THE ENTERPRISE MIDDLEWARE SOLUTION

BEA WebLogic Enterprise

Getting Started

BEA WebLogic Enterprise 4.2
Document Edition 4.2
July 1999

Getting Started

Document Edition	Part Number	Date	Software Version
4.2	861-001001-003	July 1999	BEA WebLogic Enterprise 4.2

Contents

Preface

Purpose of This Document ... vii
How to Use This Document ... viii
Related Documentation ... xi
Contact Information ... xiv

1. Understanding the WebLogic Enterprise (WLE) Product

Overview of the WLE Product ... 1-1
WLE Programming Environment ... 1-4
- IDL Compilers ... 1-4
- Development Commands ... 1-5
- Administration Tools ... 1-6
- ActiveX Application Builder ... 1-9

WLE Object Services ... 1-11
WLE Components ... 1-12
- Bootstrap Object ... 1-14
- IIOP Listener/Handler ... 1-15
- ORB ... 1-16
- TP Framework ... 1-17

How WLE Client and Server Applications Interact ... 1-19
- Step 1: The server application is initialized. ... 1-20
- Step 2: The client application is initialized. ... 1-21
- Step 3: The client application authenticates itself to the WLE domain. ... 1-22
- Step 4: The client application obtains a reference to the object needed to execute its business logic. ... 1-23
- Step 5: The client application invokes an operation on the CORBA object. ... 1-25

2. Developing WebLogic Enterprise (WLE) Applications

Overview of the Development Process for WLE Applications........................ 2-2
The Simpapp Sample Application 2-4
Step 1: Writing the OMG IDL.. 2-5
Step 2: Generating Client Stubs and Skeletons 2-6
Step 3: Writing the Server Application 2-8
    Writing the Methods That Implement Each Interface’s Operations 2-9
    Creating the Server Object 2-12
    Defining an Object’s Activation Policies 2-15
Step 4: Writing the Client Application 2-16
Step 5: Creating a Configuration File... 2-19
Step 6: Compiling the Server Application...................................... 2-22
Step 7: Compiling the Client Application 2-23
Additional WLE Sample Applications 2-24
    Univeristy Sample Applications.. 2-24
    Java Sample Applications.. 2-25

3. Using Security

Overview of the Security Service 3-1
How Security Works 3-2
The Security Sample Application... 3-4
Development Steps 3-6
    Step 1: Defining the Security Level in the Configuration File.................. 3-6
    Step 2: Writing the Client Application...................................... 3-7

4. Using Transactions

Overview of the Transaction Service 4-1
When to Use Transactional Objects 4-3
What Happens During a Transaction... 4-4
Transactions Sample Application.. 4-6
Development Steps 4-8
    Step 1: Writing the OMG IDL.. 4-9
    Step 2: Defining Transaction Policies for the Interfaces.......................... 4-12
    Step 3: Writing the Client Application..................................... 4-13
    Step 4: Writing the Server Application 4-14

Step 5: Creating a Configuration File 4-16

Preface

Purpose of This Document

This document presents an overview of the BEA WebLogic Enterprise (sometimes referred to as WLE) product and describes the development process for developing distributed client/server applications using the WebLogic Enterprise software. The *Getting Started* document does not discuss every feature of the WebLogic Enterprise product; instead, it gives a general description of building a simple transactional application. This document should be used in conjunction with the following BEA WebLogic Enterprise documents:

- *Creating Client Applications*
- *Creating C++ Server Applications*
- *Creating Java Server Applications*
- *Using Server-to-Server Communication*

Note: Effective February 1999, the BEA M3 product is renamed. The new name of the product is BEA WebLogic Enterprise (WLE).

Who Should Read This Document

This document is intended for programmers who want to familiarize themselves with the WebLogic Enterprise product and create distributed client/server applications using the WebLogic Enterprise programming environment.

How This Document Is Organized

The *Getting Started* document is organized as follows:

- Chapter 1, “Understanding the WebLogic Enterprise (WLE) Product,” describes the features, the programming environment, and the architectural components of the WLE product.
- Chapter 2, “Developing WebLogic Enterprise (WLE) Applications,” explains how to build a typical WLE application, using the Simpapp sample application as an example.
- Chapter 3, “Using Security,” describes how security is incorporated into a WLE application. The Security sample application is used as an example.
- Chapter 4, “Using Transactions,” describes how transactions are incorporated into a WLE application. The Transactions sample application is used as an example.

How to Use This Document

This document, *Getting Started*, is designed primarily as an online, hypertext document. If you are reading this as a paper publication, note that to get full use from this document you should access it as an online document via the Online Documentation CD for the BEA WebLogic Enterprise 4.2 release.

The following sections explain how to view this document online, and how to print a copy of this document.

Opening the Document in a Web Browser

To access the online version of this document, open the following file:

```
\doc\wle\v42\index.htm
```

Note: The online documentation requires Netscape Communicator version 4.0 or later, or Microsoft Internet Explorer version 4.0 or later.

Printing from a Web Browser

You can print a copy of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser. To select a chapter or appendix, click anywhere inside the chapter or appendix you want to print.

The Online Documentation CD includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document. On the CD Home Page, click the PDF Files button and scroll to the entry for the document you want to print.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

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

Convention	Item
`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` *`firstfile-syntax`*`] [-l` *`lastfile-syntax`*`]...`

`|`	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` *`firstfile-syntax`*`] [-l` *`lastfile-syntax`*`]...`

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

Related Documentation

The following sections list the documentation provided with the BEA WebLogic Enterprise software, related BEA publications, and other publications related to the technology.

BEA WebLogic Enterprise Documentation

The BEA WebLogic Enterprise information set consists of the following documents:

Installation Guide

C++ Release Notes

Java Release Notes

Getting Started (this document)

Guide to the University Sample Applications

Guide to the Java Sample Applications

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

Administration Guide

Using Server-to-Server Communication

C++ Programming Reference

Java Programming Reference

Java API Reference

JDBC Driver Programming Reference

System Messages

Glossary

Technical Articles

Note: The Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

BEA Publications

Selected BEA TUXEDO Release 6.5 for BEA WebLogic Enterprise version 4.2 documents are available on the Online Documentation CD.

To access these documents:

1. Click the Other Reference button from the main menu.
2. Click the TUXEDO Documents option.

Other Publications

For more information about CORBA, Java, and related technologies, refer to the following books and specifications:

Cobb, E. 1997. *The Impact of Object Technology on Commercial Transaction Processing*. VLDB Journal, Volume 6. 173-190.

Edwards, J. with DeVoe, D. 1997. *3-Tier Client/Server At Work*. Wiley Computer Publishing.

Edwards, J., Harkey, D., and Orfali, R. 1996. *The Essential Client/Server Survival Guide*. Wiley Computer Publishing.

Flanagan, David. May 1997. *Java in a Nutshell*, 2nd Edition. O’Reilly & Associates, Incorporated.

Flanagan, David. September 1997. *Java Examples in a Nutshell*. O’Reilly & Associates, Incorporated.

Fowler, M. with Scott, K. 1997. *UML Distilled, Applying the Standard Object Modeling Language*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Jacobson, I. 1994. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.

Mowbray, Thomas J. and Malveau, Raphael C. (Contributor). 1997. *CORBA Design Patterns*, Paper Back and CD-ROM Edition. John Wiley & Sons, Inc.

Orfali, R., Harkey, D., and Edwards, J. 1997. *Instant Corba*. Wiley Computer Publishing.

Orfali, R., Harkey, D. February 1998. *Client/Server Programming with Java and CORBA*, 2nd Edition. John Wiley & Sons, Inc.

Otte, R., Patrick, P., and Roy, M. 1996. *Understanding CORBA*. Prentice Hall PTR.

Rosen, M. and Curtis, D. 1998. *Integrating CORBA and COM Applications*. Wiley Computer Publishing.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Loresen, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

The Common Object Request Broker: Architecture and Specification. Revision 2.2, February 1998. Published by the Object Management Group (OMG).

CORBAservices: Common Object Services Specification. Revised Edition. Updated: November 1997. Published by the Object Management Group (OMG).

Contact Information

The following sections provide information about how to obtain support for the documentation and the software.

Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information about how to contact Customer Support, refer to the following section.)

Customer Support

If you have any questions about this version of the BEA WebLogic Enterprise product, or if you have problems installing and running the BEA WebLogic Enterprise software, 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

CHAPTER

1 Understanding the WebLogic Enterprise (WLE) Product

This chapter discusses the following topics:

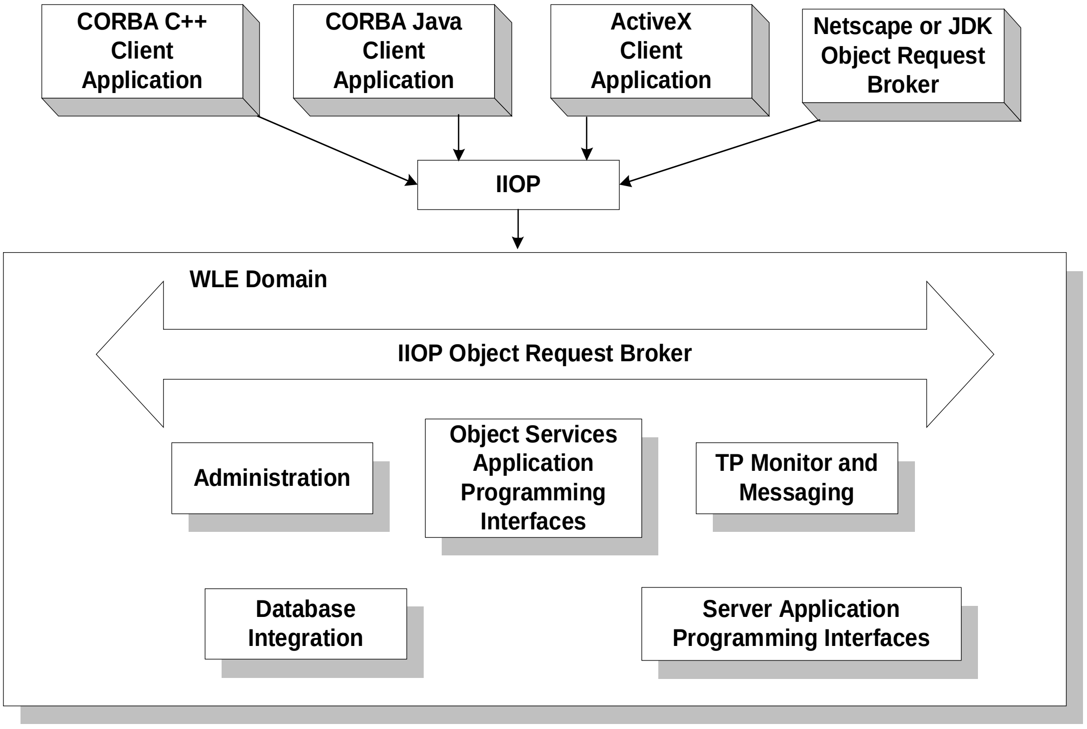
- Overview of the WLE Product
- WLE Programming Environment
- WLE Object Services
- WLE Components
- How WLE Client and Server Applications Interact

Overview of the WLE Product

The WLE product extends the Object Request Broker (ORB) model with online transaction processsing (OLTP) functions. It uses the CORBA standard as a programming model for developing enterprise applications with high performance, scalability, and reliability. The WLE deployment infrastructure delivers secure, transactional, distributed applications in a managed environment.

Objects built with the WLE product can be accessed from Web-based applications that communicate using the Object Management Group (OMG) Internet Inter-ORB Protocol (IIOP). IIOP is the standard protocol for communications running on the Internet or on an enterprise's Intranet. The WLE product has a native implementation of IIOP, ensuring high-performance, interoperable, distributed-object applications for the Internet, Intranets, and enterprise computing environments. Figure 1-1 illustrates the WLE product.

Figure 1-1 WLE Product


CORBA C++ Client Application
CORBA Java Client Application
ActiveX Client Application
Netscape or JDK Object Request Broker
IIOP
WLE Domain
IIOP Object Request Broker
Administration
Object Services Application Programming Interfaces
TP Monitor and Messaging
Database Integration
Server Application Programming Interfaces

The WLE product provides:

- A set of integrated components that can be used to build robust, distributed client/server applications. These components can be accessed from a set of C++, Java, and COM/OLE/ActiveX application programming interfaces (APIs).
- Support for hetergeneous client and server applications. The WLE product supports C++ and Java server applications, as well as C++, Java, and ActiveX client applications and Java applets.
- Interoperability with other ORB products. The use of IIOP in the WLE product allows client and server applications developed with the product to be used with applications built with CORBA 2.0 compliant ORBs from other vendors.
- Integration with other enterprise resources, such as Oracle and Microsoft SQL Server databases and legacy applications.
- A Simple Network Management Protocol (SNMP) Management Information Base (MIB) that defines the key management attributes of WLE applications.
- A CORBA and XA-compatible Transaction Service to ensure the integrity of your data even when transactions span mulitiple databases and applications.
- A CORBA-based security service that provides authentication for distributed objects and their client applications.
- An interface repository that stores meta information about object types. Meta information stored for CORBA objects includes information about modules, interfaces, operations, attributes, and exceptions.
- Dynamic Invocation Interface (DII) support. DII allows client applications to dynamically create requests for objects that were not defined at compile time.

The topics in *Getting Started* describe the features of the WLE product and the development process for building a transactional application using the WLE software. This topic does not discuss every feature of the WLE product; instead it gives an general description of building transactional applications. For more information, see the following on the Online Documentation CD:

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

Using Server-to-Server Communication

WLE Programming Environment

The WLE product offers a robust programming environment that makes the development and management of distributed objects easier. The following topics describe the features of the programming environment:

- IDL Compilers
- Development Commands
- Administration Tools
- ActiveX Application Builder

IDL Compilers

The WLE product comes with two IDL compilers that make object development easier:

- `idl`—compiles the OMG IDL file and generates client stub and server skeleton files required for interface definitions being implemented in C++
- `m3idltojava`—compiles the OMG IDL file and generates client stub and server skeleton files required for interface definitions being implemented in Java

For a description of how to use the IDL compilers, see the following on the Online Documentation CD:

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

For a description of the `idl` and `m3idltojava` commands, see the following on the Online Documentation CD:

C++ Programming Reference

Java Programming Reference

Development Commands

Table 1-1 lists the commands that the WLE product provides for developing application components and managing the Interface Repository.

Table 1-1 Development Commands

Development Command	Description
`buildjavaserver`	Constructs a server application JAR file for a Java server application.
`buildobjclient`	Constructs a C++ client application.
`buildobjserver`	Constructs a C++ server application.
`buildXAJS`	Constructs an XA resource manager to be used with a Java server application group.
`genicf`	Generates an Implementation Configuration File (ICF). The ICF file defines activation and transaction policies for C++ server applications.
`idl2ir`	Creates the Interface Repository and loads interface definitions into it.
`ir2idl`	Shows the content of the Interface Repository.
`irdel`	Deletes the specified object from the Interface Repository.

For a description of how to use the development commands to develop client and server applications, see the following on the Online Documentation CD:

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

For a description of the development commands, see the following on the Online Documentation CD:

C++ Programming Reference

Java Programming Reference

Administration Tools

The WLE product provides a complete set of tools for administering your WLE environment. You can manage the WLE application through commands, through a graphical user interface, or by including administration utilities in a script.

You can use the commands listed in Table 1-2 to perform administration tasks for your WLE application.

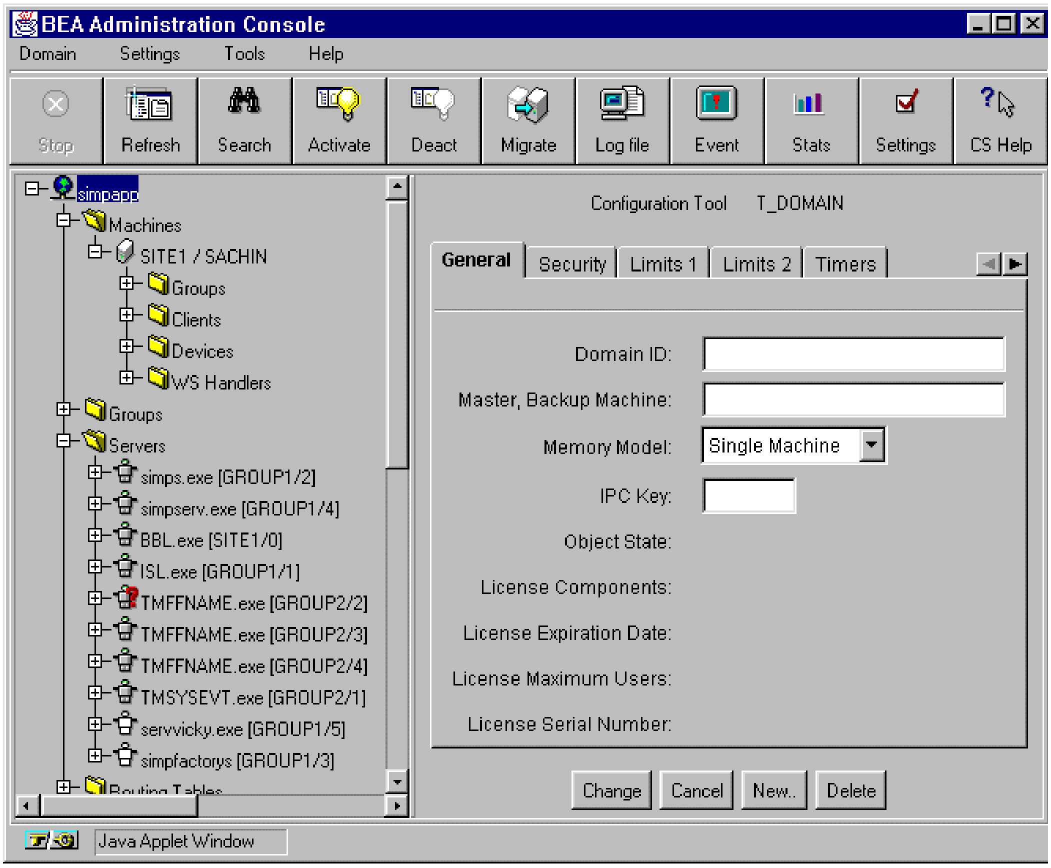
Table 1-2 Administration Commands

Administration Command	Description
`tmadmin`	Displays information about current configuration parameters.
`tmboot`	Activates the WLE application referenced in the specified configuration file. Depending on the options used, the entire application or parts of the application are started.
`tmconfig`	Dynamically updates and retrieves information about the configuration of a WLE application.
`tmloadcf`	Parses the configuration file and loads the binary version of the configuration file.

Administration Command	**Description**
`tmshutdown`	Shuts down a set of specified server applications, or removes interfaces from a configuration file.
`tmunloadcf`	Unloads the configuration file.

The Administration Console is a Java-based applet that you can download into your Internet browser and use to remotely manage your WebLogic Enterprise applications. The Administration Console allows you to perform administration tasks, such as monitoring system events, managing system resources, creating and configuring administration objects, and viewing system statistics. Figure 1-2 shows the main window of the Administration Console.

Figure 1-2 Administration Console Main Window



Domain ID:
Master, Backup Machine:
Memory Model: Single Machine
IPC Key:
Object State:
License Components:
License Expiration Date:
License Maximum Users:
License Serial Number:

In addition, a set of utilities called the AdminAPI is provided for directly accessing and manipulating system settings in the Management Information Bases (MIBs) for the WLE product. The advantage of the AdminAPI is that it can be used to automate administrative tasks, such as monitoring log files and dynamically reconfiguring an application, thus eliminating the need for human intervention.

For information about the Administration commands, see the *Adminstration Guide* on the Online Documentation CD

For a description of the Administration Console and how it works, see the *Administration Guide* on the Online Documentation CD and the online help that is integrated into the Administration Console graphical user interface (GUI).

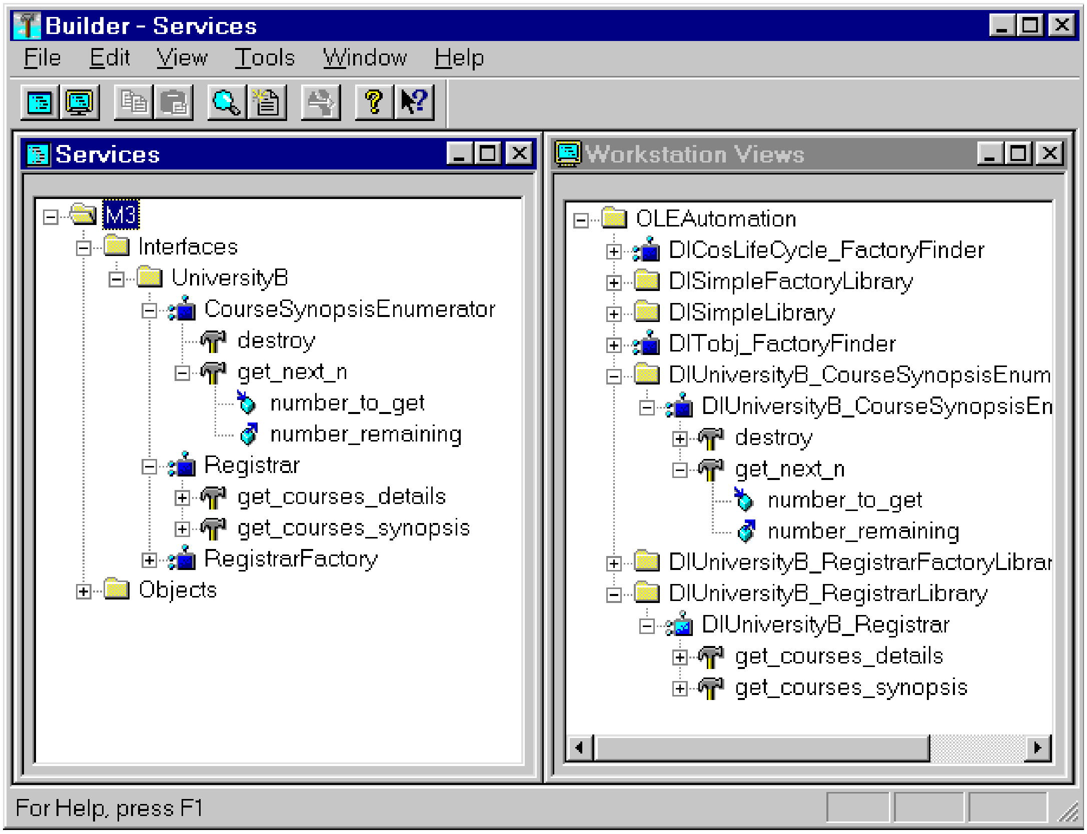
For information about the AdminAPI, see the *BEA TUXEDO Reference* on the Online Documentation CD.

ActiveX Application Builder

The ActiveX Application Builder is a development tool that you use with a client development tool (such as Visual Basic) to select which CORBA interfaces in a WLE domain you want your ActiveX client application to interact with. In addition, you use the ActiveX Application Builder to create Automation bindings for CORBA interfaces, and to create packages for deploying ActiveX views of CORBA objects to client machines.

Figure 1-3 shows the ActiveX Application Builder main window.

Figure 1-3 ActiveX Application Builder Main Window



For a description of the ActiveX Application Builder and how it works, see the online help that is integrated into the ActiveX Application Builder graphical user interface (GUI). For a description of how ActiveX client applications use CORBA objects, see *Creating Client Applications* on the Online Documentation CD.

WLE Object Services

The WLE product includes a set of environmental objects that provide object services to client applications in a WLE domain. You access the environmental objects through a bootstrapping process that accesses the services in a particular WLE domain.

The following services are provided:

- Object Life Cycle service

  The Object Life Cycle service is provided through the FactoryFinder environmental object. The FactoryFinder object is a CORBA object that can be used to locate a factory, which in turn can create object references for CORBA objects. Factories and FactoryFinder objects are implementations of the CORBAservices Life Cycle Service. WLE applications use the Object Life Cycle service to find object references.

  For information about using the Object Life Cycle Service, see the topic “How WLE Client and Server Applications Interact.”

- Security service

  The Security service is accessed through the SecurityCurrent environmental object. The SecurityCurrent object is used to authenticate a client application into a WLE domain with the proper security. The WLE software provides an implementation of the CORBAservices Security Service.

  For information about using security, see the topic “Using Security.”

- Transaction service

  The Transaction service is accessed through either the TransactionCurrent environmental object or the UserTransaction object. The TransactionCurrent object allows a client application to participate in a transaction. The WLE software provides an implementation of the CORBAservices Object Transaction Service (OTS). In addition, the UserTransaction object provides access to Sun Microsystems, Inc.’s Java Transaction API (JTA) defined in the `javax.transaction` package.

  For information about using transactions, see the topic “Using Transactions.”

- Interface Repository service

  The Interface Respository service is accessed through the IntefaceRepository object. The InterfaceRepository object is a CORBA object that contains interface definitions for all the available CORBA interfaces and the factories used to create object references to the CORBA interfaces. The Interface Repository object is used with client applications that use DII.

  For information about using DII, see *Creating Client Applications* on the Online Documentation CD.

The WLE software provides environmental objects for the following programming environments:

- C++
- Java
- Automation (used by ActiveX client applications)

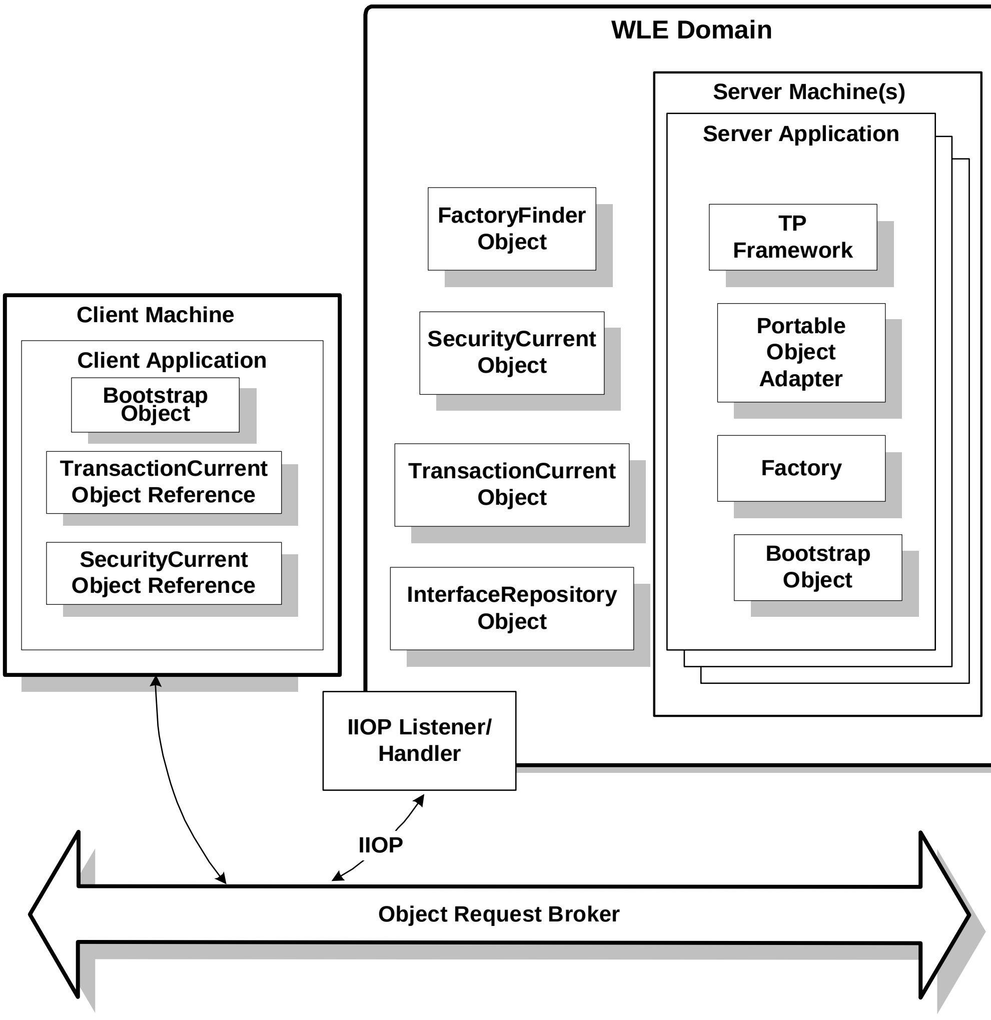
WLE Components

This section provides an introduction to the following WLE components:

- Bootstrap Object
- IIOP Listener/Handler
- ORB
- TP Framework

Figure 1-4 illustrates the components in a WLE application.

Figure 1-4 Components in a WLE Application



Bootstrap Object

The Bootstrap object establishes communication between a client application and a WLE domain. A domain is simply a way of grouping objects and services together as a management entity. A WLE domain has at least one IIOP Listener/Handler and is identified by a name. One client application can connect to multiple WLE domains using different Bootstrap objects.

One of the first things that client applications do after startup is create a Bootstrap object by supplying the host and port of the IIOP Listener/Handler as a parameter to its constructor, as follows:

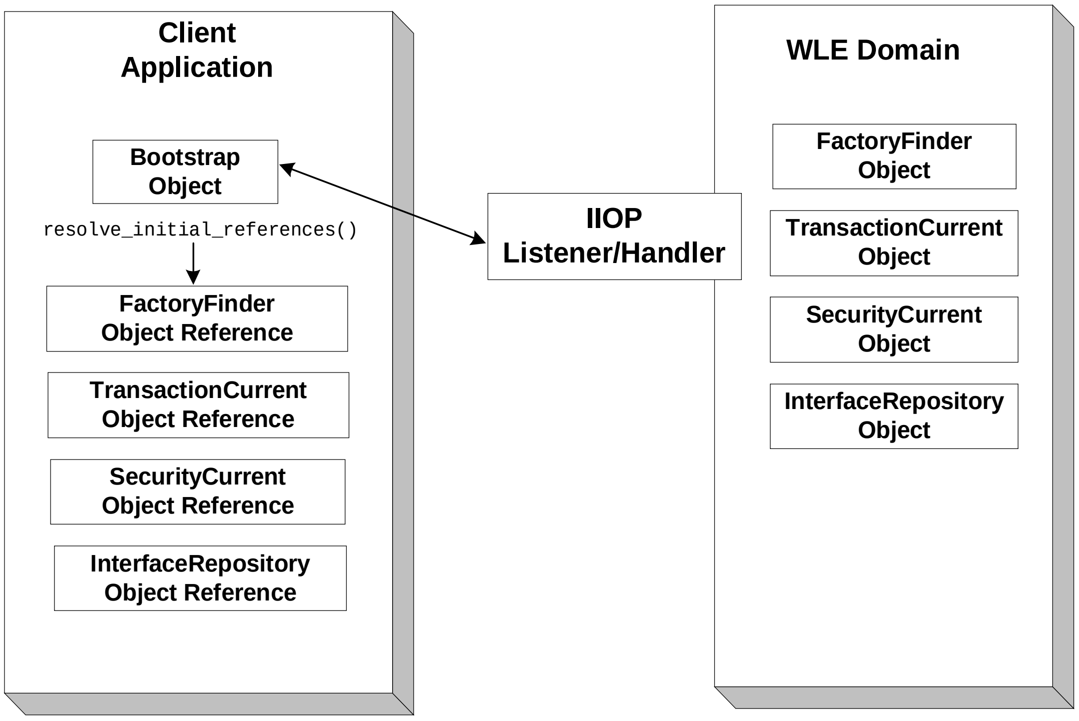
`//`*`host`*`:`*`port`*

For example, //`myserver:4000`

The client application then uses the Bootstrap object to obtain references to the objects in a WLE domain. Once the Bootstrap object is instantiated, the `resolve_initial_references` method is invoked by the client application, passing in a `string id`, to obtain a reference to the objects in the domain that provide CORBA services. The valid values for `string Id` are FactoryFinder, TransactionCurrent, SecurityCurrent, and InterfaceRepository.

Figure 1-5 illustrates how the Bootstrap object works in a WLE domain.

Figure 1-5 How the Bootstrap Object Works


Client Application
Bootstrap Object
resolve_initial_references()
FactoryFinder Object Reference
TransactionCurrent Object Reference
SecurityCurrent Object Reference
InterfaceRepository Object Reference
IIOP Listener/Handler
WLE Domain
FactoryFinder Object
TransactionCurrent Object
SecurityCurrent Object
InterfaceRepository Object

IIOP Listener/Handler

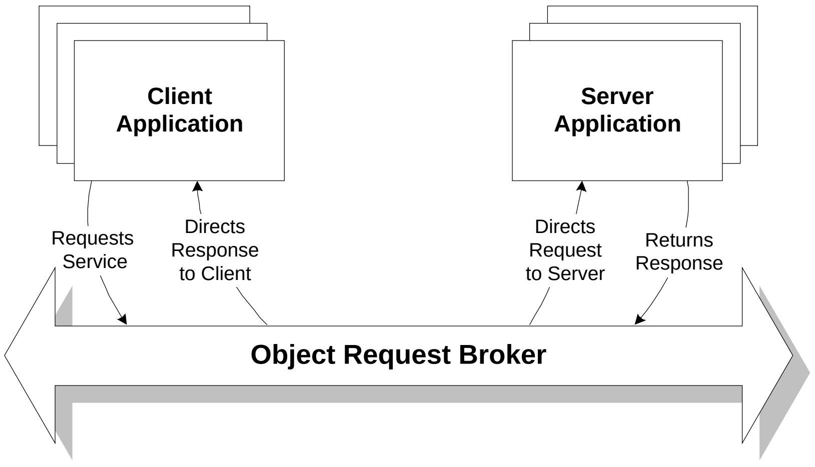
The IIOP Listener/Handler is a process that receives the client request, which is sent using IIOP, and delivers that request to the appropriate server application. The IIOP Listener/Handler serves as a communication concentrator, providing a critical scalability feature. The IIOP Listener/Handler removes from the server application the burden of maintaining client connections. For information about configuring the IIOP Listener/Handler, see the *Adminstration Guide* on the Online Documentation CD.

ORB

The ORB serves as an intermediary for requests that client applications send to server applications, so that client applications and server applications do not need to contain information about each other. The ORB is responsible for all the mechanisms required to find the implementation that can satisfy the request, to prepare an object's implementation to receive the request, and to communicate the data that makes up the request. The WLE product provides a C++ ORB and a BEA version of the Java IDL ORB provided with the Java Development Kit from Sun Microsystems, Inc.

Figure 1-6 shows the relationship between an ORB, a client application, and a server application.

Figure 1-6 The ORB in a Client/Server Environment



When the client application uses IIOP to send a request to the domain, the ORB performs the following functions:

- ♦ Validates each request and its arguments to ensure that the client application supplied all the required arguments.
- ♦ Manages the mechanisms required to find the CORBA object that can satisfy the client application's request. To do this, the ORB interacts with the Portable Object Adapter (POA). The POA prepares an object's implementation to receive the request and communicates the data in the request.

- Marshals data. The ORB on the client machine writes the data associated with the request into a standard form. The ORB receives this data and converts it into the format appropriate for the machine on which the server application is running. When the server application sends data back to the client application, the ORB marshals the data back into its standard form and sends it back to the ORB on the client machine.

TP Framework

The TP Framework provides a programming model that achieves high levels of performance while shielding the application programmer from the complexities of the CORBA interfaces. The TP Framwork supports the rapid construction of WLE applications, which makes it easier for application programmers to adhere to design patterms associated with successful TP applications.

The TP Framework interacts with the Portable Object Adapter (POA) and the WLE application, thus eliminating the need for direct POA calls in an application. In addition, the TP Framework integrates transactions and state management into the WLE application.

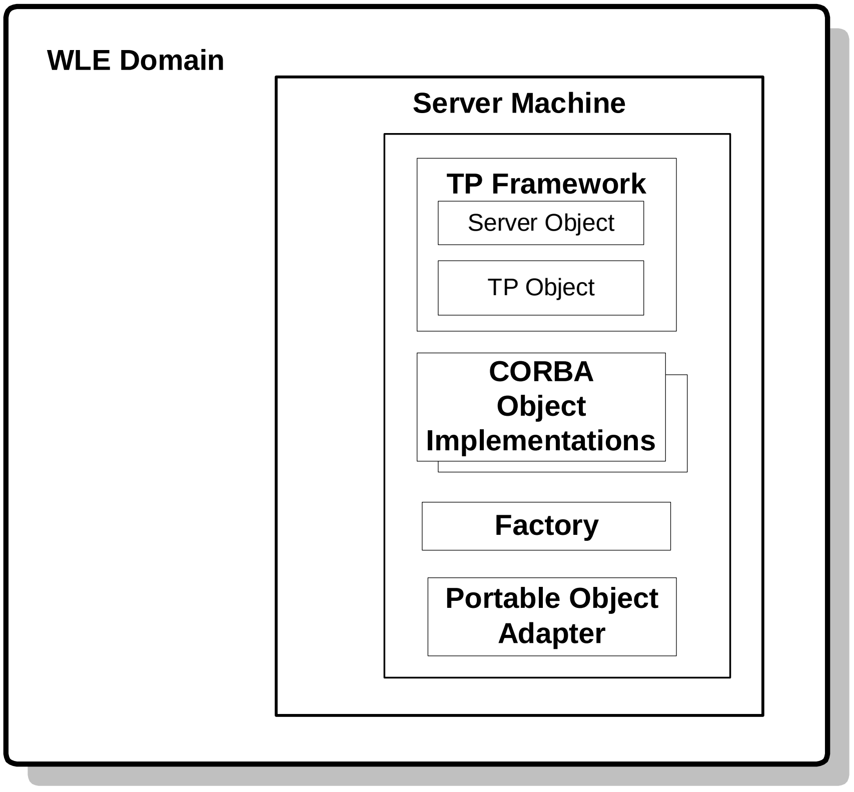
The application programmer uses an Application Programming Interface (API) that automates many of the functions required in a standard CORBA application. The application programmer is responsible only for writing the business logic of the WLE application and overriding default actions provided by the TP Framework.

The TP Framework API provides routines that perform the following functions required by a CORBA application:

- Initializing the server application and executing startup and shutdown routines
- Creating object references
- Registering and unregistering object factories
- Managing objects and object state
- Tying the server application to WLE system resources
- Getting and initializing the ORB
- Performing object housekeeping

The TP Framework ensures that the execution of a client request takes place in a coodinated, predictable manner. The TP Framework calls the objects and services available in the WLE application at the appropriate time, in the correct sequence. In addition, the TP Framework maximizes the reuse of system resources by objects. Figure 1-7 illustrates the TP Framework.

Figure 1-7 The TP Framework



The TP Framework is not a single object, but is rather a collection of objects that work together to manage the CORBA objects that contain and implement your WLE application's data and business logic.

One of the TP Framework objects is the Server object. The Server object is a user-written programming entity that implements operations that perform tasks such as initializing and releasing the server application; for server applications implemented in C++, the TP Framework instantiates the CORBA objects needed to satisfy a client request.

If a client request that requires an object that is not currently active and in-memory in the server application arrives, the TP Framework coordinates all the operations that are required to instantiate the object. This includes coordinating with the ORB and the POA to get the client request to the appropriate object implementation code.

For examples of programming with the TP Framework, see *Creating C++ Server Applications* and *Creating Java Server Applications* on the Online Documenation CD.

How WLE Client and Server Applications Interact

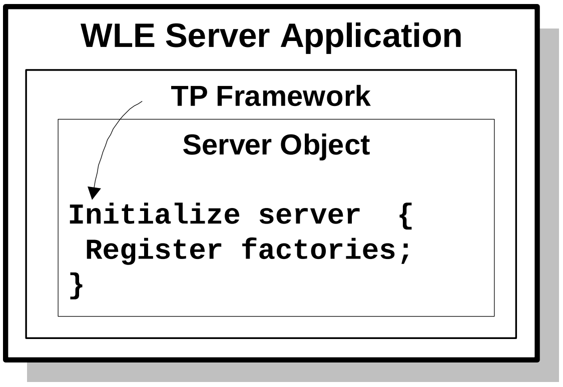
The interaction between WLE client and server applications includes the following steps:

1. The server application is initialized.
2. The client application is initialized.
3. The client application authenticates itself to the WLE domain.
4. The client application obtains a reference to the object needed to execute its business logic.
5. The client application invokes an operation on the CORBA object.

The following topics describe what happens during each step.

Step 1: The server application is initialized.

The system administrator enters the `tmboot` command on a machine in the WLE domain to start the WLE server application. The TP Framework invokes the `initialize` operation in the Server object to initialize the server application.

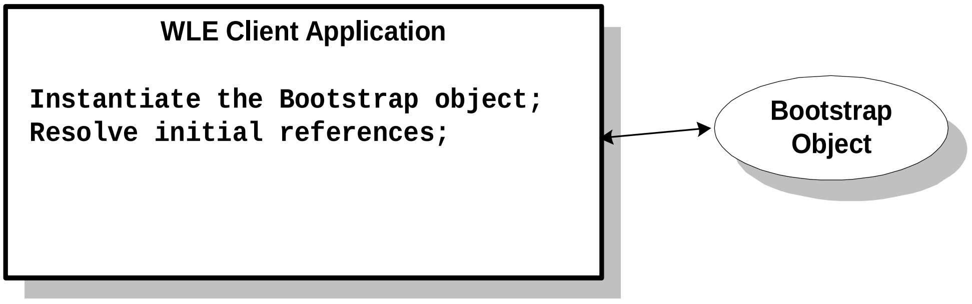


During the initialization process, the Server object does the following:

1. Gets the Bootstrap object and a reference to the FactoryFinder object.
2. Typically registers any factories with the FactoryFinder object.
3. Optionally gets an object reference to the ORB.
4. Performs any process-wide initialization.

Step 2: The client application is initialized.

During initialization, the client application uses the Bootstrap object in the domain to obtain initial references to the environmental objects available in the domain.

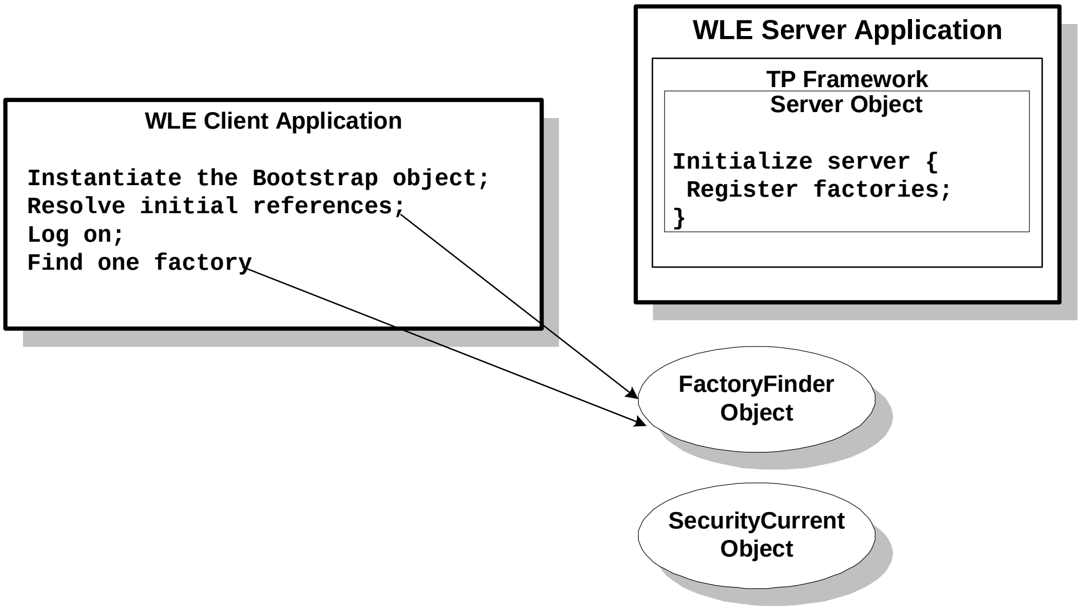

WLE Client Application
Instantiate the Bootstrap object;
Resolve initial references;
Bootstrap Object

The Bootstrap object returns references to the FactoryFinder, SecurityCurrent, TransactionCurrent, and InterfaceRepository objects in the WLE domain.

Step 3: The client application authenticates itself to the WLE domain.

If the WLE domain has a security model in effect, the client application needs to authenticate itself to the WLE domain before it can invoke any operations in the server application. To authenticate itself to the WLE domain, the client application:

1. Uses the Bootstrap object to obtain a reference to the SecurityCurrent object.
2. Invokes the `logon` operation of the `PrincipalAuthenticator` object, which is retrieved from the SecurityCurrent object.



```
Instantiate the Bootstrap object;
Resolve initial references;
Log on;
Find one factory
```

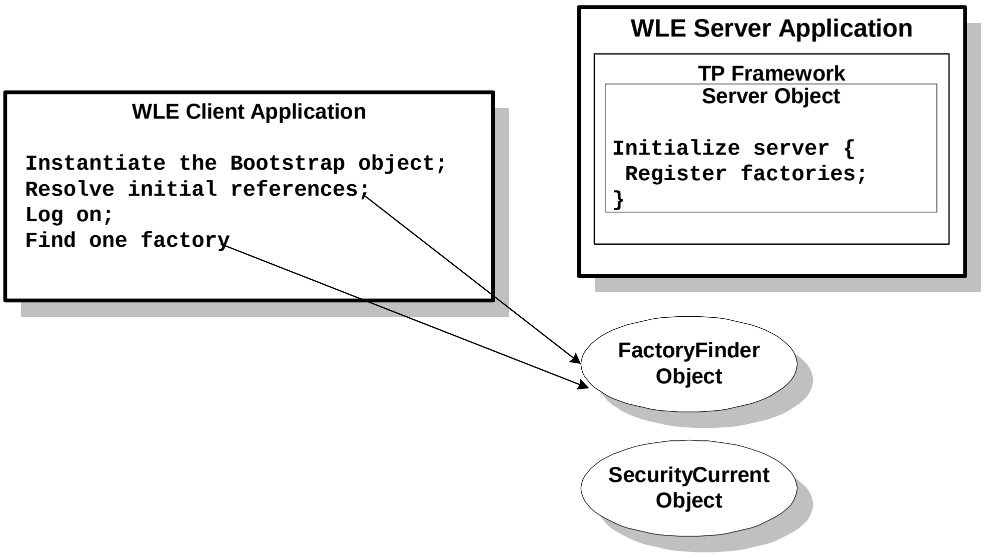
```
Initialize server {
 Register factories;
}
```

Step 4: The client application obtains a reference to the object needed to execute its business logic.

The client application needs to perform the following steps:

1. Obtain a reference to the factory for the object it needs.

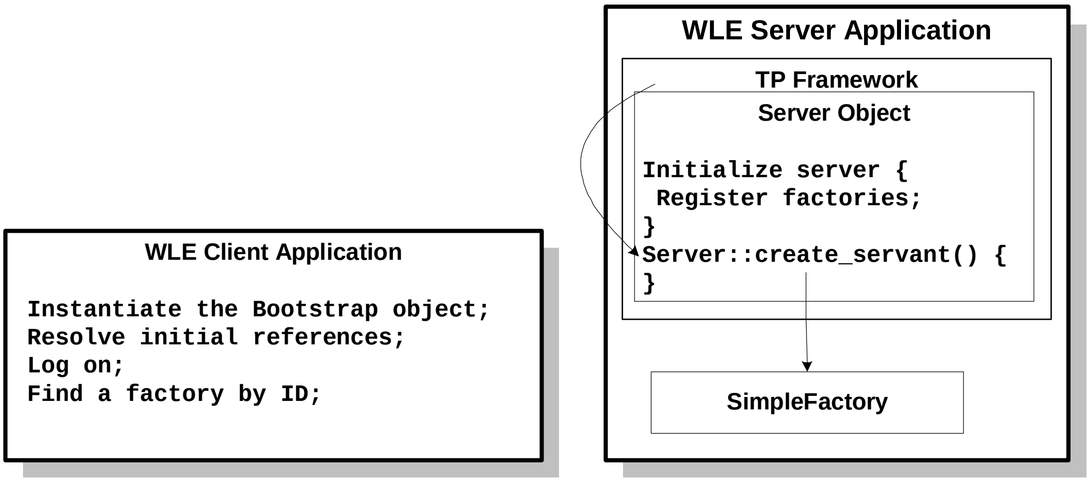
   For example, the client application needs a reference to the `SimpleFactory` object. The client application obtains this factory reference from the `FactoryFinder` object, shown in the following figure.



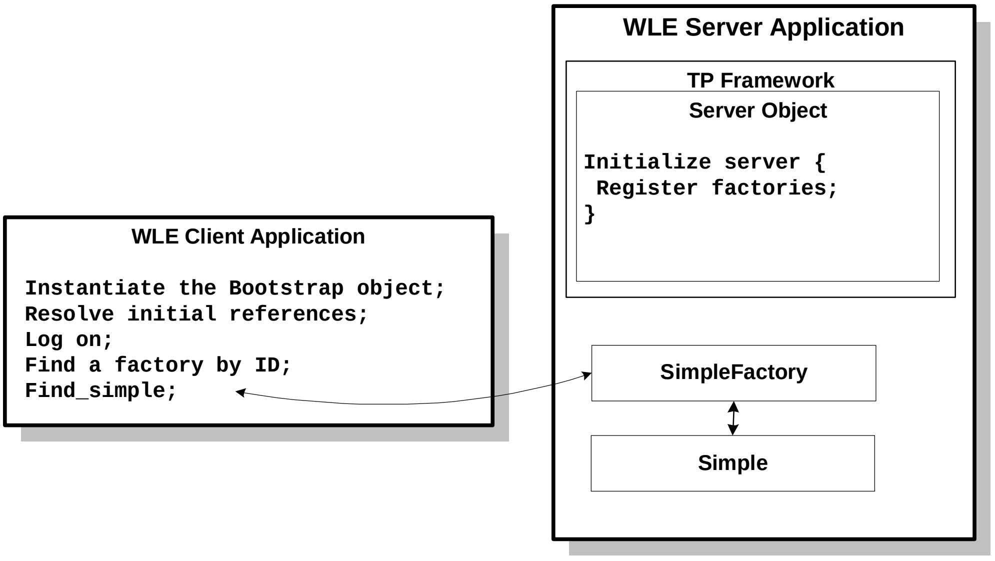
2. Invoke the `SimpleFactory` object to get a reference to the `Simple` object.

   If the `SimpleFactory` object is not active, what happens next depends on the programming language in which the server application is implemented:

   - If C++, the TP Framework instantiates the `SimpleFactory` object by invoking the `Server::create_servant` method on the Server object, shown in the following figure.
   - In Java, the WLE system instantiates the `SimpleFactory` object dynamically.



3. The TP Framework invokes the `activate_object` and `find_simple` operations on the `SimpleFactory` object to get a reference to the `Simple` object, shown in the following figure.

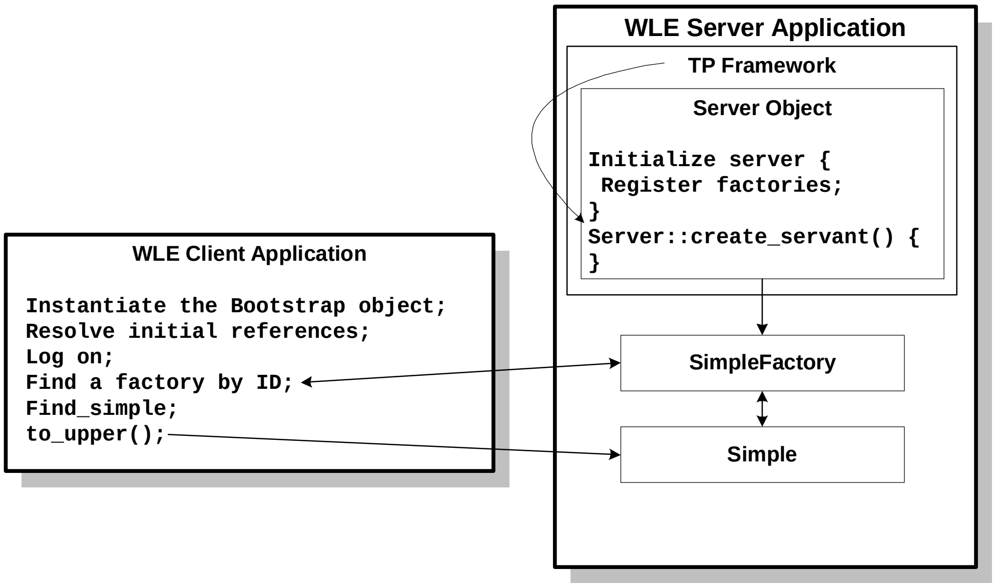


The `SimpleFactory` object then returns the object reference to the `Simple` object to the client application.

Note: Because the TP Framework activates objects by default, the Simpapp sample application does not implicitly use the `activate_object` operation for the SimpleFactory object.

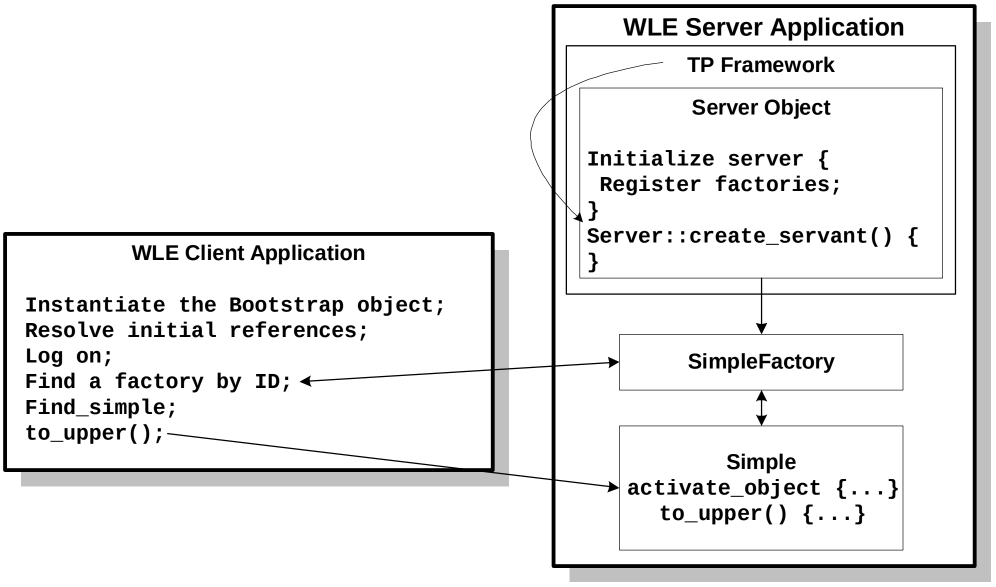
Step 5: The client application invokes an operation on the CORBA object.

Using the reference to the CORBA object that the factory has returned to the client application, the client application invokes an operation on the object. For example, now that the client application has an object reference to the `Simple` object, the client application can invoke the `to_upper` operation on it. The instance of the Simple object required for the client request is created as shown in the following figure.



If the server application were implemented in Java, the `Simple` object required for the client request is instantiated dynamically by the WLE system.

4. The TP Framework invokes the `activate_object` operation on the `Simple` object and the factory object to allow the object to initialize any object state necessary, shown in the following figure.



Object state initialization often involves reading durable state information from disk for that object.

5. The TP Framework invokes the operation on the object, returning the response to the client application.

CHAPTER

2 Developing WebLogic Enterprise (WLE) Applications

This chapter discusses the following topics:

- Overview of the Development Process for WLE Applications
- The Simpapp Sample Application
- Step 1: Writing the OMG IDL
- Step 2: Generating Client Stubs and Skeletons
- Step 3: Writing the Server Application
- Step 4: Writing the Client Application
- Step 5: Creating a Configuration File
- Step 6: Compiling the Server Application
- Step 7: Compiling the Client Application
- Additional WLE Sample Applications

Overview of the Development Process for WLE Applications

Table 2-1 outlines the development process for WLE applications.

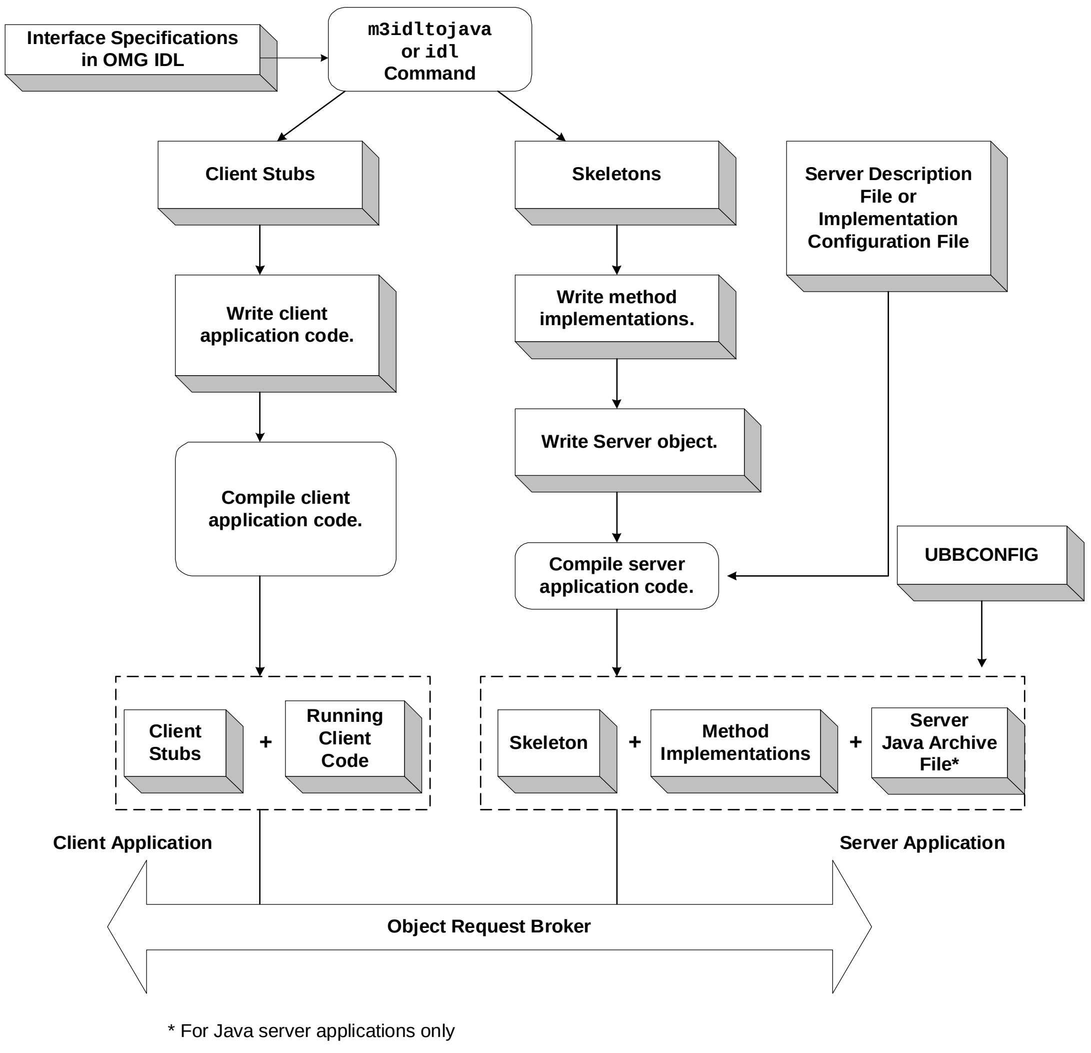
Table 2-1 Development Process for WLE Applications

Step	Description
1	Write the Object Management Group (OMG) Interface Definition Language (IDL) for each CORBA interface you want to use in your WLE application.
2	Generate the client stubs and the skeletons.
3	Write the server application.
4	Write the client application.
5	Create a configuration file.
6	Compile the server application.
7	Compile the client application.

The steps in the development process are described in the following topics.

Figure 2-1 illustrates the process for developing WLE applications.

Figure 2-1 Development Process for WLE Applications


Interface Specifications in OMG IDL
m3idltojava or idl Command
Client Stubs
Skeletons
Server Description File or Implementation Configuration File
Write client application code.
Write method implementations.
Write Server object.
Compile client application code.
Compile server application code.
UBBCONFIG
Client Stubs
+
Running Client Code
Skeleton
+
Method Implementations
+
Server Java Archive File*
Client Application
Server Application
Object Request Broker
* For Java server applications only

The Simpapp Sample Application

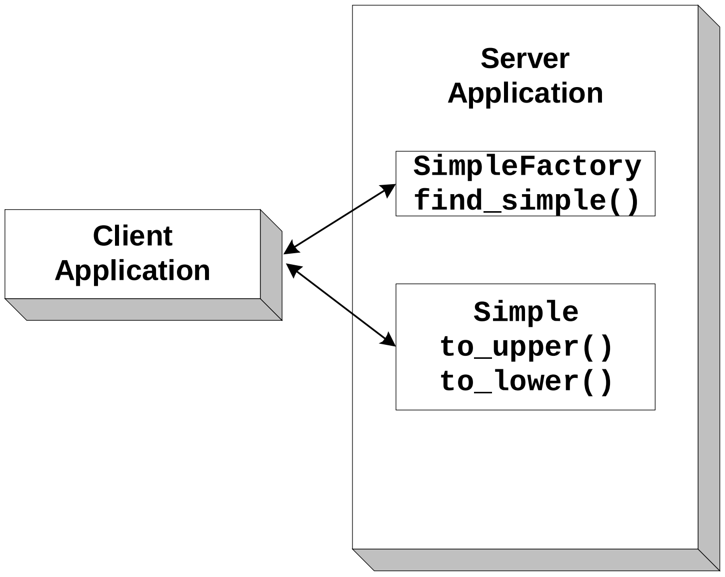
Throughout this topic, the Simpapp sample application is used to demonstrate the development steps. C++ and Java versions of the Simpapp sample application are available.

The server application in the Simpapp sample application provides an implementation of a CORBA object that has the following two methods:

- The `upper` method accepts a string from the client application and converts the string to uppercase letters.
- The `lower` method accepts a string from the client application and converts the string to lowercase letters.

Figure 2-2 illustrates how the Simpapp sample application works.

Figure 2-2 Simpapp Sample Application



The source files for the C++ and Java versions of the Simpapp sample application are located in the `\samples\corba\simpapp` and `\samples\corba\simpap_java` directories of the WLE software. Instructions for building and running the Simpapp sample applications are in the `readme` files in the directories. For the instructions for building and running the Java Simpapp sample application, see *Guide to the Java Sample Applications* on the Online Documentation CD.

Note: The Simpapp sample applications demonstrate building C++ client and server applications and Java client and server applications. For information about building a simple ActiveX client application, see the description of the Basic sample application in the *Guide to the University Sample Applications* on the Online Documentation CD.

The WLE product offers a suite of sample applications that demonstrate and aid in the development of WLE applications. For an overview of the available sample applications, see the topic “Additional WLE Sample Applications.”

Step 1: Writing the OMG IDL

The first step in writing a WLE application is to specify all of the CORBA interfaces and their methods using the Object Management Group (OMG) Interface Definition Language (IDL). An interface definition written in OMG IDL completely defines the CORBA interface and fully specifies each operation’s arguments. OMG IDL is a purely declarative language. This means that it contains no implementation details. Operations specified in OMG IDL can be written in and invoked from any language that provides CORBA bindings.

The Simpapp sample application implements the CORBA interfaces listed in Table 2-2.

Table 2-2 CORBA Interfaces for the Simpapp Sample Application

Interface	Description	Operation
`SimpleFactory`	Creates object references to the `Simple` object	`find_simple()`

`Simple`	Converts the case of a string	`to_upper()`
		`to_lower()`

Listing 2-1 shows the `simple.idl` file that defines the CORBA interfaces in the Simpapp sample application. The same OMG IDL file is used by both the C++ and Java Simpapp sample applications.

Listing 2-1 OMG IDL Code for the Simpapp Sample Application

```
#pragma prefix "beasys.com"

interface Simple
{
     //Convert a string to lower case (return a new string)
     string to_lower(in string val);

     //Convert a string to upper case (in place)
     void to_upper(inout string val);
};

interface SimpleFactory
{
     Simple find_simple();
};
```

Step 2: Generating Client Stubs and Skeletons

The interface specification defined in OMG IDL is used by the IDL compiler to generate client stubs for the client application, and skeletons for the server application. The client stubs are used by the client application for all operation invocations. You use the skeleton, along with the code you write to create the server application that implements the CORBA objects.

During the development process, use one of the following commands to compile the OMG IDL file and produce client stubs and skeletons for WLE client and server applications:

- ♦ If you are creating C++ client and server applications, use the `idl` command. For a description of the `idl` command, see *C++ Programming Reference* on the Online Documentation CD.

- If you are creating Java client and server applications, use the `m3idltojava` command. For a description of the `m3idltojava` command, see *Java Programming Reference* on the Online Documentation CD.

Table 2-3 lists the files that are created by the `idl` command.

Table 2-3 Files Created by the IDL Command

File	Default Name	Description
Client stub file	`*application*_c.cpp`	Contains generated code for sending a request.
Client stub header file	`*application*_c.h`	Contains class definitions for each interface and type specified in the OMG IDL file.
Skeleton file	`*application*_s.cpp`	Contains skeletons for each interface specified in the OMG IDL file. During run time, the skeleton maps client requests to the appropriate operation in the server application.
Skeleton header file	`*application*_s.h`	Contains the skeleton class definitions.
Implementation file	`*application*_i.cpp`	Contains signatures for the methods that implement the operations on the interfaces specified in the OMG IDL file.
Implementation header file	`*application*_i.h`	Contains the initial class definitions for each interface specified in the OMG IDL file.

Table 2-4 lists the files that are created by the `m3idltojava` command.

Table 2-4 Files Created by the `m3idltojava` Command

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 `*interface*Impl.java`, substituting the actual name of the interface in the file name. We call this new file an *object implementation file*.

File	Default Name	Description
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. During run time, 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.

Step 3: Writing the Server Application

The WLE software supports C++ and Java server applications. The steps for creating server applications are:

1. Write the methods that implement each interface's operations.
2. Create the server object.
3. Define object activation policies.

For a detailed description of how to create server applications, see the following on the Online Documentation CD:

- *Creating C++ Server Applications*
- *Creating Java Server Applications*

Writing the Methods That Implement Each Interface's Operations

After you compile the OMG IDL file, you need to write methods that implement the operations for each interface in the file. An implementation file contains the following:

- 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)
- The `activate_object` and `deactivate_object` methods (optional)

  Within the `activate_object` and `deactivate_object` methods, you write code that performs any particular steps related to activating or deactivating an object. For information about activating and deactivating objects, see *Creating C++ Server Applications* or *Creating Java Server Applications* on the Online Documentation CD.

You can write the implementation file by hand. However, both the `idl` and `m3idltojava` commands have an option that generates a template for implementation files. For information about using this template, see *Creating C++ Server Applications* or *Creating Java Server Applications* on the Online Documentation CD.

You also need to write an implementation for the factory that is used to create the objects in your application. You can include the implementation for the factory object in the same file with the other implemenations in your WLE application, or youu can include it in a seperate file.

Writing an implementation for a factory object is different than writing an implementation for other types of objects, because you need to define a specific set of information for the factory. For more information about writing implementations for factories, see *Creating C++ Server Applications* or *Creating Java Server Applications* on the Online Documentation CD.

Listing 2-2 includes the C++ implementations of the Simple and SimpleFactory interfaces in the Simpapp sample application.

Listing 2-2 C++ Implementation of the Simple and SimpleFactory Interfaces

```
// Implementation of the Simple_i::to_lower method which converts
// a string to lower case.

char* Simple_i::to_lower(const char* value)
{
     CORBA::String_var var_lower = CORBA::string_dup(value);
     for (char* ptr = v_lower; ptr && *ptr; ptr++) {
         *ptr = tolower(*ptr);
     }
     return var_lower._retn();
}

// Implementation of the Simple_i::to_upper method which converts
// a string to upper case.

void Simple_i::to_upper(char*& valuel)
{
     CORBA::String_var var_upper = value;
     var_upper = CORBA::string_dup(var_upper.in());
     for (char* ptr = var_upper; ptr && *ptr; ptr++) {
         *ptr = toupper(*ptr);
     }
     value = var_upper._retn();
}
// Implementation of the SimpleFactory_i::find_simple method which
// creates an object reference to a Simple object.

Simple_ptr SimpleFactory_i::find_simple()
{
     CORBA::Object_var var_simple_oref =
         TP::create_object_reference(
             _tc_Simple->id(),
             "simple",
             CORBA::NVList::_nil()
         );
     }
```

Listing 2-3 includes the Java implementation of the Simple interface from the Simpapp sample application.

Listing 2-3 Java Implmentation of the Simple Interface

```
import com.beasys.Tobj.TP;

/**The SimpleImpl class implements the to_upper and to_lower
/**methods.


public class SimpleImpl extends _SimpleImplBase
{
/*Converts a string to upper case.*/

       public void to_upper(org.omg.CORBA.StringHolder data)
       {
              if (data.value == null)
                   return;
              data.value = data.value.toUpperCase();
              return;
       }
/*Converts a string to lower case.*/

       public String to_lower(String data)
       {
              if (data == null)
                   return null;
              return data.toLowerCase();
       }
}
```

Listing 2-4 includes the Java implementation of the SimpleFactory interface from the Simpapp sample application.

Listing 2-4 Java Implementation of the SimpleFactory Interface

```
import com.beasys.Tobj.TP;

/**The SimpleFactoryImpl class provides code to create the Simple
/**object.


public class SimpleFactoryImpl extends _SimpleFactoryImplBase
```

```
{
/*Create an object reference to a Simple object*/

       public Simple find_simple()
       {
              org.omg.CORBA.Object simple_oref =
                  TP.create_object_reference(
                           SimpleHelper.id(), //Repository ID
                           “simple”,          //object id
                           null               //routing criteria
                           );
              //Send back the narrowed reference
                     return SimpleHelper.narrow(simple_oref);

              };
       };
};
```

Creating the Server Object

The Server object performs the following tasks:

- Initializes the server application, including registering factories, allocating resources needed by the server application, and, if necessary, opening an XA resource manager
- Performs server application shutdown and cleanup procedures
- In C++ server applications, instantiates CORBA objects needed to satisfy client requests

In C++ server applications, the Server object is already instantiated and a header file for the Server object is available. You implement methods that initialize and release the server application, and, if desired, create servant objects.

Listing 2-5 includes the C++ code from the Simpapp sample application for the Server object.

Listing 2-5 C++ Server Object

```
static CORBA::Object_var static_var_factory_reference;

// Method to start up the server


CORBA::Boolean Server::initialize(int argc, char* argv[])
{
       // Create the Factory Object Reference

       static_var_factory_reference =
           TP::create_object_reference(
              _tc_SimpleFactory->id(),
              "simple_factory",
              CORBA::NVList::_nil()
           );
       // Register the factory reference with the FactoryFinder

       TP::register_factory(
            static_var_factory_reference.in(),
           _tc_SimpleFactory->id()
       );
       return CORBA_TRUE;
}
// Method to shutdown the server

void Server::release()
{
// Unregister the factory.

  try {
       TP::unregister_factory(
           static_var_factory_reference.in(),
           _tc_SimpleFactory->id()
       );
  }
  catch (...) {
       TP::userlog("Couldn’t unregister the SimpleFactory");
  }
}
// Method to create servants

Tobj_Servant Server::create_servant(const char*
 interface_repository_id)
{
       if (!strcmp(interface_repository_id,
       _tc_SimpleFactory->id())) {
              return new SimpleFactory_i();
       }
       if (!strcmp(interface_repository_id,
       _tc_Simple->id())) {
              return new Simple_i();
       }
       return 0;
}
```

In Java server applications, you implement the 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 the Server object.

Listing 2-6 includes the Java code from the Simpapp sample application for the server object.

Listing 2-6 Java Server Object

```
import com.beasys.Tobj.TP;

public class ServerImpl
       extends com.beasys.Tobj.Server
{
       static org.omg.CORBA.Object factory_reference;


/**Method to start up the server*/

       public boolean initialize(String[] args)
       {
         try {
         // Create the factory object reference.
                factory_reference = TP.create_object_reference(
                           SimpleFactoryHelper.id(),
                           "simple_factory",
                           null
                           );

// Register the factory reference with the FactoryFinder

                    TP.register_factory(
                    factory_reference,
                      SimpleFactoryHelper.id()
                      );

return true;

      } catch (Exception e){
                      TP.userlog("Couldn’t initialize server: " +
                      e.getMessage());
                      e.printStackTrace();
                      return false;
              }
      }
/** Method to shutdown the server*/

      public void release()
      {
              try {
                      TP.unregister_factory(
                              factory_reference,
                      SimpleFactoryHelper.id()
                               );
              } catch (Exception e){
                      TP.userlog("Couldn’t unregister the
                      SimpleFactory: " + e.getMessage());
                      e.printStackTrace();
              }
      }
}
```

Defining an Object’s Activation Policies

As part of server development, you determine what events cause an object to be activated and deactivated by assigning object activation policies, as follows:

- For C++ server applications, specify object activation policies in the Implementation Configuration File (ICF). A template ICF file is created by the `genicf` command.
- For Java server applications, specify object activation policies in the Server Description File, written in Extensible Markup Language (XML).

Note: You also define transaction policies in the ICF and Server Description Files. For information about using transactions in your WLE application, see the topic “Using Transactions.”

The WLE software supports the following activation policies:

Activation Policy	Description
`method`	Causes the object to be active only for the duration of the invocation on one of the object's operations. This is the default activation policy.
`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 method `TP::deactivateEnable` (C++) or `com.beasys.Tobj.TP.deactivateEnable` (Java) has been invoked on the object.

The Simple interface in the Simpapp sample application is assigned the default activation policy of method. For more information about managing object state and defining object activation policies, see *Creating C++ Server Applications* and *Creating Java Server Applications* on the Online Documentation CD.

Step 4: Writing the Client Application

The WLE software supports the following types of client applications:

- CORBA C++
- CORBA Java
- CORBA Java applets
- ActiveX

The steps for creating client applications are as follows:

1. Initialize the ORB.
2. Use the Bootstrap environmental object to establish communication with the WLE domain.
3. Resolve initial references to the FactoryFinder environmental object.
4. Use a factory to get an object reference for the desired CORBA object.
5. Invoke methods on the CORBA object.

Note: Before you can create ActiveX client applications, you need to ensure that the OMG IDL file for the CORBA interface you want to use is loaded in the Interface Repository, and that bindings have been created for the CORBA interface. For a description of these steps, see *Creating Client Applications* on the Online Documentation CD.

The client development steps are illustrated in Listing 2-7 and Listing 2-8, which include code from the Simpapp sample application. In the Simpapp sample application, the client application uses a factory to get an object reference to the Simple object and then invokes the `to_upper` and `to_lower` methods on the Simple object.

For a detailed description of the development steps with code examples from CORBA C++, CORBA Java, and ActiveX client applications as well as Java applets, see *Creating Client Applications* on the Online Documentation CD.

Listing 2-7 C++ Client Application from the Simpapp Sample Application

```
int main(int argc, char* argv[])
{
    try {
       // Initialize the ORB
       CORBA::ORB_var var_orb = CORBA::ORB_init(argc, argv, "");

       // Create the Bootstrap object
       Tobj_Bootstrap bootstrap(var_orb.in(), "");

       // Use the Bootstrap object to find the FactoryFinder
       CORBA::Object_var var_factory_finder_oref =
         bootstrap.resolve_initial_references("FactoryFinder");

       // Narrow the FactoryFinder
```

```
        Tobj::FactoryFinder_var var_factory_finder_reference =
         Tobj::FactoryFinder::_narrow
         (var_factory_finder_oref.in());

        // Use the factory finder to find the Simple factory
        CORBA::Object_var var_simple_factory_oref =
        var_factory_finder_reference->find_one_factory_by_id(
        _tc_SimpleFactory->id()
        );

        // Narrow the Simple factory
        SimpleFactory_var var_simple_factory_reference =
          SimpleFactory::_narrow(
                                  var_simple_factory_reference.in());

        // Find the Simple object
        Simple_var var_simple =
          var_simple_factory_reference->find_simple();

        // Get a string from the user
        cout << "String?";
        char mixed[256];
        cin >> mixed;

        // Convert the string to upper case :
        CORBA::String_var var_upper = CORBA::string_dup(mixed);
        var_simple->to_upper(var_upper.inout());
        cout << var_upper.in() << endl;

        // Convert the string to lower case
        CORBA::String_var var_lower = var_simple->to_lower(mixed);
        cout << var_lower.in() << endl;

        return 0;
        }
}
```

Listing 2-8 Java Client Application from the Simpapp Sample Application

```
public class SimpleClient
{
        public static void main(String args[])

        // Initialize the ORB.
        ORB orb = ORB.init(args, null);

        // Create the Bootstrap object
        Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb, "");

        // Use the Bootstrap object to locate the FactoryFinder
        org.omg.CORBA.Object factory_finder_reference =
        bootstrap.resolve_initial_references("FactoryFinder");

        // Narrow the FactoryFinder
        FactoryFinder factory_finder_reference =
        FactoryFinderHelper.narrow(factory_finder_reference);

        // Use the FactoryFinder to find the Simple factory.
        org.omg.CORBA.Object simple_factory_reference =
        factory_finder_reference.find_one_factory_by_id
        (SimpleFactoryHelper.id());

        // Narrow the Simple factory
        SimpleFactory simple_factory_reference =
        SimpleFactoryHelper.narrow(simple_factory_reference);

        // Find the Simple object.
        Simple simple = simple_factory_reference.find_simple();

        // Get a string from the user.
        System.out.println("String?");
        String mixed = in.readLine();

        // Convert the string to upper case.
        org.omg.CORBA.StringHolder buf = new
        org.omg.CORBA.StringHolder(mixed);
        simple.to_upper(buf);
        System.out.println(buf.value);

        // Convert the string to lower case.
        String lower = simple.to_lower(mixed);
        System.out.println(lower);
         }
}
```

Step 5: Creating a Configuration File

Because the WLE software offers great flexibility and many options to application designers and programmers, no two applications are alike. An application, for example, may be small and simple (a single client and server running on one machine) or complex enough to handle transactions among thousands of client and server applications. For this reason, for every WLE application being managed, the system administrator must provide a configuration file that defines and manages the components (for example, domains, server applications, client applications, and interfaces) of that application.

When system administrators create a configuration file, they are describing the WLE application using a set of parameters that the WLE software interprets to create a runnable version of the application. During the setup phase of administration, the system administrator's job is to create a configuration file. The configuration file contains the sections listed in Table 2-5.

Table 2-5 Sections in the Configuration File for WLE Applications

Sections in the Configuration File	Description
`RESOURCES`	Defines defaults (for example, user access and the main adminstration machine) for the WLE application
`MACHINES`	Defines hardware-specific information about each mahine running in the WLE application
`GROUPS`	Defines logical groupings of server applications or CORBA interfaces
`SERVERS`	Defines the server application processes (for example, the Transaction Manager) used in the WLE application
`SERVICES`	Defines parameters for services provided by the WebLogic Enterpise application
`INTERFACES`	Defines information about the CORBA interfaces in the WLE application
`ROUTING`	Defines routing critieria for the WLE application

There are two forms of the configuration file:

- An ASCII version of the file, created and modified with any editor. Throughout the WLE documentation, the ASCII version of the configuration file is referred to as the `UBBCONFIG` file. The configuration file may, in fact, be given any file name.

- The `TUXCONFIG` file, a binary version of the `UBBCONFIG` file created using the `tmloadcf` command. When the `tmloadcf` command is executed, the environment variable `TUXCONFIG` must be set to the name and directory location of the `TUXCONFIG` file.

For information about the Configuration file and the `tmloadcf` command, see *Administration Guide* on the Online Documentation CD.

Listing 2-9 shows the configuration file for the Simpapp sample application.

Listing 2-9 Configuration File for Simpapp Sample Application

```
*RESOURCES
       IPCKEY    55432
       DOMAINID  simpapp
       MASTER    SITE1
       MODEL     SHM
       LDBAL     N

*MACHINES
       "PCWIZ"
       LMID         = SITE1
       APPDIR       = "C:\WLEDIR\MY_SIM~1"
       TUXCONFIG    = "C:\WLEDIR\MY_SIM~1\results\tuxconfig"
       TUXDIR       = "C:\WLEDIR"
       MAXWSCLIENTS = 10

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

*SERVERS
       DEFAULT:
          RESTART = Y
          MAXGEN  = 5
       TMSYSEVT
          SRVGRP  = SYS_GRP
          SRVID   = 1
       TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 2
          CLOPT   = "-A -- -N -M"
```

```
        TMFFNAME
           SRVGRP  = SYS_GRP
           SRVID   = 3
           CLOPT   = "-A -- -N"
        TMFFNAME
           SRVGRP  = SYS_GRP
           SRVID   = 4
           CLOPT   = "-A -- -F"
        simple_server
           SRVGRP  = APP_GRP
           SRVID   = 1
           RESTART = N
        ISL
           SRVGRP  = SYS_GRP
           SRVID   = 5
          CLOPT   = "-A -- -n //PCWIZ:2468"

*SERVICES
```

When creating Java server applications, include the `JavaServer` parameter in the `UBBCONFIG` file to start the Java server application. For example:

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

If you are using an XA-compliant resource manager, use the `JavaServerXA` parameter in place of the `JavaServer` parameter to associate the XA resource manager with a specified server group.

Step 6: Compiling the Server Application

You use the `buildobjserver` command to compile and link C++ server applications. The `buildobjserver` command has the following format:

```
buildobjserver [-o servername] [options]
```

In the `buildobjserver` command syntax:

- `-o` *`servername`* represents the name of the server application to be generated by this command.
- *`options`* represents the command line options to the `buildobjserver` command.

When creating Java server applications, use the `javac` compiler to create the bytecodes for all the class files that comprise your WLE application. This set of files includes the `*.java` source files generated by the `m3idltojava` compiler, plus the object implementation files and server class files you created.

You use the `buildjavaserver` command to build a Java ARchive (JAR) file and link the Java server applications. 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.

You then have to specify the location of the JAR file for your Java server application in the `APPDIR` system environment variable. On Windows NT systems, this directory must be on a local drive (not a networked drive). On Solaris, the directory can be local or remote.

Step 7: Compiling the Client Application

The final step in the development of the CORBA client application is to produce the executable client application. To do this, you need to compile the code and then link against the client stub.

When creating CORBA C++ client applications, use the `buildobjclient` command to construct a WLE client application executable. The command combines the client stubs for interfaces that use static invocation, and the associated header files, with the standard WLE libraries to form a client executable. For the syntax of the `buildobjclient` command, see *C++ Programming Reference* on the Online Documentation CD.

When creating CORBA Java client applications, see your Java ORB's documentation for information about building client executables. You need to include the `m3envobj.jar` file in your `CLASSPATH` when you compile the CORBA Java client application. The `m3envobj.jar` file contains the Java classes for the WLE environmental objects.

The `m3envobj.jar` file built against the Netscape Enterprise server is located in the following directory:

```
WLEdir/udataobj/java/netscape
```

Additional WLE Sample Applications

Sample applications demonstrate the tasks involved in developing a WLE application, and provide sample code that can be used by client and server programmers to build their own WLE application. The following additional sample applications are provided:

- University sample applications
- Java sample applications

Code from the sample applications is used throughout this manual to illustrate the development steps. A complete description of building and running the sample applications is provided in the following:

- *Guide to the University Sample Applications*
- *Guide to the Java Sample Applications*

Univeristy Sample Applications

The University sample applications are based on client and server applications implemented at a university. Each University sample application demonstrates a new WLE feature while building on the experience obtained from the previous sample application. The University sample applications are intentionally simplified to demonstrate only the steps and processes associated with using a particular feature of the WLE product.

Table 2-6 describes the University sample applications.

Table 2-6 The University Sample Applications

University Sample Application	Description
Basic	Describes how to develop WLE client and server applications and configure the WLE application. Building C++ server applications and CORBA C++, CORBA Java, and ActiveX client applications are demonstrated.
Security	Adds application-level security to the client applications and to the WLE application.
Transactions	Adds transactional objects to the C++ server application and client applications in the Basic sample application. The Transactions sample application demonstrates how to use the Implementation Configuration File (ICF) to define transaction policies for CORBA objects.
Wrapper	Demonstrates how to wrap an existing BEA TUXEDO application as a CORBA object.
Production	Demonstrates replicating server applications, creating stateless objects, and implementing factory-based routing in server applications.

Java Sample Applications

The Java sample applications demonstrate the process of developing Java server applications with the WLE product. In addition, the Java sample applications focus on using database products, such as Oracle and Microsoft SQL Server, with a WLE application. The Java sample applications listed in Table 2-7 are provided.

Table 2-7 Java Sample Applications

Java Sample Application	Description
Java Simpapp	Provides a Java client application and a Java server application. The Java server application contains two operations that manipulate strings received from the Java client application.
JDBC Bankapp	Implements an automatic teller machine (ATM) interface and uses Java Database Connectivity (JDBC) to access a database that stores account and customer information.
XA Bankapp	Implements the same ATM interface as JDBC Bankapp; however, XA Bankapp uses a database XA library to demonstrate using the Transaction Manager to coordinate transactions.

3 Using Security

This chapter discusses the following topics:

- Overview of the Security Service
- How Security Works
- The Security Sample Application
- Development Steps

Overview of the Security Service

The WLE product offers a security model based on the CORBAservices Security Service. The WLE security model implements the authentication portion of the CORBAservices Security Service.

Security information is defined on a domain basis. The security level for the domain is defined in the configuration file. Client applications use the SecurityCurrent object to provide the necessary authentication information to log on to the WLE domain.

The following levels of authentication are provided:

- TOBJ_NOAUTH

  No authentication is needed; however, the client application may still authenticate itself, and may specify a user name and a client application name, but no password.

- TOBJ_SYSAUTH

  The client application must authenticate itself to the WLE domain and must specify a user name, client application name, and application password.

- TOBJ_APPAUTH

  In addition to the TOBJ_SYSAUTH information, the client application must provide application-specific information. If the default WLE authentication service is used in the application configuration, the client application must provide a user password; otherwise, the client application provides authentication data that is interpreted by the custom authentication service in the application.

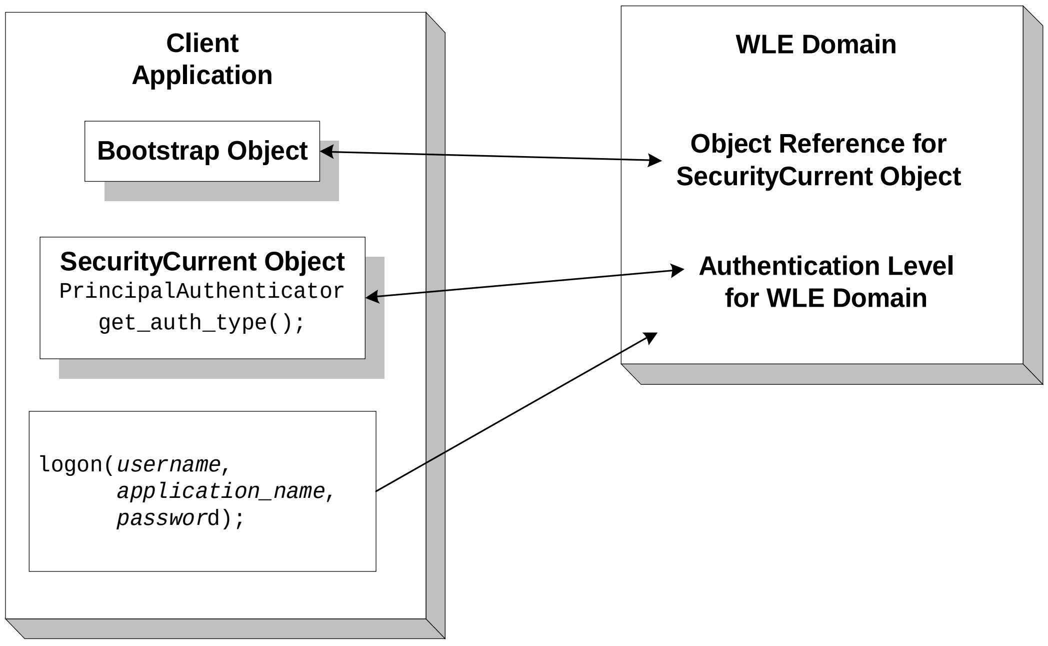
Note: If a client application is not authenticated and the security level is `TOBJ_NOAUTH`, the IIOP Listener/Handler of the WLE domain registers the client application with the user name and client application name sent to the IIOP Listener/Handler.

In the WLE software, only the PrincipalAuthenticator and Credentials properties on the SecurityCurrent object are supported. For a description of the `SecurityLevel1::Current` and `SecurityLevel2::Current` interfaces, see the *C++ Programming Reference* or the *Java Programming Reference* on the Online Documentation CD.

How Security Works

Figure 3-1 illustrates how security works in a WLE domain.

Figure 3-1 How Security Works in a WLE Domain



The steps are as follows:

1. The client application uses the Bootstrap object to return an object reference to the SecurityCurrent object for the WLE domain.
2. The client application obtains the PrincipalAuthenticator.
3. The client application uses the `Tobj::PrincipalAuthenticator::get_auth_type()` method to get the authentication level for the WLE domain.
4. The proper authentication level is returned to the client application.
5. The client application uses the `Tobj::PrincipalAuthenticator::logon()` method to log on to the WLE domain with the proper authentication information.

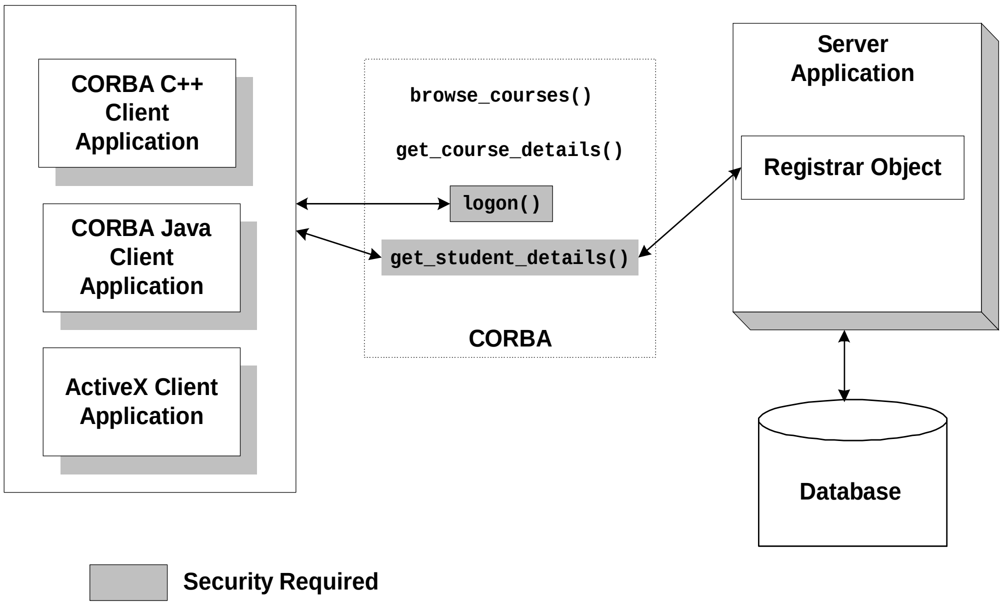
The Security Sample Application

The Security sample application demonstrates application-level security. The Security sample application requires each student using the application to have an ID and a password. The Security sample application works in the following manner:

- The client application has a logon operation. This operation invokes operations on the PrincipalAuthenticator object, which is obtained as part of the process of logging on to access the domain.
- The server application implements a `get_student_details()` operation on the `Registrar` object to return information about a student. After the user is authenticated, logon is complete, the `get_student_details()` operation accesses the student information in the database to obtain the student information needed by the client logon operation.
- The database in the Security sample application contains course and student information.

Figure 3-2 illustrates the Security sample application.

Figure 3-2 Security Sample Application



The source files for the Security sample application are located in the `\samples\corba\university` directory in the WLE software. For information about building and running the Security sample application, see the *Guide to the University Sample Applications* on the Online Documentation CD.

Development Steps

Table 3-1 lists the development steps for writing a WLE application that has security.

Table 3-1 Development Steps for WLE Applications That Have Security

Step	Description
1	Define the security level in the configuration file.
2	Write the client application.

Step 1: Defining the Security Level in the Configuration File

The security level for a WLE domain is defined by setting the `SECURITY` parameter `RESOURSES` section of the configuration file to the desired security level. Table 3-2 lists the options for the `SECURITY` parameter.

Table 3-2 Options for the `SECURITY` Parameter

Option	Definition
`NONE`	No security is implemented in the domain. This option is the default. This option maps to the TOBJ_NOAUTH level of authentication.
`APP_PW`	Requires that client applications provide an application password during initialization. The `tmloadcf` command prompts for an application password. This option maps to the TOBJ_APPAUTH level of authentication.
`USER_AUTH`	Requires an application password and performs a per-user authentication during the initialization of the client application. This option maps to the TOBJ_SYSAUTH level of authentication.

In the Security sample application, the `SECURITY` parameter is set to `APP_PW` for application-level security. For information about adding security to a WLE domain, see the *Administration Guide* on the Online Documentation CD.

Step 2: Writing the Client Application

Write client application code that does the following:

1. Uses the Bootstrap object to obtain a reference to the SecurityCurrent object for the specific WLE domain.
2. Gets the PrincipalAuthenticator object from the SecurityCurrent object.
3. Uses the `get_auth_type` operation of the PrincipalAuthenticator object to return the type of authentication expected by the WLE domain.

Listing 3-1 and Listing 3-2 include the portions of the CORBA C++ and CORBA Java client applications in the Security sample application that illustrate the development steps for security. To see an example of the code for ActiveX client applications, see the *Guide to the University Sample Applications* on the Online Documentation CD.

Listing 3-1 Example of Security in a CORBA C++ Client Application

```
CORBA::Object_var var_security_current_oref =
     bootstrap.resolve_initial_references(“SecurityCurrent”);
SecurityLevel2::Current_var var_security_current_ref =
     SecurityLevel2::Current::_narrow(var_security_current_oref.in());


//Get the PrincipalAuthenticator
SecurityLevel2::PrincipalAuthenticator_var var_principal_authenticator_oref =
     var_security_current_oref->principal_authenticator();
//Narrow the PrincipalAuthenticator
Tobj::PrincipalAuthenticator_var var_bea_principal_authenticator =
     Tobj::PrincipalAuthenticator::_narrow
                                     var_principal_authenticator_oref.in());

//Determine the security level
Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
Security::AuthenticationStatus status = var_bea_principalauthenticator->logon(
                                                   user_name,
                                                   client_name,
                                                  system_password,
                                                  user_password,
                                                  0);
```

Listing 3-2 Example of Security in a CORBA Java Client Application

```
org.omg.CORBA.Object SecurityCurrentObj =
      gBootstrapObjRef.resolve_initial_references(“SecurityCurrent”);
org.omg.SecurityLevel2.Current secCur =
     org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);

//Get the PrincipalAuthenticator
org.omg.SecurityLevel2.PrincipalAuthenticator authlevel2 =
     secCur.principal_authenticator();
//Narrow the PrincipalAuthenticator
com.beasys.Tobj.PrincipalAuthenticatorObjRef gPrinAuthObjRef =
     (com.beasys.Tobj.PrincipalAuthenticator)
     org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow(authlevel2);

//Determine the security level
com.beasys.Tobj.Authtype authType = gPrinAuthObjRef.get_auth_type();

org.omg.Security.AuthenticationStatus status = gPrinAuthObjRef.logon
     (gUserName, ClientName, gSystemPassword, gUserPassword,0);
```

4 Using Transactions

This chapter discusses the following topics:

- Overview of the Transaction Service
- When to Use Transactional Objects
- What Happens During a Transaction
- Transactions Sample Application
- Development Steps

Overview of the Transaction Service

One of the most fundamental features of the WLE product is transaction management. Transactions are a means to guarantee that database transactions are completed accurately and that they take on all the **ACID properties** (atomicity, consistency, isolation, and durability) of a high-performance transaction. The WLE system protects the integrity of your transactions by providing a complete infrastructure for ensuring that database updates are done accurately, even across a variety of resource managers. The WLE system includes the following:

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

  The WLE product provides a C++ interface to the OTS and a Java interface to the OTS and the JTS. The JTS is the Sun Microsystems, Inc. Java interface for transaction services, and is based on the OTS. The OTS and the JTS are accessed through the TransactionCurrent environmental object. For information about using the TransactionCurrent environmental object, see the *C++ Programming Reference* or the *Java Programming Reference* on the Online Documentation CD.

- The Sun Microsystems, Inc. Java Transaction API (JTA).

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

  - The `javax.transaction` package description in the *Java API Reference*.
  - The Java Transaction API specification, published by Sun Microsystems, Inc. and available from the Sun Microsystems, Inc. Web site. (See the WLE version 4.2 Release Notes for information about obtaining this document.)

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

- Creates a global transaction identifier when a client application initiates a transaction.
- Works with the TP Framework to track objects that are involved in a transaction and, therefore, need to be coordinated when the transaction is ready to commit.
- Notifies the resource managers—which are, most often, databases—when they are accessed on behalf of a transaction. Resource managers then lock the accessed records until the end of the transaction.
- Orchestrates the two-phase commit when the transaction completes, which ensures that all the participants in the transaction commit their updates simultaneously. It coordinates the commit with any databases that are being updated using Open Group's XA protocol. Almost all relational databases support this standard.
- Executes the rollback procedure when the transaction must be stopped.
- Executes a recovery procedure when failures occur. It determines which transactions were active in the machine at the time of the crash, and then determines whether the transaction should be rolled back or committed.

When to Use Transactions

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

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

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

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

  - Data is cached in memory or written to a database during or after each successive invocation.
  - Data is written to a database at the end of the conversation.
  - The client application needs the object to maintain an in-memory context between each invocation; that is, each successive invocation uses the data that is being maintained in memory across the conversation.
  - At the end of the conversation, the client application needs the ability to cancel all database write operations that may have occurred during or at the end of the conversation.

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

- Within the scope of a single client invocation on an object, the object performs multiple edits to data in a database. If one of the edits fails, the object needs a mechanism to roll back all the edits. (In this situation, the individual database edits are not necessarily CORBA invocations.)

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

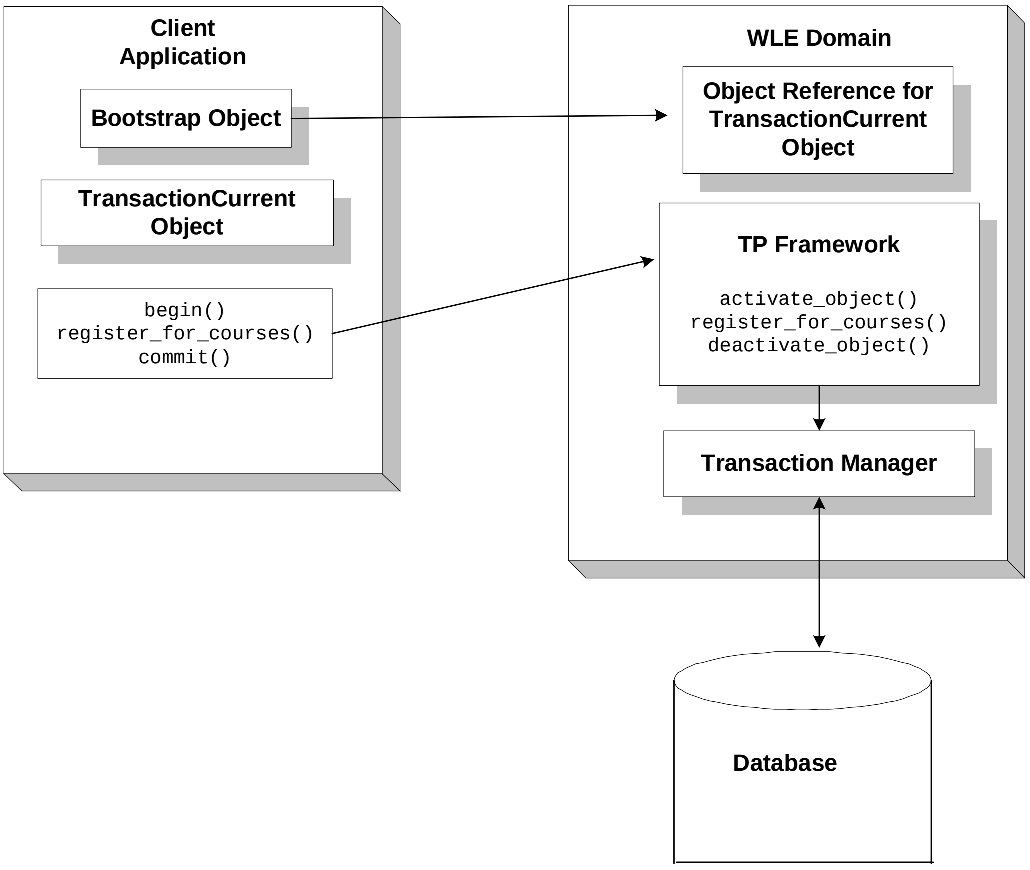
  - Invoking the debit method on one account
  - Invoking the credit method on another account

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

What Happens During a Transaction

Figure 4-1 illustrates how transactions work in a WLE application.

Figure 4-1 How Transactions Work in a WLE Application



A basic transaction works in the following way:

1. The client application uses the Bootstrap object to return an object reference to the TransactionCurrent object for the WLE domain.

2. A client application begins a transaction using the `Tobj::TransactionCurrent::begin` method, and issues a request to the CORBA interface through the TP Framework. All operations on the CORBA interface execute within the scope of a transaction.

- If a call to any of these operations raises an exception (either explicitly or as a result of a communication failure), the exception can be caught and the transaction can be rolled back.
- If no exceptions occur, the client application commits the current transaction using the `Tobj::TransactionCurrent::commit` method. This method ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

3. The `Tobj::TransactionCurrent:commit` method causes the TP Framework to call the Transaction Manager to complete the transaction.
4. The Transaction Manager updates the database.

Transactions Sample Application

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

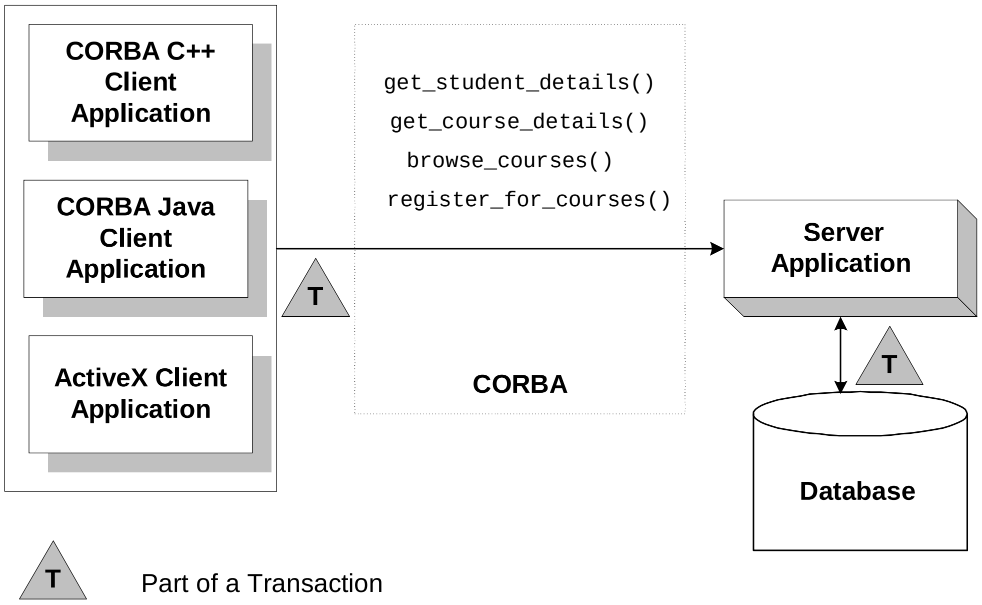
The Transactions sample application works in the following way:

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

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

Figure 4-2 illustrates how the Transactions sample application works.

Figure 4-2 Transactions Sample Application



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

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

Note: For information about how transactions are implemented in Java WLE applications, see the description of the XA Bankapp sample application in the *Guide to the Java Sample Applications* on the Online Documentation CD.

Development Steps

This topic describes the development steps for writing a WLE application that includes transactions. Table 4-1 lists the development steps.

Table 4-1 Development Steps for WLE Applications That Have Transactions

Step	Description
1	Write the OMG IDL for the transactional CORBA interface.
2	Define the transaction policies for the CORBA interface in the Implementation Configuration file (ICF) for C++ WLE applications, or in the Server Description File for Java WLE client applications.
3	Write the client application.
4	Write the server application.
5	Create a configuration file.

The Transactions sample application is used to demonstrate these development steps. The source files for the Transactions sample application are located in the `\samples\corba\university` directory of the WLE software. For information about building and running the Transactions sample application, see the *Guide to the University Sample Applications* on the Online Documentation CD.

The XA Bankapp sample application demonstrates how to use transactions in Java WLE applications. The source files for the XA Bankapp sample application are located in the `\samples\corba\bankapp_java` directory of the WLE software. For information about building and running the XA Bankapp sample application, see the *Guide to the Java Sample Applications* on the Online Documentation CD .

Step 1: Writing the OMG IDL

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

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

Listing 4-1 includes the OMG IDL for the Transactions sample application.

Listing 4-1 OMG IDL for the Transactions Sample Application

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

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

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

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

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

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

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

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

struct StudentDetails
{
       StudentId         student_id;
       string            name;
```

```
        CourseDetailsList registered_courses;
};

enum NotRegisteredReason
{
        AlreadyRegistered,
        NoSuchCourse
};

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

exception TooManyCredits
{
        unsigned short maximum_credits;
};

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

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

};

// The RegistrarFactory interface finds Registrar interfaces.

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

Step 2: Defining Transaction Policies for the Interfaces

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

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

During development, you decide which interfaces will execute in a transaction by assigning transaction policies, as follows:

- ♦ For C++ server applications, you specify transaction policies in the Implementation Configuration File (ICF). A template ICF file is created by the `genicf` command.
- ♦ For Java server applications, you specify transaction policies in the Server Description File, written in Extensible Markup Language (XML).

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

Step 3: Writing the Client Application

The client application needs code that performs the following tasks:

1. Obtains a reference to the TransactionCurrent object from the Bootstrap object.
2. Begins a transaction by invoking the `Tobj::TransactionCurrent::begin()` operation on the TransactionCurrent object.
3. Invokes operations on the object. In the Transactions sample application, the client application invokes the `register_for_courses()` operation on the `Registrar` object, passing a list of courses.

Listing 4-2 illustrates the portion of the CORBA C++ client applications in the Transactions sample application that illustrates the development steps for transactions.

For an example of a CORBA Java client application that uses transactions, see the XA Bankapp sample application in the *Guide to the Java Sample Applications* on the Online Documentation CD. For an example of using transactions in an ActiveX client application, see *Creating Client Applications* on the Online Documentation CD.

Listing 4-2 Transactions Code for CORBA C++ Client Applications

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

Step 4: Writing the Server Application

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

If your WLE application uses a database, you need to include in the server application code that opens and closes an XA Resource Manager. These operations are included in the `Server::initialize()` and `Server::release()` operations of the Server object. Listing 4-3 shows the portion of the code for the Server object in the Transactions sample application that open and closes the XA Resource Manager.

Note: For a complete example of a C++ server application that implements transactions, see the Transactions sample application in the *Guide to the University Sample Applications*.

For an example of a Java server application that implements transactions, see the description of the XA Bankapp sample application in the *Guide to the Java Sample Applications* on the Online Documentation CD.

Listing 4-3 C++ Server Object in Transactions Sample Application

```
CORBA::Boolean Server::initialize(int argc, char* argv[])
{
       TRACE_METHOD("Server::initialize");
       try {
              open_database();
              begin_transactional();
              register_fact();
              return CORBA_TRUE;
}
       catch (CORBA::Exception& e) {
              LOG(“CORBA exception : “ <<e);
       }
```

```
        catch (SamplesDBException& e) {
                LOG(“Can’t connect to database”);
        }
        catch (...) {
                LOG(“Unexpected database error : “ <<e);
        }
        catch (...) {
                LOG(“Unexpected exception”);
        }
        cleanup();
        return CORBA_FALSE;
}

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

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

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

Step 5: Creating a Configuration File

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

- In the `SERVERS` section:
  - Define a server group that includes both the server application that includes the interface and the server application that manages the database. This server group needs to be specified as transactional.
  - Replace `JavaServer` with `JavaServerXA` to associate the XA resource manager with a specified server group. (`JavaServer` uses the null RM.)
- In the `OPENINFO` parameter of the Groups section, include information to open the resource manager for the database. You obtain this information from the product documentation for your database. Note that the default version of the `com.beasys.Tobj.Server.initialize` method automatically opens the resource manager.
- Include the pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter. For more information about the transaction log, see the *Administration Guide*.

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

Listing 4-4 Configuration File for Transactions Sample Application

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

*MACHINES
       BLOTTO
```

```
        LMID = SITE1
        APPDIR = C:\TRANSACTION_SAMPLE
        TUXCONFIG=C:\TRANSACTION_SAMPLE\tuxconfig
        TLOGDEVICE=C:\APP_DIR\TLOG
        TLOGNAME=TLOG
        TUXDIR="C:\WLEdir"
        MAXWSCLIENTS=10

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

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

*SERVERS
        DEFAULT:
        RESTART = Y
        MAXGEN  = 5

        TMSYSEVT
          SRVGRP  = SYS_GRP
          SRVID   = 1

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

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

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

        TMIFRSVR
          SRVGRP  = SYS_GRP
          SRVID   = 5
```

```
       UNIVT_SERVER
         SRVGRP  = ORA_GRP
         SRVID   = 1
         RESTART = N

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

*SERVICES
```

For information about the transaction log and defining parameters in the Configuration file, see the *Administration Guide*.

Index

A

activation policies
  defining in Implementation Configuration file 2-15
  defining in Server Description file 2-15
  Simpapp sample application 2-15
  Simple interface 2-16
  supported 2-16
ActiveX application builder
  description 1-8
AdminAPI
  description 1-8
administration commands
  tmadmin command 1-6
  tmboot command 1-6
  tmconfig command 1-6
  tmloadcf command 1-6
  tmshutdown command 1-6
  tmunloadcf command 1-6
Administration console
  description 1-7
administration tools
  AdminAPI 1-8
  administration commands 1-6
  Administration console 1-7
authentication
  client application 1-20
  levels 3-1

B

Bootstrap object
  description 1-13
  illustrated 1-13
  Simpapp sample application 2-17
building
  C++ client applications 2-23
    buildobjclient command 1-5
  C++ server applications
    buildobjserver command 1-5
    genicf command 1-5
  Java client applications 2-23
  Java server applications
    buildjavaserver command 1-5
buildjavaserver command
  building Java server applications 1-5
  description 1-5
  format 2-22
  in the Simpapp sample application 2-22
buildobjclient command
  building C++ client applications 1-5
  description 1-5
  format 2-23
  in the Simpapp sample application 2-23
buildobjserver command
  building C++ server applications 1-5
  description 1-5
  format 2-22
  in the Simpapp sample application 2-22
buildXAJS command
  building an XA resource manager 1-5

description 1-5

C

client applications
authenticating into the WLE domain 1-20
initialization process 1-19
invoking objects 1-23
using transactions 4-5
writing
Security sample application 4-13
Simpapp sample application 2-16
Transactions sample application 4-13
client stubs
generating 2-6
in Simpapp sample application 2-6
code example
C++ client application for Simpapp sample application 2-17
C++ implementation of the Simple interface 2-10
C++ Server object 2-13
C++ server object that supports transactions 4-14
configuration file for Simpapp sample application 2-21
Java client application for the Simpapp sample application 2-18
Java implementation of SimpleFactory interface 2-11
Java implementation of the Simple Interface 2-10
Java Server object 2-14
OMG IDL for Simpapp sample application 2-6
OMG IDL for Transactions sample application 4-9
security in C++ client applications 3-7
security in Java client applications 3-8
transactions in C++ client application 4-13
UBBCONFIG file for Transactions sample application 4-15
compiling
C++ client applications 2-23
C++ server applications 2-22
Java client applications 2-23
Java server applications 2-22
CORBAservices Object Transaction Service
using in WLE applications 4-1
create_servant method 1-21

D

development commands
buildjavaserver 1-5
buildobjclient command 1-5
buildobjserver command 1-5
buildXAJS command 1-5
genicf command 1-5
idl2ir command 1-5
ir2idl command 1-5
irdel command 1-5
development process
activation policies 2-16
client applications
Security sample application 3-7
Simpapp sample application 2-16
Transactions sample application 4-13
defining object activation policies 2-15
illustrated 2-2
Implementation Configuration file 2-15
OMG IDL
Simpapp sample application 2-5
Transactions sample application 4-9
Security sample application 3-6
server applications
Simpapp sample application 2-8
Transactions sample

application 4-14
Server Description file 2-15
Simpapp sample application 2-4
steps for creating WLE applications 2-2
Transactions sample application 4-8
WLE applications 2-2
writing a configuration file 2-19
writing server application code 2-8
writing the client application code 2-16
writing the OMG IDL 2-5

E

environmental objects
and client initialization 1-19
description 1-10

F

factories
finding 1-21
registering 1-21
FactoryFinder object
description 1-10
example use of 1-21

G

genicf command
creating a ICF file 1-5
description 1-5

I

idl command 1-4
description 1-4
files created by 2-7
generating client stubs 2-6
generating skeletons 2-6
IDL compiler
idl command 1-4
m3idltojava command 1-4
supported 1-4
idl2ir command
description 1-5
IIOP
definition 1-2
use in WLE product 1-2
IIOP Listener/Handler
description 1-14
Implementation Configuration file
defining activation policies 2-15
defining transaction policies 4-12
initialize method
summary 1-18, 1-19
Interface Repository
creating 1-5
deleting objects from 1-5
displaying the contents 1-5
idl2ir command 1-5
ir2idl command 1-5
irdel command 1-5
loading interface definitions into 1-5
InterfaceRepository object
description 1-10
interfaces
writing methods to implement operations 2-9
ir2idl command
description 1-5
irdel command
description 1-5

J

Java client applications
required files 2-23
Java Transaction Service
using in WLE applications 4-1

M

m3idltojava command 1-4

description 1-4
files created by 2-7
generating client stubs 2-7
generating skeletons 2-7
Management Information Base
see MIB 1-3
managing
WLE applications
tmadmin command 1-6
tmboot command 1-6
tmconfig command 1-6
tmloadcf command 1-6
tmshutdown command 1-6
tmunloadcf command 1-6
method implementations
C++ 2-9
Java 2-10
writing 2-9
MIB
for WLE applications 1-3

O
Object Life Cycle service
description 1-10
object request broker
see ORB 1-15
object services
Interface Repository 1-10
Object Life Cycle service 1-10
Security service 1-10
Transaction service 1-10
objects
invoking 1-23
OMG IDL
compiling 2-6
generating client stubs 2-6
generating skeletons 2-6
Simple interface 2-5, 2-6
SimpleFactory interface 2-5, 2-6
Transactions sample application 4-9
ORB
description 1-15
illustrated 1-15

P
POA
description 1-15
interaction with TP Framework 1-16
Portable Object Adapter
see POA 1-15
PrincipalAuthenticator object
using in client applications 3-4
programming tools 1-4

R
register_factory method
example of 1-21
resolve_initial_references method 1-19

S
Security sample application
defining security level 3-6
description 3-4
development process 3-6
illustrated 3-4
location of files 3-5
PrincipalAuthenticator object 3-4
SecurityCurrent object 3-4
using the PrincipalAuthenticator object 3-7
using the SecurityCurrent object 3-7
writing the client application 3-7
Security service
description 1-10
functional description 3-2
SecurityCurrent object
description 1-10
using in client applications 3-4

server applications
defining object activation policies 2-15
Implementation Configuration file 2-15
Server Description file 2-15
writing
Simpapp sample application 2-8
Transactions sample application 4-14
method implementations 2-9
the Server object 2-12
Server Description file
defining activation policies 2-15
defining transaction policies 4-12
Server object 4-14
description 1-18
Transactions sample application 4-14
writing 2-12
Simpapp sample application
compiling
C++ client application 2-23
C++ server application 2-22
Java client application 2-23
compiling Java server application 2-22
configuration file 2-19
description 2-4
file location 2-4
illustrated 2-4
interfaces defined for 2-5
OMG IDL 2-5
using the Bootstrap object 2-17
using the buildjavaserver command 2-22
using the buildobjserver command 2-22
writing the client application code 2-16
Simple interface
activation policy 2-16
OMG IDL 2-5
Simple Network Management Protocol
see SNMP 1-3
SimpleFactory interface
OMG IDL 2-5
skeletons
generating 2-6
in Simpapp sample application 2-6
SNMP
in the WLE product 1-3
support
documentation xiv
technical xiv
supporting databases 4-14

T

TLOGDEVICE parameter 4-15
tmadmin command
description 1-6
tmboot command
description 1-6
tmconfig command
description 1-6
tmloadcf command
creating a configuraiton file 2-20
description 1-6
tmshutdown command
description 1-6
tmunloadcf command
description 1-6
Tobj_Bootstrap 1-19
TP Framework
description 1-16
illustrated 1-17
transaction policies
defined 4-12
Transaction service
description 1-10, 4-1
features 4-2
TransactionCurrent object
description 1-10
transactions
functional overview 4-4
illustrated 4-5
in client applications 4-5
OMG IDL 4-6

restrictions 4-17
when to use 4-3
Transactions sample application
description 4-6
development process 4-8
file location 4-9
illustrated 4-7
OMG IDL 4-9
starting server application 4-14
transaction policies 4-12
UBBCONFIG file 4-15
writing client applications 4-13
writing server applications 4-14
TUXCONFIG file
description 2-20

U

UBBCONFIG file
adding transactions 4-15
description 2-20
sections in 2-20
setting the security level 3-6
user exceptions
Transactions sample application 4-7
UserTransaction object
description 1-10

W

WLE applications
defining security levels 3-6
how they work 1-18
managing
tmadmin command 1-6
tmboot command 1-6
tmconfig command 1-6
tmloadcf command 1-6
tmshutdown command 1-6
tmunloadcf command 1-6
using CORBAservices Object Transaction Service 4-1
using Java Transaction Service 4-1
WLE components
IIOP Listener/Handler 1-14
illustrated 1-12
ORB 1-15
TP Framework 1-16
WLE domain
adding security to 3-4
WLE product
ActiveX application builder 1-8
administration tools 1-6
description of components 1-11
development commands 1-5
features 1-3
functionality overview 1-1
how client and server applications work 1-18
IDL compilers 1-4
illustrated 1-2
object services 1-10
programming tools 1-4