ_ADAPTER`

  C++:

  `CORBA::OBJ_ADAPTER`

  If a client that is in a transaction attempts to invoke an operation on an object that is currently in a different transaction, the client application receives the following error message:

  Java:

  `org.omg.CORBA.INVALID_TRANSACTION`

  C++:

  `CORBA::INVALID_TRANSACTION`

- For transaction-bound objects, consider doing all state handling in the `com.beasys.Tobj_Servant.deactivate_object` method (in Java) or `Tobj_ServantBase::deactivate_object()` operation (in C++). This makes it easier for the object to handle its state properly, because the outcome of the transaction is known at the time that `deactivate_object()` is invoked.

- For method-bound objects that have several operations, but only a few that affect the object's durable state, consider doing the following:
  - Assign the `optional` transaction policy.
  - Scope any write operations within a transaction, by making invocations on the `TransactionCurrent` object.

  If the object is invoked outside a transaction, the object does not incur the overhead of scoping a transaction for reading data. This way, regardless of whether the object is invoked within a transaction, all the object's write operations are handled transactionally.

- Transaction rollbacks are asynchronous. Therefore, it is possible for an object to be invoked while its transactional context is still active. If you try to invoke such an object, you receive an exception.

- If an object with the `always` transaction policy is involved in a transaction that is started by the WLE system, and not the client application, note the following:
  - If the server application marks the transaction for rollback only and the server throws a CORBA exception, the client application receives the CORBA exception.
  - If the server application marks the transaction for rollback only and the server does *not* throw a CORBA exception, the client application receives the `OBJ_ADAPTER` exception. In this case, the WLE system automatically rolls back the transaction. However, the client application is completely unaware that a transaction has been scoped in the WLE domain.

- If the client application initiates a transaction, and the server application marks the transaction for a rollback, one of the following occurs:
  - If the server throws a CORBA exception, the client application receives a CORBA exception.
  - If the server does *not* throw a CORBA exception, the client application receives the `TRANSACTION_ROLLEDBACK` exception.

Transaction Service in EJB Applications

The WLE EJB Container provides a Transaction Service that supports the two types of transactions in WLE EJB applications:

- **Container-managed transactions**. In container-managed transactions, the WLE EJB Container manages the transaction demarcation. Transaction attributes in the EJB deployment descriptor determine how the WLE 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 “UserTransaction API” on page 2-24.

For an introduction to transaction management in EJB applications, see “Transactions in WLE EJB Applications” on page 1-8, and “Transactions Sample EJB Code” on page 1-24.

Transaction Service in RMI Applications

WLE provides a Transaction Service that supports transactions in WLE 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 “UserTransaction API” on page 2-24. For an introduction to transaction management in RMI applications, see “Transactions in WLE RMI Applications” on page 1-10, and “Transactions Sample RMI Code” on page 1-27.

UserTransaction API

This topic includes the following sections:

- UserTransaction Methods
- Exceptions Thrown by UserTransaction Methods

WLE provides Sun Microsystems, Inc.'s `javax.transaction` package, which implements the Java Transaction API (JTA) for Java applications. The `javax.UserTransaction` interface supports transaction management for CORBA Java applications as well as for bean-managed transactions in EJB applications. For more information about the JTA, see Sun Microsystem's *Java Transaction API (JTA) Specification* (version1.0.1). For a detailed description of the `javax.transaction` interface, see the package description in the *WLE Javadoc*.

UserTransaction Methods

Table 2-4 describes the methods in the `UserTransaction` object.

Table 2-4 `UserTransaction` Methods

Method Name	Description
`begin`	Starts a transaction on the current thread.
`commit`	Commits the transaction associated with the current thread.

Table 2-4 `UserTransaction` Methods (Continued)

Method Name	Description
`getStatus`	Returns the transaction status, or `STATUS_NO_TRANSACTION` if no transaction is associated with the current thread. One of the following values: ■ `STATUS_ACTIVE` ■ `STATUS_COMMITTED` ■ `STATUS_COMMITTING` ■ `STATUS_MARKED_ROLLBACK` ■ `STATUS_NO_TRANSACTION` ■ `STATUS_PREPARED` ■ `STATUS_PREPARING` ■ `STATUS_ROLLEDBACK` ■ `STATUS_ROLLING_BACK` ■ `STATUS_UNKNOWN`
`rollback`	Rolls back the transaction associated with the current thread.
`setRollbackOnly`	Marks the transaction associated with the current thread so that the only possible outcome of the transaction is to roll it back.
`setTransactionTimeout`	Specifies the timeout value for the transactions started by the current thread with the `begin` method. If an application has not called the `begin` method, then the Transaction Service uses a default value for the transaction timeout.

Exceptions Thrown by UserTransaction Methods

Table 2-5 describes exceptions thrown by methods of the `UserTransaction` object.

Table 2-5 Exceptions Thrown by `UserTransaction` Methods

Exception	Description
`HeuristicMixedException`	Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.
`HeuristicRollbackException`	Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.
`NotSupportedException`	Thrown when the requested operation is not supported (such as a nested transaction).
`RollbackException`	Thrown when the transaction has been marked for rollback only or the transaction has been rolled back instead of committed.
`IllegalStateException`	Thrown if the current thread is not associated with a transaction.
`SecurityException`	Thrown to indicate that the thread is not allowed to commit the transaction.
`SystemException`	Thrown by the Transaction Manager to indicate that it has encountered an unexpected error condition that prevents future transaction services from proceeding.

CHAPTER

3 Transactions in CORBA Server Applications

This topic includes the following sections:

- Integrating Transactions in a WLE Client and Server Application
- Transactions and Object State Management
- User-Defined Exceptions

These sections describe how to integrate transactions into a WebLogic Enterprise (WLE) server application. Before you begin, you should read Chapter 1, "Introducing Transactions."

Integrating Transactions in a WLE Client and Server Application

This topic includes the following sections:

- Transaction Support in CORBA Applications
- Making an Object Automatically Transactional
- Enabling an Object to Participate in a Transaction
- Preventing an Object from Being Invoked While a Transaction Is Scoped
- Excluding an Object from an Ongoing Transaction
- Assigning Policies
- Using an XA Resource Manager
- Opening an XA Resource Manager
- Closing an XA Resource Manager

Transaction Support in CORBA Applications

WLE supports transactions in the following ways:

- The client or the server application can begin and end transactions explicitly by using calls on the `TransactionCurrent` object. For details about the `TransactionCurrent` object, see Chapter 4, “Transactions in CORBA Client Applications.”
- You can assign transactional policies to an object’s interface so that when the object is invoked, the WLE system can start a transaction automatically for that object, if a transaction has not already been started, and commit or roll back the transaction when the method invocation is complete. You use transactional policies on objects in conjunction with an XA Resource Manager and database when you want to delegate all the transaction commit and rollback responsibilities to that Resource Manager.

- Objects involved in a transaction can force a transaction to be rolled back. That is, after an object has been invoked within the scope of a transaction, the object can invoke `rollback_only()` on the `TransactionCurrent` object to mark the transaction for rollback only. This prevents the current transaction from being committed. An object may need to mark a transaction for rollback if an entity, typically a database, is otherwise at risk of being updated with corrupt or inaccurate data.
- Objects involved in a transaction can be kept in memory from the time they are first invoked until the moment when the transaction is ready to be committed or rolled back. In the case of a transaction that is about to be committed, these objects are polled by the WLE system immediately before the Resource Managers prepare to commit the transaction. In this sense, polling means invoking the object's `com.beasys.Tobj_Servant.deactivate_object` method (in Java) or `Tobj_ServantBase::deactivate_object()` operation (in C++) and passing a reason value.

  When an object is polled, the object may veto the current transaction by invoking `rollback_only()` on the `TransactionCurrent` object. In addition, if the current transaction is to be rolled back, objects have an opportunity to skip any writes to a database. If no object vetoes the current transaction, the transaction is committed.

The following sections explain how you can use object activation policies and transaction policies to determine the transactional behavior you want in your objects. Note that these policies apply to an interface and, therefore, to all operations on all objects implementing that interface.

Note: If a server application manages an object that you want to be able to participate in a transaction, the `Server` object for that application must invoke the `com.beasys.Tobj.TP.open_xa_rm` and `com.beasys.Tobj.TP.close_xa_rm` methods (in Java), or the `TP::open_xa_rm()` and `TP::close_xa_rm()` operations (in C++). For more information about database connections, see "Opening an XA Resource Manager" on page 3-8.

Making an Object Automatically Transactional

The WLE system provides the `always` transactional policy, which you can define on an object's interface to have the WLE system start a transaction automatically when that object is invoked and a transaction has not already been scoped. When an invocation on that object is completed, the WLE system commits or rolls back the transaction automatically. Neither the server application, nor the object implementation, needs to invoke the `TransactionCurrent` object in this situation; the WLE system automatically invokes the `TransactionCurrent` object on behalf of the server application.

Assign the `always` transactional policy to an object's interface when:

- The object writes to a database and you want all the database commit or rollback responsibilities delegated to an XA Resource Manager whenever this object is invoked.
- You want to give the client application the opportunity to include the object in a larger transaction that encompasses invocations on multiple objects, and the invocations must all succeed or be rolled back if any one invocation fails.

If you want an object to be automatically transactional, assign the following policies to that object's interface in the XML-based Server Description File (in Java) or Implementation Configuration File (in C++):

Activation Policies	Transaction Policy
■ `process` ■ `method` ■ `transaction`	`always`

Note: Database cursors cannot span transactions. However, in C++, the `CourseSynopsisEnumerator` object in the WLE University sample applications uses a database cursor to find matching course synopses from the University database. Because database cursors cannot span transactions, the `activate_object()` operation on the `CourseSynopsisEnumerator` object reads all matching course synopses into memory. Note that the cursor is managed by an iterator class and is thus not visible to the `CourseSynopsisEnumerator` object.

Enabling an Object to Participate in a Transaction

If you want an object to be able to be invoked within the scope of a transaction, you can assign the `optional` transaction policies to that object's interface. The `optional` transaction policy may be appropriate for an object that does not perform any database write operations, but that you want to have the ability to be invoked during a transaction.

You can use the following policies, when they are specified in the XML-based Server Description File (in Java) or Implementation Configuration File (in C++) for that object's interface, to make an object optionally transactional:

Activation Policies	Transaction Policy
■ `process` ■ `method` ■ `transaction`	`optional`

When the transaction policy is `optional`, if the `AUTOTRAN` parameter is enabled in the application's `UBBCONFIG` file, the implementation is transactional. Servers containing transactional objects must be configured within a group associated with an XA-compliant Resource Manager.

If the object does perform database write operations, and you want the object to be able to participate in a transaction, assigning the `always` transactional policy is generally a better choice. However, if you prefer, you can use the `optional` policy and encapsulate any write operations within invocations on the `TransactionCurrent` object. That is, within your operations that write data, scope a transaction around the write statements by invoking the `TransactionCurrent` object to, respectively, begin and commit or roll back the transaction, if the object is not already scoped within a transaction. This ensures that any database write operations are handled transactionally. This also introduces a performance efficiency: if the object is not invoked within the scope of a transaction, all the database read operations are nontransactional, and, therefore, more streamlined.

Note: When choosing the transaction policies to assign to your objects, make sure you are familiar with the requirements of the XA Resource Manager you are using. For example, some XA Resource Managers (such as the Oracle 7 Transaction Manager Server) require that any object participating in a transaction scope their database read operations, in addition to write operations, within a transaction (you can still scope your own transactions, however). Other Resource Managers, such as Oracle8i, do not require a transaction context for read and write operations. If an application attempts a write operation without a transaction context, Oracle8i will start a local transaction implicitly, in which case the application needs to commit the local transaction explicitly.

Preventing an Object from Being Invoked While a Transaction Is Scoped

In many cases, it may be critical to exclude an object from a transaction. If such an object is invoked during a transaction, the object returns an exception, which may cause the transaction to be rolled back. The WLE system provides the `never` transaction policy, which you can assign to an object's interface to specifically prevent that object from being invoked within the course of a transaction, even if the current transaction is suspended.

This transaction policy is appropriate for objects that write durable state to disk that cannot be rolled back, such as for an object that writes data to a disk that is not managed by an XA Resource Manager. Having this capability in your client/server application is crucial if the client application does not or cannot know if some of its invocations are causing a transaction to be scoped. Therefore, if a transaction is scoped, and an object with this policy is invoked, the transaction can be rolled back.

To prevent an object from being invoked while a transaction is scoped, assign the following policies to that object's interface in the XML-based Server Description File (in Java) or Implementation Configuration File (in C++):

Activation Policies	Transaction Policy
■ `process` ■ `method`	`never`

Excluding an Object from an Ongoing Transaction

In some cases, it may be appropriate to permit an object to be invoked during the course of a transaction but also keep that object from being a part of the transaction. If such an object is invoked during a transaction, the transaction is automatically suspended. After the invocation on the object is completed, the transaction is automatically resumed. The WLE system provides the `ignore` transaction policy for this purpose.

The `ignore` transaction policy may be appropriate for an object such as a factory that typically does not write data to disk. By excluding the factory from the transaction, the factory can be available to other client invocations during the course of a transaction. In addition, using this policy can introduce an efficiency into your server application because it minimizes the overhead of invoking objects transactionally.

To prevent any transaction from being propagated to an object, assign the following policies to that object's interface in the Server Description File (in Java) or Implementation Configuration File (in C++):

Activation Policies	Transaction Policy
■ `process` ■ `method`	`ignore`

Assigning Policies

For information about how to create a Server Description File (in Java) or Implementation Configuration File (in C++) and specify policies on objects, see "Step 5: Define the object activation and transaction policies" in "Steps for Creating a WLE Server Application" in *Creating Java Server Applications*, or "Step 4: Define the in-memory behavior of objects" in "Steps for Creating a WLE Server Application" in *Creating C++ Server Applications*.

Using an XA Resource Manager

The Transaction Manager Server (TMS) handles object state data automatically. For example, the XA Bankapp sample C++ application in the `drive:\M3dir\samples\corba\bankapp_java\XA` directory uses the Oracle7 TMS as an example of a relational database management service (RDBMS).

Using any XA Resource Manager imposes specific requirements on how different objects managed by the server application may read and write data to that database, including the following:

- Some XA Resource Managers, such as Oracle7, require that all database operations be scoped within a transaction. This means that all method invocations on the `DBaccess` object need to be scoped within a transaction because this object reads from a database. The transaction can be started either by the client or by the WLE system.

  Other XA Resource Managers, such as Oracle8i, do not require a transaction context for read and write operations. If an application attempts a write operation without a transaction context, Oracle8i will start a local transaction implicitly, in which case the application needs to commit the local transaction explicitly.

- When a transaction is committed or rolled back, the XA Resource Manager automatically handles the durable state implied by the commit or rollback. That is, if the transaction is committed, the XA Resource Manager ensures that all database updates are made permanent. Likewise, if there is a rollback, the XA Resource Manager automatically restores the database to its state prior to the beginning of the transaction.

  This characteristic of XA Resource Managers actually makes the design problems associated with handling object state data in the event of a rollback much simpler. Transactional objects can always delegate the commit and rollback responsibilities to the XA Resource Manager, which greatly simplifies the task of implementing a server application.

Opening an XA Resource Manager

This section describes how to open the XA Resource Manager in Java and C++.

Opening an XA Resource Manager in Java

If an object's interface has the `always` or `optional` transaction policy, you must invoke the `com.beasys.Tobj.TP.open_xa_rm` method in the `com.beasys.Tobj.Server.initialize` method in the Server object that supports this object. You must build a special version of the JavaServer by using the `buildXAJS` command, if your object performs database operations.

In the `SERVERS` section of the application's `UBBCONFIG` file, you must use the `JavaServerXA` element in place of `JavaServer` to associate the XA Resource Manager with a specified server group. (`JavaServer` uses the null RM.)

The Resource Manager is opened using the information provided in the `OPENINFO` parameter, which is in the `GROUPS` section of the `UBBCONFIG` file. Note that the default version of the `com.beasys.Tobj.Server.initialize` method automatically opens the Resource Manager.

If you have an object that participates in a transaction but does not actually perform database operations (the object typically has the `optional` transaction policy), you still need to include an invocation to the `com.beasys.Tobj.TP.open_xa_rm` method.

Opening an XA Resource Manager in C++

If an object's interface has the `always` or `optional` transaction policy, you must invoke the `TP::open_xa_rm()` operation in the `Server::initialize()` operation in the Server object. The Resource Manager is opened using the information provided in the `OPENINFO` parameter, which is in the `GROUPS` section of the `UBBCONFIG` file. Note that the default version of the `Server::initialize()` operation automatically opens the Resource Manager.

If you have an object that does not write data to disk and that participates in a transaction—the object typically has the `optional` transaction policy—you still need to include an invocation to the `TP::open_xa_rm()` operation. In that invocation, specify the `NULL` Resource Manager.

Closing an XA Resource Manager

If your Server object's `com.beasys.Tobj.Server.initialize` method (in Java) or `Server::initialize()` operation (in C++) opens an XA Resource Manager, you must include the following invocation in the `com.beasys.Tobj.Server.release` method (in Java) or `Server::release()` operation (in C++):

Java:

```
com.beasys.Tobj.TP.close_xa_rm();
```

C++:

```
TP::close_xa_rm();
```

Transactions and Object State Management

This topic includes the following sections:

- Delegating Object State Management to an XA Resource Manager
- Waiting Until Transaction Work Is Complete Before Writing to the Database

If you need transactions in your WLE client and server application, you can integrate transactions with object state management in a few different ways. In general, the WLE system can automatically scope the transaction for the duration of an operation invocation without requiring you to make any changes to your application's logic or the way in which the object writes durable state to disk.

Delegating Object State Management to an XA Resource Manager

Using an XA Resource Manager, such as Oracle7, generally simplifies the design problems associated with handling object state data in the event of a rollback. (The Oracle7 Resource Manager is used in the WLE University sample C++ applications). Transactional objects can always delegate the commit and rollback responsibilities to the XA Resource Manager, which greatly simplifies the task of implementing a server application. This means that process- or method-bound objects involved in a transaction can write to a database during transactions, and can depend on the Resource Manager to undo any data written to the database in the event of a transaction rollback.

Waiting Until Transaction Work Is Complete Before Writing to the Database

The `transaction` activation policy is a good choice for objects that maintain state in memory that you do not want written, or that cannot be written, to disk until the transaction work is complete. When you assign the `transaction` activation policy to an object, the object:

- Is brought into memory when it is first invoked within the scope of a transaction
- Remains in memory until the transaction is either committed or rolled back

When the transaction work is complete, the WLE system invokes each transaction-bound object’s `com.beasys.Tobj_Servant.deactivate_object` method (in Java) or `Tobj_ServantBase::deactivate_object()` operation (in C++), passing a `reason` code that can be either `DR_TRANS_COMMITTING` or `DR_TRANS_ABORTED`. If the variable is `DR_TRANS_COMMITTING`, the object can invoke its database write operations. If the variable is `DR_TRANS_ABORTED`, the object skips its write operations.

When to Assign the Transaction Activation Policy

Assigning the `transaction` activation policy to an object may be appropriate in the following situations:

- You want the object to write its persistent state to disk at the time that the transaction work is complete.

  This introduces a performance efficiency because it reduces the number of database write operations that may need to be rolled back.

- You want to provide the object with the ability to veto a transaction that is about to be committed.

If the WLE system passes the reason `DR_TRANS_COMMITTING`, the object can, if necessary, invoke `rollback_only()` on the `TransactionCurrent` object. Note that if you do make an invocation to `rollback_only()` from within the `com.beasys.Tobj_Servant.deactivate_object` method (in Java) or `Tobj_ServantBase::deactivate_object()` operation (in C++), then `deactivate_object()` is not invoked again.

- You want to provide the object with the ability to perform batch updates.
- You have an object that is likely to be invoked multiple times during the course of a single transaction, and you want to avoid the overhead of continually activating and deactivating the object during that transaction.

Transaction Policies to Use with the Transaction Activation Policy

To give an object the ability to wait until the transaction is committing before writing to a database, assign the following policies to that object's interface in the XML-based Server Description File (in Java) or Implementation Configuration File (in C++):

Activation Policy	Transaction Policy
`transaction`	`always` or `optional`

Note: Transaction-bound objects cannot start a transaction or invoke other objects from inside the `com.beasys.Tobj_Servant.deactivate_object` method (in Java) or `Tobj_ServantBase::deactivate_object()` operation (in C++). The only valid invocations transaction-bound objects can make inside `deactivate_object()` are write operations to the database.

Also, if you have an object that is involved in a transaction, the Server object that manages that object must include invocations to open and close the XA Resource Manager, even if the object does not write any data to disk. (If you have a transactional object that does not write data to disk, you specify the `NULL` Resource Manager.) For more information about opening and closing an XA Resource Manager, see "Opening an XA Resource Manager" on page 3-8 and "Closing an XA Resource Manager" on page 3-10.

User-Defined Exceptions

This topic includes the following sections:

- About User-Defined Exceptions
- Defining the Exception
- Throwing the Exception

About User-Defined Exceptions

Including a user-defined exception in a WLE client/server application involves the following steps:

1. In your OMG IDL file, define the exception and specify the operations that can use it.
2. In the implementation file, include code that throws the exception.
3. In the client application source file, include code that catches and handles the exception.

For example, the Transactions sample C++ application includes an instance of a user-defined exception, `TooManyCredits`. This exception is thrown by the server application when the client application tries to register a student for a course, and the student has exceeded the maximum number of courses for which he or she can register. When the client application catches this exception, the client application rolls back the transaction that registers a student for a course. This section explains how you can define and implement user-defined exceptions in your WLE client/server application, using the `TooManyCredits` exception as an example.

Defining the Exception

In the OMG IDL file for your client/server application:

1. Define the exception and define the data sent with the exception. For example, the `TooManyCredits` exception is defined to pass a short integer representing the maximum number of credits for which a student can register. Therefore, the definition for the `TooManyCredits` exception contains the following OMG IDL statements:

   ```
   exception TooManyCredits
   {
       unsigned short maximum_credits;
   };
   ```

2. In the definition of the operations that throw the exception, include the exception. The following example shows the OMG IDL statements for the `register_for_courses()` operation on the `Registrar` interface:

   ```
   NotRegisteredList register_for_courses(
       in StudentId        student,
       in CourseNumberList courses
   ) raises (
       TooManyCredits
   );
   ```

Throwing the Exception

In the implementation of the operation that uses the exception, write the code that throws the exception, as in the following C++ example.

```
if ( ... ) {
    UniversityZ::TooManyCredits e;
    e.maximum_credits = 18;
    throw e;
```

How the Transactions University Sample Application Works (C++ Only)

This topic includes the following sections:

- About the Transactions University Sample Application

- Transactional Model Used by the Transactions University Sample Application
- Object State Considerations for the University Server Application
- Configuration Requirements for the Transactions Sample Application

About the Transactions University Sample Application

To implement the student registration process, the Transactions sample application does the following:

- The client application obtains a reference to the `TransactionCurrent` object from the `Bootstrap` object.
- When the student submits the list of courses for which he or she wants to register, the client application:

  a. Begins a transaction by invoking the `Current::begin()` operation on the TransactionCurrent object

  b. Invokes the `register_for_courses()` operation on the `Registrar` object, passing a list of courses

- The `register_for_courses()` operation on the `Registrar` object processes the registration request by executing a loop that does the following iteratively for each course in the list:

  a. Checks to see how many credits the student is already registered for

  b. Adds the course to the list of courses for which the student is registered

  The `Registrar` object checks for the following potential problems, which prevent the transaction from being committed:

  - The student is already registered for the course.
  - A course in the list does not exist.
  - The student exceeds the maximum credits allowed.

- As defined in the application's OMG IDL, the `register_for_courses()` operation returns a parameter to the client application, `NotRegisteredList`, which contains a list of the courses for which the registration failed.

If the `NotRegisteredList` value is empty, the client application commits the transaction.

If the `NotRegisteredList` value contains any courses, the client application queries the student to indicate whether he or she wants to complete the registration process for the courses for which the registration succeeded. If the user chooses to complete the registration, the client application commits the transaction. If the user chooses to cancel the registration, the client application rolls back the transaction.

- If the registration for a course has failed because the student exceeds the maximum number of credits he or she can take, the `Registrar` object returns a `TooManyCredits` exception to the client application, and the client application rolls back the entire transaction.

Transactional Model Used by the Transactions University Sample Application

The basic design rationale for the Transactions sample application is to handle course registrations in groups, as opposed to one at a time. This design helps to minimize the number of remote invocations on the `Registrar` object.

In implementing this design, the Transactions sample application shows one model of the use of transactions, which were described in “Integrating Transactions in a WLE Client and Server Application” on page 3-2. The model is as follows:

- The client begins a transaction by invoking the `begin()` operation on the TransactionCurrent object, followed by making an invocation to the `register_for_courses()` operation on the `Registrar` object.

  The `Registrar` object registers the student for the courses for which it can, and then returns a list of courses for which the registration process was unsuccessful. The client application can choose to commit the transaction or roll it back. The transaction encapsulates this conversation between the client and the server application.

- The `register_for_courses()` operation performs multiple checks of the University database. If any one of those checks fail, the transaction can be rolled back.

Object State Considerations for the University Server Application

Because the Transactions University sample application is transactional, the University server application generally needs to consider the implications on object state, particularly in the event of a rollback. In cases where there is a rollback, the server application must ensure that all affected objects have their durable state restored to the proper state.

Because the `Registrar` object is being used for database transactions, a good design choice for this object is to make it transactional (assign the `always` transaction policy to this object's interface). If a transaction has not already been scoped when this object is invoked, the WLE system will start a transaction automatically.

By making the `Registrar` object automatically transactional, all database write operations performed by this object will always be done within the scope of a transaction, regardless of whether the client application starts one. Since the server application uses an XA Resource Manager, and since the object is guaranteed to be in a transaction when the object writes to a database, the object does not have any rollback or commit responsibilities because the XA Resource Manager takes responsibility for these database operations on behalf of the object.

The `RegistrarFactory` object, however, can be excluded from transactions because this object does not manage data that is used during the course of a transaction. By excluding this object from transactions, you minimize the processing overhead implied by transactions.

Object Policies Defined for the Registrar Object

To make the `Registrar` object transactional, the ICF file specifies the `always` transaction policy for the `Registrar` interface. Therefore, in the Transaction sample application, the ICF file specifies the following object policies for the `Registrar` interface:

Activation Policy	Transaction Policy
`process`	`always`

Object Policies Defined for the RegistrarFactory Object

To exclude the `RegistrarFactory` object from transactions, the ICF file specifies the `ignore` transaction policy for the `Registrar` interface. Therefore, in the Transaction sample application, the ICF file specifies the following object policies for the `RegistrarFactory` interface:

Activation Policy	Transaction Policy
`process`	`ignore`

Using an XA Resource Manager in the Transactions Sample Application

The Transactions sample application uses the Oracle7 Transaction Manager Server (TMS), which handles object state data automatically. Using any XA Resource Manager imposes specific requirements on how different objects managed by the server application may read and write data to that database, including the following:

- Some XA Resource Managers, such as Oracle7, require that all database operations be scoped within a transaction. This means that the `CourseSynopsisEnumerator` object needs to be scoped within a transaction because this object reads from a database.
- When a transaction is committed or rolled back, the XA Resource Manager automatically handles the durable state implied by the commit or rollback. That is, if the transaction is committed, the XA Resource Manager ensures that all database updates are made permanent. Likewise, if there is a rollback, the XA Resource Manager automatically restores the database to its state prior to the beginning of the transaction.

  This characteristic of XA Resource Managers actually makes the design problems associated with handling object state data in the event of a rollback much simpler. Transactional objects can always delegate the commit and rollback responsibilities to the XA Resource Manager, which greatly simplifies the task of implementing a server application.

Configuration Requirements for the Transactions Sample Application

The University sample applications use an Oracle7 Transaction Manager Server (TMS). To use the Oracle7 database, you must include specific Oracle-provided files in the server application build process. For more information about building, configuring, and running the Transactions sample application, see Chapter 6, “Transactions Sample CORBA C++ XA Application.” For more information about the configurable settings in the UBBCONFIG file, see “Modifying the UBBCONFIG File to Accommodate Transactions” on page 11-2.

CHAPTER

4 Transactions in CORBA Client Applications

This topic includes the following sections:

- Overview of WLE CORBA Transactions
- Summary of the Development Process for Transactions
- Step 1: Use the Bootstrap Object to Obtain the TransactionCurrent Object
- Step 2: Use the TransactionCurrent Methods

This topic describes how to use transactions in CORBA C++, CORBA Java, and ActiveX client applications for the WebLogic Enterprise (WLE) software. Before you begin, you should read Chapter 1, “Introducing Transactions.”

For an example of how transactions are implemented in working client applications, see Chapter 6, “Transactions Sample CORBA C++ XA Application.” For an overview of the TransactionCurrent object, see “Client Application Development Concepts” in *Creating CORBA Client Applications*.

Overview of WLE CORBA Transactions

Client applications use transaction processing to ensure that data remains correct, consistent, and persistent. The transactions in the WLE software allow client applications to begin and terminate transactions and to get the status of transactions. The WLE software uses transactions as defined in the **CORBAservices Object Transaction Service**, with extensions for ease of use.

Transactions are defined on **interfaces**. The application designer decides which interfaces within a WLE client/server application will handle transactions. Transaction policies are defined in the Implementation Configuration File (ICF) for C++ server applications, or in the Server Description file (XML) for Java server applications. Generally, the ICF file or the Server Description file for the available interfaces is provided to the client programmer by the application designer.

If you prefer, you can use the Transaction application programming interface (API) defined in the `javax.transaction` package that is shipped with the WLE (Java) software.

Summary of the Development Process for Transactions

To add transactions to a client application, complete the following steps:

- Step 1: Use the Bootstrap Object to Obtain the TransactionCurrent Object
- Step 2: Use the TransactionCurrent Methods

The rest of this topic describes these steps using portions of the client applications in the Transactions University sample application. For information about the Transactions University sample application, see Chapter 6, “Transactions Sample CORBA C++ XA Application.”

The Transactions University sample application is located in the following directory on the WLE software kit:

- For Microsoft Windows NT systems:
  `drive:\wledir\samples\corba\university\transactions`
- For UNIX systems:
  `drive:/wledir/samples/corba/university/transactions`

Step 1: Use the Bootstrap Object to Obtain the TransactionCurrent Object

Use the `Bootstrap` object to obtain an object reference to the `TransactionCurrent` object for the specified WLE domain. For more information about the `TransactionCurrent` object, see “Client Application Development Concepts” in *Creating CORBA Client Applications*.

The following C++, Java, and Visual Basic examples illustrate how the `Bootstrap` object is used to return the `TransactionCurrent` object.

C++ Example

```
CORBA::Object_var var_transaction_current_oref =
     Bootstrap.resolve_initial_references(“TransactionCurrent”);
CosTransactions::Current_var transaction_current_oref=
     CosTransactions::Current::_narrow(
              var_transaction_current_oref.in());
```

Java Example

```
org.omg.CORBA.Object transCurObj =
     gBootstrapObjRef.resolve_initial_references(
                         “TransactionCurrent”);
org.omg.CosTransactions.Current gTransCur=
     org.omg.CosTransactions.CurrentHelper.narrow(transCurObj);
```

Visual Basic Example

```
Set objTransactionCurrent =
          objBootstrap.CreateObject(“Tobj.TransactionCurrent”)
```

Step 2: Use the TransactionCurrent Methods

The `TransactionCurrent` object has **method**s that allow a client application to manage transactions. These methods can be used to begin and end transactions and to obtain information about the current transaction.

Note: Alternatively, a CORBA Java client could use the `UserTransaction` object instead.

Table 4-1 describes the methods in the `TransactionCurrent` object:

Table 4-1 `TransactionCurrent` Methods

Method	Description
`begin`	Creates a new transaction. Future operations take place within the scope of this transaction. When a client application begins a transaction, the default transaction timeout is 300 seconds. You can change this default, using the `set_timeout` method.
`commit`	Ends the transaction successfully. Indicates that all operations on this client application have completed successfully.
`rollback`	Forces the transaction to roll back.
`rollback_only`	Marks the transaction so that the only possible action is to roll back. Generally, this method is used only in server applications.
`suspend`	Suspends participation in the current transaction. This method returns an object that identifies the transaction and allows the client application to resume the transaction later.

Table 4-1 `TransactionCurrent` Methods (Continued)

Method	Description
`resume`	Resumes participation in the specified transaction.
`get_status`	Returns the status of a transaction with a client application.
`get_transaction_name`	Returns a printable string describing the transaction.
`set_timeout`	Modifies the timeout period associated with transactions. The default transaction timeout value is 300 seconds. If a transaction is automatically started instead of explicitly started with the `begin` method, the timeout value is determined by the value of the `TRANTIME` parameter in the `UBBCONFIG` file. For more information about setting the `TRANTIME` parameter, see Chapter 11, "Administering Transactions."
`get_control`	Returns a control object that represents the transaction.

A basic transaction works in the following way:

1. A client application begins a transaction using the `Tobj::TransactionCurrent::begin` method. This method does not return a value.

2. The operations on the CORBA interface 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 communications failure), the exception can be caught and the transaction can be rolled back.

3. Use the `Tobj::TransactionCurrent::commit` method to commit the current transaction. 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.

   The association between the transaction and the client application ends when the client application calls the `Tobj::TransactionCurrent:commit` method or the `Tobj::TransactionCurrent:rollback` method.The following C++, Java, and Visual Basic examples illustrate using a transaction to encapsulate the operation of a student registering for a class:

C++ Example

```
//Begin the transaction
transaction_current_oref->begin();
try {
//Perform the operation inside the transaction
     pointer_Registar_ref->register_for_courses(student_id, course_number_list);
     ...
//If operation executes with no errors, commit the transaction:
     CORBA::Boolean report_heuristics = CORBA_TRUE;
     transaction_current_ref->commit(report_heuristics);
}
catch (CORBA::Exception &) {
//If the operation has problems executing, rollback the
//transaction. Then throw the original exception again.
//If the rollback fails,ignore the exception and throw the
//original exception again.
try {
     transaction_current_ref->rollback();
}
catch (CORBA::Exception &) {
            TP::userlog("rollback failed");
}
throw;
}
```

Java Example

```
try{
   gTransCur.begin();
   //Perform the operation inside the transaction
   not_registered =
      gRegistrarObjRef.register_for_courses(student_id,selected_course_numbers);


   if (not_registered != null)

     //If operation executes with no errors, commit the transaction
     boolean report_heuristics = true;
     gTransCur.commit(report_heuristics);

   } else gTransCur.rollback();
} catch(org.omg.CosTransactions.NoTransaction nte) {
    System.err.println(“NoTransaction: “ + nte);
    System.exit(1);
} catch(org.omg.CosTransactions.SubtransactionsUnavailable e) {
    System.err.println(“Subtransactions Unavailable: “ + e);
    System.exit(1);
} catch(org.omg.CosTransactions.HeuristicHazard e) {
    System.err.println(“HeuristicHazard: “ + e);
    System.exit(1);
} catch(org.omg.CosTransactions.HeuristicMixed e) {
    System.err.println(“HeuristicMixed: “ + e);
    System.exit(1);
}
```

Visual Basic Example

```
' Begin the transaction
'
objTransactionCurrent.begin
'
' Try to register for courses
'
NotRegisteredList = objRegistrar.register_for_courses(mStudentID,
     CourseList, exception)
'
If exception.EX_majorCode = NO_EXCEPTION then
     ' Request succeeded, commit the transaction
     '
     Dim report_heuristics As Boolean
     report_heuristics = True
     objTransactionCurrent.commit report_heuristics
Else
     ' Request failed, Roll back the transaction
     '
     objTransactionCurrent.rollback
          MsgBox "Transaction Rolled Back"
End If
```

CHAPTER

5 Transactions Sample CORBA Java JDBC Application

This topic includes the following sections:

- How the JDBC Bankapp Sample Application Works
- Development Process for the JDBC Bankapp Sample Application
- Setting Up the Database for the JDBC Bankapp Sample Application
- Building the JDBC Bankapp Sample Application
- Compiling the Client and Server Applications
- Initializing the Database
- Starting the Server Application in the JDBC Bankapp Sample Application
- Files Generated by the JDBC Bankapp Sample Application
- Starting the ATM Client Application in the JDBC Bankapp Sample Application
- Stopping the JDBC Bankapp Sample Application
- Using the ATM Client Application

Refer to the `Readme.txt` file in the `\WLEdir\samples\corba\bankapp_java\JDBC` directory for troubleshooting information and for the latest information about using the JDBC Bankapp sample application.

How the JDBC Bankapp Sample Application Works

The JDBC Bankapp sample application implements an automatic teller machine (ATM) interface and uses Java Database Connectivity (JDBC) to access a database that stores account and customer information. This topic includes the following sections:

- Java Server Objects
- Application Workflow
- JDBC Connection Pooling

Java Server Objects

The JDBC Bankapp sample application consists of a Java server application that contains the objects listed in Table 5-1.

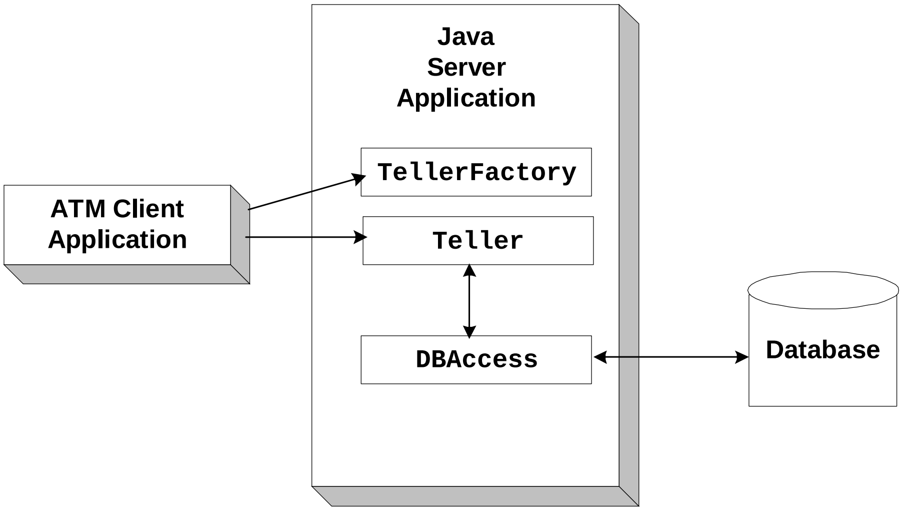
Table 5-1 Objects in the Java Server Application of the JDBC Bankapp

Object	Description
`TellerFactory`	The `TellerFactory` object creates the object references to the `Teller` object.
`Teller`	The `Teller` object receives and processes requests for banking operations from the ATM client application.
`DBAccesss`	The `DBAccess` object receives and processes requests from the `Teller` object to the database.

Application Workflow

Figure 5-1 illustrates how the JDBC Bankapp sample application works.

Figure 5-1 The JDBC Bankapp Sample Application



JDBC Connection Pooling

The JDBC Bankapp sample application demonstrates how to use JDBC database connection pooling running in a multithreaded server application. In the JDBC Bankapp sample application, WLE creates and initializes a pool of database connections that the sample application uses. All `DBAccess` objects share this pool. For more information about JDBC connection pools, see “Using JDBC Connection Pooling” in *Using the JDBC Drivers*.

A minimum number of database connections is established when the server is initialized. The number of connections is increased on demand. When a worker thread receives a request for a `DBAccess` object, the corresponding `DBAccess` method gets an available database connection from the pool. When the call to the `DBAccess` method completes, the database connection is returned to the pool. If there is no database connection available and the maximum number of database connections has been established, the worker thread waits until a database connection becomes available.

Development Process for the JDBC Bankapp Sample Application

This topic includes the following sections:

- Object Management Group (OMG) Interface Definition Language (IDL)
- Client Application
- Server Application
- Server Description File (BankApp.xml)
- UBBCONFIG File

This topic describes the development process for the JDBC Bankapp sample application.

Note: The steps in this topic have been done for you and are included in the JDBC Bankapp sample application.

Object Management Group (OMG) Interface Definition Language (IDL)

Table 5-2 lists the CORBA interfaces defined in the OMG IDL for the JDBC Bankapp sample application.

Table 5-2 CORBA Interfaces Defined in the JDBC Bankapp OMG IDL

Interface	Description	Methods
`TellerFactory`	Creates object references to the `Teller` object	`create_Teller()`

Table 5-2 CORBA Interfaces Defined in the JDBC Bankapp OMG IDL (Continued)

`Teller`	Performs banking operations	`verify_pin_number()`
		`deposit()`
		`withdraw()`
		`inquiry()`
		`transfer()`
		`report()`

`DBAccess`	Accesses the Oracle database on behalf of the `Teller` object	`get_valid_accounts()`
		`read_account()`
		`update_account()`
		`transfer_funds()`

BankApp.idl File

Listing 5-1 shows the `BankApp.idl` file that defines the `TellerFactory` and `Teller` interfaces in the JDBC Bankapp sample application. A copy of this file is included in the directory of the JDBC Bankapp sample application.

Listing 5-1 OMG IDL Code for the `TellerFactory` and `Teller` Interfaces

```
#pragma prefix "beasys.com"
#pragma javaPackage "com.beasys.samples"

#include "Bank.idl"

module BankApp{
               exception IOException {};
               exception TellerInsufficentFunds();

               struct     BalanceAmounts{
                    float fromAccount;
                    float toAccount;
               };

               struct     TellerActivity {
                    long  totalRequests;
                    long  totalSuccesses;
                    long  totalFailures;
                    float currentBalance;
               };
               //Process Object
               interface Teller  {
                     void verify_pin_number(in short  pinNo,
                                       out Bank::CustAccounts accounts)
                               raises(Bank::PinNumberNotFound, IOException);
                     float deposit(in long accountNo, in float amount)
                               raises(Bank::AccountRecordNotFound,IOException);
                     float withdraw(in long accountNo, in float amount)
                               raises(Bank::AccountRecordNotFound,
                                       Bank::InsufficentFunds,
                                       IOException, TellerInsufficientFunds);
                     float inquiry(in long accountNo)
                               raises(Bank::AccountRecordNotFound, IOException);
                     void transfer(in long fromAccountNo,
                                   in long toAccountNo,in float amount,
                                   out BalanceAmounts balAmounts)
                               raises(Bank::AccountRecordNotFound,
                                      Bank::InsufficientFunds,
                                      IOException);
                     void report(out TellerActivity tellerData)
                               raises(IOException);
               };

               interface TellerFactory{
                        Teller createTeller(in string tellerName);
               };

};
```

BankDB.idl File

Listing 5-2 shows the `BankDB.idl` file that defines the `DBAccess` interface in the JDBC Bankapp sample application. A copy of this file is included in the directory of the JDBC Bankapp sample application.

Listing 5-2 OMG IDL Code for the `DBAccess` Interface

```
#pragma prefix "beasys.com"
#pragma javaPackage "com.beasys.samples"

#include "Bank.idl"

module BankDB{
             struct AccountData{
                  long accountID;
```

```
                float balance;
           };

           interface DBAccess{
                void get_valid_accounts(in short, pinNo,
                                        out Bank::CustAccounts accounts)
                        raises(Bank::DatabaseException,
                               Bank::PinNumberNotFound);
                void read_account(inout AccountData data)
                        raises(Bank::DatabaseException,
                               Bank::AccountRecordNotFound);
                void update_account(inout AccountData data)
                        raises(Bank::DatabaseException,
                               Bank::AccountRecordNotFound,
                               Bank::InsufficientFunds);
                void transfer_funds(in float_amount,
                                        inout AccountData fromAcct,
                                        inout AccountData toAcct,
                        raises(Bank::DatabaseException,
                                   Bank::AccountRecordNotFound,
                                   Bank::InsufficientFunds);
           };

};
```

Bank.idl File

Listing 5-3 shows the `Bank.idl` file that defines common exceptions and structures. It is included by both `BankApp.idl` and `BankDB.idl`. A copy of this file is included in the directory of the JDBC Bankapp sample application.

Listing 5-3 OMG IDL Code for the Exceptions and Structures in JDBC Bankapp

```
#pragma prefix "beasys.com"
#pragma javaPackage "com.beasys.samples"

module Bank{

            exception DataBaseException {};
            exception PinNumberNotFound ();
            exception AccountRecordNotFound ();
            exception InsufficientFunds ();

            struct CustAccounts{
                  long checkingAccountID;
```

```
                long savingsAccountID;
        };

};
```

Client Application

During the development of the client application, you would write Java code that performs the following tasks:

- Initializes the ORB.
- Uses the Bootstrap environmental object to establish communication with the WebLogic Enterprise (WLE) domain.
- Resolves initial references to the `FactoryFinder` environmental object.
- Uses a factory to get an object reference for the `Teller` object.
- Invokes the `verify_pin_number`, `deposit`, `withdraw`, `inquiry`, `transfer`, and `report` methods on the `Teller` object.

A Java client application, referred to as the ATM client application, is included in the JDBC Bankapp sample application. For more information about writing Java client applications, see Chapter 4, “Transactions in CORBA Client Applications.”

Server Application

During the development of the server application, you would write the following:

- The Server object, which initializes the server application in the JDBC Bankapp sample application and registers a factory for the `Teller` object with the WLE domain. The Server object also obtains a reference to the JDBC connection pool from JNDI.
- The implementations for the methods of the `Teller` and `DBAccess` objects.

  The implementations for the `Teller` object include invoking operations on the `DBAccess` object.

Because the `Teller` object has durable state (for example, ATM statistics) that is stored in an external source (a flat file), the method implementations must also include the `activate_object` and `deactivate_object` methods to ensure the `Teller` object is initialized with its state.

The JDBC Bankapp server application is configured to be multithreaded. Writing a multithreaded WLE Java server application is the same as writing a single-threaded Java server application; you cannot establish multiple threads programmatically in your object implementations. Instead, you establish the number of threads for a Java server application in the `UBBCONFIG` file. For information about writing Java server applications and using threads in Java server applications, see Chapter 3, "Transactions in CORBA Server Applications."

Server Description File (BankApp.xml)

During development, you create a Server Description File (`BankApp.xml`) that defines the activation and transaction policies for the `TellerFactory`, `Teller`, and `DBAccess` interfaces. Table 5-3 shows the activation and transaction policies for the JDBC Bankapp sample application.

Table 5-3 Activation and Transaction Policies for JDBC Bankapp

Interface	Activation Policy	Transaction Policy
`TellerFactory`	`Process`	`Never`
`Teller`	`Method`	`Never`
`DBAccess`	`Method`	`Never`

A Server Description File for the JDBC Bankapp sample application is provided. For information about creating Server Description Files and defining activation and transaction policies on objects, see *Creating CORBA Java Server Applications.*

UBBCONFIG File

When using the WLE software, the server application is represented by a Java Archive (JAR). The JAR must be loaded into the Java Virtual Machine (JVM) to be executed. The JVM must execute in a WLE server application to be integrated in an WLE application. By default, the server application that loads the JVM is called `JavaServer`. You include the options to start `JavaServer` in the `Servers` section of the application's `UBBCONFIG` file. For information about starting the `JavaServer` and defining parameters in the `UBBCONFIG` file, see "Creating the Configuration File" in the *Administration Guide*.

Enabling Multithreaded Support

If your Java server application is multithreaded, you can establish the number of threads by using the command-line option (`CLOPT`) `-M` in the `SERVERS` section of the `UBBCONFIG` file. In Listing 5-4, the `-M 100` option enables multithreading for the `JavaServer` and specifies 100 as the maximum number of worker threads that a particular instance of `JavaServer` can support. The largest number that you can specify is 500.

Listing 5-4 Enabling Multithreaded Support in UBBCONFIG

```
JavaServer
   SRVGRP = BANK_GROUP1
   SRVID = 2
   SRVTYPE = JAVA
   CLOPT = "-A -- -M 100 Bankapp.jar TellerFactory_1 bank_pool"
   RESTART = N
```

Notes: The `SRVTYPE=JAVA` line is required when using JDBC connection pooling.

The information for the `CLOPT` parameter needs to be entered on one line.

You also need to set the `MAXACCESSERS` parameter in the `RESOURCES` section of the `UBBCONFIG` file to account for the number of worker threads that each server application is configured to run. The `MAXACCESSERS` parameter specifies the number of processes that can attach to a WLE application.

Setting Up the Connection Pool

For the JDBC Bankapp sample application, you need to include the name of the connection pool on the command-line option (`CLOPT`) in the `SERVERS` section of the `UBBCONFIG` file, as shown in Listing 5-5.

Listing 5-5 Specifying the Connection Pool Name (`bank_pool`) in UBBCONFIG

```
CLOPT = "-A -- -M 100 Bankapp.jar TellerFactory_1 bank_pool"
```

Note: The information for the `CLOPT` parameter needs to be entered on one line.

In addition, you need to include the following information on the `JDBCCONNPOOLS` section of the `UBBCONFIG` file:

- The server group and server ID of the server.
- The class name of JDBC driver:
  - `JdbcOracle734` for the jdbcKona/Oracle driver
  - `JdbcMSSQL4` for the jdbcKona/MSSQLServer driver
- Either the JDBC URL for the Oracle database, or the name of the machine where the Microsoft SQL Server database is installed
- Optionally, either the user id and password for the Oracle database, or the user name and password you defined for the master instance of the Microsoft SQL Server database
- The initial number of database connections in the pool
- The maximum number of database connections in the pool

Listing 5-6 provides an example of the `JDBCCONNPOOLS` section in the `UBBCONFIG`.

Listing 5-6 Specifying `JDBCCONNPOOLS` Information in `UBBCONFIG`

```
JDBCCONNPOOLS
   bank_pool
      SRVGRP          = BANK_GROUP1
      SRVID           = 2
      DRIVER          = "weblogic.jdbc20.oci815.Driver"
```

```
URL             = "jdbc:weblogic:oracle:Beq-local"
PROPS           = "user=scott;password=tiger;server=Beq-Local"
ENABLEXA        = N
INITCAPACITY    = 2
MAXCAPACITY     = 10
CAPACITYINCR    = 1
CREATEONSTARTUP = Y
```

For more information about configuring JDBC connection pools, see "Using JDBC Connection Pooling" in *Using the JDBC Drivers*.

Setting Up the Database for the JDBC Bankapp Sample Application

The JDBC Bankapp sample application uses a database to store all the bank data. You can use either the Oracle or the Microsoft SQL Server database with the JDBC Bankapp sample application.

Before you can build and run the JDBC Bankapp sample application, you need to follow the steps in the product documentation to install the desired database.

The jdbcKona/Oracle and jdbcKona/MSSQLServer4 drivers are installed as part of the WLE installation. For more information about the jdbcKona drivers, see *Using the JDBC Drivers*.

Note: The jdbcKona/Oracle driver supports Oracle Version 7.3.4 and Oracle8i (for Solaris and Windows NT) and versions 8.04 and 8i (for HP-UX). By default, this sample application supports Oracle Version 7.3.4 on NT/Solaris and Version 8.0.4 on HP. You can use a different Oracle version by specifying command line parameters, as described in "Step 4: Run the setupJ Command" on page 5-20.

Setting Up an Oracle Database

If you are using Oracle as the database for the JDBC Bankapp sample application, you need to install the following software:

- Visual C++ Version 5.0 with Service Pack for Visual Studio (Windows NT only)
- Sun SparcWorks Compiler 4.2 (Solaris only)
- Oracle Version 7.3.4

When using the Oracle database, you use the default database created by the Oracle installation program. You need the connection string you defined for the Oracle database and the default user id and password. See the Oracle product documentation for details about obtaining this information.

Setting Up a Microsoft SQL Server Database

If you are using the Microsoft SQL Server as the database for the JDBC Bankapp sample application, you need to install the following software:

- Visual C++ Version 5.0 with Service Pack for Visual Studio (Windows NT only)
- Sun SparcWorks Compiler 4.2 (Solaris only)
- Microsoft SQL Server Version 7.0

When using the Microsoft SQL Server database, you use the `master` database instance. You need the name of the machine where the Microsoft SQL Server database is installed and the user name and password you defined for the `master` instance of the Microsoft SQL Server database. See the Microsoft product documentation for details about obtaining this information.

Building the JDBC Bankapp Sample Application

This topic describes the following steps, which are required to build the JDBC Bankapp sample application:

- Step 1: Copy the Files for the JDBC Bankapp Sample Application into a Work Directory.

- Step 2: Change the Protection Attribute on the Files for the JDBC Bankapp Sample Application.
- Step 3: Verify the Settings of the Environment Variables.
- Step 4: Run the setupJ Command.
- Step 5: Load the UBBCONFIG File.

Step 1: Copy the Files for the JDBC Bankapp Sample Application into a Work Directory

You need to copy the files for the JDBC Bankapp sample application into a work directory on your local machine.

Source File Directories

The files for the JDBC Bankapp sample application are located in the following directories:

Windows NT

```
drive:\WLEdir\samples\corba\bankapp_java\JDBC
drive:\WLEdir\samples\corba\bankapp_java\client
drive:\WLEdir\samples\corba\bankapp_java\shared
```

UNIX

```
/usr/local/WLEdir/samples/corba/bankapp_java/JDBC
/usr/local/WLEdir/samples/corba/bankapp_java/client
/usr/local/WLEdir/samples/corba/bankapp_java/shared
```

Table 5-4 describes the contents of these directories.

Table 5-4 Source File Directories for the JDBC Bankapp Sample Application

Directory	Description
`JDBC`	Source files and commands needed to build and run the JDBC Bankapp sample application.

Table 5-4 Source File Directories for the JDBC Bankapp Sample Application

Directory	Description
`client`	Files for the ATM client application. The `images` subdirectory contains `.gif` files used by the graphical user interface in the ATM client application.
`shared`	Common files for the JDBC Bankapp and XA Bankapp sample applications.

Copying Source Files to the Work Directory

You need to manually copy only the files in the `\JDBC` directory. The other sample application files are automatically copied from the `\client` and `\shared` directories when you execute the `setupJ` command. For example:

Windows NT

```
prompt> cd c:\mysamples\bankapp_java\JDBC
```

```
prompt> copy c:\WLEdir\samples\corba\bankapp_java\JDBC\*
```

UNIX

```
ksh prompt> cd /usr/mysamples/bankapp_java/JDBC/*
```

```
ksh prompt> cp $TUXDIR/samples/bankapp_java/JDBC/* .
```

Note: You cannot run the JDBC Bankapp sample application in the same work directory as the XA Bankapp sample application, because some of the files for the JDBC Bankapp sample application have the same name as files for the XA Bankapp sample application.

Source Files Used to Build the JDBC Bankapp Sample Application

Table 5-5 lists the files used to build and run the JDBC Bankapp sample application.

Table 5-5 Files Included in the JDBC Bankapp Sample Application

File	Description
`Bank.idl`	OMG IDL code that declares common structures and extensions for the JDBC Bankapp sample application.
`BankApp.idl`	OMG IDL code that declares the `TellerFactory` and `Teller` interfaces.
`BankDB.idl`	OMG IDL code that declares the `DBAccess` interface.
`TellerFactoryImpl.java`	Java source code that implements the `createTeller` method. This file is in the `com.beasys.samples` package. It is automatically moved to the `com/beasys/samples` directory by the `setupJ` command.
`TellerImpl.java`	Java source code that implements the `verify`, `deposit`, `withdraw`, `inquiry`, `transfer`, and `report` methods. This file is in the `com.beasys.samples` package. It is automatically moved to the `com/beasys/samples` directory by the `setupJ` command.
`BankAppServerImpl.java`	Java source code that overrides the `Server.initialize` and `Server.release` methods.
`DBAccessImpl.java`	Java source code that implements the `get_valid_accounts`, `read_account`, `update_account`, and `transfer` methods. This file is in the `com.beasys.samples` package. It is automatically moved to the `com/beasys/samples` directory by the `setupJ` command.
`Atm.java`	Java source code for the ATM client application.
`BankStats.java`	Contains methods to initialize, read from, and write to the flat file that contains the ATM statistics.

Table 5-5 Files Included in the JDBC Bankapp Sample Application (Continued)

File	Description
`BankApp.xml`	Server Description File used to associate activation and transaction policy values with CORBA interfaces.
`InitDB.java`	Java program that initializes the database and ensures that JDBC is working properly.
`setupJ.cmd`	Windows NT batch file that builds and runs the JDBC Bankapp sample application.
`setupJ.ksh`	UNIX Korn shell script that builds and runs the JDBC Bankapp sample application.
`makefileJ.mk`	Make file for the JDBC Bankapp sample application on the UNIX operating system. The UNIX `make` command needs to be in the path of your machine.
`makefileJ.nt`	Make file for the JDBC Bankapp sample application on the Windows NT operating system. The Windows NT `nmake` command needs to be in the path of your machine.
`Readme.txt`	File that provides the latest information about building and running the JDBC Bankapp sample application.

Step 2: Change the Protection Attribute on the Files for the JDBC Bankapp Sample Application

During the installation of the WLE software, the files for the JDBC Bankapp sample application are marked read-only. Before you can edit or build the files in the JDBC Bankapp sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>/bin/ksh
ksh prompt>chmod u+w /workdirectory/*.*
```

Step 3: Verify the Settings of the Environment Variables

Before building and running the JDBC Bankapp sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Environment Variables

Table 5-6 lists the environment variables required to run the JDBC Bankapp sample application.

Table 5-6 Required Environment Variables for the JDBC Bankapp Sample Application

Environment Variable	Description

`TUXDIR`	The directory path where you installed the WLE software. For example:
	Windows NT
	`TUXDIR=c:\WLEdir`
	UNIX
	`TUXDIR=/usr/local/WLEdir`

`JAVA_HOME`	The directory path where you installed the JDK software. For example:
	Windows NT
	`JAVA_HOME=c:\JDK1.2`
	UNIX
	`JAVA_HOME=/usr/local/JDK1.2`

Table 5-6 Required Environment Variables for the JDBC Bankapp Sample Application

Environment Variable	Description

`ORACLE_HOME`	The directory path where you installed the Oracle software. For example:
	Windows NT
	`ORACLE_HOME=d:\orant`
	UNIX
	`ORACLE_HOME=/usr/local/oracle`
	Note: This environment variable applies only if you are using the Oracle database on the Solaris operating system.

Verifying Settings

To verify that the information defined during installation is correct:

Windows NT

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.
4. Click the Environment tab.

   The Environment page appears.
5. Check the settings for `TUXDIR` and `JAVA_HOME`.

UNIX

```
ksh prompt>printenv TUXDIR

ksh prompt>printenv JAVA_HOME

ksh prompt>printenv ORACLE_HOME
```

Changing Settings

To change the settings:

Windows NT

1. On the Environment page in the System Properties window, click the environment variable you want to change, or enter the name of the environment variable in the Variable field.
2. In the Value field, enter the correct information for the environment variable.
3. Click OK to save the changes.

UNIX

```
ksh prompt>TUXDIR=directorypath; export TUXDIR
ksh prompt>JAVA_HOME=directorypath; export JAVA_HOME
ksh prompt>JAVA_HOME=directorypath; export ORACLE_HOME
```

Note: If you are running multiple WLE applications concurrently on the same machine, you also need to set the `IPCKEY` and `PORT` environment variables. See the `Readme.txt` file for information about how to set these environment variables.

Step 4: Run the setupJ Command

The `setupJ` command automates the following steps:

1. Copy the required files from the `\client` and `\shared` directories.
2. Set the `PATH`, `TOBJADDR`, `APPDIR`, `TUXCONFIG`, and `CLASSPATH` system environment variables.
3. Create the `UBBCONFIG` file (`ubb_jdbc`).
4. Create a `setenvJ.cmd` or `setenvJ.ksh` file that can be used to reset the system environment variables.

Syntax

The syntax for the `setupJ` command is:

```
prompt>setupJ DB_DRIVER DB_SERVER DB_USER DB_PASSWORD
```

where:

Parameter	Description
`DB_DRIVER`	Name of the database driver. Valid values include: ■ `jdbcOracle734` ■ `jdbcOracle804` ■ `jdbcOracle815` ■ `jdbcMSSQL4` Default values are `jdbcOracle734` (on Solaris or NT) or `jdbcOracle804` (on Hewlett-Packard).
`DB_SERVER`	Name of the machine where the database is installed. Default values are `Beq-Local` (on NT) or `null` (on Solaris or Hewlett-Packard).
`DB_USER`	User name defined for the database. Default value is `scott`.
`DB_PASSWORD`	Password defined for the database. Default value is `tiger`.

Note: `SetupJ` uses default values unless you explicitly specify arguments. For example, to use Microsoft SQL Server, you must specify all command line parameters.

Command

To enter the `setupJ` command, complete the following steps:

Windows NT

```
prompt>cd c:\mysamples\bankapp_java\JDBC

prompt>setupJ jdbcOracle815 Beq-Local scott tiger
```

UNIX

```
prompt>/bin/ksh

prompt>cd /usr/mysamples/bankapp_java/JDBC
prompt>. ./setupJ.ksh jdbcOracle815 null scott tiger
```

Step 5: Load the UBBCONFIG File

To load the `UBBCONFIG` file, use the following command:

```
prompt>tmloadcf -y ubb_jdbc
```

Compiling the Client and Server Applications

The directory for the JDBC Bankapp sample application contains a make file that builds the client and server sample applications. During development, you use the `buildjavaserver` command to build the server application, and your Java product's development commands to build the client application. However, for the JDBC Bankapp sample application, these steps are included in the make file.

To build the client and server applications in the JDBC Bankapp sample application, use the following commands:

Windows NT

```
prompt>nmake -f makefileJ.nt
```

UNIX

```
prompt>make -f makefileJ.mk
```

Initializing the Database

This topic includes the following sections:

- Initializing an Oracle Database

- Initializing a Microsoft SQL Server Database

Initializing an Oracle Database

To initialize an Oracle database using the default arguments, enter the following command:

```
prompt>java InitDB
```

To initialize the Oracle database with user-defined attributes, enter the following command:

```
prompt>java InitDB driver_name connect_string username password
```

where

Parameter	Description
driver_name	Name of the database driver. Valid values include: ■ `jdbcOracle734` ■ `jdbcOracle804` ■ `jdbcOracle815` ■ `jdbcMSSQL4` Default values are `jdbcOracle734` (on Solaris or NT) or `jdbcOracle804` (on Hewlett-Packard).
connect_string	Connection string for the instance of the Oracle database being used with the JDBC Bankapp sample application. Default values are `Beq-Local` (on NT) or `null` (on Solaris or Hewlett-Packard).
username	User name for the Oracle database. Default value is `scott`.
password	Password for the Oracle database. Default value is `tiger`.

Initializing a Microsoft SQL Server Database

To initialize a Microsoft SQL Server database, enter the following command:

```
prompt>java InitDB JdbcMSSQL4 db_server username password
```

where:

Parameter	**Description**
`jdbcMSSQL4`	Name of the database driver for Microsoft SQL Server. This is the only valid value.
`db_server`	Name of the machine on which the Microsoft SQL Server database is installed.
`username`	User name for the `master` instance of the Microsoft SQL Server database.
`password`	Password for the `master` instance of the Microsoft SQL Server database.

Starting the Server Application in the JDBC Bankapp Sample Application

Start the server application in the JDBC Bankapp sample application by entering the following command:

```
prompt>tmboot -y
```

The `tmboot` command starts the application processes listed in Table 5-7.

Table 5-7 Application Processes Started by `tmboot` Command

Process	**Description**
`TMSYSEVT`	BEA TUXEDO system event broker.

Table 5-7 Application Processes Started by `tmboot` Command

Process	Description
`TMFFNAME`	Three `TMFFNAME` server processes are started: ■ The `TMFFNAME` server process started with the `-N` and `-M` options is the master `NameManager` service. The `NameManager` service maintains a mapping of the application-supplied names to object references. ■ The `TMFFNAME` server process started with the `-N` option is the slave `NameManager` service. ■ The `TMFFNAME` server process started with the `-F` option contains the `FactoryFinder` object.
`JavaServer`	Server process that implements the `TellerFactory`, `Teller,` and `DBAccess` interfaces.
`ISL`	IIOP Listener process.

Files Generated by the JDBC Bankapp Sample Application

Table 5-8 lists the files generated by the JDBC Bankapp sample application.

Table 5-8 Files Generated by the JDBC Bankapp Sample Application

File	Description
`ubb_jdbc`	The `UBBCONFIG` file for the JDBC Bankapp sample application. This file is generated by the `setupJ` command.
`setenvJ.cmd` and `setenvJ.ksh`	Contains the commands to set the environment variables needed to build and run the JDBC Bankapp sample application. `setenvJ.cmd` is the Windows NT version and `setenvJ.ksh` is the UNIX Korn shell version of the file.
`tuxconfig`	A binary version of the `UBBCONFIG` file. Generated by the `tmloadcf` command.

Table 5-8 Files Generated by the JDBC Bankapp Sample Application (Continued)

File	Description
`ULOG.<date>`	A log file that contains messages generated by the `tmboot` command. The log file also contains messages generated by the server applications and by the `tmshutdown` command.
`.adm/.keybd`	A file that contains the security encryption key database. The subdirectory is created by the `tmloadcf` command.

`Atm$1.class`
`Atm.class`
`AtmAppletStub.class`
`AtmArrow.class`
`AtmButton.class`
`AtmCenterTextCanvas.class`
`AtmClock.class`
`AtmScreen.class`
`AtmServices.class`
`AtmStatus.class`
	Used by the Java client application. Created when the `Atm.java` file is compiled.

`InitDB.class`	Initializes the database used by the JDBC Bankapp sample application. Created when `InitDB.java` is compiled.

`AccountRecordNotFound.java`
`AccountRecordNotFoundHelper.java`
`AccountRecordNotFoundHolder.java`
`CustAccounts.java`
`CustAccountsHelper.java`
`CustAccountsHolder.java`
`DataBaseException.java`
`DataBaseExceptionHelper.java`
`DataBaseExceptionHolder.java`
`InsufficientFunds.java`
`InsufficientFundsHelper.java`
`InsufficientFundsHolder.java`
`PinNumberNotFound.java`
`PinNumberNotFoundHelper.java`
`PinNumberNotFoundHolder.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `Bank.idl` file. These files are created in the `com/beasys/samples/Bank` directory.

Table 5-8 Files Generated by the JDBC Bankapp Sample Application (Continued)

File	Description

`BalanceAmounts.java`
`BalanceAmountsHelper.java`
`BalanceAmountsHolder.java`
`IOException.java`
`IOExceptionHelper.java`
`IOExceptionHolder.java`
`Teller.java`
`TellerActivity.java`
`TellerActivityHelper.java`
`TellerActivityHolder.java`
`TellerFactory.java`
`TellerFactoryHelper.java`
`TellerFactoryHolder.java`
`TellerInsufficientFunds.java`
`TellerInsufficientFundsHelper.java`
`TellerInsufficientFundsHolder.java`
`_TellerFactoryImplBase.java`
`_TellerFactoryStub.java`
`_TellerImplBase.java`
`_TellerStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankApp.idl` file. These files are created in the `com/beasys/samples/BankApp` subdirectory.

`AccountData.java`
`AccountDataHelper.java`
`AccountDataHolder.java`
`DBAccessHelper.java`
`DBAccessHolder.java`
`_DBAccessImplBase.java`
`_DBAccessStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankDB.idl` file. These files are created in the `com/beasys/samples/BankDB` subdirectory.

`Bankapp.ser` `Bankapp.jar`	The Server Descriptor File and Server Java Archive file generated by the `buildjavaserver` command in the make file.
`stderr`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.err.println` method sends the output to the `stderr` file instead of to the `ULOG` file.

Table 5-8 Files Generated by the JDBC Bankapp Sample Application (Continued)

File	Description
`stdout`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.out.println` method sends the output to the `stdout` file instead of to the `ULOG` file.
`tmsysevt.dat`	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command.

Starting the ATM Client Application in the JDBC Bankapp Sample Application

Start the ATM client application by entering the following command, which sets the Java CLASSPATH to include the current directory and the client JAR file (`m3envobj.jar`).

Note: You can specify the full WLE JAR file (`m3.jar`) instead of the client JAR file.

Windows NT

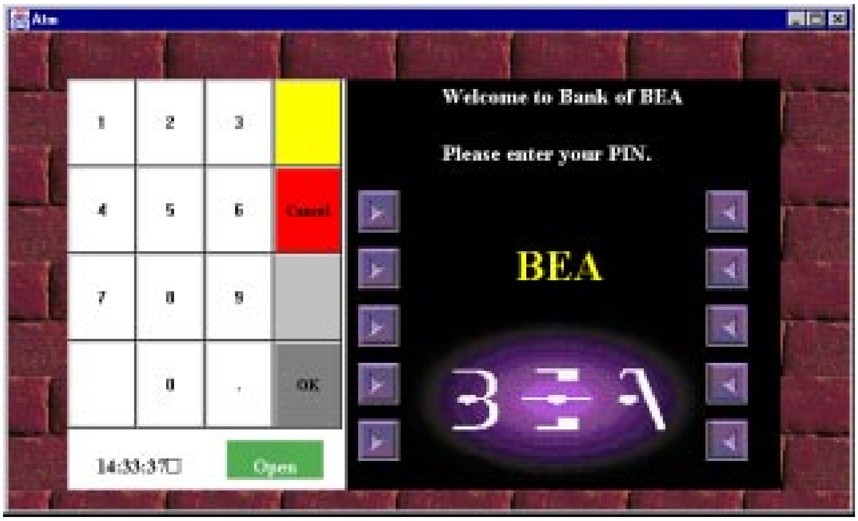
```
prompt>java -classpath .;%TUXDIR%\udataobj\java\jdk\m3envobj.jar
-DTOBJADDR=%TOBJADDR% Atm Teller1
```

UNIX

```
ksh prompt>java -classpath .:$TUXDIR/udataobj/java/jdk
/m3envobj.jar -DTOBJADDR=$TOBJADDR Atm Teller1
```

The GUI for the ATM client application appears. Figure 5-2 shows the GUI for the ATM client application.

Figure 5-2 GUI for ATM Client Application



Stopping the JDBC Bankapp Sample Application

Before using another sample application, enter the following commands to stop the JDBC Bankapp sample application and to remove unnecessary files from the work directory:

Windows NT

```
prompt>tmshutdown -y
prompt>nmake -f makefileJ.nt clean
```

UNIX

```
ksh prompt>tmshutdown -y
ksh prompt>make -f makefileJ.mk clean
```

Using the ATM Client Application

This topic includes the following sections:

- Available Banking Operations
- Available Statistics
- Keypad Functions
- Steps for Using the ATM Client Application

Available Banking Operations

In the ATM client application, a customer enters a personal identification number (PIN) and performs one of the following banking operations:

- Withdraws money from the account
- Deposits money in the account
- Inquires about the balance of the account
- Transfers money between checking and savings accounts

Available Statistics

One special PIN number (`999`) allows customers to receive statistics about the ATM machine. The following statistics are available:

- Total number of requests received by the ATM machine

  For example, an inquiry is one request, and a withdrawal is one request.
- Total number of successful requests
- Total number of failed requests

For example, when a customer attempts to withdraw more money than is in his account, the request fails.

- Total amount of cash remaining in the ATM machine

  The ATM machine starts with $10,000 and the amount decreases with each withdrawal request.

Keypad Functions

Use the keypad in the ATM client application to enter a PIN and amounts for deposit, transfer, and withdrawal. Table 5-9 describes the functions available on the keypad in the ATM client application.

Table 5-9 Keypad Functions in the ATM Client Application

Key	Function
Cancel	Use this key to cancel the current operation and exit the view.
OK	Use this key to accept the entered data. After you enter a PIN or an amount for deposit, transfer, or withdrawal, you need to click the OK button to have the action take effect.
Numerics (0 through 9)	Use these keys to enter your PIN and an amount for deposit, transfer, and withdrawal amounts.
Period (.)	Use this key to enter decimal amounts for deposit, transfer, and withdrawal.

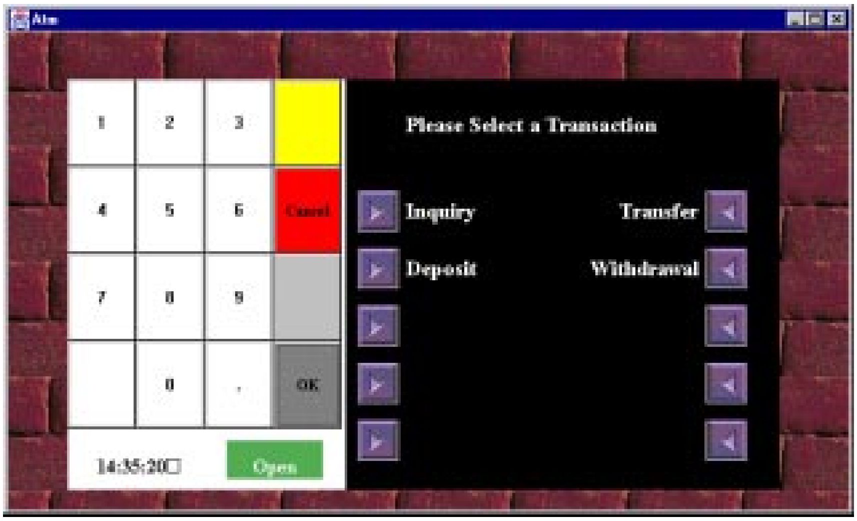
Steps for Using the ATM Client Application

To use the ATM client application in the JDBC Bankapp sample application:

1. Enter one of the following PINs: `100`, `110`, `120`, or `130`.
2. Click OK.

   The Operations view appears. Figure 5-3 shows the Operations view in the ATM client application.

Figure 5-3 Operations View in the ATM Client Application



From the Operations view, you can perform the follow banking operations:

- Inquiry
- Transfer
- Deposit
- Withdrawal

3. Click the desired banking operation.
4. Click either the Checking Acct or Savings Acct button.
5. Enter a dollar amount.
6. Click OK.

   An updated account balance appears.

   Note: After you click OK, you cannot cancel the operation. If you enter an amount and then select Cancel, the ATM client application cancels your operation and displays the previous screen.

7. Click OK.
8. Click Cancel to return to the main window of the ATM client application.

6 Transactions Sample CORBA C++ XA Application

This topic includes the following sections:

- How the Transactions Sample Application Works
- Development Process for the Transactions Sample Application
- Setting Up the Transactions Sample Application
- Compiling the Transactions Sample Application
- Running the Transactions Sample Application
- Using the Client Applications in the Transactions sample application

Refer to `Readme.txt` in the `\transactions` directory for troubleshooting information and the latest information about using the Transactions sample application.

This topic explains how to design and implement transactions in a WebLogic Enterprise (WLE) server application using the Transactions University sample application as an example. This topic also describes how the Transactions sample application works, and discusses the design considerations for implementing transactions in it. For additional general information about transactions, see "Integrating Transactions in a WLE Client and Server Application" on page 3-2.

How the Transactions Sample Application Works

The Transactions sample application uses transactions to encapsulate the task of a student registering for a set of courses. The transactional model used in this application is a combination of the conversational model and the model in which a single invocation makes multiple individual operations on a database.

In the Transactions sample application, students can register for classes. The operation of registering for courses is executed within the scope of a transaction. This topic includes the following sections:

- Application Steps
- Application Workflow

Application Steps

The Transactions sample application works in the following way:

1. Students submit a list of courses for which they want to be registered.
2. For each course in the list, the University server application checks whether:
   - The course exists in the database
   - The student is already registered for a course
   - The student exceeds the maximum number of credits the student can take
3. One of the following occurs:
   - If the course meets all the criteria, the University server application registers the student for the course.
   - If the course is not in the database or if the student is already registered for the course, the University server application adds the course to a list of registered courses for which the student could not be registered. After processing all the registration requests, the server application returns the list

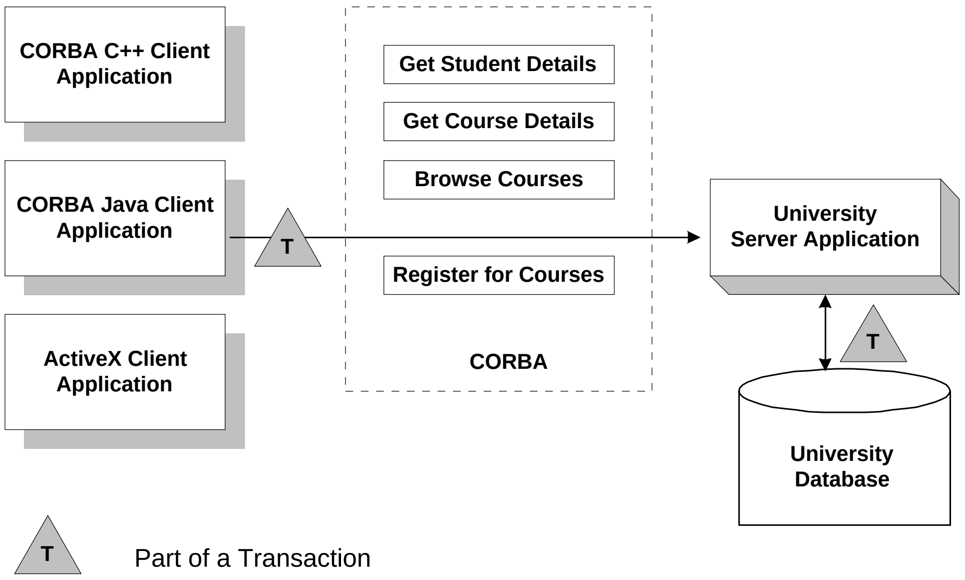
of courses for which registration failed. The client application prompts the student to either commit the transaction (thereby registering the student for the courses for which registration request succeeded) or to roll back the transaction (thus not registering the student for any of the courses).

- If the student exceeds the maximum number of credits the student can take, the University server application returns a `TooManyCredits` user exception to the client application. The client application provides a brief message explaining that the request was rejected. The client application then rolls back the transaction.

Application Workflow

Figure 6-1 illustrates how the Transactions sample application works.

Figure 6-1 The Transactions Sample Application



Development Process for the Transactions Sample Application

This topic includes the following sections:

- OMG IDL
- Client Application
- University Server Application
- UBBCONFIG File
- Implementation Configuration File (ICF)

This topic describes the steps used to add transactions to the Transactions sample application. These steps are in addition to the development process outlined in “The Basic Sample Application” in the *Guide to the University Sample Applications*.

Note: The steps in this section have been done for you and are included in the Transactions sample application.

OMG IDL

During the development process, you would define, in Object Management Group (OMG) Interface Definition Language (IDL), the `register_for_courses()` operation for the `Registrar`. The `register_for_courses()` operation has a parameter, `NotRegisteredList`, which returns to the client application the list of courses for which registration failed. If the value of `NotRegisteredList` is empty, the client application commits the transaction.

You also need to define the `TooManyCredits` user exception.

Client Application

During the development process, you would add the following to your client application:

- The `Bootstrap` environmental object to obtain a reference to the `TransactionCurrent` environmental object in the specified WLE domain.
- The operations of the `TransactionCurrent` environmental object to include a CORBA object in a transaction.
- A call to the `register_for_courses()` operation so that students can register for courses.

For information about using Transactions in client applications, see "Transactions in WLE CORBA Applications" on page 1-6.

University Server Application

During the development process, you would add the following to the University server application:

- Invocations to the `TP::open_xa_rm()` and `TP::close_xa_rm()` operations in the `Server::initialize()` and `Server::release()` operations of the Server object.
- A method implementation for the `register_for_courses()` operation.

For information about these tasks, see *Creating C++ Server Applications*.

UBBCONFIG File

During the development process, you need to specify the following information in the `UBBCONFIG` file:

- A server group that includes both the University server application and the server application that manages the database. This server group needs to be specified as transactional.

- The `OPENINFO` parameter defined according to the `XA` parameter for the Oracle database. The `XA` parameter for the Oracle database is described in the "Developing and Installing Applications that Use the XA Libraries" section of the *Oracle7 Distributed Systems* manual.

Note: If you use a database other than Oracle, refer to the product documentation for information about defining the XA parameter.

- The pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter.

For information about the transaction log and defining parameters in the `UBBCONFIG` file, see "Modifying the UBBCONFIG File to Accommodate Transactions" on page 11-2.

Implementation Configuration File (ICF)

During the development process, change the Transaction policy of the `Registrar` object from `optional` to `always`. The `always` Transaction policy indicates that this object must be part of a transaction. For information about defining Transaction policies for CORBA objects, see *Creating C++ Server Applications*.

Setting Up the Transactions Sample Application

To build the Transactions sample application, complete the following steps:

- Step 1: Copy the Files for the Transactions Sample Application into a Work Directory
- Step 2: Change the Protection on the Files for the Transactions Sample Application
- Step 3: Set the Environment Variables
- Step 4: Initialize the University Database

- Step 5: Load the UBBCONFIG File
- Step 6: Create a Transaction Log

Note: Before you can build or run the Transactions sample application, you need to perform the steps in “Setting Up Your Environment” in the *Guide to the University Sample Applications*.

Step 1: Copy the Files for the Transactions Sample Application into a Work Directory

You need to copy the files for the Transactions sample application into a work directory on your local machine.

Source File Directories

The files for the Transactions sample application are located in the following directories:

Windows NT

`drive:\WLEdir\samples\corba\university\transaction`

UNIX

`/usr/WLEdir/samples/corba/university/transaction`

In addition, you need to copy the `utils` directory into your work directory. The `utils` directory contains files that set up logging, tracing, and access to the University database.

Source Files

Table 6-1 lists the files used to create the Transactions sample application.

Table 6-1 Files Included in the Transactions Sample Application

File	Description
`univt.idl`	OMG IDL that declares the `CourseSynopsisEnumerator`, `Registrar`, and `RegistrarFactory` interfaces
`univts.cpp`	C++ source code for the University server application in the Transactions sample application
`univt_i.h` `univt_i.cpp`	C++ source code for method implementations of the `CourseSynopsisEnumerator`, `Registrar`, and `RegistrarFactory` interfaces
`univtc.cpp`	C++ source code for the CORBA C++ client application in the Transactions sample application
`frmBrowser.frm`	Visual Basic source code for the ActiveX client application in the Transactions sample application
`frmOpen.frm`	Visual Basic source code for the ActiveX client application in the Transactions sample application
`University.vbp`	Visual Basic project file for the ActiveX client application in the Transactions sample application
`University.vbw`	Visual Basic workspace file for the ActiveX client application in the Transactions sample application
`modPublicDeclarations.` `bas`	A Visual Basic file that contains the declarations for variables used in the sample applications
`frmTracing.frm` `frmTracing.frx`	Files that provide tracing capabilities to the ActiveX client application
`frmLogon.frm`	Visual Basic file that performs the security logon for the ActiveX client application
`UnivTApplet.java`	Java source code for the CORBA Java client application in the Transactions sample application
`univt_utils.h` `univt_utils.cpp`	Files that define database access functions for the CORBA C++ client application
`univt.icf`	ICF file for the Transactions sample application

Table 6-1 Files Included in the Transactions Sample Application (Continued)

File	Description
`setenvt.sh`	UNIX script that sets the environment variables needed to build and run the Transactions sample application
`setenvt.cmd`	MS-DOS command procedure that sets the environment variables needed to build and run the Transactions sample application
`ubb_t.mk`	`UBBCONFIG` file for the UNIX operating system
`ubb_t.nt`	`UBBCONFIG` file for the Windows NT operating system
`makefilet.mk`	`makefile` for the Transactions sample application on the UNIX operating system
`makefilet.nt`	`makefile` for the Transactions sample application on the Windows NT operating system
`log.cpp`, `log.h`, `log_client.cpp`, and `log_server.cpp`	Client and server applications that provide logging and tracing functions for the sample applications. These files are located in the `\utils` directory.
`oradbconn.cpp` and `oranoconn.cpp`	Files that provide access to an Oracle SQL database instance. These files are located in the `\utils` directory.
`samplesdb.cpp` and `samplesdb.h`	Files that provide print functions for the database exceptions in the sample applications. These files are located in the `\utils` directory.
`unique_id.cpp` and `unique_id.h`	C++ Unique ID class routines for the sample applications. These files are located in the `\utils` directory.
`samplesdbsql.h` and `samplesdbsql.pc`	C++ class methods that implement access to the SQL database. These files are located in the `\utils` directory.
`university.sql`	SQL for the University database. This file is located in the `\utils` directory.

Step 2: Change the Protection on the Files for the Transactions Sample Application

During the installation of the WLE software, the sample application files are marked read-only. Before you can edit the files or build the files in the Transactions sample application, you need to change the protection of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>chmod u+rw /workdirectory/*.*
```

Step 3: Set the Environment Variables

Use the following command to set the environment variables used to build the client and server applications in the Transactions sample application:

Windows NT

```
prompt>setenvt
```

UNIX

```
prompt>/bin/ksh
prompt>. ./setenvt.sh
```

Step 4: Initialize the University Database

Use the following command to initialize the University database used with the Transactions sample application:

Windows NT

```
prompt>nmake -f makefilet.nt initdb
```

UNIX

```
prompt>make -f makefilet.mk initdb
```

Step 5: Load the UBBCONFIG File

Use the following command to load the `UBBCONFIG` file:

Windows NT

```
prompt>tmloadcf -y ubb_t.nt
```

UNIX

```
prompt>tmloadcf -y ubb_t.mk
```

The build process for the `UBBCONFIG` file prompts you for an application password. This password will be used to log on to the client applications. Enter the password and press Enter. You are then prompted to verify the password by entering it again.

Step 6: Create a Transaction Log

The transaction log records the transaction activities in a WLE application. During the development process, you need to define the location of the transaction log (specified by the `TLOGDEVICE` parameter) in the `UBBCONFIG` file. For the Transactions sample application, the transaction log is placed in your work directory.

You need to perform the following steps to open the transaction log for the Transactions sample application:

1. Enter the following command to start the Interactive Administrative Interface:

   ```
   tmadmin
   ```

2. Enter the following command to create a transaction log:

   ```
   crdl -b blocks -z directorypath
   clog -m SITE1
   ```

   where

   - *`blocks`* specifies the number of blocks to be allocated for the transaction log

- *directorypath* indicates the location of the transaction log. The *directorypath* needs to match the location specified in the TLOGDEVICE parameter in the UBBCONFIG file.

The following is an example of the command on Windows NT:

```
crdl -b 500 -z c:\mysamples\university\Transaction\TLOG
```

3. Enter q to exit the Interactive Administrative Interface.

Compiling the Transactions Sample Application

This topic includes the following sections:

- Building the CORBA C++ Server and Client Applications
- Building the CORBA Java Client Application
- Building the ActiveX Client Application

During the development process, you use the buildobjclient and buildobjserver commands to build the client and server applications. You would also build a database-specific transaction manager to coordinate the transactional events in the client/server application. However, for the Transactions sample application, this step has been done for you. The directory for the Transactions sample application contains a makefile that builds the client and server sample applications and creates a transaction manager called TMS_ORA.

Note: In the makefile, the following parameter is hard coded to build a transaction manager for the Oracle database:

```
RM=Oracle_XA
```

If you use a database other than Oracle, you need to change this parameter.

For more information about the buildobjclient and buildobjserver commands, see the *C++ Programming Reference*.

Building the CORBA C++ Server and Client Applications

Use the following commands to build the CORBA C++ client and server applications in the Transactions sample application:

Windows NT

```
prompt>nmake -f makefilet.nt
```

UNIX

```
prompt>make -f makefilet.mk
```

Building the CORBA Java Client Application

To build the CORBA Java client application:

Windows NT

```
prompt>nmake -f makefilet.nt javaclient
```

UNIX

```
prompt>make -f makefilet.mk javaclient
```

Building the ActiveX Client Application

For information about starting the ActiveX client application, see “Starting the ActiveX Client Application” on page 6-16.

Running the Transactions Sample Application

To run the Transactions sample application, complete the following steps:

- Step 1: Start the Server Application
- Step 2: Start a Client Application

Step 1: Start the Server Application

Start the system and sample application server applications in the Transactions sample application by entering the following command:

```
prompt>tmboot -y
```

Table 6-2 shows the server processes started by running this command:

Table 6-2 Server Processes Started by `tmboot` Command

Process	Description
`TMSYSEVT`	BEA TUXEDO system event broker.
`TMFFNAME`	Transaction management services, including the NameManager and the FactoryFinder services.
`TMIFSRVR`	Interface Repository server process. This server process is used only by ActiveX client applications.
`univt_server`	University server process.
`ISL`	IIOP Listener/Handler process.

Before using another sample application, enter the following command to stop the system and sample application server processes:

```
prompt>tmshutdown
```

Step 2: Start a Client Application

This topic includes the following sections:

- Starting the CORBA C++ Client Application
- Starting the CORBA Java Client Application

- Starting the ActiveX Client Application

Starting the CORBA C++ Client Application

Start the CORBA C++ client application in the Transactions sample application by performing the following steps:

1. At the MS-DOS prompt, enter the following command:

   ```
   prompt>univt_client
   ```

2. At the `Enter student id:` prompt, enter any number between `100001` and `100010`.
3. Press Enter.
4. At the `Enter domain password:` prompt, enter the password you defined when you loaded the `UBBCONFIG` file.
5. Press Enter.

Starting the CORBA Java Client Application

Before you can start the CORBA Java client application, you need to change the value of the `Port` and `Host` parameters in `UnivSApplet.html` to match the host name and port number specified in the `ISL` parameter in the `UBBCONFIG` file. For example:

```
<param   name=port      value=2500>
<param   name=host      value=BEANIE>
```

Start the CORBA Java client application in the Transactions sample application by performing the following steps:

1. At the MS-DOS prompt, enter the following command:

   ```
   prompt>appletviewer UnivTApplet.html
   ```

   A logon window appears.
2. Enter a number between `100001` and `100010` in the student ID field.
3. In the Domain Password field, enter the password you defined when you loaded the `UBBCONFIG` file.
4. Double click the Logon button.

Starting the ActiveX Client Application

For the University sample applications, the task of loading the OMG IDL for the CORBA interfaces into the Interface Repository is automated by the `makefile`.

Creating ActiveX Bindings for CORBA Interfaces

Before you can start the ActiveX client application, you must use the Application Builder to create ActiveX bindings for the CORBA interfaces.

To create an ActiveX binding for a CORBA interface:

1. Click the BEA Application Builder icon in the WLE program group.

   The IIOP Listener window appears.

2. In the IIOP Listener window, enter the host name and port number that you specified in the `ISL` parameter in the `UBBCONFIG` file. You must match exactly the capitalization used in the `UBBCONFIG` file.

   The Logon window appears.

3. In the Logon window, enter a student ID between `100001` and `100010` for the user name and the password you defined when you loaded the `UBBCONFIG` file.

   The Application Builder window appears. All the CORBA interfaces loaded in the Interface Repository appear in the Services window of the Application Builder.

4. Highlight the UniversityT folder in the Services window and drag it to the Workstation Views window, or copy the UniversityT folder from the Services window and paste it into the Workstation Views window.

   A confirmation window appears.

5. Click Create to create the ActiveX bindings for the CORBA interfaces in the Transactions sample application.

   The Application Builder creates the following:

   - A binding for the CORBA interface. The binding is named `DI`*`modulename_interfacename`*. For example, the binding for the `Registrar` interface is named `DIUniversityT_Registrar`.
   - A type library. By default, the type library is placed in `\`*`WLEdir`*`\TypeLibraries.`

The type library file is named `DI`*`modulename_interfacename`*`.tlb`.

- A Windows system registry entry, including unique Program IDs for each object type, for the CORBA interface.

Running the ActiveX Client Application

Perform the following steps to run the ActiveX client application:

1. Open the `University.vbw` file in Visual Basic.
2. From the Run menu, click Start.

   The IIOP Listener window appears.
3. In the IIOP Listener window, enter the host name and port number that you specified in the `ISL` parameter in the `UBBCONFIG` file. You must match exactly the capitalization used in the `UBBCONFIG` file.

   The Logon window appears.
4. In the Logon window, enter a student ID between `100001` and `100010` for the user name and the password you defined when you loaded the `UBBCONFIG` file.

Using the Client Applications in the Transactions sample application

This topic includes the following sections:

- CORBA C++ Client Application
- CORBA Java Client Application
- ActiveX Client Application

These sections briefly explain how to use the client applications in the Transactions sample application.

CORBA C++ Client Application

The CORBA C++ client application in the Transactions sample application has the following additional option:

```
<R>    Register for Courses
```

Registering for a Course

Perform the following steps to register for a course:

1. At the Options prompt, enter `R`.
2. At the `Course Number` prompt, enter a course number followed by `-1`.
   For example:
   ```
   100011
   100039
   -1
   ```
3. Press Enter.
4. At the Options prompt, enter `L` to view a list of courses for which the student ID is registered.

Other Tasks

To exit the C++ CORBA client application, enter `E` at the Options prompt.

CORBA Java Client Application

When you log on to the CORBA Java client application, a Student Account Summary window appears. Use the Student Account Summary window to register for courses.

Registering for a Class

Perform the following steps to register for a class:

1. Obtain a list of available courses from the Student Account Summary window by entering a text string in the Search String text box, such as `computer`.
2. Click the Search Catalog button.

   A list of courses matching the search string appears in the window.
3. Select a course by clicking on its name in the lower portion of the Student Account Summary window.

   If you are already registered for a course, `Yes` appears in the Registered field on the Student Account Summary window.
4. To register for the course, click the Register button on the Student Account Summary window.

Other Tasks

To view a list of courses for which the student ID is registered, double click the Show Registration button.

To exit the CORBA Java client application, click the Logoff button in the Student Account Summary window, or choose Quit from the Applet menu.

ActiveX Client Application

When you log on to the ActiveX client application, the Course Browser window appears. Use the Course Browser window to register for courses.

Registering for a Class

Perform the following steps to register for a class:

1. In the text box next to the Find Courses button, enter a text string or use the pulldown menu to choose a curriculum subject, such as `computer`.
2. Click the Find Courses button.

   A list of all the courses that match that search string appears.
3. Select a course from the list that appears in the window next to the Get Details button, or double-click the course name.

The details for the selected course appear.

4. Click the Register for Course button or double click the course to enter the course into the schedule.

   The course appears in the student's schedule at the bottom of the window. If the student is already registered for the course, it appears in the color green. If the course conflicts with a previously registered course, it appears in the color red.

Other Tasks

While using the ActiveX client application:

- To remove a course from the schedule, double click the course in the schedule.
- To view a list of courses for which the student ID is registered, click the Get Registered Courses button.
- To complete information about a course, click the Get Details button.
- To exit the ActiveX client application, choose Exit from the File menu.

CHAPTER

7 Transactions Sample CORBA Java/C++ XA Application

This topic includes the following sections:

- How the XA Bankapp Sample Application Works
- Development Process for the XA Bankapp Sample Application
- Setting Up the Database for the XA Bankapp Sample Application
- Building the XA Bankapp Sample Application
- Compiling the Client and Server Applications
- Initializing the Oracle Database
- Starting the Server Application in the XA Bankapp Sample Application
- Files Generated by the XA Bankapp Sample Application
- Starting the ATM Client Application in the XA Bankapp Sample Application
- Stopping the XA Bankapp Sample Application
- Using the ATM Client Application

For troubleshooting information and the most recent information about using the XA Bankapp sample application, see the `Readme.txt` file in the `\WLEdir\samples\corba\bankapp_java\XA` directory.

How the XA Bankapp Sample Application Works

The XA Bankapp sample application is a CORBA application that implements the same automatic teller machine (ATM) interface as the JDBC Bankapp sample application. However, the XA Bankapp sample application uses the Oracle XA library and the WebLogic Enterprise (WLE) Transaction Manager to coordinate transactions between the WLE application and the Oracle database that stores account and customer information.

This topic includes the following sections:

- Server Applications
- Application Workflow

Server Applications

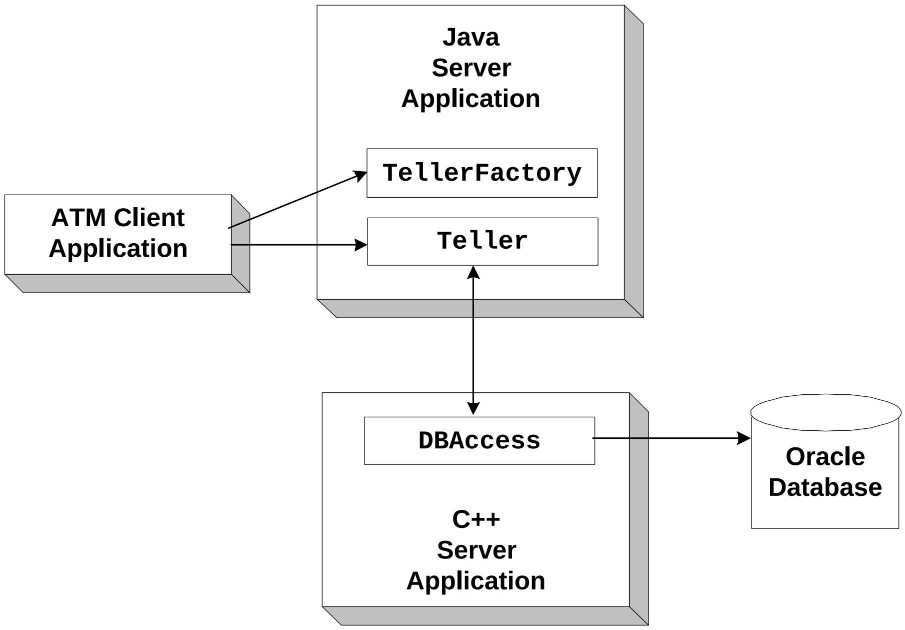
The XA Bankapp sample application consists of two server applications:

- A Java server application, which implements the `TellerFactory` and `Teller` objects.
- A C++ server application, which processes requests on objects that implement the `DBAccesss` interface.

Application Workflow

Figure 7-1 illustrates how the XA Bankapp sample application works.

Figure 7-1 The XA Bankapp Sample Application



In the XA Bankapp sample application, transactions are started and stopped in the `Teller` object using the Java Transaction Service (JTS) API. In the JDBC Bankapp sample application, transactions are started and stopped in the `DBAccess` object using the Java Database Connectivity (JDBC) API.

In the XA Bankapp sample application, the `DBAccess` object is implemented in C++ instead of Java and resides in its own server application. The object reference for the `DBAccess` object is generated in its `Server::initialize` method and is registered with the `FactoryFinder` environmental object.

Software Prerequisites

To run the XA Bankapp sample application, you need to install the following software:

- Visual C++ Version 5.0 with Service Pack 3 for Visual Studio
- Oracle Version 7.3.4

Development Process for the XA Bankapp Sample Application

This topic includes the following sections:

- Object Management Group (OMG) Interface Definition Language (IDL)
- Client Application
- Server Application
- Server Description File
- Implementation Configuration File
- UBBCONFIG File

These sections describe the development process for the XA Bankapp sample application.

Note: The steps in this section have been done for you and are included in the XA Bankapp sample application.

Object Management Group (OMG) Interface Definition Language (IDL)

The `BankApp.idl` file used in the XA Bankapp sample application defines the `TellerFactory` and `Teller` interfaces and the `Bank.idl` file defines exceptions and structures. The `transfer_funds` interface has been removed from the `BankDB.idl` because transactions are now started and stopped by the `Teller` object.

Client Application

The XA Bankapp sample application uses the same client application as the JDBC Bankapp sample application.

Server Application

For the XA Bankapp sample application, you would write the following:

- The Java Server object, which initializes the Java server application in the XA Bankapp sample application and registers a factory for the `Teller` object with the WLE domain.
- The implementation for the methods of the `Teller` object. The implementation for the `Teller` object includes invoking operations on the `DBAccess` object. Because the `Teller` object has durable state (for example, ATM statistics), which is stored in an external source (a flat file), the method implementations must also include the `activate_object()` and `deactivate_object` methods to ensure the `Teller` object is initialized with its state.
- The C++ server object which initializes the C++ server and registers the `DBAccess` object with the WLE domain.
- The implementation for the methods of the `DBAccess` object.

For information about writing server applications, see *Creating CORBA Java Server Applications* and *Creating CORBA C++ Server Applications*.

Server Description File

During development, you create a Server Description File (`BankApp.xml`) that defines the activation and transaction policies for the `TellerFactory` and `Teller` objects. Table 7-1 shows the activation and transaction policies for the XA Bankapp sample application.

Table 7-1 Activation and Transaction Policies for XA Bankapp Sample Application

Interface	Activation Policy	Transaction Policy
`TellerFactory`	`Process`	`Never`
`Teller`	`Method`	`Never`

A Server Description File for the XA Bankapp sample application is provided. For information about creating Server Description Files and defining activation and transaction policies on objects, see *Creating CORBA Java Server Applications*.

Implementation Configuration File

When writing WLE C++ server applications, you create an Implementation Configuration File (ICF). This file has been created for you and defines an activation policy of `transaction` and a transaction policy of `always` for the `DBAccess` interface.

For information about creating ICF files and defining activation and transaction policies on objects, see *Creating CORBA C++ Server Applications*.

UBBCONFIG File

During development, you need to include the following information in the `UBBCONFIG` file:

- The `OPENINFO` parameter, defined according to the `XA` parameter for the Oracle database. The `XA` parameter for the Oracle database is described the "Developing and Installing Applications that Use the XA Libraries" section of the *Oracle7 Distributed Systems* manual.
- The pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter.

For information about the transaction log and defining parameters in the `UBBCONFIG` file, see Chapter 11, "Administering Transactions."

Setting Up the Database for the XA Bankapp Sample Application

The XA Bankapp sample application uses an Oracle database to store all the bank data. Before using the XA Bankapp sample application, you need to install the following Oracle components:

- Oracle Server, Version 7.3.4
- Pro*C/C++ release 7.3.4 (for more information about supported compilers, see the Oracle product documentation)

Note: When installing the specified Oracle components, other Oracle components are also installed. However, you will not use these additional components with the XA Bankapp sample application.

You also need to start the Oracle database daemon and enable an XA Resource Manager. For information about installing the Oracle database and performing the necessary setup tasks, see the product documentation for the Oracle database.

Building the XA Bankapp Sample Application

This topic includes the following sections, which describe how to build the XA Bankapp sample application:

- Step 1: Copy the Files for the XA Bankapp Sample Application into a Work Directory
- Step 2: Change the Protection Attribute on the Files for the XA Bankapp Sample Application
- Step 3: Verify the Settings of the Environment Variables

- Step 4: Run the setupX Command
- Step 5: Load the UBBCONFIG File
- Step 6: Create a Transaction Log

Step 1: Copy the Files for the XA Bankapp Sample Application into a Work Directory

You need to copy the files for the XA Bankapp sample application into a work directory on your local machine.

Source File Directories

The files for the XA Bankapp sample application are located in the following directories:

Windows NT

`*drive:*\WLEdir\samples\corba\bankapp_java\XA`

`*drive:*\WLEdir\samples\corba\bankapp_java\client`

`*drive:*\WLEdir\samples\corba\bankapp_java\shared`

UNIX

`/usr/local/WLEdir/samples/corba/bankapp_java/XA`

`/usr/local/WLEdir/samples/corba/bankapp_java/client`

`/usr/local/WLEdir/samples/corba/bankapp_java/shared`

Table 7-2 describes the contents of these directories:

Table 7-2 Source File Directories in the XA Bankapp Sample Application

Directory	Description
`XA`	Source files and commands needed to build and run the XA Bankapp sample application.

Table 7-2 Source File Directories in the XA Bankapp Sample Application

Directory	Description
`client`	Files for the ATM client application. The `images` subdirectory contains `.gif` files used by the graphical user interface in the ATM client application.
`shared`	Common files for the JDBC Bankapp and XA Bankapp sample applications.

Copying Source Files to the Work Directory

You need only to copy the files manually in the XA directory. The other files are automatically copied from the `\client` and `\shared` directories when you execute the `setupX` command. For example:

Windows NT

```
prompt> cd c:\mysamples\bankapp_xa\XA
```

```
prompt> copy c:\WLEdir\samples\corba\bankapp_xa\XA\*
```

UNIX

```
ksh prompt> cd /usr/mysamples/bankapp_xa/XA/*
```

```
ksh prompt> cp $TUXDIR/samples/bankapp_xa/XA/*
```

Note: You cannot run the XA Bankapp sample application in the same work directory as the JDBC Bankapp sample application, because some of the files for the JDBC Bankapp sample application have the same name as files for the XA Bankapp sample application.

Source Files Used to Build the XA Bankapp Sample Application

Table 7-3 lists the files used to build and run the XA Bankapp sample application.

Table 7-3 Files Included in the XA Bankapp Sample Application

File	Description
`Bank.idl`	OMG IDL code that declares common structures and extensions for the XA Bankapp sample application.

Table 7-3 Files Included in the XA Bankapp Sample Application (Continued)

File	Description
`BankApp.idl`	OMG IDL code that declares the `TellerFactory` and `Teller` interfaces.
`BankDB.idl`	OMG IDL code that declares the `DBAccess` interface.
`BankDB.icf`	ICF file that defines activation and transaction policies for the `DBAccess` interface.
`BankDBServer.cpp`	C++ source code that implements the `Server::initialize` and `Server::release` methods for the C++ server application.
`TellerFactoryImpl.java`	Java source code that implements the `createTeller` method.
`TellerImpl.java`	Java source code that implements the `verify`, `deposit`, `withdraw`, `inquiry`, `transfer`, and `report` methods. In addition, it includes a reference to the `TransactionCurrent` environmental object and invokes operations on the `DBAccess` object within a transaction.
`BankAppServerImpl.java`	Java source code that overrides the `Server.initialize` and `Server.release` methods.
`Atm.java`	Java source code for the ATM client application.
`BankStats.java`	Contains methods to initialize, read from, and write to the flat file that contains the ATM statistics.
`BankApp.xml`	Server Description File used to associate activation and transaction policy values with CORBA interfaces.
`DBAccess_i.h` `DBAccess_i.pc`	Oracle Pro*C/C++ code that implements the `DBAccess` interface.
`InitDB.sql`	Oracle SQL *Plus script that creates and populates the database tables.

Table 7-3 Files Included in the XA Bankapp Sample Application (Continued)

File	Description
`setupX.cmd`	Windows NT batch file that builds and runs the XA Bankapp sample application.
`setupX.ksh`	UNIX Korn shell script that builds and runs the XA Bankapp sample application.
`makefileX.mk`	Make file for the XA Bankapp sample application on the UNIX operating system. The UNIX `make` command needs to be in the path of your machine.
`makefileX.nt`	Make file for the XA Bankapp sample application on the Windows NT operating system. The Windows NT `nmake` command needs to be in the path of your machine.
`Readme.txt`	Provides the latest information about building and running the XA Bankapp sample application.

Step 2: Change the Protection Attribute on the Files for the XA Bankapp Sample Application

During the installation of the WLE software, the files for the XA Bankapp sample application are marked read-only. Before you can edit or build the files in the XA Bankapp sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>/bin/ksh

ksh prompt>chmod u+w /workdirectory/*.*
```

Step 3: Verify the Settings of the Environment Variables

Before building and running the XA Bankapp sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Environment Variables

Table 7-4 lists the environment variables required to run the XA Bankapp sample application.

Table 7-4 Required Environment Variables for the XA Bankapp Sample Application

Environment Variable	Description

`TUXDIR`	The directory path where you installed the WLE software. For example:
	Windows NT
	`TUXDIR=c:\WLEdir`
	UNIX
	`TUXDIR=/usr/local/WLEdir`

`JAVA_HOME`	The directory path where you installed the JDK software. For example:
	Windows NT
	`JAVA_HOME=c:\JDK1.2`
	UNIX
	`JAVA_HOME=/usr/local/JDK1.2`

`ORACLE_HOME`	The directory path where you installed the Oracle software. For example: `ORACLE_HOME=/usr/local/oracle` You need to set this environment variable on the Solaris operating system only.

Verifying Settings

To verify that the information defined during installation is correct:

Windows NT

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.
4. Click the Environment tab.

   The Environment page appears.
5. Check the settings for `TUXDIR`, `ORACLE_HOME`, and `JAVA_HOME`.

UNIX

```
ksh prompt>printenv TUXDIR
ksh prompt>printenv JAVA_HOME
ksh prompt>printenv ORACLE_HOME
```

Changing Settings

To change the settings:

Windows NT

1. On the Environment page in the System Properties window, click the environment variable you want to change or enter the name of the environment variable in the Variable field.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

UNIX

```
ksh prompt>TUXDIR=directorypath; export TUXDIR
ksh prompt>JAVA_HOME=directorypath; export JAVA_HOME
ksh prompt>JAVA_HOME=directorypath; export ORACLE_HOME
```

Note: If you are running multiple WLE applications concurrently on the same machine, you also need to set the `IPCKEY` and `PORT` environment variables. See the `Readme.txt` file for information about how to set these environment variables.

Step 4: Run the setupX Command

The `setupX` command automates the following steps:

1. Copy the required files from the `\client` and `\shared` directories.
2. Set the `PATH`, `TOBJADDR`, `APPDIR`, `TUXCONFIG`, and `CLASSPATH` system environment variables.
3. Create the `UBBCONFIG` file.
4. Create a `setenvX.cmd` or `setenvX.ksh` file that can be used to reset the system environment variables.

Enter the `setupX` command, as follows:

Windows NT

```
prompt> cd c:\mysamples\bankapp_xa\XA

prompt>setupX
```

UNIX

```
prompt>/bin/ksh

prompt> cd /usr/mysamples/bankapp_xa/XA/*

prompt>. ./setupX.ksh
```

Step 5: Load the UBBCONFIG File

Use the following command to load the `UBBCONFIG` file:

```
prompt>tmloadcf -y ubb_xa
```

Step 6: Create a Transaction Log

The transaction log records the transaction activities in a WLE session. During the development process, you need to define the location of the transaction log (specified by the `TLOGDEVICE` parameter) in the `UBBCONFIG` file. For the XA Bankapp sample application, the transaction log is placed in your work directory.

To open the transaction log for the XA Bankapp sample application:

1. Enter the following command to start the Interactive Administrative Interface:

   ```
   tmadmin
   ```

2. Enter the following command to create a transaction log:

   ```
   crdl -b blocks -z directorypath TLOG
   crlog -m SITE1
   ```

   where

   - *`blocks`* specifies the number of blocks to be allocated for the transaction log.
   - *`directorypath`* indicates the location of the transaction log. The *`directorypath`* option needs to match the location specified in the `TLOGDEVICE` parameter in the `UBBCONFIG` file.

   The following is an example of the command on Windows NT:

   ```
   crdl -b 500 -z c:\mysamples\bankapp_java\XA\TLOG
   ```

3. Enter `quit` to exit the Interactive Administrative Interface.

Compiling the Client and Server Applications

The directory for the XA Bankapp sample application contains a make file that builds the client and server applications. During the development process, you use the `buildjavaserver` command to build the server application, and your Java product's development commands to build the client application. However, for the XA Bankapp sample application, this step is included in the make file.

Use the following commands to build the client and server applications in the XA Bankapp sample application:

Windows NT

```
prompt>nmake -f makefileX.nt
```

UNIX

```
prompt>make -f makefileX.mk
```

Initializing the Oracle Database

Use the following command to initialize the Oracle database used with the XA Bankapp sample application:

Windows NT

```
prompt>nmake -f makefileX.nt InitDB
```

UNIX

```
ksh prompt>make -f makefileX.mk InitDB
```

Starting the Server Application in the XA Bankapp Sample Application

Start the server application in the XA Bankapp sample application by entering the following command:

```
prompt>tmboot -y
```

The `tmboot` command starts the application processes listed in Table 7-5.

Table 7-5 Application Processes Started by `tmboot` Command

Process	Description
`TMSYSEVT`	BEA TUXEDO system event broker.
`TMFFNAME`	Three `TMFFNAME` server processes are started: ■ The `TMFFNAME` server process with the `-N` and `-M` options is the master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references. ■ The `TMFFNAME` server process started with the `-N` option only is the slave NameManager service. ■ The `TMFFNAME` server process started with the `-F` option contains the `FactoryFinder` object.
`TMS_ORA`	Transaction manager service.
`BankDataBase`	WLE server process that implements the `DBAccess` interface.
`JavaServerXA`	Server process that implements the `TellerFactory` and `Teller` interfaces. The `JavaServerXA` process has two options: ■ `BankApp.jar`, which is the Java Archive (JAR) file that was created by the `buildjavaserver` command. ■ `TellerFactory_1`, which is passed to the `Server.initialize` method. `JavaServerXA` is a special version of `JavaServer` that uses the same XA switch as the `BankDataBase` server process. It is created by the `buildXAJS` command.
`ISL`	IIOP Listener process.

Files Generated by the XA Bankapp Sample Application

Table 7-6 lists the files generated by the XA Bankapp sample application.

Table 7-6 Files Generated by the XA Bankapp Sample Application

File	Description
`ubb_xa`	`UBBCONFIG` file for the XA Bankapp sample application. This file is generated by the `setupX` command.
`setenvX.cmd` and `setenvX.ksh`	Contains the commands to set the environment variables needed to build and run the XA Bankapp sample application. `setenvX.cmd` is the Windows NT version and `setenvX.ksh` is the UNIX Korn shell version of the file.
`tuxconfig`	Binary version of the `UBBCONFIG` file. Generated by the `tmloadcf` command.
`TLOG`	Transaction log.
`ULOG.<`*`date`*`>`	Log file that contains messages generated by the `tmboot` command. The log file also contains messages generated by the server applications and the `tmshutdown` command.
`.adm/.keybd`	File that contains the security encryption key database. The subdirectory is created by the `tmloadcf` command.

`Atm$1.class`
`Atm.class`
`AtmAppletStub.class`
`AtmArrow.class`
`AtmButton.class`
`AtmCenterTextCanvas.class`
`AtmClock.class`
`AtmScreen.class`
`AtmServices.class`
`AtmStatus.class`
	Used by the Java client application. Created when the `Atm.java` file is compiled.

Table 7-6 Files Generated by the XA Bankapp Sample Application (Continued)

File	Description

`AccountRecordNotFound.java`
`AccountRecordNotFoundHelper.java`
`AccountRecordNotFoundHolder.java`
`CustAccounts.java`
`CustAccountsHelper.java`
`CustAccountsHolder.java`
`DataBaseException.java`
`DataBaseExceptionHelper.java`
`DataBaseExceptionHolder.java`
`InsufficientFunds.java`
`InsufficientFundsHelper.java`
`InsufficientFundsHolder.java`
`PinNumberNotFound.java`
`PinNumberNotFoundHelper.java`
`PinNumberNotFoundHolder.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `Bank.idl` file. These files are created in the `\com\beasys\samples\Bank` subdirectory.

`BalanceAmounts.java`
`BalanceAmountsHelper.java`
`BalanceAmountsHolder.java`
`IOException.java`
`IOExceptionHelper.java`
`IOExceptionHolder.java`
`Teller.java`
`TellerActivity.java`
`TellerActivityHelper.java`
`TellerActivityHolder.java`
`TellerFactory.java`
`TellerFactoryHelper.java`
`TellerFactoryHolder.java`
`TellerInsufficientFunds.java`
`TellerInsufficientFundsHelper.java`
`TellerInsufficientFundsHolder.java`
`_TellerFactoryImplBase.java`
`_TellerFactoryStub.java`
`_TellerImplBase.java`
`_TellerStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankApp.idl` file. These files are created in the `\com\beasys\samples\BankApp` subdirectory.

Table 7-6 Files Generated by the XA Bankapp Sample Application (Continued)

File	Description

`AccountData.java`
`AccountDataHelper.java`
`AccountDataHolder.java`
`DBAccessHelper.java`
`DBAccessHolder.java`
`_DBAccessImplBase.java`
`_DBAccessStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankDB.idl` file. These files are created in the `\com\beasys\samples\BankDB` subdirectory.

`Bankapp.ser` `Bankapp.jar`	Server Descriptor file and Server Java Archive file generated by the `buildjavaserver` command in the make file.

`Bank_c.cpp`
`Bank_c.h`
`Bank_s.cpp`
`Bank_s.h`
	Generated by the `idl` command for the interfaces defined in the `Bank.idl` file.

`BankDB_c.cpp`
`BankDB_c.h`
`BankDB_s.cpp`
`BankDB_s.h`
	Generated by the `idl` command for the interfaces defined in the `BankDB.idl` file.

`dbaccess_i.cpp`	Generated from the `DBAccess_i.pc` file by the Oracle Pro*C/C++ compiler.
`BankDataBase.exe`	WLE server application that implements the `DBAccess` interface.
`TMS_ORA.exe`	Server process for the Transaction Manager service.
`JavaServerXA`	Special version of the `JavaServer` that uses the same XA switches as the BankDataBase server process.
`stderr`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.err.println` method sends the output to the `stderr` file instead of to the `ULOG` file.

Table 7-6 Files Generated by the XA Bankapp Sample Application (Continued)

File	Description
`stdout`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.out.println` method sends the output to the `stdout` file instead of to the `ULOG` file.
`tmsysevt.dat`	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command.

Starting the ATM Client Application in the XA Bankapp Sample Application

Start the ATM client application by entering the following command, which sets the Java CLASSPATH to include the current directory and the client JAR file (`m3envobj.jar`).

Note: You can specify the full WLE JAR file (`m3.jar`) instead of the client JAR file.

Windows NT

```
prompt>java -classpath .;%TUXDIR%\udataobj\java\jdk\m3envobj.jar
-DTOBJADDR=%TOBJADDR% Atm Teller2
```

UNIX

```
ksh prompt>java -classpath .:$TUXDIR/udataobj/java/jdk
/m3envobj.jar -DTOBJADDR=$TOBJADDR Atm Teller2
```

The GUI for the ATM client application appears.

Stopping the XA Bankapp Sample Application

Before using another sample application, enter the following commands to stop the XA Bankapp sample application and to remove unnecessary files from the work directory:

Windows NT

```
prompt>tmshutdown -y
```

```
prompt>nmake -f makefileX.nt clean
```

UNIX

```
ksh prompt>tmshutdown -y
```

```
ksh prompt>make -f makefileX.mk clean
```

Using the ATM Client Application

The ATM client application in the XA Bankapp sample application works the same as in the JDBC Bankapp sample application. For instructions, see "Using the ATM Client Application" on page 5-30.

CHAPTER

8 Transactions in EJB Applications

This topic includes the following sections:

- Before You Begin
- General Guidelines
- Transaction Attributes
- Participating in a Transaction
- Transaction Semantics
- Session Synchronization
- Setting Transaction Timeouts

This topic describes how to integrate transactions in Enterprise JavaBeans (EJBs) applications that run under BEA WebLogic Enterprise (WLE). Before you begin, you should read Chapter 1, “Introducing Transactions.”

Before You Begin

This document describes the BEA implementation of transactions in Enterprise JavaBeans. The information in this document supplements the Sun Microsystems, Inc. evolving *Enterprise JavaBeans 1.1 Specification* (Public Release 2 dated October 18, 1999). In this document, *all* references to this specification pertain to the Public Release version.

Note: Before proceeding with the rest of this chapter, you must be thoroughly familiar with the *entire* contents of Sun Microsystem's specification document, particularly the concepts and material presented in Chapter 11, "Support for Transactions."

For general information about implementing Enterprise JavaBeans in WLE applications, see *The WLE Enterprise JavaBeans (EJB) Programming Environment*.

General Guidelines

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

- WLE fully supports Sun Microsystem's evolving *Enterprise JavaBeans 1.1 Specification* (Public Release 2 dated October 18, 1999). EJB applications must comply fully with this specification, including all of the various rules, requirements, and limitations that apply to entity beans, stateful session beans, and stateless session beans.
- 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 more information, see Chapter 10, "Transactions and the WLE JDBC/XA Driver."

- For EJB applications running under WLE, the `AUTOTRAN` setting (if specified) in the `INTERFACES` section of the `UBBCONFIG` file is ignored.
- Use standard programming techniques to optimize transaction processing. For example, properly demarcate transaction boundaries and complete transactions quickly.

For general guidelines about the WLE Transaction Service, see “Capabilities and Limitations” on page 2-2.

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 WLE 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, “Deployment Descriptor,” in Sun Microsystem’s *Enterprise JavaBeans Specification, v1.1* 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.

WLE fully supports method-level transaction attributes as defined in Section 11.4.1 in Sun Microsystem’s *Enterprise JavaBeans Specification, v1.1*.

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 Chapter 16, "Deployment Descriptor," in Sun Microsystem's *Enterprise JavaBeans Specification, v1.1*.

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 WLE EJB Container responds to these `trans-attribute` settings, see section 11.6.2 in Sun Microsystem's *Enterprise JavaBeans Specification, v1.1*.

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 Sun Microsystem's *Enterprise JavaBeans Specification, v1.1*.

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 WLE EJB Container maintains associations between the bean and the non-terminated 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 Sun Microsystem’s *Enterprise JavaBeans Specification, v1.1* 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

Sun Microsystem's *Enterprise JavaBeans Specification, v1.1* describes semantics that govern transaction processing behavior based on the EJB type (entity bean, stateless session bean, or stateful session bean) and the transaction type (container-managed or bean-managed). These semantics describe the transaction context at the time a method is invoked and define whether the EJB can access methods in the `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 8-1 describes the transaction semantics for stateful session beans in container-managed transactions.

Table 8-1 Transaction Semantics for Stateful Session Beans in Container-Managed Transactions

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

Table 8-1 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?
`afterBegin()`	Yes	No
`beforeCompletion()`	Yes	No
`afterCompletion()`	No	No

Transaction Semantics for Stateless Session Beans

Table 8-2 describes the transaction semantics for stateless session beans in container-managed transactions.

Table 8-2 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 8-3 describes the transaction semantics for entity beans in container-managed transactions.

Table 8-3 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 8-4 describes the transaction semantics for stateful session beans in bean-managed transactions.

Table 8-4 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 8-5 describes the transaction semantics for stateful session beans in bean-managed transactions.

Table 8-5 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 Sun Microsystem's *Enterprise JavaBeans Specification, v1.1*.

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

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

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.

Timeouts are specified according to the transaction type:

- **Container-managed transactions**. The Bean Provider configures the `trans-timeout-seconds` XML element in the `weblogic-ejb-extensions.xml` file. For more information, see the *WebLogic EJB Extensions Reference*.
- **Bean-managed transactions**. An application calls the `UserTransaction.setTransactionTimeout` method.

Handling Exceptions in EJB Transactions

WLE EJB applications need to catch and handle specific exceptions thrown during transactions. Sun Microsystem's *Enterprise JavaBeans Specification, v1.1* provides detailed information about handling exceptions in Chapter 12, "Exception handling."

For more information about how exceptions are thrown by business methods in EJB transactions, see the following tables in Section 12.3: Table 8 (for container-managed transactions) and Table 9 (for bean-managed transactions).

For a client's view of exceptions, see Section 12.4, particularly Section 12.4.1 (application exceptions), Section 12.4.2 (`java.rmi.RemoteException`), Section 12.4.2.1 (`javax.transaction.TransactionRolledBackException`), and Section 12.4.2.2 (`javax.transaction.TransactionRequiredException`).

9 Transactions in RMI Applications

This topic includes the following sections:

- Before You Begin
- General Guidelines

This topic describes how to integrate transactions in RMI applications that run under BEA WebLogic Enterprise (WLE).

Before You Begin

Before you begin, you should read Chapter 1, “Introducing Transactions,” particularly the following topics:

- “Transactions in WLE RMI Applications” on page 1-10
- “Transactions Sample RMI Code” on page 1-27

For more information about RMI applications, see *Using RMI in a WebLogic Enterprise Environment*.

General Guidelines

The following general guidelines apply when implementing transactions in RMI applications for WLE:

- WLE allows for flat transactions only. Transactions cannot be nested.
- For RMI applications running under WLE, the `AUTOTRAN` setting (if specified) in the `INTERFACES` section of the `UBBCONFIG` file is ignored.
- Use standard programming techniques to optimize transaction processing. For example, properly demarcate transaction boundaries and complete transactions quickly.

For general guidelines about the WLE Transaction Service, see “Capabilities and Limitations” on page 2-2.

CHAPTER

10 Transactions and the WLE JDBC/XA Driver

This topic includes the following sections:

- Before You Begin
- About Transactions and the WLE JDBX/XA Driver
- JDBC Accessibility in Java Methods
- Using the JDBC/XA Driver
- Implementing Distributed Transactions

This topic describes how to integrate transactions with CORBA Java, EJB, and RMI applications that use the WLE JDBC/XA driver and run under BEA WebLogic Enterprise (WLE). Before you begin, you should read Chapter 1, “Introducing Transactions.”

Before You Begin

This chapter describes handling transactions in CORBA Java, EJB, and RMI applications that use the WLE JDBC/XA driver to connect to resources.

For EJB applications, the information in this document supplements the Sun Microsystems, Inc. evolving *Enterprise JavaBeans 1.1 Specification* (Public Release 2 dated October 18, 1999). In this document, *all* references to this specification pertain to the Public Release version. For general information about implementing Enterprise JavaBeans in WLE applications, see *The WLE Enterprise JavaBeans (EJB) Programming Environment*.

About Transactions and the WLE JDBX/XA Driver

This topic includes the following sections:

- Support for Transactions Using the WLE JDBC/XA Driver
- Local Versus Distributed (Global) Transactions
- Transaction Contexts in WLE JDBC/XA Connections

Support for Transactions Using the WLE JDBC/XA Driver

WLE provides a multithreaded JDBC/XA driver for Oracle Corporation's Oracle8*i* database management system. The WLE JDBC/XA driver fully supports XA, the bidirectional system-level interface between a transaction manager and a Resource Manager of the X/Open Distributed Transaction Processing (DTP) model. This driver is available to CORBA Java, EJB, and RMI applications and runs in the WLE environment only.

Pooled Connections

Java applications use the WLE JDBC/XA driver to establish concurrent connections to multiple Oracle8*i* databases via their associated Resource Managers. For distributed transactions, applications must obtain database connections from the JDBC connection pool. (However, this is not a requirement for other jdbcKona drivers in local transaction mode or for third-party drivers.) Thereafter, applications perform database operations using standard JDBC API calls.

A JDBC connection is governed by the pooled connection lifecycle in the JDBC connection pool. As such, the application server might implicitly close JDBC/XA connections to enforce certain personality-specific transactional resource restrictions, as described in "JDBC Accessibility in Java Methods" on page 10-8. For more information about using WLE JDBC connection pools with WLE JDBC/XA driver, see "Using JDBC Connection Pooling" in *Using the JDBC Drivers*.

Characteristics of JavaServerXA

The `JavaServerXA` server hosts the WLE JDBC/XA driver. The JavaServerXA has the following characteristics:

- `JavaServerXA` is truly multithreaded.
- Multithreaded `JavaServerXA` cannot use JNI to make database access calls. If an application intends to use JNI to make database access calls, `JavaServerXA` must be configured to be single-threaded.
- `JavaServerXA` is still subject to other general multithreaded Java server constraints, as described in "Configuring Multithreaded Java Servers" in *Tuning and Scaling Applications*.
- Each `JavaServerXA` application can host the WLE JDBC connection pools that connect to one Resource Manager only (the Resource Manager of the TUXEDO group).

Supported JDBC Standards

WLE fully supports the JDBC 1.22 API (core functionality), the JDBC 2.0 Core API, and the distributed transactions (the `javax.sql.DataSource` API), connection pooling, and JNDI capabilities in the JDBC 2.0 Optional Package API. See *Using the JDBC Drivers* for a complete list of WLE-supported JDBC 2.0 features.

Local Versus Distributed (Global) Transactions

WLE applications using the WLE JDBC/XA driver can perform local transactions as well as distributed (also called global) transactions. A local transaction involves updates to a single Resource Manager (such as a database), while a distributed transaction involves updates across multiple Resource Managers.

The WLE JDBC/XA driver never starts a local transaction on behalf of an application. However, if the application performs database operations without first explicitly starting a distributed transaction, then these database operations occur within an "unspecified transaction context" and WLE delegates the responsibility of handling this situation to the database.

In Oracle8i, for example, the database might start a local transaction to perform such database operations.

- If autocommit is disabled, then it is the application's responsibility to explicitly complete the local transaction by calling the `javax.sql.Connection.commit` or `javax.sql.Connection.rollback` methods.
- If autocommit is enabled, then operations are committed automatically.

Failure to commit a local transaction may result in `XAER_OUTSIDE` error (indicating that the Resource Manager is performing work outside a distributed transaction) on subsequent distributed transaction operations, which includes beginning a distributed transaction. It is the responsibility of the application to be aware of the transaction context at any point and to complete distributed or local transactions appropriately.

Differences Between Local and Distributed Transactions

Table 10-1 lists differences between local and distributed transactions.

Table 10-1 Differences Between Local and Distributed Transactions

Category	Local Transactions	Distributed Transactions
Resource Managers/Databases	Single database / Resource Manager	Can span across multiple Resource Managers.

Table 10-1 Differences Between Local and Distributed Transactions

Category	Local Transactions	Distributed Transactions
Transaction Demarcation API	Can use the following API: `java.sql.Connection`	Can use either of following APIs: CORBA API: `org.omg.CosTransaction.TransactionCurrent` API. EJB API: `javax.transaction.UserTransaction` API.
Autocommit	Can be enabled or disabled.	Must be disabled.

Configuring the ENABLEXA Parameter in the UBBCONFIG

To use the WLE JDBC/XA driver, you must specify the `ENABLEXA` parameter (`ENABLEXA=Y`) in the `JDBCCONNPOOLS` section of the `UBBCONFIG`, as shown in Listing 10-1. In this example, distributed transactions are *enabled* for the `bank_pool` connection pool.

Note: This setting applies only to the WLE JDBC/XA driver.

Listing 10-1 Specifying `JDBCCONNPOOLS` Information in `UBBCONFIG`

```
JDBCCONNPOOLS
   bank_pool
      SRVGRP          = BANK_GROUP1
      SRVID           = 2
      DRIVER          = "weblogic.jdbc20.oci815.Driver"
      URL             = "jdbc:weblogic:oracle:Beq-local"
     PROPS          = "user=scott;password=tiger;server=Beq-Local"
      ENABLEXA        = Y
      INITCAPACITY    = 2
      MAXCAPACITY     = 10
      CAPACITYINCR    = 1
      CREATEONSTARTUP = Y
```

For more information about configuring JDBC connection pools, see “Using JDBC Connection Pooling” in *Using the JDBC Drivers*.

Demarcating Transaction Boundaries for Local and Distributed Transaction Contexts

Applications must carefully and explicitly demarcate transaction boundaries between distributed and local transaction contexts. For example, when an application uses the WLE JDBC/XA driver to connect to a database:

- By default, the autocommit feature is automatically disabled because it is assumed that transactions will be distributed.
- For that application to perform local transactions with autocommit (*after* completing the distributed transaction), it must explicitly enable autocommit by calling `javax.sql.Connection.setAutoCommit(true)`.

  After completing local transactions, the application must then disable autocommit *before* beginning a new distributed transaction. Listing 10-2 provides a simple example to illustrate switching between a distributed and local transaction.

Listing 10-2 Switching Between Distributed and Local Transactions

```
// Assumes that javax.transaction.UserTransaction (tx) and
// java.sql.Connection (con) were initialized previously

// Begin a distributed transaction
System.out.println("Beginning distributed transaction...");
tx.begin();
// Database operations within scope of transaction tx
if(gotException){
   try{
      tx.rollback();
      System.out.println("rolled back transaction");
      }catch(Exception e){}
   }
   elseif{
      tx.commit();
      System.out.println("committed transaction");
   }
// Local transactions
conn.setAutoCommit(true)
...[Database operations]...
conn.setAutoCommit(false)
// Begin another distributed transaction
System.out.println("Beginning distributed transaction...");
```

```
tx.begin();
...
```

Transaction Contexts in WLE JDBC/XA Connections

For WLE JDBC/XA connections, database operations will always be performed in the current transaction context. For example, an application might obtain a JDBC/XA connection in a `NULL` transaction context, begin a distributed transaction, and then perform database operations using that connection. These database operations will be performed in the context of the current distributed transaction.

Applications use WLE JDBC/XA connection API in the same way as other jdbcKona connections except that, while *within* a distributed transaction context:

- Attempting to enable autocommit mode by calling the `javax.sql.Connection.setAutoCommit` method on the WLE JDBC/XA connection will throw a `SQLException`.
- Attempting to complete the distributed transaction by calling the `javax.sql.Connection.commit` or `javax.sql.Connection.commit` methods on the WLE JDBC/XA connection will throw a `SQLException`.

Listing 10-3 shows, in a sample CORBA Java application, how to determine the current transaction context and commit a local or global transaction accordingly.

Listing 10-3 Determining Whether the Application is in a Distributed Transaction

```
// Assumes that org.omg.CosTransactions.Current (tc) and
// java.sql.Connection (con) were initialized before
// database operations were attempted
if (tc.get_status() !=
org.omg.CosTransactions.Status.StatusNoTransaction)
   {
   // Application is currently in a distributed transaction
   tc.commit(true);
   }
   else
   {
      // Application is currently in a local transaction
```

```
        con.commit();
    }
```

Similarly, for bean-managed transactions in an EJB application, the application can determine whether the application is currently in a distributed transaction by calling the `UserTransaction.getStatus()` method and testing for a returned `STATUS_NO_TRANSACTION`.

JDBC Accessibility in Java Methods

This topic includes the following sections:

- JDBC/XA Accessibility in CORBA Methods
- JDBC/XA Accessibility in EJB Methods

Note: Attempting to use a WLE JDBC/XA connection in a method where it is not supported may have undefined behavior and possibly raise a `SQLException`.

JDBC/XA Accessibility in CORBA Methods

Table 10-2 lists which methods in CORBA methods can access JDBC/XA connections.

Table 10-2 JDBC/XA Connection Accessibility for CORBA Objects

Server Method	Accessibility
constructor	Not Supported
`initialize`	Supported, after `open_xa_rm`
`activate_obj`	Supported
`deactivate_obj`	Supported
business method	Supported
`release`	Supported, before `close_xa_rm`

After completing the `initialize` method, WLE automatically closes any open connections and writes a warning message to the `ULOG`.

For transaction-bound and process-bound objects, the CORBA framework allows open connections to be retained at method end, and the transaction context of the retained connections will be as described in "Transaction Contexts in WLE JDBC/XA Connections" on page 10-7 upon subsequent method invocations. However, for method-bound objects, applications *must* explicitly close open connections before method end. If not, WLE automatically closes any open connections and writes a warning message to the `ULOG`.

JDBC/XA Accessibility in EJB Methods

For EJB methods, accessibility to JDBC/XA connections varies depending on the EJB type. For details about retaining JDBC/XA connections across method invocations (for stateful session beans only), including examples, see Section 11.3.3 in Sun Microsystem's *Enterprise JavaBeans Specification 1.1*.

Note: For all bean types, after completing the `ejbCreate` method, WLE automatically closes any open connections and writes a warning message to the `ULOG`.

Stateful Session Beans

Table 10-3 lists which stateful session bean methods can access JDBC/XA connections.

Table 10-3 JDBC/XA Connection Accessibility for Stateful Session Beans

Bean method	Container-managed transaction	Bean-managed transaction
constructor	Not Supported	Not Supported
`setSessionContext`	Not Supported	Not Supported

Table 10-3 JDBC/XA Connection Accessibility for Stateful Session Beans

Bean method	Container-managed transaction	Bean-managed transaction
`ejbCreate` `ejbRemove` `ejbActivate` `ejbPassivate`	Supported, but in unspecified transaction context (as defined in Sun Microsystem's EJB 1.1 specification).	Supported, but in unspecified transaction context (as defined in Sun Microsystem's EJB specification), unless the bean explicitly begins a transaction using `UserTransaction`.
business method	Supported	Supported
`afterBegin`	Supported	N/A
`beforeCompletion`	Supported	N/A
`afterCompletion`	Supported	N/A

For stateful session beans, the Bean Provider must close all JDBC connections in `ejbPassivate` and assign the instance's fields storing the connections to null. However, after completing the `ejbPassivate` method, WLE automatically closes any open connections and writes a warning message to the `ULOG`.

Stateless Session Beans

Table 10-4 lists which stateless session bean methods can access JDBC/XA connections.

Table 10-4 JDBC/XA Connection Accessibility for Stateless Session Beans

Bean Method	Container-Managed Transaction	Bean-Managed Transaction
constructor	Not Supported	Not Supported
`setSessionContext`	Not Supported	Not Supported
`ejbCreate`	Not Supported	Not Supported
`ejbRemove`	Not Supported	Not Supported

Table 10-4 JDBC/XA Connection Accessibility for Stateless Session Beans

Bean Method	Container-Managed Transaction	Bean-Managed Transaction
business method	Supported	Supported

Note: For stateless session beans, after completing a business method, WLE automatically closes any open connections and writes a warning message to the `ULOG`.

Entity Beans

Table 10-5 lists which entity bean methods can access JDBC/XA connections.

Table 10-5 JDBC/XA Connection Accessibility for Entity Beans

Bean Method	Accessibility
constructor	Not Supported
`setEntityContext`	Not Supported
`unsetEntityContext`	Not Supported
`ejbCreate`	Supported
`ejbPostCreate`	Supported
`ejbRemove`	Supported
`ejbFind`	Supported
`ejbActivate`	Not Supported
`ejbPassivate`	Not Supported
`ejbLoad`	Supported
`ejbStore`	Supported
business method	Supported

Using the JDBC/XA Driver

Before applications can use the WLE JDBC/XA driver, the JDBC/XA driver must be integrated into your development environment by completing the following steps:

1. Build the multithreaded `JavaServerXA` application, binding it with the Oracle8i Resource Manager, as described in "Using the WLE JDBC/XA Driver" in *Using the JDBC Drivers*.

2. In the `UBBCONFIG`, configure the `OPENINFO` parameter in the `GROUPS` section according to the definition of the `XA` parameter for the Oracle database. Listing 10-4 shows an example of an `OPENINFO` setting in a sample `UBBCONFIG`.

Listing 10-4 `OPENINFO` Setting in Sample `UBBCONFIG`

```
*GROUPS
   SYS_GRP
      LMID    = SITE1
      GRPNO   = 1
   BANK_GROUP1
      LMID    = SITE1
      GRPNO   = 2
   OPENINFO =
"ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=.+DbgFl=0
x7+MaxCur=15+Threads=true"
   TMSNAME  = TMS_ORA
   TMSCOUNT = 2
```

For more information about the `XA` parameter, see the "A Oracle XA" chapter in the Fundamentals section of Oracle Corporation's *Oracle8i Application Developer's Guide*.

3. If you want the `JavaServerXA` to be multithreaded, you must specify the `-M` option for the `CLOPT` parameter, which is defined in the `JavaServerXA` entry in the `SERVERS` section of the `UBBCONFIG` file.

Note: For single-threaded `JavaServerXA` operation, skip this step.

Listing 10-5 shows an example of `JavaServerXA` configured for multithreading in a sample `UBBCONFIG`.

Listing 10-5 Multithreaded Server Configuration in Sample `UBBCONFIG`

```
*SERVERS
   DEFAULT:
      RESTART = Y
      MAXGEN  = 5
    ...
    JavaServerXA
       SRVGRP  = BANK_GROUP1
       SRVID   = 2
       SRVTYPE = JAVA
       CLOPT   = "-A -- -M 10 BankApp.jar TellerFactory_1 bank_pool"
       RESTART = N
```

To specify connection pooling, you need to specify `SRVTYPE=JAVA` in the `SERVERS` section.

4. In the `UBBCONFIG`, configure the WLE JDBC/XA driver in the WLE JDBC Connection Pool, as described in“Using the WLE JDBC/XA Driver” in *Using the JDBC Drivers*. Listing 10-6 shows an example of JDBC connection pool settings for a connection pool named `bank_pool` in a sample `UBBCONFIG`.

Listing 10-6 JDBC Connection Pool Settings in Sample `UBBCONFIG`

```
*JDBCCONNPOOLS
   bank_pool
      SRVGRP         = BANK_GROUP1
      SRVID          = 2
      DRIVER         = "weblogic.jdbc20.oci815.Driver"
      URL            = "jdbc:weblogic:oracle:beq-local"
      PROPS          = "user=scott;password=tiger;server=Beq-Local"
      ENABLEXA       = Y
      INITCAPACITY   = 2
      MAXCAPACITY    = 10
      CAPACITYINCR   = 1
      CREATEONSTARTUP = Y
```

5. Boot the `JavaServerXA` application, as described in “Using the WLE JDBC/XA Driver” in *Using the JDBC Drivers*.

Implementing Distributed Transactions

This topic includes the following sections:

- Importing Packages
- Initializing the TransactionCurrent Object Reference
- Finding the Connection Pool via JNDI
- Setting Up XA Distributed Transactions
- Performing a Distributed Transaction

In addition to the fully supported examples supplied on the CD-ROM with this release of WLE, the BEA WLE team provides several unsupported code examples on a password protected web site for WLE customers. The code samples in this topic come from a version of the WLE XA Bankapp sample application that is available from the unsupported samples WLE web site. The URL for the unsupported samples WLE web site is specified in the product Release Notes under “About This BEA WLE Release” in the subsection “Unsupported Samples and Tools Web Page.”

This application is different from the one described in Chapter 7, “Transactions Sample CORBA Java/C++ XA Application.”

Note: This topic does not attempt to fully describe this sample application. It merely uses code fragments to illustrate the use of the JDBC/XA driver in a CORBA application.

Importing Packages

Listing 10-7 shows the packages that the application imports. In particular, note that:

- The `java.sql.*` and `javax.sql.*` packages are required for database operations.
- The `javax.naming.*` package is required for performing a JNDI lookup on the pool name, which is passed in as a command-line parameter upon server startup. The pool name must be registered on that server group.

Listing 10-7 Importing Required Packages

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import com.beasys.Tobj.*;
```

Initializing the TransactionCurrent Object Reference

Listing 10-8 shows initializing the `TransactionCurrent` object reference, which will be used by the `Teller` operations to start and stop transactions.

Listing 10-8 Initializing the `TransactionCurrent` Object Reference

```
static org.omg.CosTransactions.Current trans_cur_ref;

org.omg.CORBA.Object trans_cur_oref =
TP.bootstrap().resolve_initial_references("TransactionCurrent");
```

Finding the Connection Pool via JNDI

Listing 10-9 shows finding the connection pool via JNDI. The connection pool name is registered on the server group and is passed in as a command-line parameter upon server startup. Subsequent database connections are obtained from this pool.

Listing 10-9 Finding the Connection Pool via JNDI

```
static DataSource pool;

...

public void get_connpool(String pool_name)
    throws Exception
  {
    try {
      javax.naming.Context ctx = new InitialContext();
      pool = (DataSource)ctx.lookup("jdbc/" + pool_name);
    }
    catch (javax.naming.NamingException ex){
      TP.userlog("Couldn’t obtain JDBC connection pool: " +
pool_name);
      throw ex;
    }
  }
}
```

Setting Up XA Distributed Transactions

Listing 10-10 shows setting up XA distributed transactions by calling the `open_xa_rm` method (in `server.initialize`) and obtaining a reference to the `TransactionCurrent` object.

Note: This step is required for CORBA applications but not for EJB or RMI applications.

Listing 10-10 Setting Up XA Distributed Transactions

```
TP.open_xa_rm();

org.omg.CORBA.Object trans_cur_oref =
TP.bootstrap().resolve_initial_references("TransactionCurrent");

trans_cur_ref =
org.omg.CosTransactions.CurrentHelper.narrow(trans_cur_oref);
```

Performing a Distributed Transaction

Listing 10-11 shows a complete distributed transaction that involves the transfer of money from one bank account to another.

Sequence of Tasks

The application performs the distributed application in the following sequence:

1. The application calls the `begin` method to start the transaction.

2. The application performs the following database operations:
   - withdrawing the money from one account
   - depositing the money into another account
3. The application updates balances.
4. The application catches any exceptions thrown during the database operations.
5. The application closes the distributed transaction and updates teller statistics.
   - If an exception was thrown during the database operations, the application rolls back the transaction by calling the `rollback` method.
   - If no exceptions were thrown, the application commits the transaction by calling the `commit` method.

Listing 10-11   Performing a Distributed Transaction

```
public void transfer(int fromAccountID, int toAccountID, float
amount, BalanceAmountsHolder balances)
    throws AccountRecordNotFound, IOException, InsufficientFunds
  {
    boolean success = false;

    try {
      // Increment the number of requests the teller has received.
      tellerStats.totalTellerRequests += 1;

      // Begin the global transaction.
      BankAppServerImpl.trans_cur_ref.begin();

      // Flag this as a transfer.
      transferInProgress = true;

      // Perform the withdrawal first.
      float withdrawalBalance = withdraw(fromAccountID, amount);

      // Perform the deposit next.
      float depositBalance = deposit(toAccountID, amount);

      balances.value = new BalanceAmounts();
      balances.value.fromAccount = withdrawalBalance;
      balances.value.toAccount   = depositBalance;

      success = true;
    // Catch any exceptions thrown during database operations
    }
    catch (AccountRecordNotFound e) {
      throw e;
    }
    catch (InsufficientFunds e) {
      throw e;
    }
    catch (IOException e) {
      throw e;
    }
    catch(Exception e) {
      TP.userlog("Exception caught in transfer(): "
               + e.getMessage());
      e.printStackTrace();
      throw new org.omg.CORBA.INTERNAL();
    }
    finally {
      try {
       // Complete the distributed transaction and
       // update the Teller statistics.
       if (success) {
         tellerStats.totalTellerSuccess += 1;
         BankAppServerImpl.trans_cur_ref.commit(true);
        } else {
         tellerStats.totalTellerFail += 1;
         BankAppServerImpl.trans_cur_ref.rollback();
        }
      }
      catch(Exception e) {
       TP.userlog("Unexpected Exception thrown during commit or
rollback: " + e.getMessage());
        e.printStackTrace();
        throw new org.omg.CORBA.INTERNAL();
      }
      transferInProgress = false;
    }
  }
```

The withdraw Method

Listing 10-12 shows the `withdraw` method that is invoked in Listing 10-11. The `withdraw` method shows accessing the database to withdraw money from the specified account.

Listing 10-12 `withdraw` method

```
public float withdraw(int accountID, float amount)
  throws AccountRecordNotFound,
        IOException,
        InsufficientFunds,
        TellerInsufficientFunds
{
  boolean success = false;

  try {
    if (!transferInProgress) {
     // This is just a plain withdrawal; it is NOT a transfer.

     // Increment the number of requests that this teller
     // has received.
     tellerStats.totalTellerRequests += 1;

     // Decrement the balance left in the Teller's ATM machine.
     tellerStats.totalTellerBalance -= amount;

     // Begin the global transaction.
     BankAppServerImpl.trans_cur_ref.begin();

     // Check to see if the minimum TELLER threshold balance
     // has not been reached; if so, amount will be added back in
     // in the finally clause.
     if (tellerStats.totalTellerBalance < MinTellerBalance)
       throw new TellerInsufficientFunds();
    }

    AccountDataHolder accountDataHolder =
     new AccountDataHolder(new AccountData());
    accountDataHolder.value.accountID = accountID;
    accountDataHolder.value.balance = -amount;

    // Withdraw the money from the account.
    theDBAccess_ref.update_account(accountDataHolder);
    success = true;
    return(accountDataHolder.value.balance);
  }
  catch (AccountRecordNotFound e) {
    throw e;
  }
  catch (InsufficientFunds e) {
    throw e;
  }
  catch (TellerInsufficientFunds e) {
        throw e;
      }
      catch (DataBaseException e) {
        throw new IOException();
      }
      catch(Exception e) {
        TP.userlog("Exception caught in withdraw(): "
                 + e.getMessage());
        e.printStackTrace();
        throw new org.omg.CORBA.INTERNAL();
      }
      finally {
       // Terminate the transaction and update the Teller statistics.
        if (!transferInProgress) {
         try {
           if (success) {
             tellerStats.totalTellerSuccess += 1;
             BankAppServerImpl.trans_cur_ref.commit(true);
            } else {
             tellerStats.totalTellerFail += 1;
             tellerStats.totalTellerBalance += amount;
             BankAppServerImpl.trans_cur_ref.rollback();
            }
          }
         catch(Exception e) {
           TP.userlog("Unexpected Exception thrown during commit or
rollback: " + e.getMessage());
            e.printStackTrace();
            throw new org.omg.CORBA.INTERNAL();
          }
        }
      }
    }
```

The deposit Method

Listing 10-13 shows the `deposit` method that is invoked in Listing 10-11. The `deposit` method shows accessing the database deposit money to the specified account.

Listing 10-13 `deposit` method

```
  public float deposit(int accountID, float amount)
    throws AccountRecordNotFound, IOException
  {
    boolean success = false;

    try {
      // If this is a transfer request, then the global transaction
      // was started in the TellerImpl.transfer method; otherwise,
      // start the transaction here.
      if (!transferInProgress) {

       // This is just a plain deposit; it is NOT a transfer.
       // Increment the number of requests that this teller
       // has received.
       tellerStats.totalTellerRequests += 1;

       // Begin the global transaction.
       BankAppServerImpl.trans_cur_ref.begin();
      }

      AccountDataHolder accountDataHolder =
       new AccountDataHolder(new AccountData());
      accountDataHolder.value.accountID = accountID;
      accountDataHolder.value.balance = amount;

      // Deposit the money in the account.
      theDBAccess_ref.update_account(accountDataHolder);

      success = true;
      return(accountDataHolder.value.balance);
    }
    catch (AccountRecordNotFound e) {
      throw e;
    }
    catch (DataBaseException e) {
      throw new IOException();
    }
    catch(Exception e) {
      TP.userlog("Exception caught in BankApp.deposit(): "
               + e.getMessage());
      e.printStackTrace();
      throw new org.omg.CORBA.INTERNAL();
    }
    finally {
      try {
       // Terminate the transaction and update the Teller
statistics.
       if (!transferInProgress) {
         if (success) {
           tellerStats.totalTellerSuccess += 1;
           BankAppServerImpl.trans_cur_ref.commit(true);
         } else {
           tellerStats.totalTellerFail += 1;
           BankAppServerImpl.trans_cur_ref.rollback();
         }
        }
      }
      catch(Exception e) {
       TP.userlog("Unexpected Exception thrown during commit or
rollback: "
                 + e.getMessage());
       e.printStackTrace();
       throw new org.omg.CORBA.INTERNAL();
      }
    }
  }
```

CHAPTER

11 Administering Transactions

This topic includes the following sections:

- Modifying the UBBCONFIG File to Accommodate Transactions
- Modifying the Domain Configuration File to Support Transactions (WLE Servers)
- Sample Distributed Application Using Transactions

Before you begin, you should read Chapter 1, “Introducing Transactions.” In addition, for container-managed transaction demarcation in EJB applications, you can configure the transaction timeout setting, as described in “Setting Transaction Timeouts” on page 8-11.

Modifying the UBBCONFIG File to Accommodate Transactions

This topic includes the following sections:

- Summary of Steps
- Step 1: Specify Application-Wide Transactions in the RESOURCES Section
- Step 2: Create a Transaction Log (TLOG)
- Step 3: Define Each Resource Manager (RM) and the Transaction Manager Server in the GROUPS Section
- Step 4: Enable an Interface to Begin a Transaction

Summary of Steps

To accommodate transactions, you must modify the `RESOURCES`, `MACHINES`, `GROUPS`, and the `INTERFACES` or `SERVICES` sections of the application's `UBBCONFIG` file in the following ways:

- In the `RESOURCES` section, specify the application-wide number of allowed transactions and the value of the commit control flag.
- In the `MACHINES` section, create the `TLOG` information for each machine.
- In the `GROUPS` section, indicate information about each Resource Manager and about the transaction manager server.
- In the `INTERFACES` section (WLE System for CORBA applications only) or the `SERVICES` section (BEA TUXEDO System), enable the automatic transaction option. This option does *not* apply to EJB or RMI applications.

For instructions about modifying these sections in the `UBBCONFIG`, see "Creating a Configuration File" in the *Administration Guide*.

Step 1: Specify Application-Wide Transactions in the RESOURCES Section

Table 11-1 provides a description of transaction-related parameters in the `RESOURCES` section of the configuration file.

Table 11-1 Transaction-Related Parameters in the `RESOURCES` Section

Parameter	Meaning
`MAXGTT`	Limits the total number of global transaction identifiers (`GTRID`s) allowed on one machine at one time. The maximum value allowed is `2048`, the minimum is `0`, and the default is `100`. You can override this value on a per-machine basis in the `MACHINES` section. Entries remain in the table only while the global transaction is active, so this parameter has the effect of setting a limit on the number of simultaneous transactions.
`CMTRET`	Specifies the initial setting of the `TP_COMMIT_CONTROL` characteristic. The default is `COMPLETE`. Following are its two settings: ■ `LOGGED`—The `TP_COMMIT_CONTROL` characteristic is set to `TP_CMT_LOGGED`, which means that `tpcommit()` returns when all the participants have successfully precommitted. ■ `COMPLETE`—The `TP_COMMIT_CONTROL` characteristic is set to `TP_CMT_COMPLETE`, which means that `tpcommit()` will not return until all the participants have successfully committed. **Note:** You should consult with the RM vendors to determine the appropriate setting. If any RM in the application uses the *late commit* implementation of the XA standard, the setting should be `COMPLETE`. If all the Resource Managers use the *early commit* implementation, the setting should be `LOGGED` for performance reasons. (You can override this setting with `tpscmt()`.)

Step 2: Create a Transaction Log (TLOG)

This section discusses creating a transaction log (`TLOG`), which refers to a log in which information on transactions is kept until the transaction is completed.

Creating the UDL

The Universal Device List (UDL) is like a map of the BEA TUXEDO file system. The UDL gets loaded into shared memory when an application is booted. To create an entry in the UDL for the `TLOG` device, create the UDL on each machine using global transactions. If the `TLOGDEVICE` is mirrored between two machines, it is unnecessary to do this on the paired machine. The Bulletin Board Liaison (BBL) then initializes and opens the `TLOG` during the boot process.

To create the UDL, enter a command using the following format, before the application has been booted:

```
tmadmin -c crdl -z config -b blocks
```

where:

`-z` *`config`*	Specifies the full path name for the device where you should create the UDL.
`-b` *`blocks`*	Specifies the number of blocks to be allocated on the device.
`config`	Should match the value of the `TLOGDEVICE` parameter in the `MACHINES` section of the `UBBCONFIG` file.

Note: In general, the value that you supply for `blocks` should not be less than the value for `TLOGSIZE`. For example, if `TLOGSIZE` is specified as 200 blocks, specifying `-b 500` would not cause a degradation.

For more information about storing the `TLOG`, see the *Installation Guide*.

Defining Transaction-Related Parameters in the MACHINES Section

You can define a global transaction log (`TLOG`) using several parameters in the `MACHINES` section of the `UBBCONFIG` file. You must manually create the device list entry for the `TLOGDEVICE` on each machine where a `TLOG` is needed. You can do this either before or after `TUXCONFIG` has been loaded, but it must be done before the system is booted.

Note: If you are not using transactions, the `TLOG` parameters are not required.

Table 11-2 provides a description of transaction-related parameters in the `MACHINES` section of the configuration file.

Table 11-2 Transaction-Related Parameters in the `MACHINES` Section

Parameter	Meaning
`TLOGNAME`	The name of the `DTP` transaction log for this machine.
`TLOGDEVICE`	Specifies the WLE or BEA TUXEDO file system that contains the `DTP` transaction log (`TLOG`) for this machine. If this parameter is not specified, the machine is assumed not to have a `TLOG`. The maximum string value length is `64` characters.
`TLOGSIZE`	The size of the `TLOG` file in physical pages. Its value must be between `1` and `2048`, and its default is `100`. The value should be large enough to hold the number of outstanding transactions on the machine at a given time. One transaction is logged per page. The default should suffice for most applications.
`TLOGOFFSET`	Specifies the offset in pages from the beginning of `TLOGDEVICE` to the start of the `VTOC` that contains the transaction log for this machine.The number must be greater than or equal to `0` and less than the number of pages on the device. The default is `0`. `TLOGOFFSET` is rarely necessary. However, if two `VTOCs` share the same device or if a `VTOC` is stored on a device (such as a file system) that is shared with another application, you can use `TLOGOFFSET` to indicate a starting address relative to the address of the device.

Creating the Domains Transaction Log (BEA TUXEDO Servers)

This section applies to the BEA TUXEDO system only.

You can create the Domains transaction log before starting the Domains gateway group by using the following command:

```
dmadmin(1) crdmlog (crdlog) -d local_domain_name
```

Create the Domains transaction log for the named local domain on the current machine (the machine on which `dmadmin` is running). The command uses the parameters specified in the `DMCONFIG` file. This command fails if the named local domain is active on the current machine or if the log already exists. If the transaction log has not been created, the Domains gateway group creates the log when it starts up.

Step 3: Define Each Resource Manager (RM) and the Transaction Manager Server in the GROUPS Section

Additions to the `GROUPS` section fall into two categories:

- Defining the transaction manager servers that perform most of the work that controls global transactions:
  - The `TMSNAME` parameter specifies the name of the server executable.
  - The `TMSCOUNT` parameter specifies the number of such servers to boot (the minimum is 2, the maximum is 10, and the default is 3).

  A null transactional manager server does not communicate with any Resource Manager. It is used to exercise an application's use of the transactional primitives before actually testing the application in a recoverable, *real* environment. This server is named `TMS` and it simply begins, commits, or terminates without talking to any Resource Manager.

- Defining opening and closing information for each Resource Manager:
  - `OPENINFO` is a string with information used to open a Resource Manager.
  - `CLOSEINFO` is used to close a Resource Manager.

Sample GROUPS Section

The following sample `GROUPS` section derives from the `bankapp` banking application:

```
BANKB1 GRPNO=1 TMSNAME=TMS_SQL TMSCOUNT=2
OPENINFO=”TUXEDO/SQL:<APPDIR>/bankdl1:bankdb:readwrite”
```

Table 11-3 describes the transaction values specified in this sample `GROUPS` section.

Table 11-3 Transaction Values in the `GROUPS` Section of Sample `UBBCONFIG`

Transaction Value	Meaning
`BANKB1 GRPNO=1 TMSNAME=TMS_SQL\ TMSCOUNT=2`	Contains the name of the transaction manager server (`TMS_SQL`) and the number (2) of these servers to be booted in the group `BANKB1`
`TUXEDO/SQL`	Published name of the Resource Manager

Table 11-3 Transaction Values in the `GROUPS` Section of Sample `UBBCONFIG`

Transaction Value	Meaning
`<APPDIR>/bankdl1`	Includes a device name
`bankdb`	Database name
`readwrite`	Access mode

Characteristics of the TMSNAME, TMSCOUNT, OPENINFO, and CLOSEINFO Parameters

Table 11-4 lists the characteristics of the `TMSNAME`, `TMSCOUNT`, `OPENINFO`, and `CLOSEINFO` parameters.

Table 11-4 Characteristics of `TMSNAME`, `TMSCOUNT`, `OPENINFO`, and `CLOSEINFO` Parameters

Parameter	Characteristics
`TMSNAME`	Name of the transaction manager server executable. Required parameter for transactional configurations. `TMS` is a null transactional manager server.
`TMSCOUNT`	Number of transaction manager servers (must be between 2 and 10). Default is 3.
`OPENINFO` `CLOSEINFO`	Represents information to open or close a Resource Manager. Content depends on the specific Resource Manager. Starts with the name of the Resource Manager. Omission means the Resource Manager needs no information to open.

Step 4: Enable an Interface to Begin a Transaction

To enable an interface to begin a transaction, you change different sections in the `UBBCONFIG` file, depending on whether you are configuring a WLE CORBA server or BEA TUXEDO server:

- Changing the INTERFACES Section (WLE CORBA Servers)
- Changing the SERVICES Section (BEA TUXEDO Servers)

Changing the INTERFACES Section (WLE CORBA Servers)

The `INTERFACES` section in the `UBBCONFIG` file supports WLE CORBA interfaces:

- For each CORBA interface, set `AUTOTRAN` to `Y` if you want a transaction to start automatically when an operation invocation is received. `AUTOTRAN=Y` has no effect if the interface is already in transaction mode. The default is `N`. The effect of specifying a value for `AUTOTRAN` depends on the transactional policy specified by the developer in the Implementation Configuration File (ICF) in C++, or the Server Description File (XML) in Java, for the interface. This transactional policy will become the transactional policy attribute of the associated `T_IFQUEUE MIB` object at run time. The only time this value affects the behavior of the application is if the developer specified a transaction policy of `optional`.

  Note: To work properly, this feature depends on collaboration between the system designer and the administrator. If the administrator sets this value to Y without prior knowledge of the transaction policy defined by the developer in the interface's ICF or XML file, the actual run time effect of the parameter might be unknown.

- If `AUTOTRAN` is set to `Y`, you must set the `TRANTIME` parameter, which specifies the transaction timeout, in seconds, for the transactions to be created. The value must be greater than or equal to zero and must not exceed `2,147,483,647` (2^{31} - 1, or about 70 years). A value of zero implies there is no timeout for the transaction. (The default is `30` seconds.)

Note: For EJB and RMI applications, the `AUTOTRAN` and `TRANTIME` settings are ignored.

Table 11-5 describes the characteristics of the `AUTOTRAN`, `TRANTIME`, and `FACTORYROUTING` parameters.

Table 11-5 Characteristics of `AUTOTRAN`, `TRANTIME`, and `FACTORYROUTING` Parameters

Parameter	Characteristics
`AUTOTRAN`	■ Makes an interface the initiator of a transaction. ■ To work properly, it is dependent on collaboration between the system designer and the system administrator. If the administrator sets this value to `Y` without prior knowledge of the ICF or XML transaction policy set by the developer, the actual run-time effort of the parameter might be unknown. ■ The only time this value affects the behavior of the application is if the developer specified a transaction policy of `optional`. ■ If a transaction already exists, a new one is not started. ■ Default is `N`.
`TRANTIME`	■ Represents the timeout for the `AUTOTRAN` transactions. ■ Valid values are between 0 and 2^{31} - 1, inclusive. ■ Zero (0) represents no timeout. ■ Default is 30 seconds.
`FACTORYROUTING`	■ Specifies the name of the routing criteria to be used for factory-based routing for this CORBA interface. ■ You must specify a FACTORYROUTING parameter for interfaces requesting factory-based routing.

Changing the SERVICES Section (BEA TUXEDO Servers)

The following are three transaction-related features in the `SERVICES` section:

- If you want a service (instead of a client) to begin a transaction, you must set the `AUTOTRAN` flag to `Y`. This is useful if the service is not needed as part of any larger transaction, and if the application wants to relieve the client of making transaction decisions. If the service is called when there is already an existing transaction, this call becomes part of it. (The default is `N`.)

  Note: Generally, clients are the best initiators of transactions because a service has the potential of participating in a larger transaction.

- If AUTOTRAN is set to Y, you must set the TRANTIME parameter, which is the transaction timeout, in seconds, for the transactions to be created. The value must be greater than or equal to 0 and must not exceed 2,147,483,647 (2^{31} - 1, or about 70 years). A value of zero implies there is no timeout for the transaction. (The default is 30 seconds.)

Note: For EJB and RMI applications, the AUTOTRAN and TRANTIME settings are ignored.

- You must specify a ROUTING parameter for transactions that request data-dependent routing.

Table 11-6 describes the characteristics of the AUTOTRAN, TRANTIME, and ROUTING parameters:

Table 11-6 Characteristics of AUTOTRAN, TRANTIME, and ROUTING Parameters

Parameter	Characteristics
AUTOTRAN	Makes a service the initiator of a transaction. Relieves the client of the transactional burden. If a transaction already exists, a new one is not started. Default is N.
TRANTIME	Represents the timeout for the AUTOTRAN transactions. Valid values are between 0 and 2^{31} - 1, inclusive. 0 represents no timeout. Default is 30 seconds.
ROUTING	Points to an entry in the ROUTING section where data-dependent routing is specified for transactions that request this service.

Modifying the Domain Configuration File to Support Transactions (WLE Servers)

This topic includes the following sections:

- Characteristics of the DMTLOGDEV, DMTLOGNAME, DMTLOGSIZE, MAXRDTRAN, and MAXTRAN Parameters
- Characteristics of the AUTOTRAN and TRANTIME Parameters (WLE CORBA and TUXEDO Servers)

To enable transactions across domains, you need to set parameters in both the `DM_LOCAL_DOMAINS` and the `DM_REMOTE_SERVICES` sections of the Domains configuration file (`DMCONFIG`). Entries in the `DM_LOCAL_DOMAINS` section define local domain characteristics. Entries in the `DM_REMOTE_SERVICES` section define information on services that are *imported* and that are available on remote domains.

Characteristics of the DMTLOGDEV, DMTLOGNAME, DMTLOGSIZE, MAXRDTRAN, and MAXTRAN Parameters

The `DM_LOCAL_DOMAINS` section of the Domains configuration file identifies local domains and their associated gateway groups. This section must have an entry for each gateway group (Local Domain). Each entry specifies the parameters required for the Domains gateway processes running in that group.

Table 11-7 provides a description of the five transaction-related parameters in this section: `DMTLOGDEV`, `DMTLOGNAME`, `DMTLOGSIZE`, `MAXRDTRAN`, and `MAXTRAN`.

Table 11-7 Characteristics of `DMTLOGDEV`, `DMTLOGNAME`, `DMTLOGSIZE`, `MAXRDTRAN`, and `MAXTRAN` Parameters

Parameter	Characteristics
`DMTLOGDEV`	Specifies the BEA TUXEDO file system that contains the Domains transaction log (`DMTLOG`) for this machine. The `DMTLOG` is stored as a BEA TUXEDO `VTOC` table on the device. If this parameter is not specified, the Domains gateway group is not allowed to process requests in transaction mode. Local domains running on the same machine can share the same `DMTLOGDEV` file system, but each local domain must have its own log (a table in the `DMTLOGDEV`) named as specified by the `DMTLOGNAME` keyword.

Table 11-7 Characteristics of DMTLOGDEV, DMTLOGNAME, DMTLOGSIZE, MAXRDTRAN, and MAXTRAN Parameters (Continued)

Parameter	Characteristics
DMTLOGNAME	Specifies the name of the Domains transaction log for this domain. This name must be unique when the same DMTLOGDEV is used for several local domains. If a value is not specified, the value defaults to the string DMTLOG. The name must contain 30 characters or less.
DMTLOGSIZE	Specifies the numeric size of the Domains transaction log for this machine (in pages). It must be greater than zero and less than the amount of available space on the BEA TUXEDO file system. If a value is not specified, the value defaults to 100 pages. **Note:** The number of domains in a transaction determine the number of pages you must specify in the DMTLOGSIZE parameter. One transaction does not necessarily equal one log page.
MAXRDTRAN	Specifies the maximum number of domains that can be involved in a transaction. It must be greater than zero and less than 32,768. If a value is not specified, the value defaults to 16.
MAXTRAN	Specifies the maximum number of simultaneous global transactions allowed on this local domain. It must be greater than or equal to zero, and less than or equal to the MAXGTT parameter specified in the TUXCONFIG file. If not specified, the default is the value of MAXGTT.

Characteristics of the AUTOTRAN and TRANTIME Parameters (WLE CORBA and TUXEDO Servers)

The DM_REMOTE_SERVICES section of the Domains configuration file identifies information on services *imported* and available on remote domains. Remote services are associated with a particular remote domain.

Table 11-8 describes the two transaction-related parameters in this section: AUTOTRAN and TRANTIME.

Note: For EJB and RMI applications, these settings are ignored.

Table 11-8 Characteristics of AUTOTRAN and TRANTIME Parameters

Parameter	Characteristics
AUTOTRAN	Used by gateways to automatically start/terminate transactions for remote services. This capability is required if you want to enforce reliable network communication with remote services. You specify this capability by setting the AUTOTRAN parameter to Y in the corresponding remote service definition.
TRANTIME	Specifies the default timeout value in seconds for a transaction automatically started for the associated service. The value must be greater than or equal to zero, and less than 2147483648. The default is 30 seconds. A value of zero implies the maximum timeout value for the machine.

Sample Distributed Application Using Transactions

This topic includes the following sections:

- RESOURCES Section
- MACHINES Section
- GROUPS and NETWORK Sections
- SERVERS, SERVICES, and ROUTING Sections

This topic describes a sample configuration file for the bankapp application, a sample CORBA application that enables transactions and distributes the application over three sites. The application includes the following features:

- Data-dependent routing on ACCOUNT_ID.
- Data distributed over three databases.
- BRIDGE processes communicating with the system via the ATMI interface.

- System administration from one site.

The configuration file includes seven sections: RESOURCES, MACHINES, GROUPS, NETWORK, SERVERS, SERVICES, and ROUTING.

Note: Although this sample is a CORBA application, the principles apply to EJB applications as well, except that the ROUTING section is *not* used in EJB applications, nor are the TRANTIME and AUTOTRAN parameters in the INTERFACES section.

RESOURCES Section

The RESOURCES section shown in Listing 11-1 specifies the following parameters:

- MAXSERVERS, MAXSERVICES, and MAXGTT are less than the defaults. This makes the Bulletin Board smaller.
- MASTER is SITE3 and the backup master is SITE1.
- MODEL is set to MP and OPTIONS is set to LAN, MIGRATE. This allows a networked configuration with migration.
- BBLQUERY is set to 180 and SCANUNIT is set to 10. This means that DBBL checks of the remote BBLs are done every 1800 seconds (one half hour).

Listing 11-1 Sample RESOURCES Section

```
*RESOURCES
#
IPCKEY       99999
UID          1
GID          0
PERM         0660
MAXACCESSERS 25
MAXSERVERS   25
MAXSERVICES  40
MAXGTT       20
MASTER       SITE3, SITE1
SCANUNIT     10
SANITYSCAN   12
BBLQUERY     180
BLOCKTIME    30
```

```
DBBLWAIT     6
OPTIONS      LAN, MIGRATE
MODEL        MP
LDBAL         Y
```

MACHINES Section

The `MACHINES` section shown in Listing 11-2 specifies the following parameters:

- `TLOGDEVICE` and `TLOGNAME` are specified, which indicate that transactions will be done.
- The `TYPE` parameters are all different, which indicates that encode/decode will be done on all messages sent between machines.

Listing 11-2 Sample `MACHINES` Section

```
*MACHINES
Gisela       LMID=SITE1
              TUXDIR=”/usr/tuxedo”
              APPDIR=”/usr/home”
              ENVFILE=”/usr/home/ENVFILE”
              TLOGDEVICE=”/usr/home/TLOG”
              TLOGNAME=TLOG
              TUXCONFIG=”/usr/home/tuxconfig”
              TYPE=”3B600”

romeo         LMID=SITE2
              TUXDIR=”/usr/tuxedo”
              APPDIR=”/usr/home”
              ENVFILE=”/usr/home/ENVFILE”
              TLOGDEVICE=”/usr/home/TLOG”
              TLOGNAME=TLOG
              TUXCONFIG=”/usr/home/tuxconfig”
              TYPE=”SEQUENT”

juliet       LMID=SITE3
             TUXDIR=”/usr/tuxedo”
             APPDIR=’/usr/home”
             ENVFILE=”/usr/home/ENVFILE”
             TLOGDEVICE=”/usr/home/TLOG”
             TLOGNAME=TLOG
```

```
TUXCONFIG=”/usr/home/tuxconfig”
TYPE=”AMDAHL”
```

GROUPS and NETWORK Sections

The `GROUPS` and `NETWORK` sections shown in Listing 11-3 specify the following parameters:

- The `TMSCOUNT` is set to `2`, which means that only two `TMS_SQL` transaction manager servers will be booted per group.
- The `OPENINFO` string indicates that the application will perform database access.

Listing 11-3 Sample `GROUPS` and `NETWORK` Sections

```
*GROUPS
DEFAULT:        TMSNAME=TMS_SQL       TMSCOUNT=2
BANKB1          LMID=SITE1            GRPNO=1
 OPENINFO=”TUXEDO/SQL:/usr/home/bankdl1:bankdb:readwrite”
BANKB2          LMID=SITE2            GRPNO=2
 OPENINFO=”TUXEDO/SQL:/usr/home/bankdl2:bankdb:readwrite”
BANKB3           LMID=SITE3            GRPNO=3
 OPENINFO=”TUXEDO/SQL:/usr/home/bankdl3:bankdb:readwrite”

*NETWORK
SITE1            NADDR=”0X0002ab117B2D4359”
                 BRIDGE=”/dev/tcp”
                 NLSADDR=”0X0002ab127B2D4359”

SITE2            NADDR=”0X0002ab117B2D4360”
                 BRIDGE=”/dev/tcp”
                 NLSADDR=”0X0002ab127B2D4360”

SITE3            NADDR=”0X0002ab117B2D4361”
                 BRIDGE=”/dev/tcp”
                 NLSADDR=”0X0002ab127B2D4361”
```

SERVERS, SERVICES, and ROUTING Sections

The `SERVERS`, `SERVICES`, and `ROUTING` sections shown in Listing 11-4 specify the following parameters:

- The `TLR` servers have a `-T number` passed to their *`tpsrvrinit()`* functions.
- All requests for the services are routed on the `ACCOUNT_ID` field.
- None of the services will be performed in `AUTOTRAN` mode.

Note: The `ROUTING` section is *not* used in EJB applications.

Listing 11-4 Sample `SERVERS`, `SERVICES`, and `ROUTING` Sections

```
*SERVERS
DEFAULT: RESTART=Y MAXGEN=5 REPLYQ=N CLOPT=”-A”
TLR      SRVGRP=BANKB1    SRVID=1    CLOPT=”-A -- -T 100"
TLR      SRVGRP=BANKB2    SRVID=3    CLOPT=”-A -- -T 400"
TLR      SRVGRP=BANKB3    SRVID=4    CLOPT=”-A -- -T 700"
XFER     SRVGRP=BANKB1    SRVID=5    REPLYQ=Y
XFER     SRVGRP=BANKB2    SRVID=6    REPLYQ=Y
XFER     SRVGRP=BANKB3    SRVID=7    REPLYQ=Y

*SERVICES
DEFAULT:    AUTOTRAN=N
WITHDRAW        ROUTING=ACCOUNT_ID
DEPOSIT         ROUTING=ACCOUNT_ID
TRANSFER        ROUTING=ACCOUNT_ID
INQUIRY         ROUTING=ACCOUNT_ID

*ROUTING
ACCOUNT_ID      FIELD=ACCOUNT_ID     BUFTYPE=”FML”
                     RANGES=”MON - 9999:*,
                     10000 - 39999:BANKB1
                     40000 - 69999:BANKB2
                     70000 - 100000:BANKB3
                        “”
```

Index