BEA WebLogic Enterprise

Creating CORBA Java Server Applications

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Creating CORBA Java Server Applications

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

Contents

About This Document

This document describes how programmers can implement key features in the BEA WebLogic Enterprise (WLE) product to design and implement scalable, high-performance, Java server applications that run in a WLE domain. The Java examples shown in this book are based on the sample applications described in the *Guide to the Java Sample Applications*.

This document covers the following topics:

- Chapter 1, "Java Server Application Concepts," presents a number of basic concepts about creating WLE server applications and describes the programming entities you create for a WLE server application.
- Chapter 2, "Steps for Creating a Java Server Application," lists and describes the basic steps you follow to create a WLE server application.
- Chapter 3, "Integrating Transactions into a Java Server Application," describes how the WLE system supports transactions in a WebLogic Enterprise domain and how you can implement transactions into your server applications.
- Chapter 4, "Scaling a Java Server Application," describes the key scalability features that you can build into your WLE applications to make them highly scalable, including replicated server processes and groups, factory-based routing, and object state management.

What You Need to Know

This document is intended for programmers who are interested in creating secure, scalable, transaction-based server applications. It assumes you are knowledgeable with the BEA TUXEDO system, CORBA, and Java programming.

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.

Convention	Item
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`*
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`*`]...`

Convention	Item
|	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 Java Server Application Concepts

This chapter presents the following WLE topics:

- Overview
- The Entities You Create to Build a WLE Java Server Application
- Understanding Object References and Object State
- Choosing Between Stateless and Stateful Objects

For background information about WLE server applications and how they work, see *Getting Started*.

Overview

This section provides an overview of the Java server application creation process. The file names shown are based on the Bankapp sample application that is included with the WLE software. Many steps have been omitted from this simple overview. The purpose here is to give you an idea of the overall process, before you read about CORBA object state management and other key concepts in the remainder of this chapter, and before you read about detailed build steps in subsequent chapters.

To create a Java server application:

1. Create a text file that describes the interfaces for your CORBA objects.

   The descriptions are written in the Object Management Group Interface Definition Language (OMG IDL). For example, the `BankApp.idl` file describes the `Teller` and `TellerFactory` interfaces.

2. Compile the IDL files by using the `m3idltojava` compiler to generate, for each interface, Java object implementation files, client stubs, server skeletons, Helper classes, and Holder classes. Do not edit these files.

3. Copy each object implementation file to a new file.

   For example, compiling the `BankApp.idl` file with the `m3idltojava` compiler generates a `Teller.java` file and a `TellerFactory.java` file. To create Java files that you can use as a starting point for adding your business logic and object implementations, you can:

   a. Copy `Teller.java` to `TellerImpl.java.`

   b. Copy `TellerFactory.java` to `TellerFactoryImpl.java.`

4. Edit your object implementation files, adding the business logic to each object's methods.

5. Create the Server object, which is code that performs the initialize and release functions for the server application.

6. Use the `javac` compiler to compile all the `*.java` files into Java bytecodes (`*.class` files).

7. Create a text file called a Server Description File, which is expressed in the XML language.

   To see a sample file, open the `BankApp.xml` file that is included with the WLE software in the following directory:

   Windows NT

   `drive:\M3dir\samples\corba\bankapp_java\jdbc\`

   UNIX

   `/usr/local/M3dir/samples/corba/bankapp_java/jdbc/`

   In your Server Description File, you assign the activation and transaction policies for the interfaces implemented in your server application. This XML file also contains a server declaration, which includes the name of the Server object and the name of the server descriptor file (SER). You can also identify the Java class files that comprise the server application's Java Archive (JAR) file.

8. Compile the XML-based Server Description File with the `buildjavaserver` command and generate the SER file and JAR file.
9. Deploy your Java server application.

The Entities You Create to Build a WLE Java Server Application

To build a WLE Java server application, you create the following entities:

- The Java implementation of the CORBA objects that execute your server application's business logic. This topic is explained in the next section.
- The Java Server object, which performs the initialize and release functions for the server application, and may perform other functions. This topic is explained in the section "The Server Object" on page 1-8.
- A Java Archive (JAR) file that contains the Java bytecodes (`class` files) that comprise your server application. In the WLE Java environment, you can optionally use an `<ARCHIVE>` section of the Server Description File to identify and collect the class files and packages. The Server Description File is written in XML.

  The JAR file also contains a server descriptor, which is a Java object that contains information about all the servant classes implemented by the server application, along with the policies attached to the interfaces. Also stored in the JAR file is the name of the Server object that is used to initialize and stop the server.

There are also a number of files that you work with that are generated by the `m3idltojava` compiler and that you build into an WLE server application. These files are listed and described in Chapter 2, "Steps for Creating a Java Server Application."

The Implementation of the CORBA Objects for Your Java Server Application

Having a clear understanding of what CORBA objects are, and how they are defined, implemented, instantiated, and managed is critical for the person who is designing or creating an WLE Java server application.

The CORBA objects for which you have defined interfaces in the Object Management Group Interface Definition Language (OMG IDL) contain the business logic and data for your WLE Java server applications. All client application requests involve invoking operations on a CORBA object. The code you write that implements the operations defined for an interface is called an object implementation. For example, in Java, the object implementation is a Java class.

This section discusses the following topics:

- How OMG IDL interface definitions establish the operations that can be invoked on a CORBA object
- How you implement the operations on a CORBA object
- How client applications access and manipulate your application's CORBA objects

How Interface Definitions Establish the Operations on a CORBA Object

A CORBA object's interface identifies the operations that can be performed on it. A distinguishing characteristic of CORBA objects is that an object's interface definition is separate from its implementation. The definition for the interface establishes how the operations on the interface must be implemented, including what the valid parameters are that can be passed to and returned from an operation.

An interface definition, which is expressed in OMG IDL, establishes the client/server contract for an application. That is, for a given interface, the server application is bound to do the following:

- Implement the operations defined for that interface
- Always use the parameters defined with each operation

How the server application implements the operations may change over time. This is acceptable behavior as long as the server application continues to meet the requirement of implementing the defined interface and using the defined parameters. In this way, the client stub is always a reliable proxy for the object implementation on the server machine. This underscores one of the key architectural strengths of CORBA -- that you can change how a server application implements an object over time without requiring the client application to be modified or even to be aware that the object implementation has changed.

The interface definition also determines the content of both the client stub and the skeleton in the server application; these two entities, in combination with the ORB and the Portable Object Adapter (POA), ensure that a client request for an operation on an object can be routed to the code in the server application that can satisfy the request.

Once the system designer has specified the interfaces of the business objects in the application, the programmer's job is to implement those interfaces. This book explains how.

For more information about OMG IDL, see *Creating CORBA Client Applications*.

How You Implement the Operations on a CORBA Object

As stated earlier, the code that implements the operations defined for a CORBA object's interface is called an object implementation. For Java, this code consists of a set of methods, one for each of the operations defined for the interfaces in your application's OMG IDL file.

In the WLE Java environment, you define an object implementation file by copying the *`interface`*`.java` file generated by the `m3idltojava` compiler and editing the copy. For example, using the file names in the Bankapp sample application, copy the `Teller.java` file to `TellerImpl.java`. Then, you edit `TellerImpl.java`, adding your business logic to create the Teller object's implementation file. The suggested modification steps are described in the section "Creating an Object Implementation File" on page 2-6.

You also define the object's default in-memory behavior in a separate file, the XML-based Server Description File. In this XML file, you define the default activation and transaction policies for each interface that is implemented in the server application. You then provide this file as input to the `buildjavaserver` command.

How Client Applications Access and Manipulate Your Application's CORBA Objects

Client applications access and manipulate the CORBA objects managed by the server application via **object references** to those objects. Client applications invoke operations (that is, requests) on an object reference. These requests are sent as messages to the server application, which invokes the appropriate operations on CORBA objects. The fact that these requests are sent to the server application and invoked in the server application is completely transparent to the client; client applications appear simply to be making invocations on the client stub.

Client applications may manipulate a CORBA object only by means of an object reference. One primary design consideration is how to create object references and return them to the client applications that need them in a way that is appropriate for your application.

Typically, object references to CORBA objects are created in the WLE system by **factories**. A factory is any CORBA object that returns, as one of its operations, a reference to another CORBA object. You implement your application's factories the same way that you implement other CORBA objects defined for your application.

You can make your factories widely known to the WLE domain, and to clients connected to the WLE domain, by registering them with the FactoryFinder. Registering a factory is an operation typically performed by the Server object, which is described in the section "The Server Object" on page 1-8. For more information about designing factories, see the section "Generating Object References" on page 1-10.

The Content of an Object Reference

From the client application's perspective, an object reference is opaque; it is like a black box that client applications use without having to know what is inside. However, object references contain all the information needed for the WLE system to locate a specific object instance and to locate any state data that is associated with that object.

An object reference contains the following information:

- The interface name

  This is the Interface Repository ID of the objects' OMG IDL interface.

- The object ID (OID)

The OID uniquely identifies the instance of the object to which the reference applies. If the object has data in external storage, the OID also typically includes a key that the server machine can use to locate the object's data.

- Group ID

  The group ID identifies the server group to which the object reference is routed when a client application makes a request using that object reference. Generating a nondefault group ID is part of a key WLE feature called factory-based routing, which is described in the section "Factory-based Routing" on page 4-13.

Note: The combination of the three items in the preceding list uniquely identifies the CORBA object. It is possible for an object with a given interface and OID to be simultaneously active in two different groups, if those two groups both contain the same object implementation.

If you need to guarantee that only one object instance of a given interface name and OID is available at any one time in your domain, either: use factory-based routing to ensure that objects with a particular OID are always routed to the same group, or configure your domain so that a given object implementation is in only one group. This assures that if multiple clients have an object reference containing a given interface name and OID, the reference is always routed to the same object instance.

For more information about factory-based routing, see the section "Factory-based Routing" on page 4-13.

The Lifetime of an Object Reference

Object references created by server applications running in a WLE domain have a usable lifespan that extends beyond the life of the server process that creates them. WLE object references can be used by client applications regardless of whether the server processes that originally created them are still running. In this way, object references are not tied to a specific server process.

The Server Object

The Java Server object is the other programming code entity that you create for an WLE server application. The Java Server object implements operations that execute the following tasks:

- Performing basic server application initialization operations, which may include registering factories managed by the server application and allocating resources needed by the server application. If the server application is transactional, the Server object also implements the code that opens an XA resource manager.
- Performing server process shutdown and cleanup procedures when the server application has finished servicing requests. For example, if the server application is transactional, the Server object also implements the code that closes the XA resource manager.

You implement this Server object by creating a new class that derives from `com.beasys.Tobj.Server` and overrides the `initialize` and `release` methods. In the server application code, you can also write a public default constructor. You create the Server object class from scratch using a text editor.

For example:

```
import com.beasys.Tobj.*;

/**
 * Provides code to initialize and stop the server invocation.
 * BankAppServerImpl is specified in the BankApp.XML input file
 * as the name of the Server object.
 */

public class BankAppServerImpl
       extends com.beasys.Tobj.Server {

       public boolean initialize(string[] args)
              throws com.beasys.TobjS.InitializeFailed;

       public boolean release()
              throws com.beasys.TobjS.ReleaseFailed;

}
```

In the XML-coded Server Description File, which you process with the `buildjavaserver` command, you identify the name of the Server object.

The `create_servant` method, used in the C++ environment of WLE, is not used in the Java environment. In Java, objects are created dynamically, without prior knowledge of the classes being used.

In the Java environment of WLE, a servant factory is used to retrieve an implementation class, given the interface repository ID. This information is stored in a server descriptor file created by the `buildjavaserver` command for each implementation class.

When a method request is received, and no servant is available for the interface, the servant factory looks up the interface and creates an object of the appropriate implementation class.

This collection of the object's implementation and data compose the run-time, active instance of the CORBA object.

For more information about creating the Server object, see Chapter 2, "Steps for Creating a Java Server Application."

Understanding Object References and Object State

This section presents important background information about the following topics, which have a major influence on how you design and implement WLE server applications:

- Generating object references
- Managing object state
- Reading and writing an object's data stored on disk
- Using design patterns

It is not essential that you read these topics before proceeding to the next chapter; however, this information is located here because it applies broadly to fundamental design and implementation issues for all WLE server applications.

Generating Object References

One of the most basic functions of a WLE server application is providing client applications with object references to the objects they need to execute their business logic. WLE client applications typically get object references to the initial CORBA objects they use from the following two sources:

- The **Bootstrap object**
- **Factories** managed in the WLE domain

Client applications use the Bootstrap object to resolve initial references to a specific set of objects in the WLE domain, such as the FactoryFinder and the SecurityCurrent objects. The Bootstrap object is described in *Getting Started* and in *Creating CORBA Client Applications*.

Factories, however, are designed, implemented, and registered by you, and they provide the means by which client applications get references to objects in the WLE server application, particularly the initial server application object. At its simplest, a factory is a CORBA object that returns an object reference to another CORBA object. The client application typically invokes an operation on a factory to obtain an object reference to a CORBA object of a specific type. Planning and implementing your factories carefully is an important task when developing WLE server applications.

Client applications are able to locate via the FactoryFinder the factories managed by your server application. When you develop the Server object, you typically include code that registers with the FactoryFinder any factories managed by the server application.

It is via this registration operation that the FactoryFinder keeps track of your server application’s factories and can provide object references to them to the client applications that request them. We recommend that you use factories and register them with the FactoryFinder; this model makes it simple for client applications to find the objects in your WLE server application.

Note: In WLE 4.2, references to objects implemented in Java can be created only by factories that are also implemented in Java. You cannot mix and match factories and objects with regards to implementation language.

Managing Object State

Object state management is a fundamentally important concern of large-scale client/server systems, because it is critical that such systems optimize throughput and response time. The majority of high-throughput applications, such as applications you run in a WLE domain, tend to be stateless, meaning that the system flushes state information from memory after a service or an operation has been fulfilled.

Managing state is an integral part of writing CORBA-based server applications. Typically, it is difficult to manage state in these server applications in a way that scales and performs well. The WLE software provides an easy way to manage state and simultaneously ensure scalability and high performance.

The scalability qualities that you can build into a WLE server application help the server application function well in an environment that includes hundreds or thousands of client applications, multiple machines, replicated server processes, and a proportionately greater number of objects and client invocations on those objects.

About Object State

In a WLE domain, **object state** refers specifically to the process, or in-memory, state of an object across client invocations on it. The WLE software uses the following definitions of stateless and stateful objects:

Object	Behavior Characteristics
Stateless	The object is mapped into memory only for the duration of an invocation on one of the object's operations, and is deactivated and has its process state flushed from memory after the invocation is complete; that is, the object's state is not maintained in memory after the invocation is complete.

Object	Behavior Characteristics
Stateful	The object remains activated between invocations on it, and its state is maintained in memory across those invocations. The state remains in memory until a specific event occurs, such as: ■ The server process in which the object exists is stopped or is shut down. ■ The transaction in which the object is participating is either committed or rolled back. ■ The object invokes the `com.beasys.Tobj.TP.deactivateEnable` method on itself and the method completes. Each of these events is discussed in more detail in this section.

Both stateless and stateful objects have data; however, stateful objects may have nonpersistent data in memory that is required to maintain context (state) between operation invocations on those objects. Thus, subsequent invocations on such a stateful object always go to the same servant. Conversely, invocations on a stateless object can be directed by the WLE system to any available server process that can activate the object.

State management also involves how long an object remains active, which has important implications on server performance and the use of machine resources. The section “How to Manage Object State” on page 1-13 explains the various mechanisms the WLE system provides to control object state.

Object state is transparent to the client application. Client applications implement a conversational model of interaction with distributed objects. As long as a client application has an object reference, it assumes that the object is always available for additional requests, and the object appears to be maintained continuously in memory for the duration of the client application interaction with it.

To achieve optimal application performance, you need to carefully plan how your application’s objects manage state. Objects are required to save their state to durable storage, if applicable, before they are deactivated. Objects must also restore their state from durable storage, if applicable, when they are activated. For more information about reading and writing object state information, see the section “Reading and Writing an Object’s Data” on page 1-19.

How to Manage Object State

WLE provides two basic means to control object state:

- By defining **object activation policies** on an object's interface in the Server Description File. Object activation policies are described in the section "Object Activation Policies" on page 1-13.
- By using a TP Framework feature called **application-controlled deactivation**, described in the section "Application-Controlled Deactivation" on page 1-15.

Object Activation Policies

The WLE system provides three object activation policies that you can assign to an object's interface to determine how long an object remains in memory after it has been invoked by a client request. These policies determine whether the object to which they apply is generally stateless or stateful.

The three policies are listed and described in the following table.

Policy	Description
`Method`	Causes the object to be active only for the duration of the invocation on one of the object's operations; that is, the object is activated at the beginning of the invocation, and is deactivated at the end of the invocation. An object with this activation policy is called a *method-bound object*. The `method` activation policy is associated with stateless objects. This activation policy is the default.

Policy	Description
`Transaction`	Causes the object to be activated when an operation is invoked on it. If the object is activated within the scope of a transaction, the object remains active until the transaction is either committed or rolled back. If the object is activated outside the scope of a transaction, its behavior is the same as that of a method-bound object. An object with this activation policy is called a *transaction-bound object.* For more information about object behavior within the scope of a transaction, and general guidelines about using this policy, see Chapter 3, “Integrating Transactions into a Java Server Application.” The `transaction` activation policy is associated with stateful objects for a limited time and under specific circumstances.
`Process`	Causes the object to be activated when an operation is invoked on it, and to be deactivated only under the following circumstances: ■ The server process that manages this object is shut down. ■ An operation on this object invokes the `com.beasys.Tobj.TP.deactivateEnable` method, which causes this object to be deactivated when the method completes. (This is part of a key WLE feature called application-controlled deactivation, which is described in the section “Application-Controlled Deactivation” on page 1-15. An object with this activation policy is called a *process-bound object.* The `process` activation policy is associated with stateful objects.

You determine what events cause an object to be deactivated by assigning object activation policies. For more information about how you assign object activation policies to an object’s interface, see the section “Step 5: Define the object activation and transaction policies.” on page 2-13.

Application-Controlled Deactivation

Application-controlled deactivation provides a means for an application to deactivate an object during run time. The TP Framework provides the `com.beasys.Tobj.TP.deactivateEnable` method, which a process-bound object can invoke on itself. When invoked, the `deactivateEnable` method causes the object in which it exists to be deactivated upon completion of the current client invocation on that object. An object can invoke this method only on itself; you cannot invoke this method on any object but the object in which the invocation is made.

The application-controlled deactivation feature is particularly useful when you want an object to remain in memory for the duration of a limited number of client invocations on it, and you want the client application to be able to tell the object that the client is finished with the object. At this point, the object takes itself out of memory.

Application-controlled deactivation, therefore, allows an object to remain in memory in much the same way that a process-bound object can: the object is activated as a result of a client invocation on it, and it remains in memory after the initial client invocation on it is completed. You can then deactivate the object without having to shut down the server process in which the object exists.

An alternative to application-controlled deactivation is to scope a transaction to maintain a conversation between a client application and an object; however, transactions are inherently more costly, and transactions are generally inappropriate in situations where the duration of the transaction may be indefinite.

A good rule of thumb to use when choosing between application-controlled deactivation and transactions for a conversation is whether there are any disk writing operations involved. If the conversation involves read-only operations, or involves maintaining state only in memory, then application-controlled deactivation is appropriate. If the conversation involves writing data to disk during or at the end of the conversation, transactions may be more appropriate.

Note: If you use application-controlled deactivation to implement a conversational model between a client application and an object managed by the server application, make sure that the object eventually invokes the `com.beasys.Tobj.TP.deactivateEnable` method. Otherwise, the object remains idle in memory indefinitely. (Note that this can be a risk if the client application crashes before the `deactivateEnable` method is invoked. Transactions, on the other hand, implement a time-out mechanism to prevent the situation in which the object remains idle for an indefinite period. This may be another consideration when choosing between the two conversational models.)

You implement application-controlled deactivation in an object using the following procedure:

1. In the implementation file, insert an invocation to the `deactivateEnable` method at the appropriate location within the operation of the interface that uses application-controlled deactivation.
2. In the Server Description File (XML), assign the `process` activation policy to the interface that contains the operation that invokes the `deactivateEnable` method.
3. Build and deploy your application as described in the sections "Step 7: Finish the Server Description File." on page 2-17 and "Step 8: Deploy the server application." on page 2-18.

Choosing Between Stateless and Stateful Objects

In general, you need to balance the costs of implementing stateless objects against the costs of implementing stateful objects.

In the case where the cost to initialize an object with its durable state is expensive (because, for example, the object's data takes up a great deal of space, or the durable state is located on a disk very remote to the servant that activates it), it may make sense to keep the object stateful, even if the object is idle during a conversation. In the case where the cost to keep an object active is expensive in terms of machine resource usage, it may make sense to make such an object stateless.

By managing object state in a way that is efficient and appropriate for your application, you can maximize your application's ability to support large numbers of simultaneous client applications that use large numbers of objects. You generally do this by assigning the `method` activation policy to these objects, which has the effect of deactivating idle object instances so that machine resources can be allocated to other object instances. However, your specific application characteristics and needs may vary.

When You Want Stateless Objects

Stateless objects generally provide good performance and optimal usage of server resources, because server resources are never used when objects are idle. Stateless objects are generally a good approach to implementing server applications. Stateless objects are particularly appropriate in the following situations:

- The client application typically waits for user input between invocations on the object.
- The client request typically contains all the data needed by the server application, and the server can process the client request using only that data.
- The object has very high access rates, but low access rates from any one particular client application.

By making an object stateless, you can generally assure that server application resources are not being tied up for an arbitrarily long time waiting for input from the client application.

Note the following characteristics about an application that employs a stateless object model:

- Information about and associated with an invocation is not maintained after the server application has finished executing a client request.
- An incoming client request is sent to the first available server process. After the request has been satisfied, the application state vanishes and the server application is available for another client application request.
- Durable state information for the object exists outside the server process. With each invocation on this object, the durable state is read into memory.
- The WLE domain may direct successive requests on an object from a given client application to a different server process.
- The overall system performance of a machine that is running stateless objects is usually enhanced.

When You Want Stateful Objects

A stateful object, once activated, remains in memory until a specific event occurs, such as the process in which the object exists is shut down, or the transaction in which the object is activated is completed.

Stateful objects are typically appropriate in the following situations:

- When an object is used very frequently by a large number of client applications. This is the case for long-lived, well-known objects like factories. When the server application keeps these objects active, the client application typically experiences minimal response time in accessing them. Since these active objects are shared by many client applications, there are relatively few objects of this type in memory.

Note: Plan carefully how process objects are potentially involved in a transaction. Any object that is involved in a transaction cannot be invoked by another client application or object. Process objects meant to be used by a large number of client applications can create problems if they are involved in transactions frequently or for long durations.

- When a client application must invoke successive operations on an object to complete a transaction, and the client application is not idle while waiting for user input between those invocations. In this case, if the object were deactivated between invocations, there would be a degradation of response time because state would be written and read between each invocation; such behavior may not be appropriate for transactions. You can trade holding server resources for better response time.

Note the following behavior with stateful objects:

- State information is maintained between server invocations, and the servant typically remains dedicated to a given client application for a specified duration.
- Even though data is sent and received between the client and server applications, the server process maintains additional context or application state information in memory.
- In cases where one or more stateful objects are using a lot of machine resources, server performance for tasks and processes not associated with the stateful object may be worse than with a stateless server model.

For example, if an object has a lock on a database and is caching a lot of data in memory, that database and the memory used by that stateful object are unavailable to other objects, potentially for the entire duration of a transaction.

Reading and Writing an Object's Data

Many of the CORBA objects managed by the server application may have data that is in external storage. This externally stored data may be regarded as the *persistent* or *durable* state of the object. You must address durable state handling at appropriate points in the object implementation for object state management to work correctly.

Because of the wide variety of requirements you may have for your client/server application with regards to reading and writing an object's durable state, the TP Framework cannot automatically handle durable object state on disk. In general, if an object's durable state is modified as a result of one or more client invocations, you must make sure that durable state is saved before the object is deactivated, and you should plan carefully how the object's data is stored or initialized while the object is active.

The sections that follow describe the mechanisms available to you to handle an object's durable state, and give some general advice about how to read and write object state under specific circumstances. The specific topics presented include:

- The available mechanisms for reading and writing an object's durable state
- Reading state at object activation
- Reading state within individual operations on an object
- Stateless objects and durable state
- Stateful objects and durable state
- Your responsibilities for object deactivation
- Avoiding unnecessary I/O

How you choose to read and write durable state invariably depends on the specific requirements of your client/server application, especially with regard to how the data is structured. In general, your priority should be to minimize the number of disk operations, especially where a database controlled by an XA resource manager is involved.

Available Mechanisms for Reading and Writing an Object's Durable State

Table 1-1 and Table 1-2 describe the available mechanisms for reading and writing an object's durable state.

Table 1-1 Available Mechanisms for Reading an Object's Durable State

Mechanism	Description
`com.beasys.Tobj_Servant.activate_object` method	After the TP Framework creates the servant for an object, the TP Framework invokes the `activate_object` method on that servant. This method is defined on the `Tobj_Servant` class, from which all the CORBA objects you define for your client/server application inherit. You may choose not to define and implement the `activate_object` method on your object, in which case nothing happens regarding specific object state handling when the TP Framework activates your object. However, if you define and implement this method, you can choose to include code in this method that reads some or all of an object's durable state into memory. Therefore, the `activate_object` method provides your server application with its first opportunity to read an object's durable state into memory. Note that if an object's OID contains a database key, the `activate_object` method provides the only means the object has to extract that key from the OID. For more information about implementing the `activate_object` method, see "Step 2: Write the methods that implement each interface's operations." on page 2-5.
Operations on the object	You can include inside the individual operations that you define on the object the code that reads an object's durable state.

Table 1-2 Available Mechanisms for Writing an Object's Durable State

Mechanism	Description
`com.beasys.` `Tobj_Servant.` `deactivate_object` method	When an object is being deactivated by the TP Framework, the TP Framework invokes this operation on the object as the final step of object deactivation. As with the `activate_object` method, the `deactivate_object` method is defined on the `Tobj_Servant` class. You implement the `deactivate_object` method on your object optionally if you have specific object state that you want flushed from memory or written to a database. The `deactivate_object` method provides the final opportunity your server application has to write durable state to disk before the object is deactivated. If your object keeps any data in memory, or allocates memory for any purpose, you implement the `deactivate_object` method so your object has a final opportunity to flush that data from memory. Flushing any state from memory before an object is deactivated is critical in avoiding memory leaks.
Operations on the object	As you may have individual operations on the objects that read durable state from disk, you may also have individual operations on the object that write durable state back to disk. For method-bound and process-bound objects in general, you typically perform database write operations within these operations and not in the `deactivate_object` method. For transaction-bound objects, however, writing durable state in the `deactivate_object` method provides a number of object management efficiencies that may make sense for your transactional server applications.

Note: If you use the `deactivate_object` method to write any durable state to disk, any errors that occur while writing to disk are not reported to the client application. Therefore, the only circumstances under which you should write data to disk in this operation is when the object is transaction-bound (that is, it has the `transaction` activation policy assigned to it), or you scope the disk write operations within a transaction by invoking the `TransactionCurrent` object.

Any errors encountered while writing to disk during a transaction can be reported back to the client application. For more information about using the `deactivate_object` method to write object state to disk, see the section "Caveat for State Handling in com.beasys.Tobj_Servant.deactivate_object" on page 2-28.

Reading State at Object Activation

Using the `com.beasys.Tobj_Servant.activate_object` method on an object to read durable state may be appropriate when either of the following conditions exist:

- Object data is always used or updated in all the object's operations.
- All the object's data is capable of being read in one operation.

The advantages of using the `activate_object` method to read durable state include:

- You write code to read data only once, instead of duplicating the code in each of the operations that use that data.
- None of the operations that use an object's data need to perform any reading of that data. In this sense, you can write the operations in a way that is independent of state initialization.

Reading State Within Individual Operations on an Object

With all objects, regardless of activation policy, you can read durable state in each operation that needs that data. That is, you handle the reading of durable state outside the `com.beasys.Tobj_Servant.activate_object` method. Cases where this approach may be appropriate include the following:

- Object state is made up of discrete data elements that require multiple operations to read or write.
- Objects do not always use or update state data at object activation.

For example, consider an object that represents a customer's investment portfolio. The object contains several discrete records for each investment. If a given operation affects only one investment in the portfolio, it may be more efficient to allow that operation to read the one record than to have a general-purpose `activate_object` method that automatically reads in the entire investment portfolio each time the object is invoked.

Stateless Objects and Durable State

In the case of stateless objects -- that is, objects defined with the `method` activation policy -- you must ensure the following:

- That any durable state needed by the request is brought into memory by the time the operation's business logic starts executing
- That any changes to the durable state are written out by the end of the invocation

The TP Framework invokes the `com.beasys.Tobj_Servant.activate_object` method on an object at activation. If an object has an OID that contains a key to the object's durable state on disk, the `activate_object` method provides the only opportunity the object has to retrieve that key from the OID.

If you have a stateless object that you want to be able to participate in a transaction, we generally recommend that if the object writes any durable state to disk that it be done within individual methods on the object. However, if you have a stateless object that is always transactional -- that is, a transaction is always scoped when this object is invoked -- you have the option to handle the database write operations in the `deactivate_object` method, because you have a reliable mechanism in the XA resource manager to commit or roll back database write operations accurately.

Note: Even if your object is method-bound, you may have to take into account the possibility that two server processes are accessing the same disk data at the same time. In this case, you may want to consider a concurrency management technique, the easiest of which is transactions. For more information about transactions and transactional objects, see Chapter 3, "Integrating Transactions into a Java Server Application."

Stateful Objects and Durable State

For stateful objects, you should read and write durable state only at the point where it is needed. This may introduce the following optimizations:

- In the case of process-bound objects, you avoid the situation in which an object allocates a large amount of memory over a long period.
- In the case of transaction-bound objects, you can postpone writing durable state until the `com.beasys.Tobj_Servant.deactivate_object` method is invoked, when the transaction outcome is known.

In general, transaction-bound objects must depend on the XA resource manager to handle all database write or rollback operations automatically.

Note: Data written to external storage that is not managed by an XA resource manager will not be coordinated within the scope of a transaction; if the transaction is rolled back, the data is not rolled back.

For more information about objects and transactions, see Chapter 3, "Integrating Transactions into a Java Server Application."

Your Responsibilities for Object Deactivation

As mentioned in the preceding sections, you implement the `com.beasys.Tobj_Servant.deactivate_object` method to write an object's durable state to disk. You should also implement this operation on an object to flush any remaining object data from memory so that the object's servant can be used to activate another instance of that object. You should not assume that an invocation to an object's `deactivate_object` method also results in an invocation of that object's destructor.

Avoiding Unnecessary I/O

Be careful not to introduce inefficiencies into the application by doing unnecessary I/O in objects. Situations to be aware of include the following:

- If many operations in an object do not use or affect object state, it may be inefficient to read and write state each time these operations are invoked. Design these objects so that they handle state only in the operations that need it; in such cases, you may not want to have all of the object's durable state read in at object activation.
- If object state is made up of data that is read in multiple operations, try to do only the necessary operations at object activation by doing one of the following:
  - Read only the state that is common to all the operations in the `com.beasys.Tobj_Servant.activate_object` method. Defer the reading of additional state to only the operations that require it.
  - Write out only the state that has changed. You can do this by managing flags that indicate the data that was changed during an activation, or by comparing before and after data images.

A general optimization is to initialize a `dirtyState` flag on activation and to write data in the `com.beasys.Tobj_Servant.deactivate_object` method only if the flag has been changed while the object was active.

Sample Activation Walkthrough

For examples of the sequence of activity that takes place when an object is activated, see *Getting Started*.

Using Design Patterns

It is important to structure the business logic of your application around a well-formed design. The WLE software provides a set of design patterns to address this need. A design pattern is simply a structured solution to a specific design problem. The value of a design pattern lies in its ability to be expressed in a form you can reuse and apply to other design problems.

The WLE design patterns are structured solutions to enterprise-class application design problems. You can use them to design successful large-scale client/server applications.

The design patterns summarized here are a guide to using good design practices in WLE client and server applications. They are an important and integral part of designing WLE client and server applications, and the chapters in this book show examples of using these design patterns to implement the Bankapp sample applications.

The Process-Entity design pattern applies to a large segment of enterprise-class client/server applications. This design pattern is referred to as the flyweight pattern in *Object-Oriented Design Patterns*, Gamma et al., and as the Model-View-Controller in other publications.

In this pattern, the client application creates a long-lived process object that the client application interacts with to make requests. For example, in the WLE University sample applications, this object might be the registrar that handles course browsing operations on behalf of the client application. The courses themselves are database entities and are not made visible to the client application.

The advantages of the Process-Entity design pattern include:

- You can achieve the advantages of a fine-grained object model without implementing fine-grained objects. Instead, you use CORBA `struct` datatypes to simulate objects.
- Machine resource usage is optimized because there is only a single object mapped into memory: the process object. By contrast, if each database entity were activated into memory as a separate object instance, the number of objects that would need to be handled could overwhelm the machine's resources quickly in a large-scale deployment.
- Because they are not exposed to the client application, database entities need not be implemented as CORBA objects. Instead, entities can be implemented as local language objects in the server process. This is a fundamental principle of three-tier designs, but it also accurately models the way in which many businesses operate (for example, a registrar at a real university). The individual who serves as the registrar at a university can handle a large course database for multiple students; you do not need an individual registrar for each individual student. Thus, the process object state is distinct from the entity object state.

For complete details on the Process-Entity design pattern, see *Technical Articles* on the Online Documentation CD.

CHAPTER

2 Steps for Creating a Java Server Application

This chapter describes the basic steps involved in creating a WLE Java server application. The steps shown in this chapter are not definitive; there may be other steps you may need to take for your particular server application, and you may want to change the order in which you follow some of these steps. However, the development process for every WLE server application has each of these steps in common.

This chapter presents the following topics:

- Summary of the Java Server Application Development Process
- Development and Debugging Tips

This chapter begins with a summary of the steps, and also lists the development tools and commands used throughout this book. Your particular deployment environment might use additional software development tools, so the tools and commands listed and described in this chapter are also not definitive.

The chapter uses examples from the Bankapp sample application, which is provided with the WLE software. For complete details about the sample application, see the *Guide to the Java Sample Applications*. For complete information about the tools and commands used throughout this book, see the *Java Programming Reference*.

Summary of the Java Server Application Development Process

The basic steps involved in the creation of a server application are summarized in the following table:

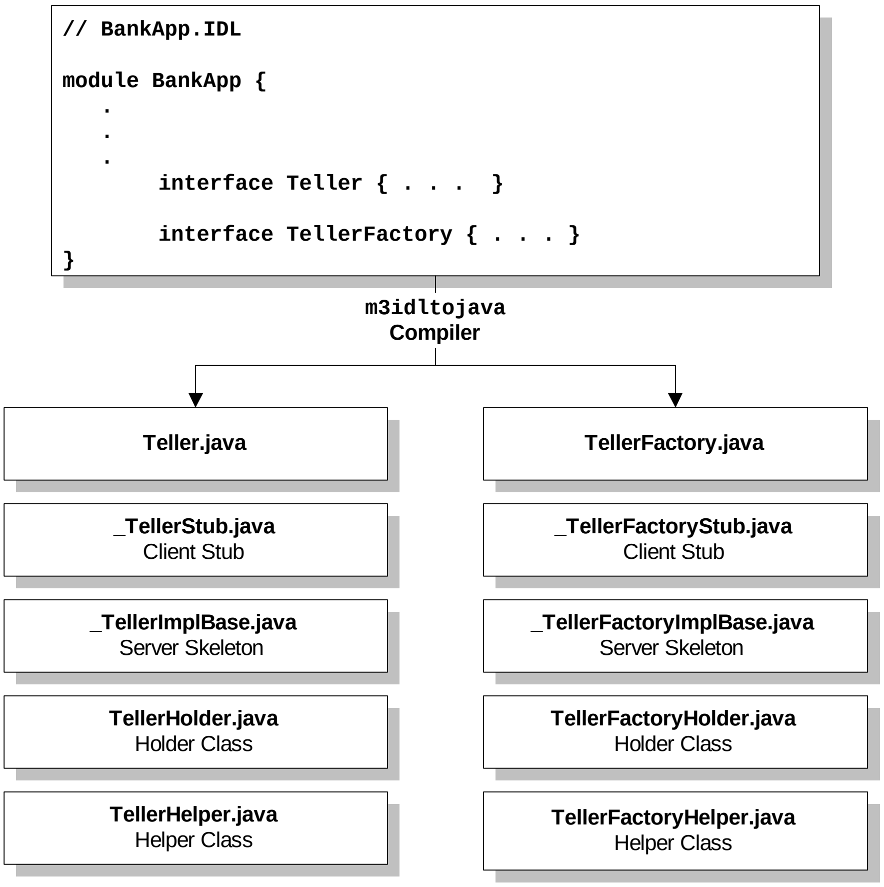
Step 1: Compile the OMG IDL file for the server application.
Step 2: Write the methods that implement each interface's operations.
Step 3: Create the Server object.
Step 4: Compile the Java source files.
Step 5: Define the object activation and transaction policies.
Step 6: Verify the environment variables.
Step 7: Finish the Server Description File.
Step 8: Deploy the server application.

The WLE software also provides the following development tools and commands:

Tool	Description
`m3idltojava`	Compiles your application's OMG IDL file.
`buildjavaserver`	Creates a JAR file containing your Java server class files; also creates a server descriptor file (SER).
`buildXAJS`	For applications that use an XA-compliant resource manager, creates an XA-specific version of the JavaServer.
`tmloadcf`	Creates the `TUXCONFIG` file, a binary file for the WLE domain that specifies the configuration of your server application.
`tmadmin`	Among other things, creates a log of transactional activities, which is used in some of the sample applications.

Step 1: Compile the OMG IDL file for the server application.

The basic structure of the client and server portions of the application that runs in the WLE domain are determined by statements in the application's OMG IDL file. When you compile your application's OMG IDL file, the `m3idltojava` compiler generates many files, some of which are shown in the following diagram:


```
// BankApp.IDL

module BankApp {
   .
   .
   .
        interface Teller { . . .  }

        interface TellerFactory { . . . }
}
```

The preceding diagram shows some of the files generated when the sample `BankApp.IDL` file is compiled by the `m3idltojava` command.

These files are described in Table 2-1.

Note: Do not modify these files.

Table 2-1 Sample Files Produced by the `m3idltojava` Compiler

File	Default Name	Description
Base interface class file	*`interface`*`.java`	Contains an implementation of the interface, written in Java. Copy this file to create a new file and add your business logic to the new file. By convention in our samples and in this document, we name this file `interfaceImpl.java`, substituting the actual name of the interface in the file name. We call this new file an object implementation file.
Client stub file	`_`*`interface`*`Stub.java`	Contains generated code for sending a request.
Server skeleton file	`_`*`interface`*`ImplBase.java`	Contains Java skeletons for each interface specified in the OMG IDL file. The skeleton maps client requests to the appropriate operation in the Java server application during run time.
Holder class file	*`interface`*`Holder.java`	Contains the implementation of the Holder class. The Holder class provides operations for `out` and `inout` arguments, which CORBA has, but which do not map exactly to Java.
Helper class file	*`interface`*`Helper.java`	Contains the implementation of the Helper class. The Helper class provides auxiliary functionality, notably the `narrow` method.

Using the m3idltojava Compiler

To generate the files listed in Table 2-1, enter the following command:

```
m3idltojava [options] idl-filename
```

In the `m3idltojava` command syntax:

- `options` represents one or more command-line options to the IDL compiler. The command-line options are described in the *Java Programming Reference*.
- *`idl-filename`* represents the name of your application's OMG IDL file.

For more information about the `m3idltojava` compiler, including details on the `m3idltojava` command, see the *Java Programming Reference*.

Note: The `m3idltojava` compiler supports all the functionality provided by the `idltojava` compiler from Sun Microsystems, Inc. For more information about the `idltojava` compiler, refer to the following Web site:

```
http://java.sun.com/products/jdk/idl/
```

Step 2: Write the methods that implement each interface's operations.

As the server application programmer, your task is to write the methods that implement the operations for each interface you have defined in your application's OMG IDL file.

The Java object implementation file contains:

- Method declarations for each operation specified in the OMG IDL file
- Your application's business logic
- Constructors for each interface implementation (implementing these is optional)
- Optionally, the `com.beasys.Tobj_Servant.activate_object` and `com.beasys.Tobj_Servant.deactivate_object` methods

  Within the `activate_object` and `deactivate_object` methods, you write code that performs any particular steps related to activating or deactivating an object. This includes reading and writing the object's durable state from and to disk, respectively. For background information on this topic, see the section "Reading and Writing an Object's Data" on page 1-19.

Creating an Object Implementation File

Although you can create your server application's object implementation file manually, you can save time and effort by using the `m3idltojava` compiler to generate a file for each interface. The `*interface*.java` file contains Java signatures for the methods that implement each of the operations defined for your application's interfaces.

To take advantage of this shortcut, use the following steps:

1. Create a copy of the `*interface*.java` file, which was created when you compiled your OMG IDL file with the `m3idltojava` command, and name it `*interface*Impl.java`. For example, using the Bankapp sample file names, you would copy `Teller.java` to a new file named `TellerImpl.java`.

2. Open the new `*interface*Impl.java` file. For example, in the previously unedited `TellerImpl.java` file, we changed:

   ```
   public interface Teller extends org.omg.CORBA.Object {
   ```

   to:

   ```
   public class TellerImpl extends Bankapp._TellerImplBase {
   ```

   `Bankapp._TellerImplBase` is the class defined in the server skeleton file that was generated by the `m3idltojava` compiler for the `Teller` object.

3. For each method in `TellerImpl.java`, we added the `public` keyword. For example, we changed:

   ```
   float deposit(int accountID, float amount)
   ```

   to:

   ```
   public float deposit(int accountID, float amount)
   ```

Repeat this procedure to create `*interface*Impl.java` object implementation files for your interfaces, and add the business logic for your Java server application.

Implementing a Factory Object

As mentioned in the section "How Client Applications Access and Manipulate Your Application's CORBA Objects" on page 1-6, you need to create factories so that client applications can easily locate the objects managed by your server application. A factory is like any other CORBA object that you implement, with the exception that you register it with the FactoryFinder object. Registering a factory is described in the section "Writing the Code That Creates and Registers a Factory" on page 2-10.

The primary function of a factory is to create object references, which it does by invoking the `com.beasys.Tobj.TP.create_object_reference` method. The `create_object_reference` method requires the following input parameters:

- The Interface Repository ID of the object's OMG IDL interface
- The object ID (OID) in string format
- Optionally, routing criteria

For example, in the Bankapp sample application, the `TellerFactory` interface specifies the following operations in the `TellerFactoryImpl.java` file.

Note: In this code fragment, the Import statement appeared earlier in the source file and is not shown here.

```
org.omg.CORBA.Object teller_oref =
       TP.create_object_reference(
       BankApp.TellerHelper.id(),  // Repository ID
       tellerName,                 // Object ID
       null                        // Routing Criteria
       );
```

In the previous code example, notice the following:

- The following parameter specifies the `Teller` object's Interface Repository ID by extracting it from its typecode:

  `BankApp.TellerHelper.id()`

- The `null` parameter specifies that no routing criteria are used, with the result that an object reference created for the `Teller` object is routed to the same group as the `TellerFactory` object that created the object reference.

  For information about specifying routing criteria that affect the group to which object references are routed, see Chapter 4, "Scaling a Java Server Application."

Note: In WLE 4.2, references to objects implemented in Java can be created only by factories that are also implemented in Java. You cannot mix and match factories and objects with regards to implementation language.

Using Threads with WLE

WLE supports the ability to configure multithreaded JavaServers. For each JavaServer, you can establish the maximum number of threads in the application's `UBBCONFIG` file.

For information about the tradeoffs of using single-threaded JavaServers or multithreaded JavaServers, see the section "Enabling Multithreaded JavaServers" on page 4-18. For information about defining the `UBBCONFIG` parameters, see Chapter 3 of the *Administration Guide*.

Step 3: Create the Server object.

In Java, you use a Server object to initialize and release the server application. You implement this Server object by creating a new class that derives from the `com.beasys.Tobj.Server` class and overrides the `initialize` and `release` methods. In the server application code, you can also write a public default constructor.

For example:

```
import com.beasys.Tobj.*;

/**
 * Provides code to initialize and stop the server invocation.
 * BankAppServerImpl is specified in the BankApp.xml input file
 * as the name of the Server object.
 */

public class BankAppServerImpl
       extends com.beasys.Tobj.Server {

       public boolean initialize(string[] args)
              throws com.beasys.TobjS.InitializeFailed;

       public boolean release()
              throws com.beasys.TobjS.ReleaseFailed;

}
```

In the XML-coded Server Description File, which you process with the `buildjavaserver` command, you identify the name of the Server object.

The `create_servant` method, used in the C++ environment of WLE, is not used in the Java environment. In Java, objects are created dynamically, without prior knowledge of the classes being used. In the Java environment of WLE, a servant factory is used to retrieve an implementation class, given the interface repository ID. This information is stored in a server descriptor file created by the `buildjavaserver` command for each implementation class. When a method request is received, and no servant is available for the interface, the servant factory looks up the interface and creates an object of the appropriate implementation class.

This collection of the object's implementation and data compose the run-time, active instance of the CORBA object.

When your Java server application starts, the TP Framework creates the Server object specified in the XML file. Then, the TP Framework invokes the `initialize` method. If the method returns true, the server application starts. If the method throws the `com.beasys.TobjS.InitializeFailed` exception, or returns `false`, the server application does not start.

When the server application shuts down, the TP Framework invokes the `release` method on the Server object.

Any command-line options specified in the `CLOPT` parameter for your specific server application in the `SERVERS` section of the WLE domain's `UBBCONFIG` file are passed to the `public boolean initialize(string[] args)` operation as `args`. For more information about passing arguments to the server application, see the *Administration Guide*. For examples of passing arguments to the server application, see the *Guide to the Java Sample Applications*.

Within the `initialize` method, you can include code that does the following, if applicable:

- Creates and registers factories
- Allocates any machine resources
- Initializes any global variables needed by the server application
- Opens the databases used by the server application
- Opens the XA resource manager

Writing the Code That Creates and Registers a Factory

If your server application manages a factory that you want client applications to be able to locate easily, you need to write the code that registers that factory with the FactoryFinder object, which is invoked typically as the final step of the server application initialization process.

To write the code that registers a factory managed by your server application, do the following:

1. Create an object reference to the factory.

   This step involves creating an object reference as described in the section “Implementing a Factory Object” on page 2-6. In this step, you include an invocation to the `com.beasys.Tobj.TP.create_object_reference` method, specifying the Interface Repository ID of the factory’s OMG IDL interface.The following Bankapp example, from the `BankAppServerImpl.java` file, creates an object reference, represented by the variable `fact_oref`, to the `TellerFactory` factory:

```
// Save the Teller factory name.
tellerFName = new String(args[0]);

// Create the Teller factory object reference.

fact_oref = TP.create_object_reference(
       BankApp.TellerFactoryHelper.id(),  // factory Repository id
       tellerFName,                       // object id
       null                               // no routing criteria
       );
```

2. Register the factory with the WLE domain.

   This step involves invoking the following operation for each of the factories managed by the server application:

```
// Register the factory reference with the factory finder.

TP.register_factory(
       fact_oref,                 // factory object reference
       tellerFName                // factory name
       );
```

   The `com.beasys.Tobj.TP.register_factory` method registers the server application’s factories with the FactoryFinder object. This operation requires the following input parameters:

- The object reference for the factory, created in step 1 above.
- A string identifier, which in the Bankapp example is based on the Teller factory name that is specified as a command-line option in the CLOPT parameter for the Bankapp server application. This string is used in the call to the `com.beasys.Tobj.TP.unregister_factory` method. It is also used in the invocation of the `find_one_factory_by_id` method that is called by clients of this interface.

Releasing the Server Application

When the WLE system administrator enters the `tmshutdown` command, the TP Framework invokes the following operation in the Server object of each running server application in the WLE domain:

```
public void release()
```

Within the `release()` operation, you may perform any application-specific cleanup tasks that are specific to the server application, such as:

- Unregistering object factories managed by the server application
- Deallocating resources
- Closing any databases
- Closing an XA resource manager

Once a server application receives a request to shut down, the server application can no longer receive requests from other remote objects. This has implications on the order in which server applications should be shut down, which is an administrative task. For example, do not shut down one server process if a second server process contains an invocation in its `release()` operation to the first server process.

During server shutdown, you may want to include an invocation to unregister each of the server application’s factories. For example, the following example is from the `BankAppServerImpl.java` file:

```
// Unregister the factory.
// Use a try block since cleanup code shouldn't throw exceptions.

try {
       TP.unregister_factory(
                fact_oref,              // factory object reference
                TellerFName             // factory interface id
         );

} catch (Exception e){
       TP.userlog("Couldn't unregister the TellerFactory: " +
       e.getMessage());
       e.printStackTrace();
}
```

The invocation of the `com.beasys.Tobj.TP.unregister_factory` method should be one of the first actions in the `release()` implementation. The `unregister_factory` method unregisters the server application's factories. This operation requires the following input arguments:

- The object reference for the factory
- A string identifier, which in the Bankapp sample is based on the Teller factory name that is specified as a command-line option in the CLOPT parameter for the Bankapp server application

Step 4: Compile the Java source files.

After you have implemented your application's objects and the Server object, use the `javac` compiler to create the bytecodes for all the class files that comprise your application. This set of files includes the `*.java` source files generated by the `m3idltojava` compiler, plus the object implementation files and server class file that you created.

Step 5: Define the object activation and transaction policies.

As stated in the section “Managing Object State” on page 1-11, you determine what events cause an object to be deactivated by assigning object activation policies, transaction policies, and, optionally, using the application-controlled deactivation feature.

You specify default object activation and transaction policies in the Server Description File, which is expressed in XML, and you implement application-controlled deactivation via the `com.beasys.Tobj.TP.deactivateEnable` method in your Java code. This section explains how you implement one of the mechanisms, using the Bankapp WLE sample application as an example.

Specifying Policies in XML

The WLE software supports the following activation policies, described in “Object Activation Policies” on page 1-13:

Activation Policy	Description
`method`	Causes the object to be active only for the duration of the invocation on one of the object’s operations.
`transaction`	Causes the object to be activated when an operation is invoked on it. If the object is activated within the scope of a transaction, the object remains active until the transaction is either committed or rolled back.
`process`	Causes the object to be activated when an operation is invoked on it, and to be deactivated only when one of the following occurs: ■ The process in which the server application exists is shut down. ■ The object has invoked the `com.beasys.Tobj.TP.deactivateEnable` method on itself.

The WLE software also supports the following transaction policies, described in Chapter 3, "Integrating Transactions into a Java Server Application":

Transaction Policy	**Description**
`always`	When an operation on this object is invoked, this policy causes the TP Framework to begin a transaction for this object, if there is not already an active transaction. If the TP Framework starts the transaction, the TP Framework commits the transaction if the operation completes successfully, or rolls back the transaction if the operation raises an exception. If `always` is specified, the `AUTOTRAN` parameter in the application's `UBBCONFIG` file is ignored.
`optional`	The implementation may be transactional. Objects can be invoked either inside or outside the scope of a transaction. 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. `Optional` is the default transaction policy.
`never`	Causes the TP Framework to generate an error condition if this object is invoked during a transaction. If `never` is specified, the `AUTOTRAN` parameter in the application's `UBBCONFIG` file is ignored.
`ignore`	If a transaction is currently active when an operation on this object is invoked, the transaction is suspended until the operation invocation is complete. This transaction policy prevents any transaction from being propagated to the object to which this transaction policy has been assigned. If `ignore` is specified, the `AUTOTRAN` parameter in the application's `UBBCONFIG` file is ignored.

To assign these policies to the objects in your application, create the Server Description File, which is written in the Extensible Markup Language (XML). Specify the activation policies for each of your application's interfaces.

Note: For information about the XML tags used with the WLE Server Description File, see the *CORBA Java Programming Reference*.

The following example shows a portion of the `BankApp.xml` file that was created for the WLE Bankapp sample application. Notice that there are no default policy settings in the XML file; the policies are explicitly assigned.

```
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">

<M3-SERVER
server-implementation="com.beasys.samples.BankAppServerImpl"
       server-descriptor-name="BankApp.ser">

       <MODULE name="com.beasys.samples">
         <IMPLEMENTATION
              name="TellerFactoryImpl"
              activation="process"
              transaction="never"
         />

         <IMPLEMENTATION
              name="TellerImpl"
              activation="method"
              transaction="never"
         />

         <IMPLEMENTATION
              name="DBAccessImpl"
              activation="method"
              transaction="never"
         />
       </MODULE>
   .
   .
   .
</M3-SERVER>
```

Step 6: Verify the environment variables.

Several environment variables are defined by the WLE software when the product is installed, but it is always a good idea to verify the following key environment variables prior to the `buildjavaserver` compilation step. The environment variables are:

- `JAVA_HOME`, the directory where the JDK is installed

- `CLASSPATH`, which must point to:
  - The location of the WLE JAR archive, which contains all the class files
  - The location of the WLE message catalogs
- `TUXDIR`, the directory where the WLE software is installed

To verify whether an environment variable has been set, you can use the `echo` command, as shown in the following examples:

On Windows NT systems:

```
echo %JAVA_HOME%
```

On Solaris systems:

```
echo $JAVA_HOME
```

If you discover that required WLE system variables are not set on your system, you can set them as shown in the following examples.

On Windows NT systems:

```
set JAVA_HOME=c:\jdk1.2

set CLASSPATH=.;%TUXDIR%\udataobj\java\jdk\m3.jar;%TUXDIR%\locale\java\M3

set PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;%JAVA_HOME%\jre\bin\classic;
%TUXDIR%\lib;%TUXDIR%\bin;%PATH%
```

On Solaris systems:

```
JAVA_HOME=/usr/kits/jdk1.2

CLASSPATH=.:$TUXDIR/udataobj/java/jdk/M3.jar:$TUXDIR/locale/java/M3

PATH=$JAVA_HOME/bin:$TUXDIR/bin:$PATH

LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/sparc/native_threads:
$JAVA_HOME/jre/lib/sparc/classic:$JAVA_HOME/jre/lib/sparc:$TUXDIR/lib

THREADS_FLAG=native

export JAVA_HOME CLASSPATH PATH LD_LIBRARY_PATH THREADS_FLAG
```

Note that during the deployment step, you must also define the environment variables APPDIR and TUXCONFIG. These variables are described in subsequent sections of this chapter.

Step 7: Finish the Server Description File.

After you have compiled the Java source code and defined the environment variables, enter additional information in the XML-based Server Description File, and then supply the Server Description File as input to the buildjavaserver command.

Edit your Server Description File to identify the Server object and the name of the file that will contain your Java application's server descriptor. This portion of the XML file is called the server declaration; its location in the file is immediately after the prolog. The required prolog contains the following two lines:

```
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">
```

Note: The DTD file type stands for Document Type Definition. In XML, a DTD file is used to specify software descriptions or to format documents. The m3.dtd file is supplied by the WLE system and specifies the set of elements (or tags, such as <IMPLEMENTATION>) that are parsed by the buildjavaserver compiler. The compiler understands the attributes attached to each element, and which elements can be used with another element.

The server declaration used in the sample BankApp.xml file is as follows:

```
<M3-SERVER
    server-implementation="com.beasys.samples.BankAppServerImpl"
    server-descriptor-name="BankApp.ser">
```

In the XML file for your Java server application, you can also include elements that will cause buildjavaserver to create a Java Archive (JAR) file. This section of the XML file is optional, because you could use the JAR command to assemble your application's classes into a JAR file. However, the <ARCHIVE> element provides help by simplifying the process of collecting the files.

For example, the BankApp.XML file contains the following elements:

```
<ARCHIVE name="BankApp.jar">
       <PACKAGE-RECURSIVE name="com.beasys.samples"/>
</ARCHIVE>
```

The archive element must be the last element inside the <`M3-SERVER`> element. It must be located after all the modules and implementations.

If the XML file contains instructions to create an archive, both the class specified by `server_name` and the file specified by `server_descriptor` are stored in the archive. The `server_descriptor` file is inserted in the archive manifest with the `M3-Server` tag; this insertion makes the server descriptor the entry point during server execution.

If you do not include the archive element, the `buildjavaserver` command generates only the server descriptor and writes it in the file specified in the `server-descriptor-name` attribute of the `M3-SERVER` element.

For more information about the elements and options in the XML-based Server Description File, see the *Java Programming Reference*.

When you have completed your edit to the Server Description File, you are ready to use the `buildjavaserver` command. (This step assumes that you have already defined the environment variables that are identified in the section “Step 6: Verify the environment variables.” on page 2-15.)

The `buildjavaserver` command has the following format:

```
buildjavaserver [-s searchpath] input_file.xml
```

In the `buildjavaserver` command syntax:

- `-s` *`searchpath`* is used to locate the classes and packages when building the archive. If this optional value is not specified, it defaults to the value of the `CLASSPATH` environment variable.
- `input_file` is the name of the XML Server Description File.

Step 8: Deploy the server application.

You or the system administrator deploy the WLE server application by using the procedure summarized in this section. For complete details on building and deploying the WLE Bankapp sample application, see the *Guide to the Java Sample Applications*.

To deploy the server application:

1. Place the server application JAR file in the directory listed in `APPDIR`. On NT systems, this directory must be on a local drive (not a networked drive). On Solaris, the directory can be local or remote.

2. If your Java server application uses an XA-compliant resource manager such as Oracle, you must build an XA-specific version of the JavaServer by using the `buildXAJS` command at a system prompt. Provide as input to the command the resource manager that is associated with the server. In your application's `UBBCONFIG` file, you also must use the `JavaServerXA` element in place of `JavaServer` to associate the XA resource manager with a specified server group. See the *Java Programming Reference* for details about the `buildXAJS` command.

3. Create the application's configuration file, also known as the `UBBCONFIG` file, in a text editor. Include the parameters to start `JavaServer` or `JavaServerXA`. For example:

```
*SERVERS
   .
   .
   .
      JavaServer
          SRVGRP = BANK_GROUP2
          SRVID = 8
          CLOPT = "-A -- -M 10 BankApp.jar TellerFactory_1"
          SYSTEM_ACCESS=FASTPATH
          RESTART = N
```

   Note: There is a strict order to starting servers in WLE Java. Also, you can specify a fully qualified path to the location of the JAR file; or, JavaServer looks for the application's JAR file in the value for the `APPDIR` environment variable. See Chapter 3 of the *Administration Guide* for `UBBCONFIG` file details.

4. Set the following additional environment variables on the machine from which you are booting the WLE server application:

   - `TUXCONFIG`, which must match the `TUXCONFIG` entry in the `UBBCONFIG` file. This variable represents the location or path of the binary version of the application's `UBBCONFIG` file.

   - `APPDIR`, which represents the directory in which the application's executable file exists.

5. If you have not already done so, set the `TUXDIR` environment variable on all machines that are running in the WLE domain or that are connected to the WLE domain. This environment variable points to the location where the WLE software is installed.

6. Enter the following command to create the `TUXCONFIG` file:

   ```
   prompt> tmloadcf -y application-ubbconfig-file
   ```

   The command-line argument *`application`*`-ubbconfig-`*`file`* represents the name of your application's `UBBCONFIG` file. Note that you may need to remove any old `TUXCONFIG` files to execute this command.

7. Enter the following command to start the WLE server application:

   ```
   prompt> tmboot -y
   ```

   You can reboot a server application without reloading the `UBBCONFIG` file.

For complete details about configuring the JDBC Bankapp and XA Bankapp sample applications, see the *Guide to the Java Sample Applications*. For complete details on creating the `UBBCONFIG` file for WLE applications, see the *Administration Guide*.

Development and Debugging Tips

The following topics are discussed in this section:

- Use of CORBA and WLE exceptions and the user log
- Detecting error conditions in the callback methods
- Common pitfalls of OMG IDL interface versioning and modification
- Caveat for state handling in the `com.beasys.Tobj_Servant.deactivate_object` method

Use of CORBA and WLE Exceptions and the User Log

This section discusses the following topics:

- The client application view of exceptions
- The server application view of exceptions

Client Application View of Exceptions

When a client application invokes an operation on a CORBA object, an exception may be returned as a result of the invocation. The only valid exceptions that can be returned to a client application are the following:

- Standard CORBA-defined exceptions that are known to every CORBA-compliant ORB
- Exceptions that are defined in OMG IDL and known to the client application via either its stub or the Interface Repository

The WLE system works to ensure that these CORBA-defined restrictions are not violated, which is described in the section “Server Application View of Exceptions” on page 2-22.

Because the set of exceptions exposed to the client application is limited, client applications may occasionally catch exceptions for which the cause is ambiguous. Whenever possible, the WLE system supplements such exceptions with descriptive messages in the user log, which serves as an aid in detecting and debugging error conditions. These cases are described in the following section.

Server Application View of Exceptions

This section presents the following topics:

- Exceptions raised by the WLE system that can be caught by application code
- The WLE system's handling of exceptions raised by application code during the invocation of operations on CORBA objects

Exceptions Raised by the WLE System that Can Be Caught by Application Code

The WLE system may return the following types of exceptions to an application when operations on the TP object are invoked:

- CORBA-defined system exceptions
- CORBA `UserExceptions` defined in the file `TobjS.idl`

  The OMG IDL code for the exceptions is as follows.

Note: This code fragment is from an IDL file that is not distributed with WLE systems. A separate file that shares the name `TobjS.idl` is distributed with WLE systems. The two files are slightly different.

```
#ifndef _OBJTM_TOBJS_IDL
#define _OBJTM_TOBJS_IDL

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

module TobjS {

    // Enumerations

    enum DeactivateReasonValue {
          DR_METHOD_END,
          DR_SERVER_SHUTDOWN,
          DR_TRANS_COMMITTING,
          DR_TRANS_ABORTED
};
// Exceptions
          exception ActivateObjectFailed { string reason; };
          exception ApplicationProblem { };
          exception CannotProceed { };
          exception CreateServantFailed { string reason; };
          exception DeactivateObjectFailed { string reason; };
          exception IllegalInterface { };
          exception IllegalOperation { };
          exception InitializeFailed { string reason; };
          exception InvalidDomain { };
          exception InvalidInterface { };
          exception InvalidName { };
          exception InvalidObject { };
          exception InvalidObjectId { };
          exception InvalidServant { };
          exception NilObject { string reason; };
          exception NoSuchElement { };
          exception NotFound { };
          exception OrbProblem { };
          exception OutOfMemory { };
          exception OverFlow { };
          exception RegistrarNotAvailable { };
          exception ReleaseFailed { string reason; };
          exception UnknownInterface { };
};
#endif /* _OBJTM_TOBJS_IDL */
```

The WLE System's Handling of Exceptions Raised by Application Code during the Invocation of Operations on CORBA Objects

A server application can raise exceptions in the following places in the course of servicing a client invocation:

- In the `com.beasys.Tobj_Servant.activate_object` and `com.beasys.Tobj_Servant.deactivate_object` callback methods
- In the implementation code for the invoked operation

It is possible for the server application to raise any of the following types of exceptions:

- A CORBA-defined system exception
- A CORBA user-defined exception defined in OMG IDL
- A CORBA user-defined exception defined for WLE

The following exceptions are intended to be used in server applications to help the WLE system send messages to the user log, which can help with troubleshooting:

```
interface TobjS {
          exception ActivateObjectFailed { string reason; };
          exception DeactivateObjectFailed { string reason; };
          exception InitializeFailed { string reason; };
          exception ReleaseFailed { string reason; };
}
```

- Any other Java exception type

All exceptions raised by server application code that are not caught by the server application are caught by the WLE system. When these exceptions are caught, one of the following occurs:

- The exception is returned to the client application without alteration.
- The exception is converted to a standard CORBA exception, which is then returned to the client application.
- The exception is converted to a standard CORBA exception, and the following actions occur:
  - The exception is returned to the client application.
  - One or more messages containing descriptive information about the error are sent to the user log. The descriptive information may originate from either the server application code or from the WLE system.

The following sections show how the WLE system handles exceptions raised by the server application during the course of a client invocation on a CORBA object.

Exceptions raised in the `com.beasys.Tobj_Servant.activate_object` method

If any exception is raised in the `activate_object` method:

- The `org.omg.CORBA.OBJECT_NOT_EXIST` exception is returned to the client application.
- If the exception raised is `com.beasys.TobjS.ActivateObjectFailed`, a message is sent to the user log. If a reason string is supplied in the constructor for the exception, the reason string is also written as part of the message.
- Neither the operation requested by the client nor the `com.beasys.Tobj_Servant.deactivate_object` method is invoked.

Exceptions raised in operation implementations

The WLE system requires operation implementations to throw either CORBA system exceptions, or user-defined exceptions defined in OMG IDL that are known to the client application. If these types of exceptions are thrown by operation implementations, then the WLE system returns them to the client application, unless one of the following conditions exists:

- The object has the `always` transaction policy, and the WLE system automatically started a transaction when the object was invoked. In this case, the transaction is automatically rolled back by the WLE system. Because the client application is unaware of the transaction, the WLE system then raises the `org.omg.CORBA.OBJ_ADAPTER` CORBA system exception, and not the `org.omg.CORBA.TRANSACTION_ROLLEDBACK` exception, which would have been the case had the client initiated the transaction.

- The exception is defined in the file `TobjS.idl.` In this case, the exception is converted to the `org.omg.CORBA.BAD_OPERATION` exception and `BAD_OPERATION` is returned to the client application. In addition, the following message is sent to the user log:

  ```
  "WARN: Application didn't catch TobjS exception. TP Framework
  throwing org.omg.CORBA.BAD_OPERATION."
  ```

  If the exception is `com.beasys.TobjS.IllegalOperation`, the following supplementary message is written to warn the programmer of a possible coding error in the application:

  ```
  "WARN: Application called com.beasys.Tobj.TP.deactivateEnable()
  illegally and didn't catch TobjS exception."
  ```

  This can occur if the `com.beasys.Tobj.TP.deactivateEnable` method is invoked inside an object that has the `transaction` activation policy. (Application-controlled deactivation is not supported for transaction-bound objects.)

- The WLE system raised an internal system exception following the client invocation. In this case, the `org.omg.CORBA.INTERNAL` exception is returned to the client.

Exceptions raised in the `com.beasys.Tobj_Servant.deactivate_object` method

If any exception is raised in the `deactivate_object` method, the following occurs:

- The exception is not returned to the client application.

- If the exception raised is `com.beasys.TobjS.DectivateObjectFailed`, a message is sent to the user log. If a reason string is supplied in the constructor for the exception, the reason string is also written as part of the message.
- A message is sent to the user log for exceptions other than the `TobjS.DeactivateObjectFailed` exception, indicating the type of exception caught by the WLE system.

Detecting Error Conditions in the Callback Methods

The WLE system provides a set of predefined exceptions that allow you to specify message strings that the TP Framework writes to the user log if application code gets an error in any of the following callback methods:

- `com.beasys.Tobj_Servant.activate_object`
- `com.beasys.Tobj_Servant.deactivate_object`
- `com.beasys.Tobj.Server.initialize`
- `com.beasys.Tobj.Server.release`

You can use these exceptions as a useful debugging aid that allows you to send unambiguous information about why an exception is being raised. Note that the TP Framework writes these messages to the user log only. They are not returned to the client application.

You specify these messages with the following exceptions, which have an optional reason string:

Exception	Callback Methods That Can Raise This Exception
`ActivateObjectFailed`	`com.beasys.Tobj_Servant.activate_object`
`DeactivateObjectFailed`	`com.beasys.Tobj_Servant.deactivate_object`
`InitializeFailed`	`com.beasys.Tobj.Server.initialize`
`ReleaseFailed`	`com.beasys.Tobj.Server.release`

To send a message string to the user log, specify the string in the exception, as in the following example:

```
throw new InitializeFailed("Unable to Initialize Bankapp server");
```

Note the following:

- When you throw these exceptions, the reason string parameter is optional. If you do not need to specify a message string, omit the string parameter, as in the following example:

  ```
  throw new com.beasys.TobjS.ActivateObjectFailed();
  ```

- If you choose to use the `InitializedFailed` exception in your code, be sure to either fully qualify that object or include the following import declaration prior to the `InitializeFailed` exception:

  ```
  import com.beasys.TobjS.*;
  ```

Common Pitfalls of OMG IDL Interface Versioning and Modification

An object is instantiated based on its Interface Repository ID. It is crucial that this interface ID is the same as the one supplied in the factory when the factory invokes the `com.beasys.Tobj.TP.create_object_reference` method.

It is possible for this condition to arise if, during the course of development, different versions of the interface are being developed or many modifications are being made to the IDL file. Even if you typically use the *`interface`*`Helper.id` method to specify the interface repository ID, it is possible for a mismatch to occur.

If the interface IDs do not match, the following message is placed in the user log (`ULOG`) and the `create_object_reference` method returns a null object reference:

```
IJTPFW_CAT:38: ERROR: TP.create.object.reference() could not create
object reference for: Interface = Interface-ID OID= oid-number
```

Caveat for State Handling in com.beasys.Tobj_Servant.deactivate_object

The `deactivate_object` method is invoked when the activation boundary for an object is reached. You may, optionally, write durable state to disk in the implementation of this operation. It is important to understand that exceptions raised in this operation are not returned to the client application. The client application will be unaware of any error conditions raised in this operation unless the object is participating in a transaction. Therefore, in cases where it is important that the client application know whether the writing of state via this operation is successful, we recommend that transactions be used.

If you decide to use the `deactivate_object` method for writing state, and the client application needs to know the outcome of the write operations, we recommend that you do the following:

- Ensure that each operation that affects object state is invoked within a transaction, and that deactivation occurs within the transaction boundaries. This can be done by using either the `method` or `transaction` activation policies, and is possible with the `process` activation policy if the `com.beasys.Tobj.TP.deactivateEnable` method is invoked within the transaction boundary.
- If an error occurs during the writing of object state, invoke the `org.omg.CosTransactions.Current.rollback_only` method to ensure that the transaction is rolled back. One of the following actions is taken:
  - If there is no transaction associated with the client thread, the `OBJ_ADAPTER` 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.

If transactions are not used, we recommend that you write object state within the scope of individual operations on the object, rather than via the `deactivate_object` method. This way, if an error occurs, the operation can raise an exception that is returned to the client application.

3 Integrating Transactions into a Java Server Application

This chapter describes how to integrate transactions into a WLE server application and covers the following topics:

- Overview of Transactions in the WLE System
- Integrating Transactions in a WLE Client and Server Application
- Transactions and Object State Management
- Notes on Using Transactions in the WLE System

Overview of Transactions in the WLE System

The WLE system provides transactions as 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.

That is, you have a requirement to perform multiple write operations on durable storage, and you must be guaranteed that the operations succeed; if any one of the operations fails, the entire set of operations is rolled back.

Transactions typically are appropriate in the situations described in the following list. Each situation encapsulates a transactional model supported by the WLE system.

- The client application needs to make invocations on several different 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 so far. For example, if the client application cannot book a flight from Los Angeles to Honolulu on a given date, the client application needs to cancel the flight reservations made up to that point.

- The client needs a conversation with an object managed by the server application, and the client 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 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 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 application. The user of the client application browses through an online catalog and makes multiple purchase selections. When the user is done choosing all the items he or she wants to buy, the user clicks on a button to make the purchase, where the user may enter credit card information. If the credit card check fails (for example, the user cannot provide valid credit card information) the shopping application needs a way to cancel all the pending purchase selections 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. (And in this situation, the individual database edits are not necessarily CORBA 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.

Integrating Transactions in a WLE Client and Server Application

The WLE system supports two transaction API models:

- The Java Transaction Service defined by Sun Microsystems—This service is the Java mapping of the Object Transaction Service (OTS) that is specified as part of CORBA: `org.omg.CosTransactions.Current.`
- The Java Transaction API defined by Sun Microsystems—Only the application-level transaction demarcation interface is supported: `javax.transaction.UserTransaction.`

In this document, we refer generically to these mappings as the TransactionCurrent object. For specifics about `org.omg.CosTransactions.Current` and `javax.transaction.UserTransaction`, see the *WLE Javadoc* and the *CORBA Java Programming Reference*.

The WLE system 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 *Creating 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 the `rollback_only` method 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 and passing a reason value.)

  When an object is polled, the object may veto the current transaction by invoking the `rollback_only` method 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 vetos the current transaction, the transaction is committed.

The following sections explain how you can use object activation policies and transaction policies to get 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. For more information about database connections, see the section "Opening an XA Resource Manager" on page 3-9.

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.

Assigning the `always` transactional policy to an object's interface is appropriate 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:

Activation Policy	Transaction Policy
`process`, `method`, or `transaction`	`always`

Note: Database cursors cannot span transactions. For an example, see *Creating C++ Server Applications*.

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 for that object's interface, to make an object optionally transactional:

Activation Policy	Transaction Policy
`process`, `method`, or `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: Some XA resource managers used in the WLE system require that any object participating in a transaction scope their database read operations, in addition to write operations, within a transaction. (However, you can still scope your own transactions.) For example, using the Oracle7 TMS with the WLE system has this requirement. 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.

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.

This transaction policy is appropriate for objects that write durable state to disk that cannot be rolled back; for example, 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:

Activation Policy	**Transaction Policy**
`process` or `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:

Activation Policy	Transaction Policy
`process` or `method`	`ignore`

Assigning Policies

For information about how to create a Server Description File and specify policies on objects, see the section "Step 5: Define the object activation and transaction policies." on page 2-13.

Using an XA Resource Manager

The XA Bankapp sample application in the `drive:\M3dir\samples\corba\bankapp_java\XA` directory uses the Oracle7 Transaction Manager Server (TMS) as an example of a relational database management service (RDBMS). TMS 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 (for example, 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.
- 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

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.

Closing an XA Resource Manager

If your Server object's `com.beasys.Tobj.Server.initialize` method opens an XA resource manager, you must include the following invocation in the `com.beasys.Tobj.Server.release` method:

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

Transactions and Object State Management

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.

The following sections address some key points regarding transactions and object state management.

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. Transactional objects can always delegate the commit and rollback responsibilities to the XA resource manager, which greatly eases 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, 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.

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

- You want the object to write its durable 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 the `rollback_only` method on the TransactionCurrent object. Note that if you do make an invocation to the `rollback_only` method from within the `com.beasys.Tobj_Servant.deactivate_object` method, the `deactivate_object` method is not invoked again.

- 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.

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:

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. The only valid invocations transaction-bound objects can make inside the `deactivate_object` method 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. For more information about opening and closing an XA resource manager, see the sections "Opening an XA Resource Manager" on page 3-9 and "Closing an XA Resource Manager" on page 3-10.

Notes on Using Transactions in the WLE System

Note the following about integrating transactions into your WLE client/server applications:

- The following transactions are not permitted in the WLE system:
  - Nested transactions

    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.)
  - Recursive transactions

    A transactional object cannot call a second object, which in turn calls the first object.
- 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.
- Objects can be involved with only one transaction at one time. The WLE system does not support concurrent transactions.

- The WLE system does not queue requests to objects that are currently involved in a transaction. If a nontransactional client application attempts to invoke an operation on an object that is currently in a transaction, the client application receives the following error message:

  `org.omg.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:

  `org.omg.CORBA.INVALID_TRANSACTION`

- For transaction-bound objects, you might consider doing all state handling in the `com.beasys.Tobj_Servant.deactivate_object` method. This makes it easier for the object to handle its state properly, since the outcome of the transaction is known at the time that the `com.beasys.Tobj_Servant.deactivate_object` method is invoked.

- For method-bound objects that have several operations, but only a few that affect the object's durable state, you may want to consider 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 an exception is raised inside an operation on that object, the client application receives an `OBJ_ADAPTER` exception. In this situation, 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 and returns a CORBA exception, the client application receives only a transaction rollback exception but not the CORBA exception.

CHAPTER

4 Scaling a Java Server Application

This chapter shows how you can take advantage of several key scalability features of the WLE system. The descriptions demonstrate scalability features that achieve the following goals:

- Adding parallel processing capability, enabling the WLE domain to process multiple client requests simultaneously
- Spreading the processing load on the server applications in the Bankapp sample application across multiple machines

Some of the Bankapp examples in this chapter include sample code that is not implemented in the product sample's Bankapp files.

This chapter discusses the following topics:

- Overview of the Scalability Features Available in the WLE System
- Scaling a WLE Server Application. This section includes the following topics:
  - Replicating Server Processes and Server Groups
  - Scaling the Application Via Object State Management
  - Factory-based Routing
  - Enabling Multithreaded JavaServers
- How the Bankapp Server Application Can Be Scaled Further

Overview of the Scalability Features Available in the WLE System

Supporting highly scalable applications is one of the strengths of the WLE system. Many applications may perform well in an environment characterized by 1 to 10 server processes, and 10 to 100 client applications. However, in an enterprise environment, applications need to support:

- Hundreds of execution contexts, where the context can be a thread or a process
- Tens of thousands of client applications
- Millions of objects

Deploying a Java application with such demands quickly reveals the resource shortcomings and performance bottlenecks in your application. The WLE system supports such large-scale deployments in several ways, including:

- Replicated server processes and server groups
- Object state management
- Factory-based routing
- Multithreaded JavaServers (appropriate for certain types of applications and processing environments, as outlined in the section “Enabling Multithreaded JavaServers” on page 4-18)

Other features provided in the WLE system to make an application highly scalable include the IIOP Listener/Handler, which is summarized in *Getting Started* and described fully in the *Administration Guide*.

Scaling a WLE Server Application

Using the JDBC Bankapp sample application as an example, this section explains how to scale an application to meet a significantly greater processing capability. The basic design goal for the JDBC Bankapp sample application is to greatly scale up the number of client applications it can accommodate by doing the following:

- Processing in parallel (and on one machine) the client requests on multiple objects that implement the same interface
- Directing requests on behalf of some bank automated teller machines (ATMs) to one machine, and other ATMs to other machines
- Adding more machines across which to spread the processing load

To accommodate these design goals, the JDBC Bankapp sample application has been extended as follows:

- Replicates the Teller and TellerFactory server processes within the groups in which they are configured.
- Replicates the groups described above on an additional machine.
- Implements a stateless object model to scale up the number of client requests the server process can manage simultaneously.
- Assigns unique object IDs (OIDs) to the following objects so that they can be instantiated multiple times simultaneously in their respective groups. This makes these objects available on a per-client-application (and not per-process) basis, thereby accommodating a parallel-processing capability.
  - `TellerFactory`
  - `Teller`
- Implements factory-based routing to direct client requests on behalf of some ATMs to one machine, and other ATMs to another machine.
- Setting up threads for the Teller objects, as discussed in the *Guide to the Java Sample Applications*. For related information, also see the section “Enabling Multithreaded JavaServers” on page 4-18.

The sections that follow describe how the JDBC Bankapp sample application uses replicated server processes and server groups, object state management, and factory-based routing to meets its scalability goals. The first section that follows provides a description of the OMG IDL changes implemented in the Bankapp sample application.

Replicating Server Processes and Server Groups

The WLE system offers a wide variety of choices for how you may configure your server applications, such as:

- One machine with one server process that implements one interface.
- One machine with multiple server processes implementing one interface.
- One machine with multiple server processes implementing multiple interfaces, with or without factory-based routing.
- One machine with a multithreaded JavaServer offering one or multiple interfaces. For information about the tradeoffs of single-threaded JavaServers versus multithreaded JavaServers, see the section "Enabling Multithreaded JavaServers" on page 4-18.
- Multiple machines with multiple server processes and multiple interfaces, with or without factory-based routing.

In summary:

- To add more parallel processing capability to your client/server application, replicate your server processes.
- To add more machines to your deployment environment, add more groups and implement factory-based routing.
- To add more capacity (for certain types of applications only), add more threads. For information about the tradeoffs of single-threaded JavaServers versus multithreaded JavaServers, see the section "Enabling Multithreaded JavaServers" on page 4-18.

The following sections describe replicated server processes and groups, and also explain how you can configure them in the WLE system.

Replicated Server Processes

When you replicate the server processes in your application:

- You obtain a means to balance the load of incoming requests on that server application. As requests arrive in the WLE domain for the server group, the WLE system routes the request to the least busy server process within that group.
- You can improve the server application's performance. Instead of having one server process that can process one client request at one time, you can have multiple server processes available that can process multiple client requests simultaneously. (Note that to make this work, you need to make each object unique, which you can do by having your server application's factory assign unique OIDs.)
- You obtain a useful failover protection in the event that one of the server images stops.

To achieve the full benefit of replicated server processes, make sure that the objects instantiated by your server application generally have unique IDs. This way, a client invocation on an object can cause the object to be instantiated on demand, within the bounds of the number of server processes that are available, and not queued up for an already active object.

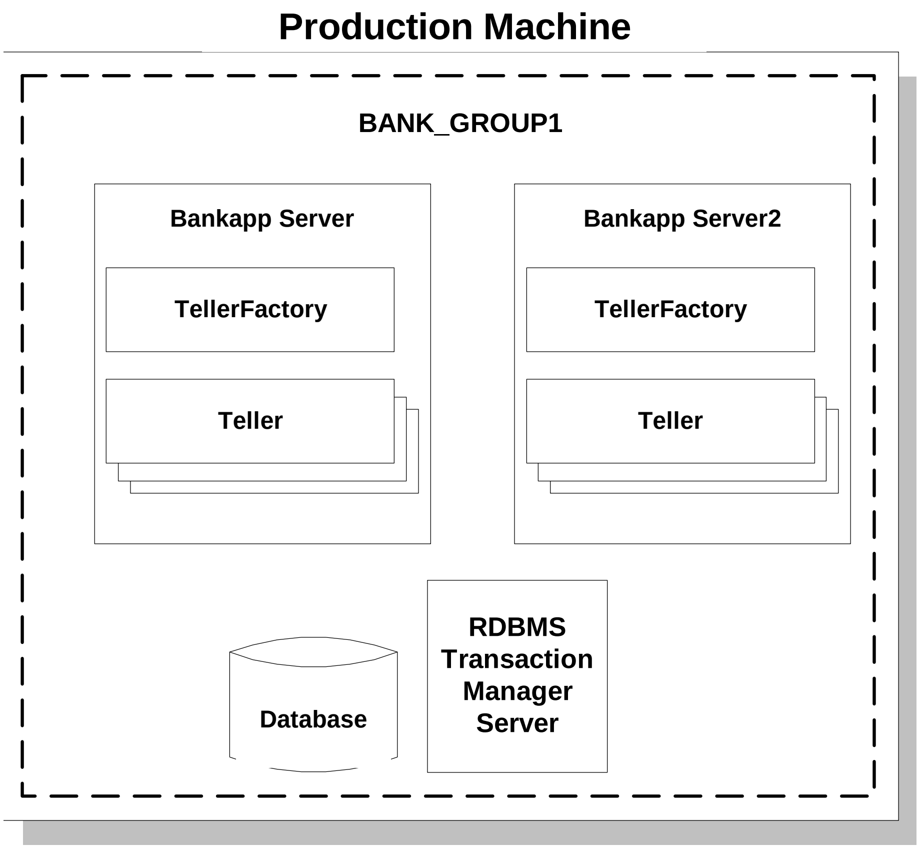
As you design your application, keep in mind that there is a tradeoff between providing:

- Better application recovery, via multiple processes
- More efficient performance, via threads (for some types of application patterns and processing environments)

Better failover occurs only by adding processes, and not by adding threads. This section discusses the technique of adding processes. For information about the tradeoffs of single-threaded JavaServers versus multithreaded JavaServers, see the section "Enabling Multithreaded JavaServers" on page 4-18.

Figure 4-1 shows the Bankapp server application replicated in the `BANK_GROUP1` group. The replicated servers are running on a single machine.

Figure 4-1 Replicated Servers in the Bankapp Sample



When a request arrives for this group, the WLE domain has several server processes available that can process the request, and the WLE domain can choose the server process that is least busy.

In Figure 4-1, note the following:

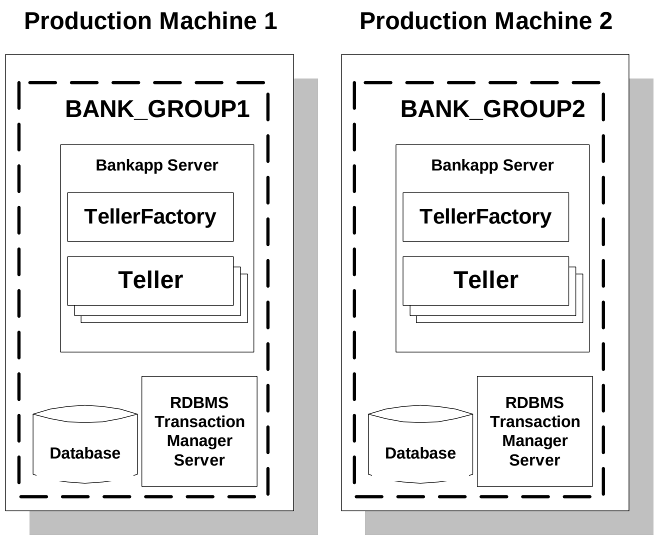
- At any time, there may be no more than one instance of the `TellerFactory` object within a given server process.
- There may be any number of `Teller` objects in any Bankapp server process.

Replicated Server Groups

The notion of server groups is specific to the WLE system and adds value to a CORBA implementation; server groups are an important part of the scalability features of the WLE system. Basically, to add more machines to a deployment, you need to add more groups.

Figure 4-2 shows the Bankapp sample application groups replicated on another machine, as specified in the application's `UBBCONFIG` file.

Figure 4-2 Replicating Server Groups Across Machines



Note: In the simple example shown in Figure 4-2, the content of the databases on Production Machines 1 and 2 is identical. Each database would contain all of the account records for all of the account IDs. Only the processing would be distributed, based on the ATM (`atmID` field). A more realistic example, one not readily adapted to the Bankapp sample application, would distribute the data and processing based on ranges of bank account IDs.

The way in which server groups are configured, where they run, and the ways in which they are replicated is specified in the `UBBCONFIG` file. When you replicate a server group, you can do the following:

- Have a means to spread processing load for a given application or set of applications across additional machines.
- Use factory-based routing to send one set of requests on a given interface to one machine, and another set of requests on the same interface to another machine.

The effect of having multiple server groups includes the following:

- When a client request arrives in the WLE domain, the WLE system checks the group ID specified in the object reference.
- The WLE domain sends the request to the least busy server process within the group to which the request is routed that can process the request.

The section “Factory-based Routing” on page 4-13 shows how the Bankapp sample application uses factory-based routing to spread the application’s processing load across multiple machines.

Configuring Replicated Server Processes and Groups

To configure replicated server processes and groups in your WLE domain:

1. Bring your application’s `UBBCONFIG` file into a text editor, such as WordPad.
2. In the `GROUPS` section, specify the names of the groups you want to configure.
3. In the `SERVERS` section, enter the following information for the server process you want to replicate:
   - A server application name. For java, this is the name of the Java server, plus the name of the JAR file.
   - The `GROUP` parameter, which specifies the name of the group to which the server process belongs. If you are replicating a server process across multiple groups, specify the server process once for each group.
   - The `SRVID` parameter, which specifies a numeric identifier, giving the server process a unique identity.
   - The `MIN` parameter, which specifies the number of instances of the server process to start when the application is booted.
   - The `MAX` parameter, which specifies the maximum number of server processes that can be running at any one time.

Thus the MIN and MAX parameters determine the degree to which a given server application can process requests on a given interface in parallel. During run time, the system administrator can examine resource bottlenecks and start additional server processes, if necessary. In this sense, the application is designed so that the system administrator can scale it.

Note: The following example shows lines from the GROUPS and SERVERS sections of the UBBCONFIG file for a Bankapp sample application. These configuration settings are not used with the Bankapp sample provided with the WLE software.

```
*RESOURCES
      IPCKEY      55432
      DOMAINID    simple
      MASTER      SITE1
      MODEL       SHM
      LDBAL       Y

*MACHINES
      "TRIXIE"
                  LMID         = SITE1
                  APPDIR       = "c:\bankapp\jdbc\."
                  TUXCONFIG    = "c:\bankapp\jdbc\.\tuxconfig"
                  TUXDIR       = "c:\m3dir"
                  MAXCLIENTS  = 10

*GROUPS
      SYS_GRP
                  LMID        = SITE1
                  GRPNO       = 1
      BANK_GROUP1
                  LMID        = SITE1
                  GRPNO       = 2
      BANK_GROUP2
                  LMID        = SITE1
                  GRPNO       = 3

*SERVERS
      # By default, restart a server if it crashes, up to 5 times
      # in 24 hours.
      #
      DEFAULT:
                  RESTART = Y
                  MAXGEN = 5

    # Start the Tuxedo System Event Broker.  This event broker
    # must be started before any servers providing the
    # NameManager Service.
```

```
 #
   TMSYSEVT
               SRVGRP = SYS_GRP
               SRVID = 1

 # TMFFNAME is a M3 provided server that runs the
 # object-transactional management services. This includes the
 # NameManager and FactoryFinder services.

 # The NameManager service is a M3-specific service
 # that maintains a mapping of application-supplied names to
 # object references.

 # Start the NameManager Service (-N option).  This name
 # manager is being started as a Master (-M option).
 #

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

 # Start a slave NameManager Service
 #

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

 # Start the FactoryFinder (-F) service
 #

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

 # Start the JavaServer in Bank_Group1
 #
   JavaServer
             SRVGRP = BANK_GROUP1
             SRVID = 5
             CLOPT = "-A -- -M 10 BankApp.jar TellerFactory_1"
             SYSTEM_ACCESS=FASTPATH
             RESTART = N

# Start the JavaServer in Bank_Group2
#
```

```
      JavaServer
                SRVGRP = BANK_GROUP2
                SRVID = 6
                CLOPT = "-A -- -M 10 BankApp.jar TellerFactory_1"
                SYSTEM_ACCESS=FASTPATH
                RESTART = N

    # Start the listener for IIOP clients
    #
    # Specify the host name of your server machine as
    # well as the port.  A typical port number is 2500
    #

      ISL
                  SRVGRP = SYS_GRP
                  SRVID = 7
                  CLOPT = "-A -- -n //TRIXIE:2468"

*SERVICES

*INTERFACES
        "IDL:beasys.com/BankApp/Teller:1.0"
         FACTORYROUTING=atmID

*ROUTING
     atmID
            TYPE = FACTORY
            FIELD = "atmID"
            FIELDTYPE = LONG
            RANGES = "1-5:BANK_GROUP1,
                      6-10: BANK_GROUP2,
                      *:BANK_GROUP1
```

Scaling the Application Via Object State Management

As stated in Chapter 1, “Java Server Application Concepts,” object state management is a fundamentally important concern of large-scale client/server systems because it is critically important that such systems achieve optimized throughput and response time. This section explains how you can use object state management to increase the scalability of the objects managed by a WLE server application, using the `Teller` objects in the Bankapp sample applications as an example.

The following table summarizes how you can use the object state management models supported in the WLE system to achieve major gains in scalability in your WLE applications.

State Model	How You Can Use It to Achieve Scalability
Method-bound	Method-bound objects are brought into the machine's memory only for the duration of the client invocation on them. When the invocation is complete, the object is deactivated and any state data for that object is flushed from memory. You can use method-bound objects to create a stateless server model in your application, in which thousands of objects are managed by your application. From the client application view, all the objects are available to service requests. However, because the server application is mapping objects into memory only for the duration of client invocations on them, only comparatively few of the objects managed by the server application are in memory at any given moment. A method-bound object is said in this document to be a stateless object.
Process-bound	Process-bound objects remain in memory from the time they are first invoked until the server process in which they are running is shut down. If appropriate for your application, process-bound objects with a large amount of state data can remain in memory to service multiple client invocations, and the system's resources need not be tied up reading and writing the object's state data on each client invocation. A process-bound object is said in this document to be a stateful object. (Note that transaction-bound objects can also be considered stateful, since they can remain in memory between invocations on them within the scope of a transaction.)

As an example of achieving scalability, the Bankapp sample `Teller` object could use the `method` activation policy. The `method` activation policy assigned to this object means that the object is activated whenever a client request arrives for it. The `Teller` object stays in memory only for the duration of one client invocation, which is appropriate in cases where the Process-Entity design pattern is recommended. As the number of clients issuing requests on the `Teller` object increases, the WLE domain is able to:

- Instantiate the `Teller` object for each client request that arrives. Client requests are not queued for an existing `Teller` object, which would likely be the case if the `Teller` object were process-bound.
- Perform load balancing by instantiating the `Teller` object in the least busy server process or group.

Factory-based Routing

Factory-based routing is a powerful feature that provides a means to send a client request to a specific server group. Using factory-based routing, you can spread that processing load for a given application across multiple machines, because you can determine the group, and thus the machine, in which a given object is instantiated.

You can use factory-based routing to expand upon the variety of load-balancing and scalability capabilities in the WLE system. In the case of the Bankapp sample application, you can use factory-based routing to send requests to a subset of ATMs to one machine, and requests for another subset of ATMs to another machine. As you add machines to ramp up your application's processing capability, the WLE system makes it easy to modify the factory-based routing in your application to add more machines.

The chief benefit of factory-based routing is that it provides a simple means to scale up an application, and invocations on a given interface in particular, across a growing deployment environment. Spreading the deployment of an application across additional machines is strictly an administrative function that does not require any recoding or rebuilding of the application.

The chief design consideration regarding implementing factory-based routing in your client/server application is in choosing the value on which routing is based. The sections that follow describe how factory-based routing works, using the extended JDBC Bankapp sample application, which uses factory-based routing in the following way. Client application requests to the `Teller` object are routed based on a teller number. Requests for one subset of teller numbers go to one group; and requests on behalf of another subset of teller numbers go to another group.

How Factory-based Routing Works

Your factories implement factory-based routing by changing the way they create object references. All object references contain a group ID, and by default the group ID is the same as the factory that creates the object reference. However, using factory-based routing, the factory creates an object reference that includes routing criteria that determines the group ID. Then when client applications send an invocation using such an object reference, the WLE system routes the request to the group ID specified in the object reference. This section focuses on how the group ID is generated for an object reference.

To implement factory-based routing, you need to coordinate the following:

- Data in the `INTERFACES` and `ROUTING` sections of the `UBBCONFIG` file.
- Groups, machines, and databases configured in the `UBBCONFIG` file.
- How the factory specifies routing criteria. The interface definition for the factory needs to specify the parameter that represents the routing criteria used to determine the group ID.

To describe the data that needs to be coordinated, the following two sections discuss configuring for factory-based routing in the `UBBCONFIG` file, and implementing factory-based routing in the factory.

Configuring for Factory-based Routing in the UBBCONFIG File

For each interface for which requests are routed, you need to establish the following information in the `UBBCONFIG` file:

- Details about the data in the routing criteria
- For each kind of criteria, the values that route to specific server groups

To configure for factory-based routing, the `UBBCONFIG` file needs to specify the following data in the `INTERFACES` and `ROUTING` sections, and also in how groups and machines are identified:

1. The `INTERFACES` section lists the names of the interfaces for which you want to enable factory-based routing. For each interface, this section specifies what kinds of criteria the interface routes on. This section specifies the routing criteria via an identifier, `FACTORYROUTING`, as in the following example:

   ```
   *INTERFACES
       "IDL:beasys.com/BankApp/Teller:1.0"
           FACTORYROUTING = atmID
   ```

   The preceding example shows the fully qualified Interface Repository ID for an interface in the extended Bankapp sample in which factory-based routing is used. The FACTORYROUTING identifier specifies the name of the routing value, atmID.

2. The ROUTING section specifies the following data for each routing value:

   - The TYPE parameter, which specifies the type of routing. In the Bankapp sample, the type of routing is factory-based routing. Therefore, this parameter is defined to FACTORY.
   - The FIELD parameter, which specifies the name that the factory inserts in the routing value. In the extended Bankapp sample, the field parameter is atmID.
   - The FIELDTYPE parameter, which specifies the data type of the routing value. In the Bankapp sample, the field type for atmID is LONG.
   - The RANGES parameter, which specifies the values that are routed to each group.

   The following example shows the ROUTING section of the UBBCONFIG file used in the Bankapp sample application:

```
*ROUTING
     atmID
            TYPE = FACTORY
            FIELD = "atmID"
            FIELDTYPE = LONG
            RANGES = "1-5:BANK_GROUP1,
                      6-10: BANK_GROUP2,
                      *:BANK_GROUP1
```

   The preceding example shows that Teller object references for ATMs in one range are routed to one server group, and Teller object references for ATMs in other ranges are routed to other groups. As illustrated in Figure 4-2, BANK_GROUP1 and BANK_GROUP2 reside on different production machines.

Implementing Factory-based Routing in a Factory

Factories implement factory-based routing by the way the invocation to the com.beasys.Tobj.TP.create_object_reference method is implemented.

This operation has the following Java binding:

```
public static org.omg.CORBA.Object
 create_object_reference(java.lang.String interfaceName,
                         java.lang.String stroid,
                         org.omg.CORBA.NVList criteria)
```

```
                      throws InvalidInterface,
                             InvalidObjectId
```

The `criteria` specifies a list of named values that can be used to provide factory-based routing for the object reference. The use of factory-based routing is optional and is dependent on the use of this argument. If you do not want to use factory-based routing, you can pass a value of 0 (zero) for this argument. The work of implementing factory-based routing in a factory is in building the `NVlist`.

As stated previously, the `TellerFactory` object in the Bankapp sample application specifies the value `atmID`. This value must match exactly the following in the `UBBCONFIG` file:

- The routing name, type, and allowable values specified by the `FACTORYROUTING` identifier in the `INTERFACES` section
- The routing criteria name, field, and field type specified in the `ROUTING` section

Note: The following example is not part of the Bankapp sample code, but is shown here to illustrate the factory-based routing feature. The `TellerFactory` object inserts the bank account number into the `NVlist` using the following code:

```
// Put the atmID (which is the routing criteria)
// into a CORBA NVList. The atmID comes from the
// tellerName that is passed in as an input parameter;
// tellerName should have the form: Teller<atmID>

int atmID = Integer.parseInt (tellerName.substring(6));
any.insert_long(atmID);

// Create the NVlist and add the atmID to the list.

org.omg.CORBA.NVList criteria = TP.orb().create_list(1);
criteria.add_value("atmID", any, 0);

// Create the object reference.

       org.omg.CORBA.Object teller_oref =
               TP.create_object_reference(
               BankApp.TellerHelper.id(),  // Repository ID
               tellerName,                 // Object ID
               criteria                    // Routing Criteria
               );
```

Note: It is possible for an object with a given interface and OID to be simultaneously active in two different groups, if those two groups both contain the same object implementation. (However, if your factories generate unique OIDs, this situation is very unlikely.) If you need to guarantee that only one object instance of a given interface name and OID is available at any one time in your domain, either: use factory-based routing to ensure that objects with a particular OID are always routed to the same group, or configure your domain so that a given object implementation is in only one group. This assures that if multiple clients have an object reference containing a given interface name and OID, the reference is always routed to the same object instance.

To enable routing on an object's OID, specify the OID as the routing criterion in the `com.beasys.Tobj.TP.create_object_reference` method, and set up the `UBBCONFIG` file appropriately.

What Happens at Run Time

When you implement factory-based routing in a factory, the WLE system generates an object reference. The following example shows how the client application gets an object reference to a `Teller` object when factory-based routing is implemented:

1. The client application invokes the `TellerFactory` object, requesting a reference to a `Teller` object. Included in the request is a teller name that includes an `atmID`.
2. The `TellerFactory` inserts the `atmID` into an `NVlist`, which is used as the routing criteria.
3. The `TellerFactory` invokes the `com.beasys.Tobj.TP::create_object_reference` method, passing the `Teller` Interface Repository ID, a unique OID, and the `NVlist`.
4. The WLE system compares the content of the routing tables with the value in the `NVlist` to determine a group ID.
5. The WLE system inserts the group ID into the object reference.

When the client application subsequently does an invocation on an object using the object reference, the WLE system routes the request to the group specified in the object reference.

Note: Be careful how you implement factory-based routing if you use the process-entity design pattern. The object can service only those entities that are contained in the group's database.

Enabling Multithreaded JavaServers

WLE supports the ability to configure multithreaded JavaServers. For each JavaServer, you can establish the maximum number of worker threads in the application's `UBBCONFIG` file.

A worker thread is a thread that is started and managed by the WLE Java software, as opposed to threads started and managed by an application program. Internally, WLE Java manages a pool of available worker threads. When a client request is received, an available worker thread from the thread pool is scheduled to execute the request. When the request is done, the worker thread is returned to the pool of available threads.

In the current WLE Java release, BEA recommends that you not establish threads programmatically. Only worker threads that are created by the run-time WLE JavaServer may access the WLE Java infrastructure. This restriction means that your Java application should not create a Java thread from a worker thread and then try to begin a new transaction in the thread. You can, however, start threads in your application to perform other, non-WLE work.

Deploying multithreaded JavaServers may not be appropriate for all applications. The potential for a performance gain from a multithreaded JavaServer depends on:

- The application pattern
- Whether the application is running on a single-processor machine or a multiprocessor machine

If the application is running on a single-processor machine and the application is CPU-intensive only, without any I/O or delays, in most cases the multithreaded JavaServer will not perform better. In fact, due to the overhead of switching between threads, the multithreaded JavaServer in this configuration may perform worse than a single-threaded JavaServer.

A performance gain is more likely with a multithreaded JavaServer when the application has some delays or is running on a multiprocessor machine.

Multithreaded WLE server applications appear the same as single-threaded applications, codewise. However, if you are planning to configure your Java server applications to be multithreaded, or if you want to have the flexibility to do so at some point in the future, keep the following recommendations in mind when writing your object implementations in Java:

- Do not start your own threads in your Java code. Threading should remain transparent in your source files.
- Write thread-safe code. Because static variables are shared across all instances of a class that could be executed in different server threads, make sure that access to those variables is synchronized properly when objects that use them are executed in a multithreaded configuration. You should use standard Java synchronization techniques to make sure that the use of static variables is properly synchronized.

  For more information about Java synchronization techniques, see the *Java Language Specification*, available at the Sun Microsystems, Inc. Web site at the following URL:

  `http://java.sun.com`
- If your application uses JNI code to access ATMI, `JavaServer` must be configured as single-threaded.
- If an XA-enabled version of JavaServer is built using `buildXAJS`, the server supports only the single-threaded mode; in this case, the WLE system ignores the `-M number` command line argument for multithreading (if specified).
- If your application is sending messages to the User Log (`ULOG`), note that it is not helpful to use the process ID to distinguish among the different threads. Instead, you can include in each message one of the following:
  - The object ID
  - The thread name
  - The transaction ID (if your object is transactional)

For information about defining the `UBBCONFIG` parameters to implement a multithreaded JavaServer, see Chapter 3 of the *Administration Guide*.

Additional Design Considerations for the Teller Object

The principal considerations that influence the design of the `Teller` object include:

- How to ensure that the `Teller` object works properly for the Bankapp deployment environment; namely, across multiple replicated server processes and multiple groups.

- How to ensure that client requests for account inquiries, withdrawls, and transfers in a given account go to the correct server group, given that the four server groups in the extended Bankapp WLE domain each deal with different databases.

The primary implications of these considerations are that these objects must:

- Have unique object IDs (OIDs)
- Be method-bound (that is, have the `method` activation policy assigned to them)

The remainder of this section discusses these considerations and implications in detail.

Instantiating the Teller Object

Because the extended Bankapp server is now replicated, the WLE domain must have a means to differentiate between multiple instances of the `Teller` object. That is, if there are two Bankapp server processes running in a group, the WLE domain must have a means to distinguish between, say, the `Teller` object running in the first Bankapp server process and the `Teller` object running in the second Bankapp server process.

The way to provide the WLE domain with the ability to distinguish among multiple instances of these objects is to make each object instance unique.

To make each `Teller` object unique, the factories for those objects must change the way in which they make object references to them. For example, when the `TellerFactory` object in the original Bankapp sample application created an object reference to the `Teller` object, the `com.beasys.Tobj.TP::create_object_reference` method specified an OID that consisted only of the string `tellerName`. However, in the extended Bankapp sample application discussed in this chapter, the same `create_object_reference` method uses a generated unique OID instead.

A consequence of giving each `Teller` object a unique OID is that there may be multiple instances of these objects running simultaneously in the WLE domain. This characteristic is typical of the stateless object model, and is an example of how the WLE domain can be highly scalable and at the same time offer high performance.

And last, because unique `Teller` objects need to be brought into memory for each client request on them, it is critical that these objects be deactivated when the invocations on them are completed so that any object state associated with them does not remain idle in memory. The Bankapp server application addresses this issue by assigning the `method` activation policy to the Teller object in the XML-based Server Description File.

Ensuring That Account Updates Occur in the Correct Server Group

The chief scalability advantage of having replicated server groups is to be able to distribute processing across multiple machines. However, if your application interacts with a database, which is the case with the JDBC Bankapp sample application, it is critical that you consider the impact of these multiple server groups on the database interactions.

In many cases, you may have one database associated with each machine in your deployment. If your server application is distributed across multiple machines, you must consider how you set up your databases.

The JDBC Bankapp sample application uses factory-based routing to send one set of requests to one machine, and another set to the other machine. As mentioned earlier, factory-based routing is implemented in the `TellerFactory` object by the way in which references to `Teller` objects are created.

How the Bankapp Server Application Can Be Scaled Further

In the future, the system administrator of the Bankapp sample application may want to add capacity to the WLE domain. For example, the bank may eventually have a large increase in automated teller machines (ATMs). This can be done without modifying or rebuilding the application.

The system administrator has the following tools available to continually add capacity:

- Replicating the Bankapp sample application server groups across additional machines

  Doing this requires modifying the `UBBCONFIG` file to specify the additional groups, what server processes run in those groups, and what machines they run on.

- Changing the factory-based routing tables

  For example, instead of routing to the four groups shown earlier in this chapter, the system administrator can modify the routing rules in the `UBBCONFIG` file to partition the application further among the new groups added to the WLE domain. Any modification to the routing tables must be consistent with any changes or additions made to the server groups and machines configured in the `UBBCONFIG` file.

Note: If you add capacity to an application that uses a database, you must also consider the impact on how the database is set up, particularly when you are using factory-based routing. For example, if the Bankapp sample application is spread across six machines, the database on each machine must be set up appropriately and in accordance with the routing tables in the `UBBCONFIG` file.

Index