BEA WebLogic Enterprise

Getting Started

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Copyright

Copyright © 1999 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, ObjectBroker, TOP END, and Tuxedo are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Connect, BEA Manager, BEA MessageQ, BEA Jolt, M3, eSolutions, eLink, WebLogic, and WebLogic Enterprise are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

Getting Started

Document Edition	Part Number	Date	Software Version
5.0	861-001001-004	December 1999	BEA WebLogic Enterprise 5.0

Contents

3. The WLE Enterprise JavaBeans (EJB) Programming Environment

Part II. Developing WLE CORBA Applications

4. Developing WebLogic Enterprise (WLE) CORBA Applications

5. Using Security

6. Using Transactions

Part III. Developing WLE Enterprise JavaBeans

7. Designing and Developing Enterprise JavaBeans for the WLE System

8. Building and Deploying Enterprise JavaBeans (EJBs)

About This Document

This document presents an overview of the BEA WebLogic Enterprise (WLE) product and describes the development process for developing distributed CORBA and Enterprise JavaBeans (EJB) applications using the WLE software.

The *Getting Started* document does not discuss every feature of the WLE product; instead, it gives a general description of building a typical application or bean using the WLE programming environment. For information about all the WLE features, see the Developer Guides page in the WebLogic Enterprise online documentation.

This document covers the following topics:

- Chapter 1, "Overview of the WLE Product," presents an overview of the WLE product.
- Chapter 2, "The WLE CORBA Programming Environment," describes the CORBA programming environment available in the WLE product and the architectural components of the CORBA programming environment.
- Chapter 3, "The WLE Enterprise JavaBeans (EJB) Programming Environment," describes the EJB programming environment available in the WLE product and the architectural components of the EJB programming environment.
- Chapter 4, "Developing WebLogic Enterprise (WLE) CORBA Applications," explains how to build a typical WLE application, using the Simpapp sample application as an example.
- Chapter 5, "Using Security," describes how security is incorporated into a WLE CORBA application. The Security sample application is used as an example.
- Chapter 6, "Using Transactions," describes how transactions are incorporated into a WLE CORBA application. The Transactions sample application is used as an example.

- Chapter 7, “Designing and Developing Enterprise JavaBeans for the WLE System,” explains how to create a typical EJB using the WLE programming environment.
- Chapter 8, “Building and Deploying Enterprise JavaBeans (EJBs),” describes the tasks required to build and deploy an EJB in the WLE programming environment.

What You Need to Know

This document is intended for programmers who want to familiarize themselves with the WLE programming environment and create either distributed CORBA or Enterprise JavaBeans applications using the WLE product.

e-docs Web Site

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

How to Print the Document

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

A PDF version of this document is available on the WebLogic Enterprise documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document

(or a portion of it) in book format. To access the PDFs, open the WebLogic Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

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

Related Information

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

Contact Us!

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

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

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSupport at www.beasys.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

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

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
. . .	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Part I Overview of the WLE Product and Programming Environments

1 Overview of the WLE Product

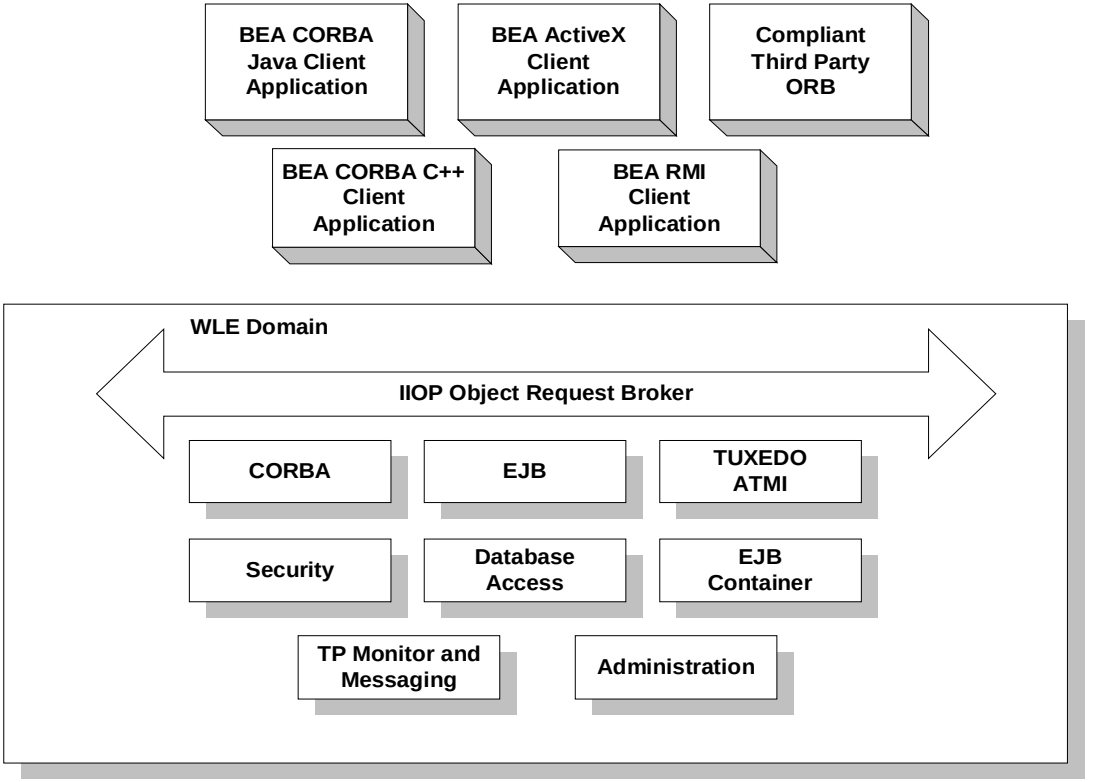
The BEA WebLogic Enterprise (WLE) product allows you to build, deploy, and manage component-based solutions for your enterprise. WLE brings together the Object Request Broker (ORB) and online transaction processing (OLTP) functions with industry programming models such as ATMI, CORBA, and EJB. The result is a platform that enables you to deliver scalable, secure, and transactional e-commerce applications in a well-managed environment.

The WLE product now lets you add Enterprise JavaBeans (EJBs) to your existing WLE applications.WLE CORBA objects and EJBs share the same WLE transaction, security, configuration, and monitoring infrastructure. Therefore, you can write transactions that span TUXEDO services, CORBA objects, and EJBs. In addition, the WLE product supports client applications built using Remote Method Invocation (RMI). RMI provides a pure Java application programming interface (API) for accessing remote objects. Using RMI, client applications can directly access EJBs.

WLE CORBA objects use the Object Management Group (OMG) Internet Inter-ORB Protocol (IIOP) in a WebLogic Enterprise environment. 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



The WLE product provides:

- A set of integrated components that can be used to build robust, distributed e-commerce applications. These components can be accessed from C++, Java, COM, or RMI client applications.
- A choice of server-side components. One or more EJB, RMI, CORBA Java, CORBA C++, or TUXEDO ATMI server-side components can be deployed in a single WLE application.
- Interoperability with third-party vendor CORBA ORB products. IIOP-compliant ORBs may send client requests to a WLE application.
- Access to databases via a Java API is provided with JDBC drivers for Oracle and Microsoft SQL server databases. A JDBC-XA driver is provided for Oracle 8.1.5 databases. These drivers can also be deployed in a more optimal manner using the WLE connection pooling feature.
- A Management Information Base (MIB) that defines the key management attributes of WLE applications.
- Support for the Java Naming and Directory Interface (JNDI) used for finding and registering EJB Home objects and RMI objects.
- Support for the CORBA and Java transaction services to ensure the integrity of your data even when transactions span multiple programming models, databases, and applications.
- A security service that handles authentication for principals that need to access resources in a CORBA or EJB server environment. Access control lists (ACLs) are also provided for EJBs in your WLE application.
- An interface repository that stores meta information about WLE CORBA objects. Meta information includes information about modules, interfaces, operations, attributes, and exceptions.
- Dynamic Invocation Interface (DII) support. DII allows WLE CORBA client applications to dynamically create requests for objects that were not defined at compile time.

The topics in *Getting Started* describe the CORBA and EJB programming environments of the WLE product and the development process for building a transactional application using either programming environment.

2 The WLE CORBA Programming Environment

This topic includes the following sections:

- Overview of the WLE CORBA Programming Features
- WLE CORBA Object Services
- WLE CORBA Components
- How WLE CORBA Client and Server Applications Interact

Overview of the WLE CORBA Programming Features

The WLE product offers a robust CORBA programming environment that simplifies the development and management of distributed objects. 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++
- `idltojava`—Compiles IDL files to Java source code based on IDL to Java mappings defined by the OMG. The `idltojava` compiler provided with BEA WebLogic Enterprise (WLE) includes several enhancements, extensions and additions that are not present in the original Sun Microsystems, Inc. version of the compiler. The WLE specific revisions are summarized here.
  - Differs from that described in the Sun Microsystems, Inc. documentation in behavior and defaults of the flags.
  - Includes a new #pragma tag: `#pragma ID <name> <Repostitory_id>`
  - Includes a new #pragma tag: `#pragma version <name> <m.n>`
  - Extends the `#pragma prefix` to work on inner scope. A blank prefix reverts.
  - Allows unions with boolean discriminators
  - Allow declarations nested inside complex types
- `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 Chapter 4, "Developing WebLogic Enterprise (WLE) CORBA Applications."

For a description of the `idl, idltojava,` and `m3idltojava` commands, see *WLE Reference* in the WebLogic Enterprise online documentation.

Development Commands

Table 2-1 lists the commands that the WLE product provides for developing CORBA application components and managing the Interface Repository.

Table 2-1 WLE CORBA 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.
`id12ir`	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 Chapter 4, "Developing WebLogic Enterprise (WLE) CORBA Applications."

For a description of the development commands, see *WLE Commands Reference* in the WebLogic Enterprise online documentation.

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.

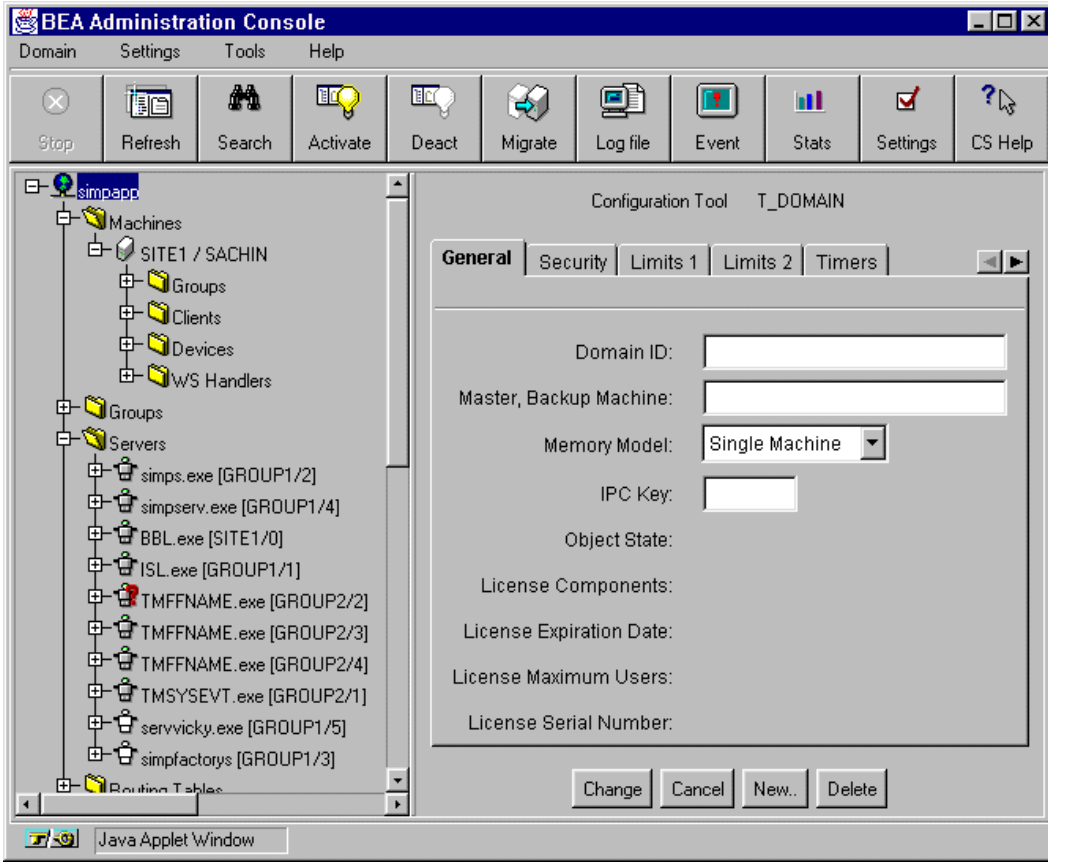
You can use the commands listed in Table 2-2 to perform administration tasks for your WLE application.

Table 2-2 WLE 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.
`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 2-1 shows the main window of the Administration Console.

Figure 2-1 Administration Console Main Window



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 manual intervention.

For information about the Administration commands, see *WLE Commands Reference* and *Administration Guide* in the WebLogic Enterprise online documentation.

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

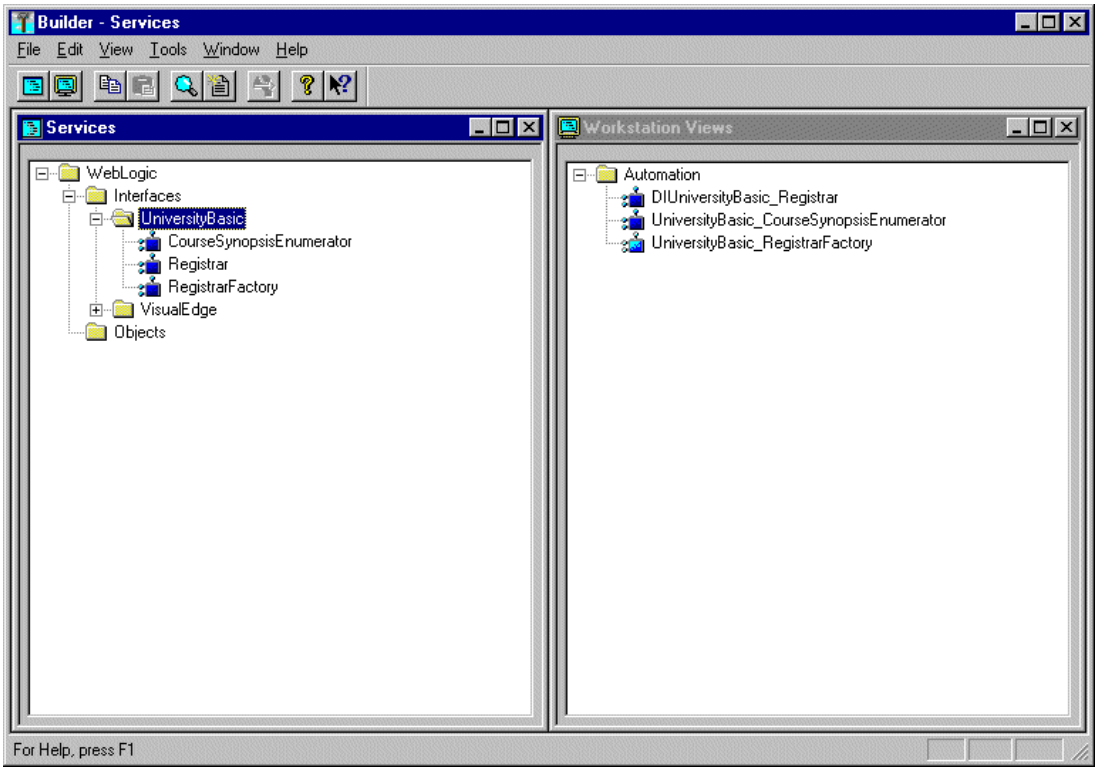
For information about the AdminAPI, see *BEA TUXEDO Reference* in the WebLogic Enterprise online documentation.

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 2-2 shows the ActiveX Application Builder main window.

Figure 2-2 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 information about creating ActiveX client applications, see *WLE ActiveX Client Developer's Guide* in the WebLogic Enterprise online documentation.

WLE CORBA 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 *How WLE CORBA Client and Server Applications Interact* in the WebLogic Enterprise online documentation.

- 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 *Using Security* in the WebLogic Enterprise online documentation.

- 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 the Sun Microsystems, Inc. Java Transaction API (JTA) defined in the `javax.transaction` package.

  For information about using transactions, see *Using Transactions* in the WebLogic Enterprise online documentation.

- Interface Repository service

  The Interface Repository 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 *Using DII* in the WebLogic Enterprise online documentation.

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

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

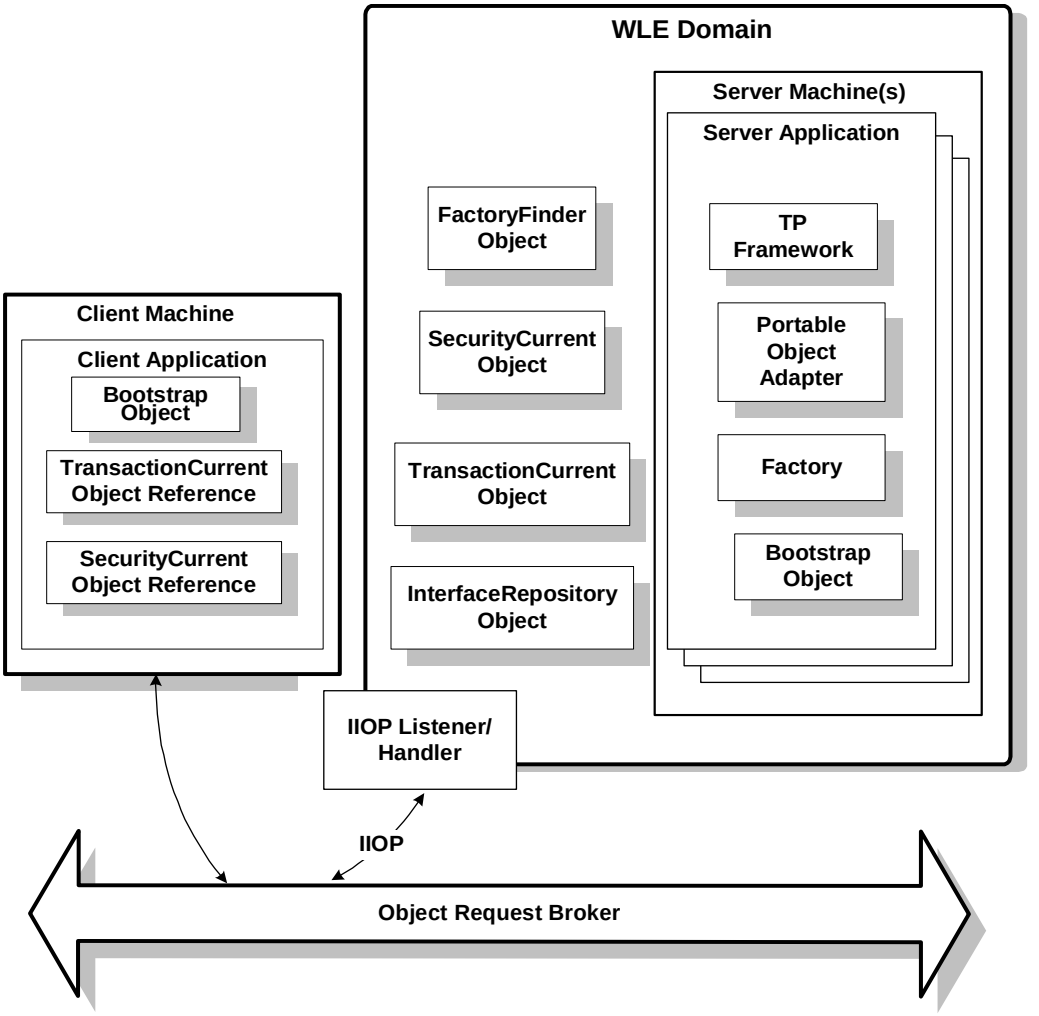
WLE CORBA Components

This section provides an introduction to the following WLE components:

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

Figure 2-3 illustrates the components in a WLE application.

Figure 2-3 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 using one of the following URL address formats:

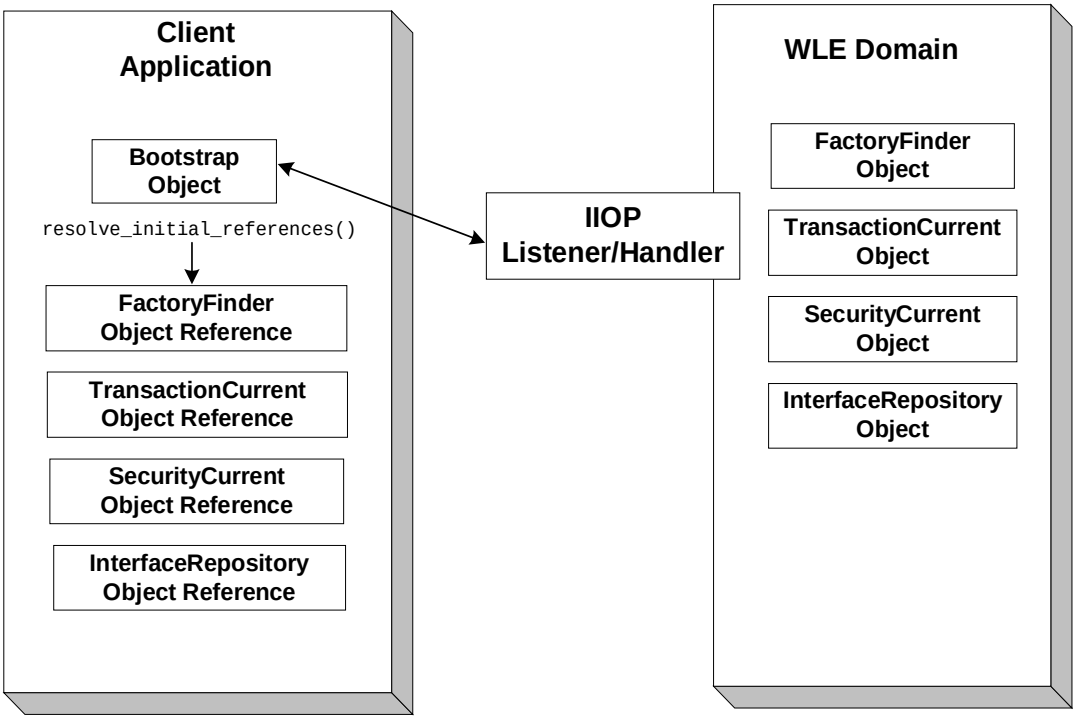
- `//host:port`
- `corbaloc://host:port`
- `corbalocs://host:port`

For more information about the Bootstrap URL address formats, see *Using Security* in the WebLogic Enterprise online documentation.

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 2-4 illustrates how the Bootstrap object works in a WLE domain.

Figure 2-4 How the Bootstrap Object Works in a WLE Domain



IIOP Listener/Handler

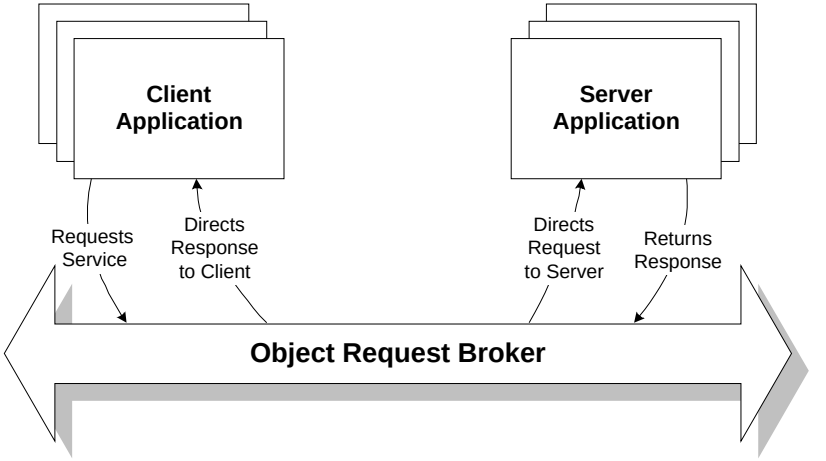
The IIOP Listener/Handler is a process that receives the client request, which is sent using IIOP, and delivers that request to the appropriate server application. The IIOP Listener/Handler serves as a communication concentrator, providing a critical scalability feature. The IIOP Listener/Handler removes from the server application the burden of maintaining client connections. For information about configuring the IIOP Listener/Handler, see *Adminstration Guide* and the description of the ISL command in the *WLE Reference* in the WebLogic Enterprise online documentation.

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 (JDK) from Sun Microsystems, Inc.

Figure 2-5 shows the relationship between an ORB, a client application, and a server application.

Figure 2-5 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 Framework supports the rapid construction of WLE applications, which makes it easier for application programmers to adhere to design patterns 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.

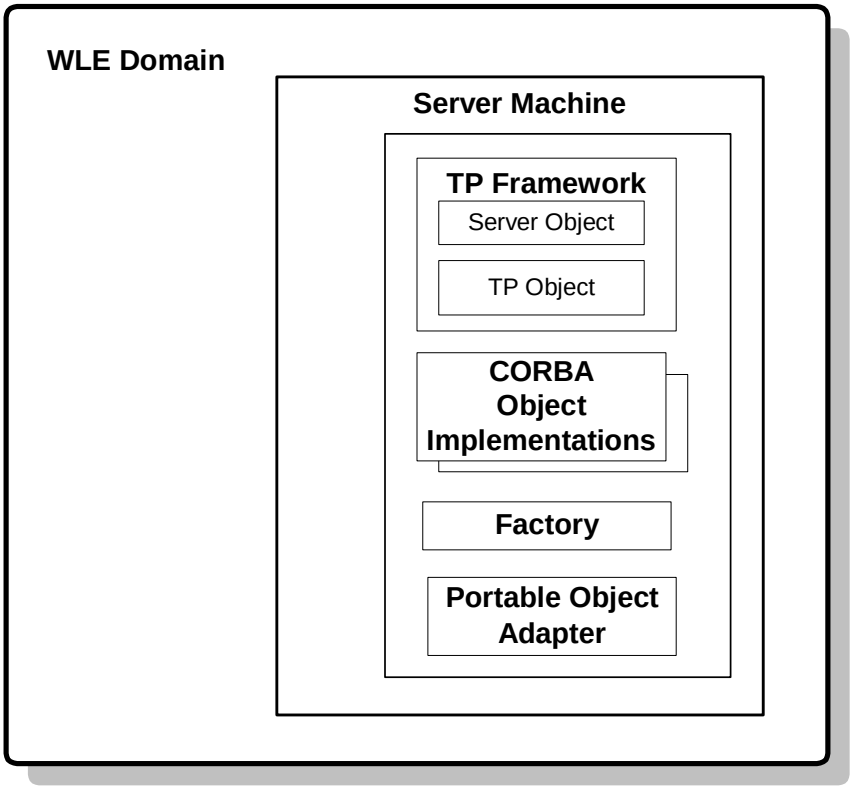
The application programmer uses an application programming interface (API) that automates many of the functions required in a standard CORBA application. The application programmer is responsible only for writing the business logic of the WLE application and overriding default actions provided by the 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 coordinated, 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 2-6 illustrates the TP Framework.

Figure 2-6 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.

How WLE CORBA Client and Server Applications Interact

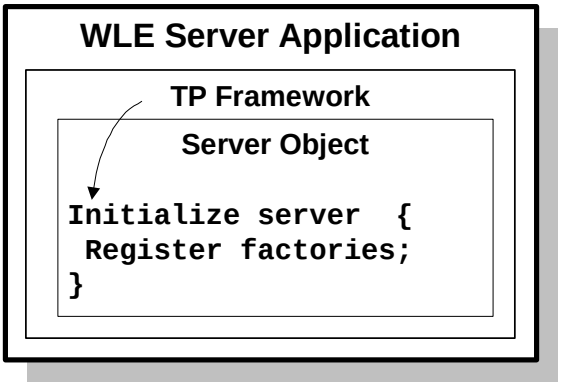
The interaction between WLE CORBA client and server applications includes the following steps:

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.

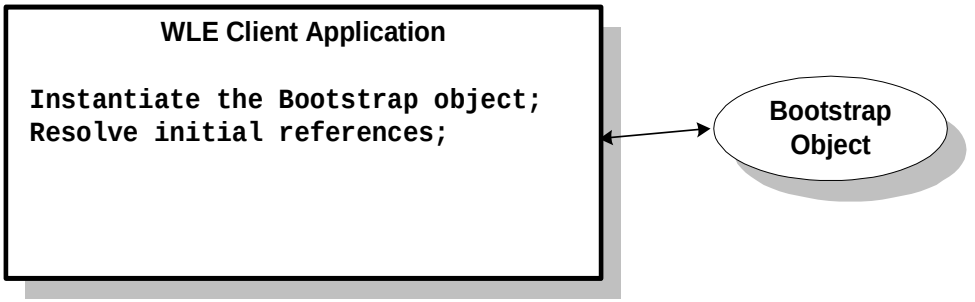


During the initialization process, the Server object does the following:

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.



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 using TUXEDO authentication, 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.

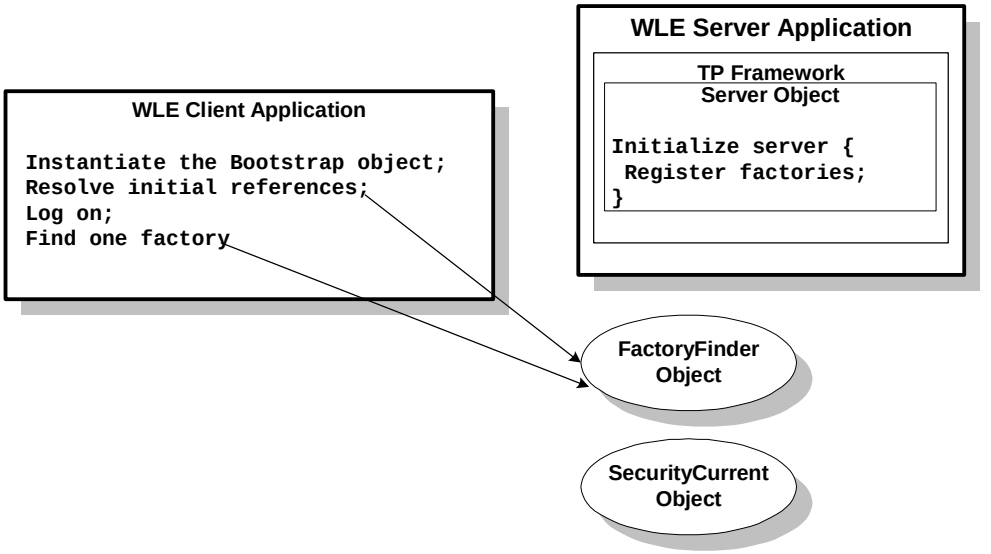
Note: For information about using certificate-based authentication, see *Using Security* in the WebLogic Enterprise online documentation.

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.

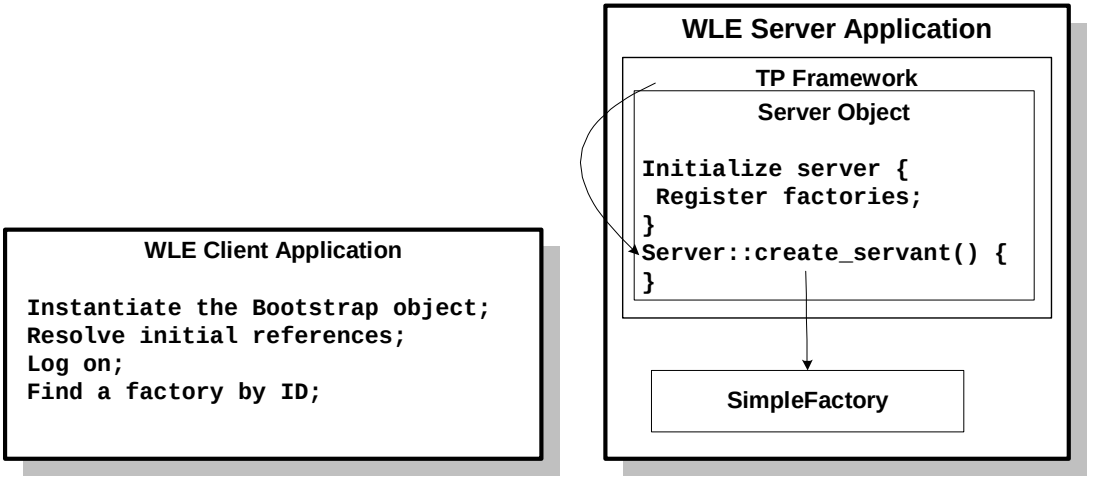
   For example, the client application needs a reference to the `SimpleFactory` object. The client application obtains this factory reference from the `FactoryFinder` object, shown in the following figure.


WLE Server Application
TP Framework
Server Object
Initialize server {
Register factories;
}
WLE Client Application
Instantiate the Bootstrap object;
Resolve initial references;
Log on;
Find one factory
FactoryFinder Object
SecurityCurrent Object

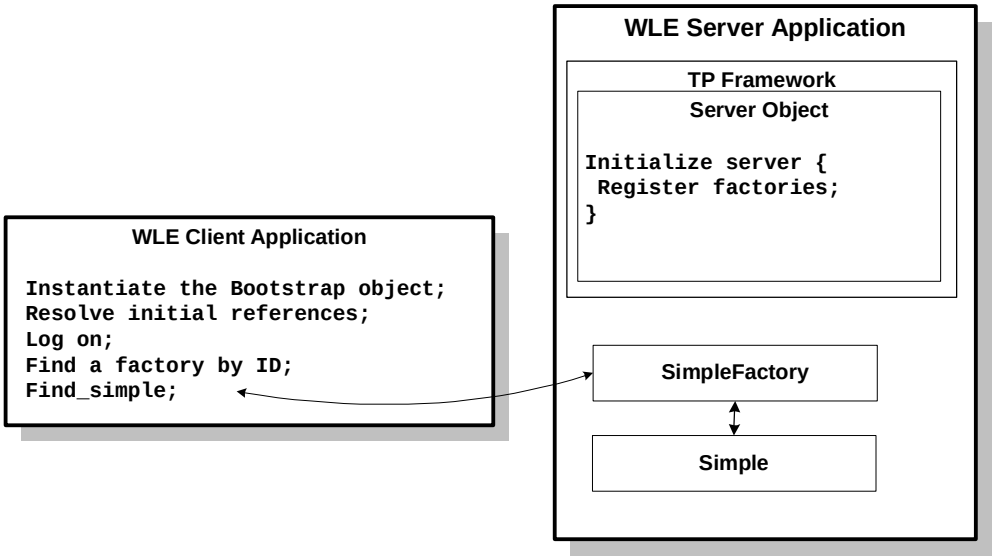
2. Invoke the `SimpleFactory` object to get a reference to the `Simple` object.

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

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



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.

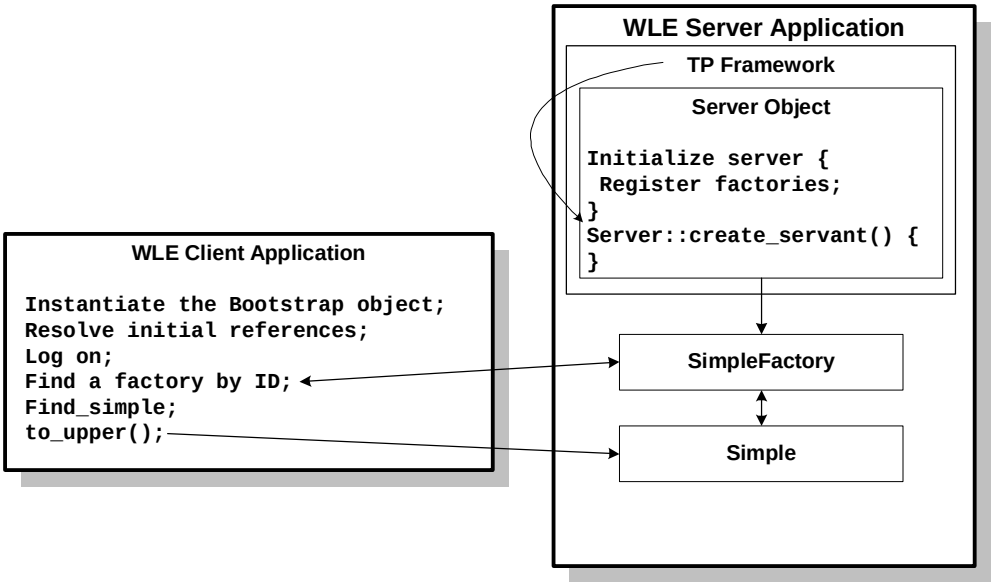


The `SimpleFactory` object then returns the object reference to the `Simple` object to the client application.

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.

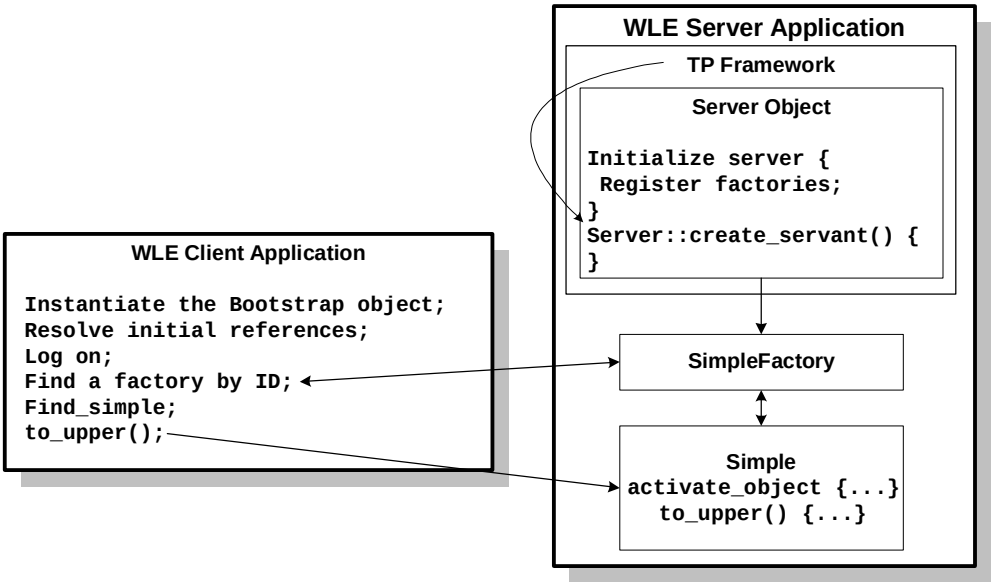
Step 5: The client application invokes an operation on the CORBA object.

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.

The TP Framework invokes the `activate_object()` operation on the `Simple` object and the `SimpleFactory` 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. The TP Framework invokes the operation on the object, returning the response to the client application.

3 The WLE Enterprise JavaBeans (EJB) Programming Environment

This topic includes the following sections:

- Overview of the WLE EJB Programming Environment
- Types of Beans Supported in WLE
- EJBs and Persistence
- Roles of People Who Develop, Build, Deploy, and Administer EJBs
- Items You Create for an EJB Application
- Tools and Facilities Provided for Building and Deploying EJBs
- EJBs and Failover in the WLE Environment

Overview of the WLE EJB Programming Environment

The Enterprise JavaBeans Specification 1.1, published by Sun Microsystems, Inc., defines a component architecture for building distributed, object-oriented business applications in Java. The EJB architecture addresses the development, deployment, and run-time aspects of an enterprise application's lifecycle.

An EJB encapsulates business logic inside a component framework that manages the details of security, transaction, and state management. Low-level details, such as the following, are handled by the EJB container:

- Multithreading
- Resource pooling
- Scaling
- Distributed naming
- Automatic persistence
- Remote invocation
- Transaction boundary management
- Distributed transaction management

This built-in, low-level support allows the EJB to focus on the business problem to be solved.

With the WLE EJB model, you can write or buy business components (such as invoices, bank accounts, and shipping routes) and, during deployment into a certain project, specify how the component should be used -- which users have access to which methods, whether the container should automatically start a transaction or whether it should inherit the caller's transaction, and so on. In this scenario, an EJB contains the business logic (methods) and the customization needed for a particular application (deployment descriptor), and the EJB will run within any standard implementation of the EJB container. An EJB is, in essence, a distributed object for which transactions and security can be specified declaratively in deployment descriptors.

The spirit of "write once, run anywhere" carries through into EJB: any vendors's EJB container (that conforms to the EJB Specification) can run any third-party EJBs (that also conform to the EJB Specification) to create an application. Nuances of the security mechanisms and specific distributed transaction monitors are entirely abstracted out of the application code (unless the Bean Provider chooses to make such calls explicitly).

Types of Beans Supported in WLE

With the WLE system, you can build and deploy standard, portable EJBs. The EJB Specification defines three types of beans:

Stateless session bean	An instance of a stateless session bean has no conversational state for the client that created the instance. This instance is not assigned permanently to the client. The EJB container can maintain a pool of instances and allocate method invocations coming from any client to any available instance (that is, not processing a request for a particular client). Therefore, any instance can receive method invocations from any client, and these requests can be processed on behalf of different transactions and security contexts. The EJB container decides the life of an instance; that is, the container can destroy an instance when resources are required or according to other policies. However, the client decides the life of the reference to the bean. The reference obtained from the bean's home interface is valid until the client destroys it. Note that stateless session beans cannot use the `SessionSynchronization` interface to synchronize with the starting and stopping of a transaction.

Stateful session bean	An instance of a stateful session bean maintains a conversational state for the client that created the instance. Therefore, instances of a stateful session bean are assigned to a particular client and are destroyed only when the client decides to remove the EJB object. Instances of stateful session beans do not survive a crash of the EJB container (which in WLE spans all the processes in the same group where the bean is deployed) or a redeployment of the bean. The EJB container can passivate inactive instances to maximize the use of the system resources -- that is, to deactivate the bean with its state saved to be restored at a later time during the bean's reactivation. Stateful session beans can use the `SessionSynchronization` interface to synchronize with the starting and stopping of a transaction.
Entity bean	An instance of an entity bean has a unique identity called the **primary key**. Object references to an entity bean should be usable for a long time and clients should be able to reuse them across server crash or restart. The reference becomes invalid when a client application removes the EJB or when the EJB is reconfigured. **Note:** If a server group crashes, and the System Administrator restarts that group using the same group ID and persistence store, the EJB container can process requests for beans in that group again. The EJB container for stateless session beans spans the entire domain in which the beans are deployed. Multiple client applications can access an entity bean instance; the EJB container is responsible for synchronizing the access to the instance. Typically, an entity bean has a persistent state, and application designers can choose between managing the persistence directly from the bean (bean-managed) or letting the EJB container manage the persistence (container-managed). In either case, the EJB container determines *when* an entity bean instance can be passivated (which also triggers the persistent storage of the state of the instance). An entity bean cannot use the `SessionSynchronization` interface to synchronize with the starting and stopping of a transaction.

EJBs and Persistence

An entity EJB can save its state in any transactional or nontransactional persistent storage, or it can ask the EJB container to save its nontransient instance variables automatically. The WLE system allows both choices. An EJB that manages its own persistence is referred to as having **bean-managed persistenc**e; an EJB that delegates to the EJB container the saving and restoring of its state is referred to as having **container-managed persistence**.

You control the persistence characteristics of a bean, such as where its data is maintained in durable storage, in its deployment descriptor; in the case of bean-managed persistence, you implement the specific method invocations in the bean that load and store state.

For more information about development and deployment considerations with regards to persistence, see the following sections in this guide:

- "Development Considerations for EJBs and Persistence" on page 7-16
- "Specifying Persistence Information" on page 8-8

Roles of People Who Develop, Build, Deploy, and Administer EJBs

The Enterprise JavaBeans Specification describes the six roles regarding who develops, builds, deploys, and administers an EJB application. These roles are summarized in the following table to help clarify what needs to be done, by whom, and when during the entire life cycle of an EJB in a way that is consistent with the EJB Specification.

Enterprise Bean Provider	The Enterprise Bean Providers (or Bean Providers) produce enterprise beans. Their output is an EJB JAR file that contains one or more enterprise beans. The Bean Provider is responsible for: ■ The Java classes that implement the enterprise bean's business methods ■ The definition of the bean's remote and home interfaces ■ The bean's deployment descriptor The deployment descriptor includes the structural information (for example, the name of the enterprise bean class) of the enterprise bean and declares all the enterprise bean's external dependencies (for example, the names and types of resources that the enterprise bean uses).
Application Assembler	The Application Assembler combines enterprise beans into larger deployable application units. The input to the Application Assembler is one or more EJB JAR files produced by the Bean Providers. The Application Assembler outputs one or more EJB JAR files that contain the enterprise beans along with their application assembly instructions. The Application Assembler has inserted the application assembly instruction into the deployment descriptors. Bean providers cooperate with the Application Assembler to combine EJBs into larger deployable units. In the WLE environment, creating these larger deployable units is more efficient if the Application Assembler takes into account the scalability and resource management capabilities provided by the WLE environment. For example, EJBs that access the same resources should be packaged together. The Application Assembler also specifies the security required by the application by associating client role names with the methods of the different beans.

Deployer	The Deployer uses the EJB container tools to customize one or more EJB JAR files produced by a Bean Provider or Application Assembler so that the beans can run in the corresponding EJB environment. These tools generate the additional classes required to manage the beans. The Deployer is primarily focused on the individual EJBs. In the WLE environment, the Deployer uses the `ejbc` command or the WebLogic EJB Deployer for this purpose. These tools can also be used by the Application Assembler to construct an EJB package, which is the EJB JAR file containing all the bean implementations and the assembly instructions. The Deployer also ensures that the security role names defined by the Application Assembler map to existing user groups and accounts that exist in the EJB environment. The Deployer must resolve all the external dependencies declared by the Bean Provider (for example, Deployers must ensure that all resources used by the enterprise beans are present in the operational environment, and they must bind them to the resource manager connection factory references declared in the deployment descriptor), and must follow the application assembly instructions defined by the Application Assembler.
EJB Server Provider	The EJB Server Provider (in the WLE system, this is BEA) is a specialist in the area of distributed transaction management, distributed objects, and other lower-level, system-level services.
EJB Container Provider	The EJB Container Provider (in the WLE system, this is BEA) provides: ■ The deployment tools necessary for the deployment of enterprise beans ■ The run-time support for the deployed enterprise beans' instances From the perspective of the enterprise beans, the container is a part of the target operational environment. The container run time provides the deployed enterprise beans with transaction and security management, network distribution of clients, scalable management of resources, and other services that are generally required as part of a manageable server platform.

System Administrator	The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure, which includes the EJB server and container. The System Administrator is also responsible for overseeing the well-being of the deployed enterprise bean applications at run time. The System Administrator cooperates with the Deployer to define the environment needed by the application. The System Administrator configures the WLE domain by defining the different machines, server groups, and other resources needed by the application (for example, JDBC connection pools and XA resource managers). The System Administrator also needs to add any security information needed by the application (for example, new user groups). The administrator is also responsible for monitoring the application and performing any run-time changes needed to adapt the operational environment to failures or other conditions.

Items You Create for an EJB Application

This section summarizes all the items you need to create for an EJB application that runs in the WLE environment, regardless of which role you are assuming, and explains where you can find more information about creating the item.

Item	Description	Where to Find More Information
One or more EJBs	The basic beans containing your application's business logic.	"Developing EJB Applications for the WLE System" on page 7-9

Item	Description	Where to Find More Information
Deployment descriptor	An XML file, created by one of the following methods, that specifies basic configuration and run-time information relevant to the deployment of the EJBs: ■ DDGenerator command ■ WebLogic EJB Deployer ■ Manually, using a common text editor	"Step 2: Create a deployment descriptor." on page 7-10
EJB JAR file	A Java Archive (JAR) file, produced by the `ejbc` command, that contains all the Java class files for the EJBs in the application. This file is created initially by the Bean Provider, and is then modified by the Bean Deployer and Application Assembler.	"Step 3. Package the EJB components into a Java Archive (JAR) file." on page 7-15 and "Step 4: Produce the deployable EJB JAR file." on page 8-13
WebLogic EJB extensions to the deployment descriptor DTD	An XML file, specifying configuration information pertinent to the WLE environment.	"Step 3: Create the WebLogic EJB extensions to the deployment descriptor DTD." on page 8-6
Module initializer object	A Java object specifying the module initializer class. This entity is optional.	"Specifying the Module Initializer Class" on page 8-6

Tools and Facilities Provided for Building and Deploying EJBs

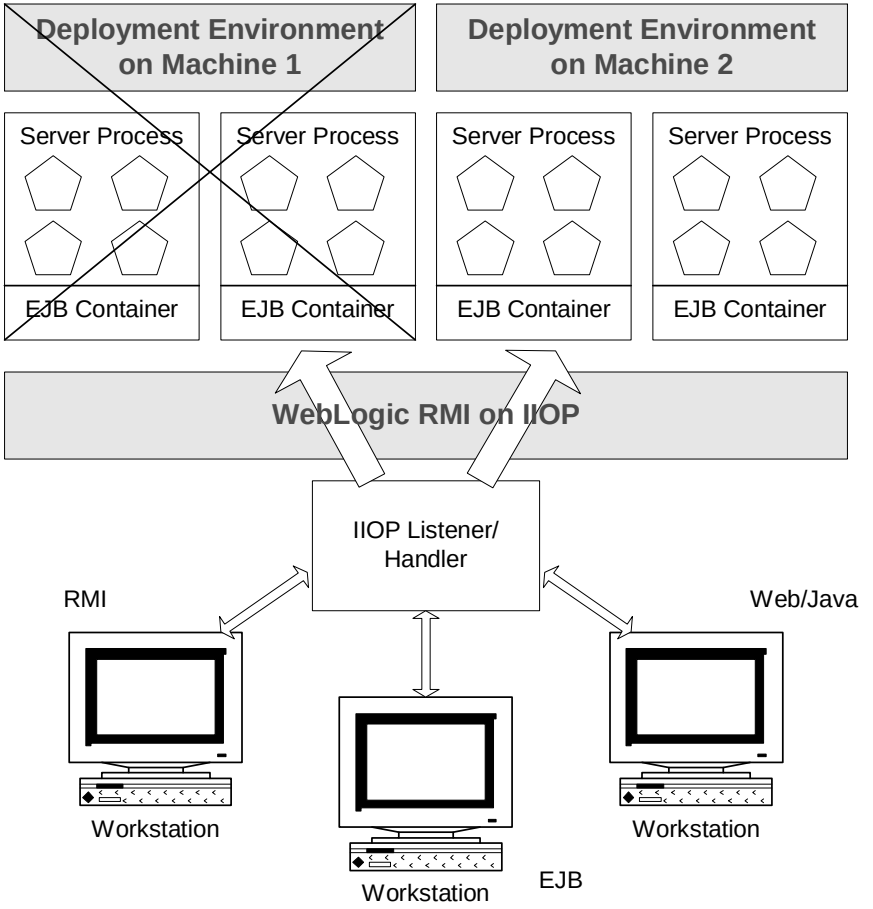
To help application programmers and deployers build EJBs that fully leverage the WLE system, the WLE software provides the tools and facilities listed in the following table:

`ejbc` command	Used by application programmers, Application Assemblers, and deployers as a command-line alternative to the WebLogic EJB Deployer to construct a deployable EJB JAR file.
`DDGenerator` command	Used by the Bean Provider to create the initial deployment descriptor file.
WebLogic EJB Deployer	Used by the Bean Provider, Bean Deployer, and Application Assembler to configure and deploy EJBs for use with your WLE server. You can use the WebLogic EJB Deployer to: ■ Examine an existing EJB and the configurable properties in its deployment descriptor. ■ Modify the properties and save the changes to a file (`.xml` or `.jar` format). ■ Generate EJB interface classes for a particular WebLogic environment. ■ Generate deployment classes for the beans. The WebLogic EJB Deployer is documented in the online help available from that tool's Help menu.
`UBBCONFIG` file	Used for configuring the EJB container and the Java server in which the EJB container is run and which loads the JVM and other modules needed by the EJB application. (You can also use the TMIB in place of the `UBBCONFIG` file.)

For more information about deploying and administering EJB applications in the WLE environment, see Chapter 8, "Building and Deploying Enterprise JavaBeans (EJBs)."

EJBs and Failover in the WLE Environment

The WLE system provides the following failover characteristics of EJB applications deployed in a WLE domain. Note that client applications cannot control where EJBs are instantiated. The following figure shows how, in the instance of a machine crash, failover is managed wholly by the WLE system.



Stateless session beans. If the server process hosting a stateless session bean fails, the bean is automatically instantiated in a different server process (on the same server or on another group within the domain), provided that the server process that is capable of supporting the session bean is available.

Entity beans. If one group that hosts one or more entity beans fails and in cases when the client application receives a `RemoteException`, the client application can invoke the `findByPrimaryKey` method to find the home interface for the entity bean, with the specified unique key, on another group in the domain. This works as long as the other group is configured to support that entity bean. Application developers can write client application code within a loop that begins by invoking the `findByPrimaryKey` method; this way, if a group fails, the client application retries the invocation on a different group.

Note that, for bean-managed persistence, the Bean Provider must implement this method explicitly; for container-managed persistence, this method is generated automatically.

Stateful session beans. If one group fails, the administrator must dynamically configure the group on a different machine. For more information, see *Configuring a Running System* in the WebLogic Enterprise online documentation.

For file-based persistence, recovery depends on whether persistence storage resides on a file system that is still network accessible (for example, an NFS-mounted volume). Because the `persistence-store-directory-root` element in the WLE EJB extensions to the deployment descriptor DTD specifies the path for persistent storage, the bean's state can be recovered. (Note that this file persistence mechanism is internal to the WLE system.)

For JDBC-based persistence, the application simply reconnects to the database, as long as the DBMS node is running and the network is accessible to the new node.

Note: In general, you should use JDBC-based persistence for production applications because it is more robust than file-based persistence. File-based persistence is typically appropriate only for development and prototyping purposes.

Part II Developing WLE CORBA Applications

CHAPTER

4 Developing WebLogic Enterprise (WLE) CORBA Applications

This topic includes the following sections:

- Overview of the Development Process for WLE CORBA Applications
- The Simpapp Sample Application
- Step 1: Write the OMG IDL code.
- Step 2: Generate client stubs and skeletons.
- Step 3: Write the server application.
- Step 4: Write the client application.
- Step 5: Create an XA resource manager.
- Step 6: Create a configuration file.
- Step 7: Create the TUXCONFIG file.
- Step 8: Compile the server application.
- Step 9: Compile the client application.
- Step 10: Start the WLE CORBA application.
- Additional WLE CORBA Sample Applications

For an in-depth discussion of creating WLE CORBA client and server applications, see the following in the WebLogic Enterprise online documentation:

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

Overview of the Development Process for WLE CORBA Applications

Table 4-1 outlines the development process for WLE CORBA applications.

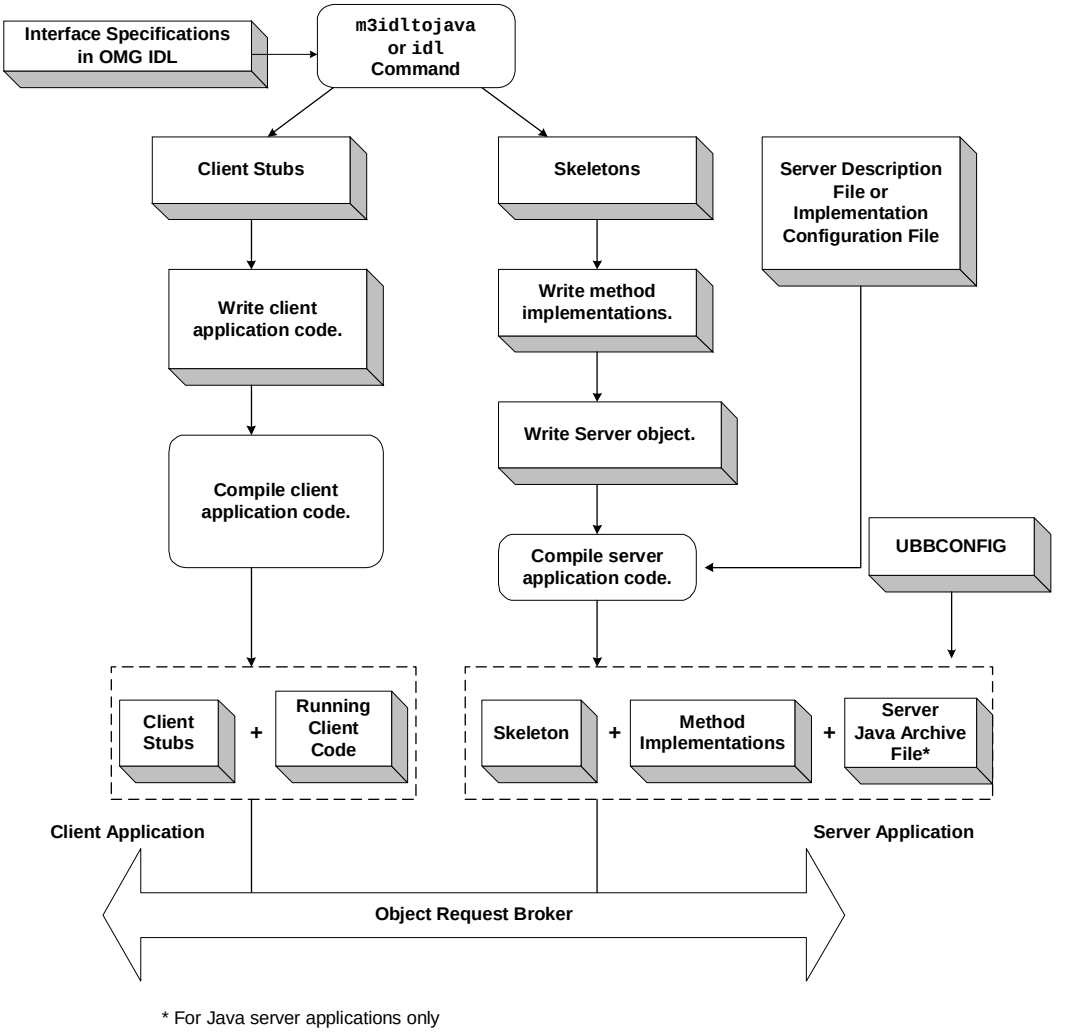
Table 4-1 Development Process for WLE CORBA Applications

Step	Description
1	Write the Object Management Group (OMG) Interface Definition Language (IDL) code 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 an XA resource manager.
6	Create a configuration file.
7	Create a `TUXCONFIG` file.
8	Compile the server application.
9	Compile the client application.
10	Start the WLE CORBA application.

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

Figure 4-1 illustrates the process for developing WLE CORBA applications.

Figure 4-1 Development Process for WLE CORBA Applications



* For Java server applications only

The Simpapp Sample Application

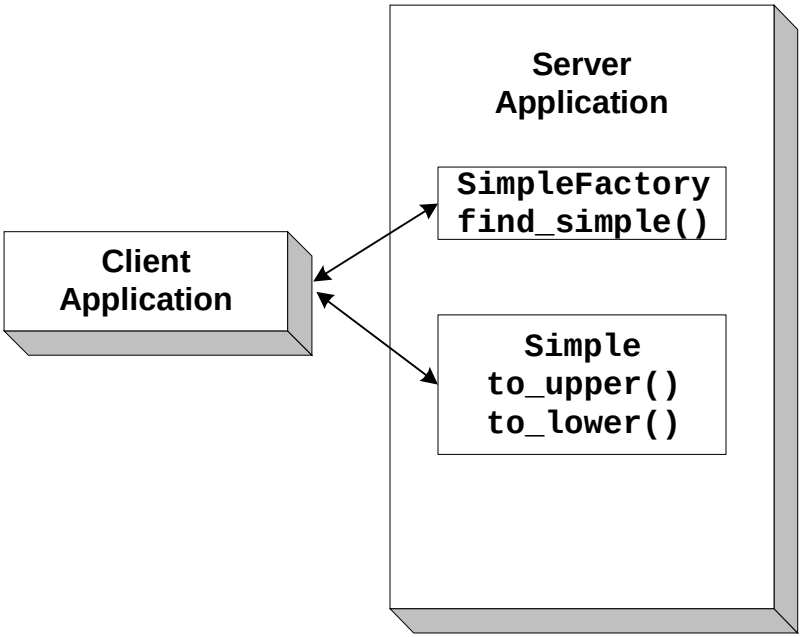
Throughout this topic, the Simpapp sample application is used to demonstrate the development steps. C++ and Java versions of the Simpapp sample application are available.

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 4-2 illustrates how the Simpapp sample application works.

Figure 4-2 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 instructions for building and running the C++ and Java Simpapp sample applications, see *Samples* in the WebLogic Enterprise online documentation.

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 *Basic sample application* in the WebLogic Enterprise online documentation.

The WLE product offers a suite of sample applications that demonstrate and aid in the development of WLE CORBA applications. For an overview of the available sample applications, see *Samples* in the WebLogic Enterprise online documentation.

Step 1: Write the OMG IDL code.

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 4-2.

Table 4-2 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 4-1 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 4-1 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: Generate 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 *WLE Reference* in the WebLogic Enterprise online documentation.

- If you are creating Java client and server applications, use the `m3idltojava` command. For a description of the `m3idltojava` command, see *WLE Reference* in the WebLogic Enterprise online documentation.

Table 4-3 lists the files that are created by the `idl` command.

Table 4-3 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 4-4 lists the files that are created by the `m3idltojava` command.

Table 4-4 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 the samples and in this document, this file is named *`interface`*`Impl.java`. Substitute the actual name of the interface in the file name. This new file is called an *object implementation file*.
Client stub file	`_`*`interface`*`Stub.java`	Contains generated code for sending a request.
Server skeleton file	`_`*`interface`*`ImplBase.java`	Contains Java skeletons for each interface specified in the OMG IDL file. 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: Write 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.
4. Create and register a factory.
5. Release the server application.

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 the object. For more information, see *Creating CORBA C++ Server Applications* and *Creating CORBA Java Server Applications* in the WebLogic Enterprise online documentation.

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.

Listing 4-2 includes the C++ implementation of the Simple and SimpleFactory interfaces in the Simpapp sample application.

Listing 4-2 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 4-3 includes the Java implementation of the Simple interface from the Simpapp sample application.

Listing 4-3 Java Implementation 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 4-4 includes the Java implementation of the SimpleFactory interface from the Simpapp sample application.

Listing 4-4 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 4-5 includes the C++ code from the Simpapp sample application for the Server object.

Listing 4-5 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. When creating Java server applications, you identify the name of the Server object in the Server Description File.

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

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

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

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

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

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

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

Listing 4-6 includes the Java code from the Simpapp sample application for the Server object.

Listing 4-6 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 CORBA application, see *Using Transactions* in the WebLogic Enterprise online documentation.

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 CORBA C++ Server Applications* and *Creating CORBA Java Server Applications* in the WebLogic Enterprise online documentation.

Creating and Registering a Factory

If your server application manages a factory that you want client applications to be able to locate easily, you need to write the code that registers that factory with the FactoryFinder object.

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

1. Create an object reference to the factory.

   You include an invocation to the `create_object_reference()` method, specifying the Interface Repository ID of the factory's OMG IDL interface or the object ID (OID) in string format. In addition, you can specify routing criteria.

2. Register the factory with the WLE domain.

   Use the `register_factory()` method to register the factory with the FactoryFinder object in the WLE domain. The `register_factory()` method requires the object reference for the factory and a string identifier.

Listing 4-7 includes the code from the C++ Simpapp sample application that creates and registers a factory.

Listing 4-7 C++ Example of Creating and Registering a Factory

```
...
CORBA::Object_var v_reg_oref =
       TP:create_object_reference(
              _tc.SimpleFactory->id(),    //Factory Interface ID
              “simplefactory”,            //Object ID
              CORBA::NVList::_nil()       //Routing Criteria
       );
```

```
       TP::register_factory(
              CORBA::Object_var v_reg_oref.in(),
              _tc_SimpleFactory->id(),
       );
...
```

In Listing 4-7, notice the following:

- `tc.SimpleFactory->id()` specifies the SimpleFactory object's Interface Repository ID by extracting it from its typecode.
- `CORBA::NVList::_nil()` specifies that no routing criteria are used, with the result that an object reference created for the Simple object is routed to the same group as the SimpleFactory object that created the object reference.

Listing 4-8 includes the code from the Java Simpapp sample application that creates and registers a factory.

Listing 4-8 Java Example of Creating and Registering a Factory

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

org.omg.CORBA.Object simple_oref =
                  TP.create_object_reference(
                  SimpApp.SimpleHelper.id(),  // Repository ID
                  SimpleFName,                 // Object ID
                   null                       // Routing Criteria
                    );
// Register the factory reference with the factory finder.

       TP.register_factory(
              fact_oref,               // factory object referenc
              SimpleFName                // factory name
              );
...
```

Releasing the Server Application

You need to include code in your server application to perform a graceful shutdown of the server application. The `release()`method is provided for that purpose. Within the `release()` method, you may perform any application-specific cleanup tasks that are specific to the server application, such as:

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

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

During server shutdown, you may want to unregister each of the server application's factories. The invocation of the `unregister_factory()` method should be one of the first actions in the `release()` implementation. The `unregister_factory()` method unregisters the server application's factories. This operation requires the following input arguments:

- The object reference for the factory
- A string identifier, based on the factory object's interface typecode, used to identify the Interface Repository ID of the object's OMG IDL interface

Listing 4-9 includes C++ code that releases a server application and unregistered the factories in the server application.

Listing 4-9 C++ Example of Releasing a WLE Server Application

```
...
public void release()
{
       TP.unregister_factory(
              factory_reference,
```

```
                SimpleFactoryHelper.id
                );
}

...
```

Listing 4-10 includes Java code that releases a server application and unregistered the factories in the server application.

Listing 4-10 Java Example of Releasing a WLE Server Application

```
...
/**
* Method to shutdown the server.
*/
public void release)()
{
  //This method will only be called if Server.initialize()
  //succeeded, therefore, we know that the factory has been
  //registered with the factory finder.


  //Unregister the factory.
  //Use a try block since cleanup code should not throw
  //exceptions.

try{
TP.unregister_factory(
   fact_ref,                //factory object reference
   SimpleFactoryHelper.id() //factory repository id
   );
}catch (Exception e){
  //Some exception occurred. The call to TP.userlog()
  //will put the message in the ULOG file.
TP.userlog("Couldn't unregister the SimpleFactory:"
           +e.getMessage());
e.printStackTrace();
}
}
...
```

Step 4: Write 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: For information about creating an ActiveX client application, see *WLE ActiveX Client Developer's Guide* in the WebLogic Enterprise online documentation.

The client development steps are illustrated in Listing 4-11 and Listing 4-12, 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.

Listing 4-11 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 4-12 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: Create an XA resource manager.

When using transactions in a WLE CORBA application, you need to create a server process for the resource manager that interacts with a database on behalf of the WLE CORBA application. The resource manager you use must conform to the X/OPEN XA specification and you need the following information about the resource manager:

- The name of the structure of type `xa_switch_t` that contains the name of the XA resource manager.
- Flags indicating the capabilities of the XA resource manager and function pointers for the actual XA functions.
- The name of the object files that provide the services of the XA interface.
- The commands needed to open and close the XA resource manager. This information is specified in the `OPENINFO` and `CLOSEINFO` parameters in the `UBBCONFIG` configuration file.

When integrating a new XA resource manager into the WLE system, the file `$TUXDIR/udataobj/RM` must be updated to include information about the XA resource manager. The information is used to include the correct libraries for the XA resource manager and to automatically and properly set up the interface between the transaction manager and the XA resource manager. The format of this file is as follows:

rm_name:*rm_structure_name*:*library_names*

where *rm_name* is the name of the XA resource manager, *rm_structure_name* is the name of the `xa_switch_t` structure that defines the name of the XA resource manager, and *library_names* is the list of the object files for the XA resource manager. White space (tabs and/or spaces) is allowed before and after each of the values and may be embedded within the *library_names*. The colon (:) character may not be embedded within any of the values. Lines beginning with a pound sign (#) are treated as comments and are ignored.

Use the `buildtms` command to build a server process for the XA resource manager. The files that result from the `buildtms` command need to be installed in the `$TUXDIR/bin` directory.

For more information about the `buildtms` command, see *WLE Reference* in the WebLogic Enterprise online documentation.

Step 6: Create 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 CORBA 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 4-5.

Table 4-5 Sections in the Configuration File for WLE CORBA Applications

Sections in the Configuration File	Description
`RESOURCES`	Defines defaults (for example, user access and the main administration machine) for the WLE CORBA application
`MACHINES`	Defines hardware-specific information about each machine running in the WLE CORBA 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 CORBA application

Sections in the Configuration File	Description
SERVICES	Defines parameters for services provided by the WLE application
INTERFACES	Defines information about the CORBA interfaces in the WLE CORBA application
JDBCONNPOOLS	Describes the pooling of JDBC connections for Java servers
ROUTING	Defines routing criteria for the WLE CORBA application

Listing 4-13 shows the configuration file for the Simpapp sample application.

Listing 4-13 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
              SRVTYPE = JAVA
              MODULES = Bankapp.jar
              SRVGRP = APP_GRP
              SRVID = 2
              SYSTEM_ACCESS = FASTPATH
              CLOPT = "-A -- -M 10 TellerFactory_1"
              RESTART = N
```

If you are using an XA resource manager, use the `JavaServerXA` parameter in place of the `JavaServer` parameter to associate the XA resource manager with a specified server group. You need to include the information to open and close the resource manager in the `OPENINFO` and `CLOSEINFO` parameters in the `GROUPS` section of the `UBBCONFIG` file. The information needed to open and close the resource manager should be provided by the manufacturer of the resource manager.

Step 7: Create the TUXCONFIG file.

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. The `tmloadcf` command converts the configuration file to binary form and writes it to the location specified in the command.

For more information about the `tmloadcf` command, see *WLE Reference* in the WebLogic Enterprise online documentation.

Step 8: Compile 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 CORBA 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 need to specify in the `APPDIR` system environment variable the location of the JAR file for your Java server application. On Windows NT systems, this directory must be on a local drive (not a networked drive). On Solaris systems, the directory can be local or remote.

Step 9: Compile 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 *WLE Reference* in the WebLogic Enterprise online documentation.

When creating CORBA Java client applications, see your Java ORB's documentation for information about building client executables. You need to include the `wledir\udataobj\java\jdk\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.

Step 10: Start the WLE CORBA application.

Use the `tmboot` command to start the server processes in your WLE CORBA application. The WLE CORBA application is usually booted from the machine designated as the `MASTER` in the `RESOURCES` section of the `UBBCONFIG` file.

For the `tmboot` command to find executables, the WLE system processes must be located in `$TUXDIR/bin`. Server applications should be in `APPDIR`, as specified in the configuration file.

When booting server applications, the `tmboot` command uses the `CLOPT`, `SEQUENCE`, `SRVGRP`, `SRVID`, and `MIN` parameters from the configuration file. Server applications are booted in the order in which they appear in the configuration file.

For more information about using the `tmboot` command, see *WLE Reference* in the WebLogic Enterprise online documentation.

Additional WLE CORBA Sample Applications

Sample applications demonstrate the tasks involved in developing a WLE CORBA application, and provide sample code that can be used by client and server programmers to build their own WLE CORBA application. Code from the sample applications are used throughout the information topics in the WLE product to illustrate the development and administrative steps. For information about building and running the sample applications, see *Samples* in the WebLogic Enterprise online documentation.

Table 4-6 describes the additional WLE CORBA sample applications.

Table 4-6 The WLE CORBA Sample Applications

WLE CORBA Sample Application	Description
Simpapp	Provides a C++ client application and a C++ server application. The C++ server application contains two operations that manipulate strings received from the C++ client application.
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.
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	Demonstrates adding TUXEDO authentication to a WLE application. For information about building and running the Security sample application, see *Using Security* in the WebLogic Enterprise online documentation.
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. For information about building and running the Transactions sample application, see *Using Transactions* in the WebLogic Enterprise online documentation.
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.

WLE CORBA Sample Application	**Description**
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. For information about building and running the JDBC Bankapp sample application, see *Using Transactions* in the WebLogic Enterprise online documentation.
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. For information about building and running the XA Bankapp sample application, see *Using Transactions* i in the WebLogic Enterprise online documentation.
Secure Simpapp	Implements the necessary development and administrative changes to the Simpapp sample application to support certificate-based authentication. Java and C++ versions are provided. For information about building and running the Secure Simpapp sample application, see *Using Security* in the WebLogic Enterprise online documentation.
Introductory Events	Demonstrates how to use joint client/server applications and callback objects to implement events in a WLE CORBA application. The C++ version uses the BEA Simple Events API and the Java version uses the CosNotification API. For information about building and running the Introductory Events sample application, see *Using the Notification Service* in the WebLogic Enterprise online documentation.
Advanced Events	Provides a more complex implementation of events in a WLE CORBA application with transient and persistent subscriptions and data filtering. The C++ version uses the BEA Simple Events API and the Java version uses the CosNotification API. For information about building and running the Advanced Events sample application, see *Using the Notification Service* in the WebLogic Enterprise online documentation.

5 Using Security

This topic includes the following sections:

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

Note: This chapter describes using username/password authentication. For a complete description of all the security features available in the WLE product and instructions for implementing the security features, see *Using Security* in the WebLogic Enterprise online documentation.

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.

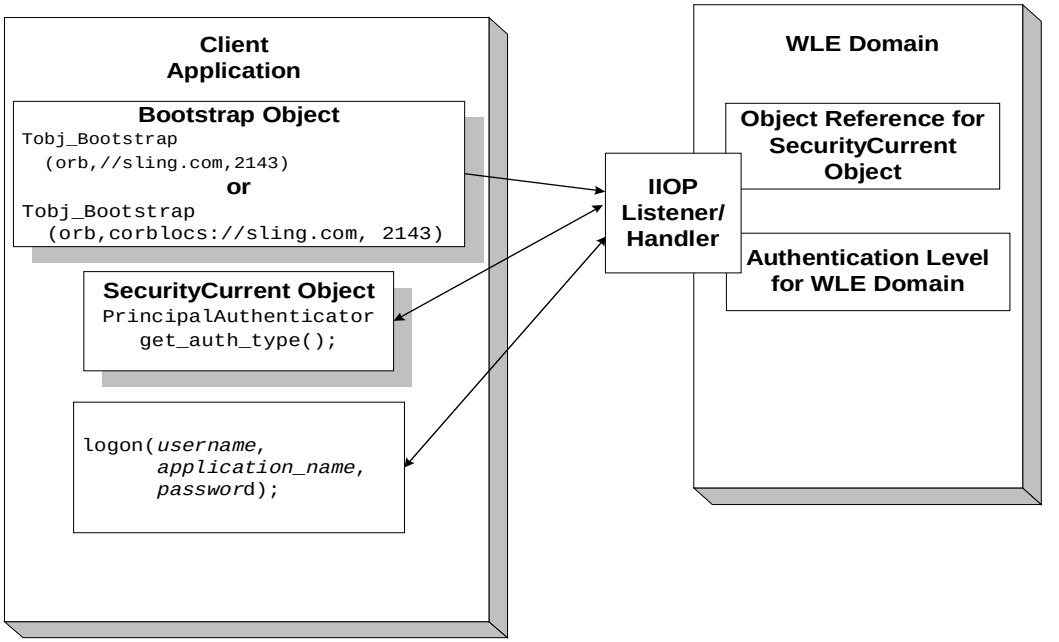
Note: If a client application is not authenticated and the security level is `TOBJ_NOAUTH`, the IIOP Listener/Handler of the WLE domain registers the client application with the user name and client application name sent to the IIOP Listener/Handler.

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++ and Java topics in *Reference* in the WebLogic Enterprise online documentation.

How Security Works

Figure 5-1 illustrates how security works in a WLE domain.

Figure 5-1 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.

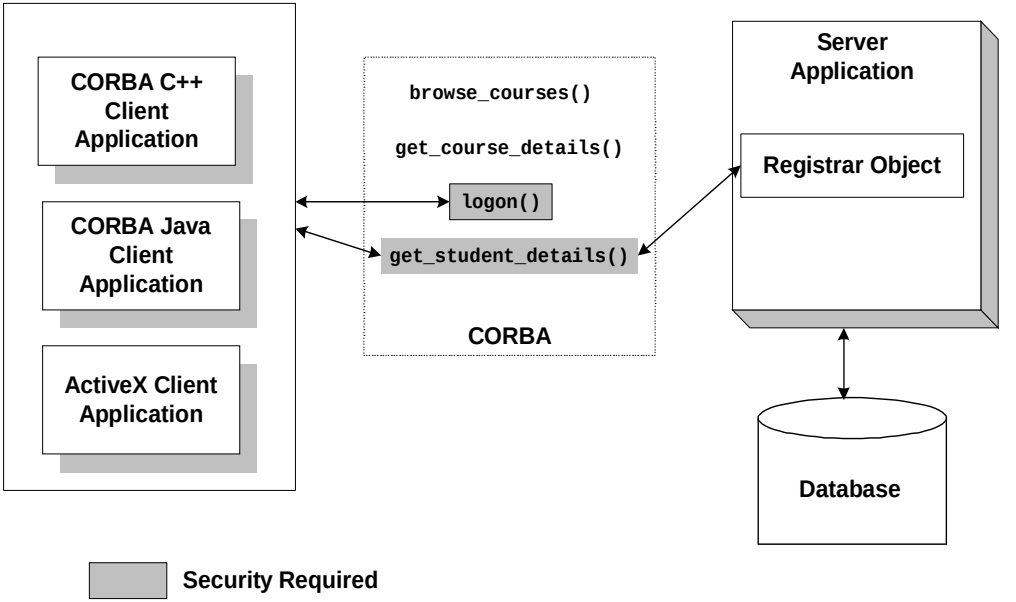
The Security Sample Application

The Security sample application demonstrates username/password authentication. 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 and 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 5-2 illustrates the Security sample application.

Figure 5-2 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 *Using Security* in the WebLogic Enterprise online documentation.

Development Steps

Table 5-1 lists the development steps for writing a WLE CORBA application that has username/password authentication security.

Table 5-1 Development Steps for WLE CORBA Applications That Have Security

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

Step 1: Define the security level in the configuration file.

The security level for a WLE domain is defined by setting the `SECURITY` parameter in the `RESOURSES` section of the configuration file to the desired security level. Table 5-2 lists the options for the `SECURITY` parameter.

Table 5-2 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 CORBA application, see *Using Security* in the WebLogic Enterprise online documentation.

Step 2: Write the CORBA 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 5-1 and Listing 5-2 include the portions of the CORBA C++ and CORBA Java client applications in the Security sample application that illustrate the development steps for security.

Listing 5-1 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_ref->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 5-2 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);
```

6 Using Transactions

This topic includes the following sections:

- Overview of the Transaction Service
- What Happens During a Transaction
- Transactions Sample Application
- Development Steps

Note: This topic describes using the C++ interface to the CORBAservices Object Transaction service. For a complete description of all the transaction features available in the WLE product and instructions for implementing the transaction features, see *Using Transactions* in the WebLogic Enterprise online documentation.

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 via 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 *Programming Reference* in the WebLogic Enterprise online documentation.

- 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

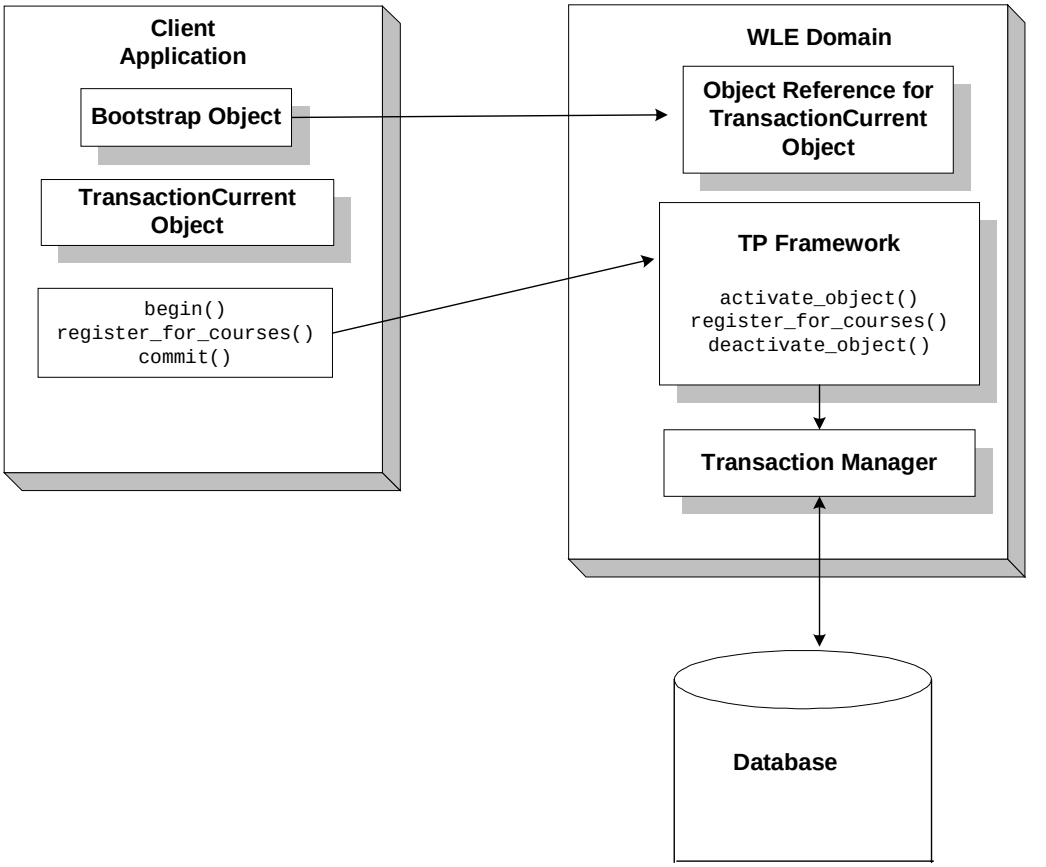
OTS, JTS, and JTA each provide the following support for your business transactions:

- 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 The Open Group 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.

What Happens During a Transaction

Figure 6-1 illustrates how transactions work in a WLE CORBA application.

Figure 6-1 How Transactions Work in a WLE CORBA 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 invokes multiple individual operations on a database.

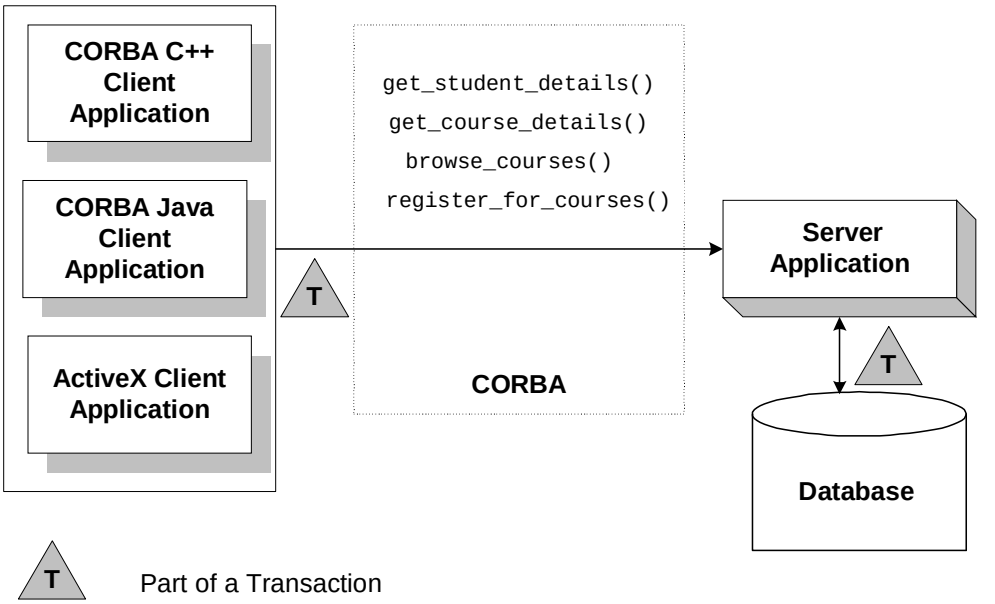
The Transactions sample application works in the following way:

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 6-2 illustrates how the Transactions sample application works.

Figure 6-2 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 WLE CORBA Java applications, see the description of the XA Bankapp sample application in *Using Transactions* in the WebLogic Enterprise online documentation.

Development Steps

This topic describes the development steps for writing a WLE CORBA application that includes transactions. Table 6-1 lists the development steps.

Table 6-1 Development Steps for WLE CORBA Applications That Have Transactions

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

Step	Description
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 *Samples* in the WebLogic Enterprise online documentation.

The XA Bankapp sample application demonstrates how to use transactions in Java WLE CORBA 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 *Samples* in the WebLogic Enterprise online documentation.

Step 1: Write the OMG IDL code.

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 6-1 includes the OMG IDL code for the Transactions sample application.

Listing 6-1 OMG IDL Code 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: Define 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: Write the CORBA client application.

The CORBA 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 6-2 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 *Guide to the Java Sample Applications* in the WebLogic Enterprise online documentation.

Listing 6-2 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: Write 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 CORBA application uses a database, you need to include code in the server application 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 6-3 shows the portion of the code for the Server object in the Transactions sample application that opens and closes the XA resource manager.

Note: For a complete example of a C++ server application that implements transactions, see the Transactions sample application in *Using Transactions* in the WebLogic Enterprise online documentation.

For an example of a Java server application that implements transactions, see the description of the XA Bankapp sample application in *Using Transactions* in the WebLogic Enterprise online documentation.

Listing 6-3 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 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: Create a configuration file.

You need to add the following information to the configuration file for a transactional WLE CORBA 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 and CLOSEINFO parameters of the GROUPS section, include information to open and close the XA 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() operation 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 *Administration* in the WebLogic Enterprise online documentation.

Listing 6-4 includes the portions of the configuration file that define this information for the Transactions sample application.

Listing 6-4 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 *Creating a Configuration File* in the WebLogic Enterprise online documentation.

Part III Developing WLE Enterprise JavaBeans

7 Designing and Developing Enterprise JavaBeans for the WLE System

This topic includes the following sections:

- Designing EJB Applications for the WLE System
- Developing EJB Applications for the WLE System

The information in this chapter supplements the Sun Microsystems, Inc. evolving Enterprise JavaBeans 1.1 Specification (Public Release 2, October 18, 1999).

Designing EJB Applications for the WLE System

The WLE software complies with the EJB 1.1 Specification. However, to design EJB applications that take advantage of the WLE architecture, you need to follow certain design rules and patterns. This section describes these design considerations with respect to the following perspectives on the WLE EJB environment:

- The client application programmer's view
- The bean programmer's view

The Client Application Programmer's View

Client application programmers using EJBs have a uniform development model they can use for beans regardless of whether the beans are local or remote. For each EJB, client programmers have access to the following:

The bean's home interface	Each EJB has a home interface (factory) that creates instances of the bean. The home interface defines the methods used by client programmers to create, remove, and find objects of the corresponding EJB type. To find a reference to a particular bean home interface, client applications should use the `lookup` method of the `InitialContext` object with the `PortableRemoteObject` class.
The bean's remote interface	Each EJB has a well-defined remote interface that defines the business methods that can be invoked by a client.
The object identity	Each EJB instance lives in a home and has a unique identity within its home. The identity of a session bean is generated by the EJB container and is not exposed to the client. The Bean Provider generates the identity of an entity bean (the primary key) and a client can retrieve the primary key from the entity object reference.
The bean's metadata interface	The metadata interface allows clients (typically, application assembly tools) to discover the metadata information about the bean.
The object handle	The handle identifies the object in a portable way. The handle can be serialized. Having a serialized handle lets you store the handle and then use it at a later time, possibly in a different process or in a different system, or by another bean or object. Handles are more useful with entity beans than with session beans.

To access any EJB, a client application needs to obtain a reference from the bean's home interface (factory); however, because the home is also an object, the client needs also to obtain a reference to it. Getting a home reference to the client application presents a bootstrapping problem. However, when you register home references with a directory service -- namely, JNDI -- client applications have a means to obtain a reference to the bean's home. This is exactly what the WLE EJB container provides.

From the WLE EJB container perspective, client applications (including JSP and servlets acting as clients) are nontrusted entities that require authentication. They typically require a network connection to access the WLE EJB container because they run on nontrusted machines.

How to set up this network connection is another bootstrapping problem. This is solved in WLE by providing a JNDI implementation that runs within the EJB container trusted environment, and by establishing the network connection when the JNDI initial context is created. The parameters required for the initialization are Java properties (name/value pairs) passed as arguments to the constructor of the `InitialContext` object.

A WLE client application can set the following properties:

Property Name	Values	Description
`WLEContext.INITIAL_CONTEXT_FACTORY`	`com.beasys.jndi.WLEInitialContextFactory`	Specifies the WLE JNDI.
`WLEContext.PROVIDER_URL`	`corbaloc://<host:port>` or `corbalocs://<host:port>`	Defines the address of the entry points to the WLE environment. The identifier `corbaloc` indicates that the protocol is WebLogic RMI on IIOP. The identifier `corbalocs` indicates that the protocol is WebLogic RMI on IIOP with SSL.
`WLEContext.SECURITY_AUTHENTICATION`	`none | simple | strong`	Defines the type of authentication: ■ `none` means no authentication (this is the default). ■ `simple` means WLE authentication. ■ `strong` means SSL authentication (certificate-based).
`WLEContext.SECURITY_PRINCIPAL`	`<Principal Identifier>`	Specifies the security principal name. For more information, see *Using Security* in the WebLogic Enterprise online documentation.

Property Name	Values	Description
WLEContext. SECURITY_CREDENTALS	<SSL credentials> or <User Password>	Specifies the credentials when authentication is strong, or the user password when authentication is simple.
WLEContext.CLIENT_NAME	<Security role>	Specifies the security role name used by simple authentication. For more information, see *Using Security* in the WebLogic Enterprise online documentation.
WLEContext. SYSTEM_PASSWORD	<password>	Specifies the system password if the simple authentication is in effect.
WLEContext.CODEBASE	<url>	Specifies the URL for remote class loading.

The client application is implicitly associated with the security context specified when the InitialContext object is created. To specify a new security context -- for example, to invoke objects in a different WLE domain -- the client application needs to close the current context and establish a new context with the new security attributes. To find a reference to a particular bean home interface, client applications should use the lookup method of the InitialContext object with the PortableRemoteObject class. Client applications can also use the lookup method to obtain a reference to the UserTransaction object. WLE client applications cannot modify the WLE JNDI naming context; that is, client applications can perform only lookup operations on this context.

The client can use the bean home interface to find or create session or entity bean instances. The create method provided by the EJB home interface creates the requested EJB and returns a reference to it. The client uses the reference for as long as it needs, and when it finishes, can invalidate the reference (and eventually the EJB instance) by invoking the ejbRemove method on the EJB instance. Between these method invocations, the client can invoke (optionally within a transaction) any of the business methods provided by the EJB.

The WLE system complies with the EJB 1.1 Specification, and client programmers must be aware of the subtle programming differences provided by the different bean types (refer to the EJB Specification for more details).

Also note that the WebLogic RMI on IIOP protocol is not currently supported for applets running on a Web browser.

The EJB Programmer's View

As an EJB programmer, or **Bean Provider**, as identified in the EJB Specification, you must follow the conventions and programming restrictions established in the EJB 1.1 Specification for the different EJB types. The following are the principal design considerations to take into account when implementing beans with the WLE EJB environment:

- Choosing Between Session and Entity Beans
- Server Startup
- Home Interface Registration
- Bean Activation and Passivation
- EJBs As Client Applications

The sections that follow discuss each of these considerations in detail.

Choosing Between Session and Entity Beans

When to use one bean type or another depends upon the design pattern that bean programmers want to use. There are a few commonly used rules:

- Stateless session beans provide a capability similar to the service model provided by the Tuxedo system. They are highly recommended for short interactions with the business data when there is no need to retain state. Therefore, they do not need special operations to activate or deactivate their state. EJB containers can freely pool instances, allocate instances as needed, and apply load balancing strategies to distribute the load across different servers.
- Stateful session beans are recommended when it is necessary to retain in-memory state across multiple method invocations made by the client. These beans are more expensive than stateless session beans because they allocate and exclusively reserve resources during the private conversation with the client application.

  When developing a stateful session bean, you must implement the `ejbPassivate` and `ejbActivate` methods in such a way that resources like JDBC connections and network connections are handled properly. While the EJB container is responsible for saving the conversational state in a portable way and

for reconstructing that state during activation, some precautions must be taken by the Bean Provider to ensure that the EJB container handles state correctly. (See section 6.4.1 in the EJB 1.1 Specification for more information.) You can also decide if the bean's state needs to be synchronized when the bean is involved in a distributed transaction.

- Entity beans are recommended when it is necessary to associate a bean instance with a particular application-defined identity, which is similar to the CORBA model, and the bean's state must be persistent (that is, the state cannot be lost). Entity beans cannot use the `SessionSynchronization` interface to synchronize with the starting and stopping of a transaction.

  Entity beans can be used in many ways; for example, to implement a persistent variant of CORBA objects or to provide an object representation of entities stored in a database. You must be careful when you use entity beans to model objects stored in a database, because these beans could introduce inefficiencies, such as having most of the business logic on the client application rather than in the server application. Moving the business logic to the server application reduces the number of invocations needed to perform a particular business transaction.

Server Startup

The WLE EJB container gives you the flexibility to specify an object that is invoked by the EJB container when a WLE server process loads an EJB JAR file. This object is an instance of a class that implements the `Server` interface, and is thus referred to as a **module initializer object**. Implementing the module initializer object is described in the section "Step 6: Specify the module initializer object in the WebLogic EJB extensions to the deployment descriptor DTD." on page 8-14.

You can use the module initializer object to perform specialized initialization for some objects, such as instantiating RMI objects. In WLE EJB applications, you can specify this class in the `<module-initializer-class>` element in the WebLogic EJB extensions to the deployment descriptor DTD, which is a special deployment descriptor that you create along with the standard deployment descriptor.

Home Interface Registration

When an EJB JAR file is deployed, the WLE container recognizes the EJB home interfaces (factories) and automatically registers them within the WLE JNDI context. The information about the home interfaces is retrieved from the deployment descriptor.

Bean Activation and Passivation

EJB containers have complete control of the passivation of EJBs. This allows the container to pool instances of a bean and to decide when an instance can be passivated (or removed from the pool) to provide better use of system resources.

As a bean programmer, you cannot make any assumptions about when a bean object is passivated. This passivation can happen at any time. This is particularly important when the bean accesses a database via cursors, because these cursors could become invalid after the passivation; the EJB container can reactivate the bean in another server process.

The WLE EJB container currently follows a passivation model that is similar to the model used by the WLE TP Framework for CORBA applications. If resources are scarce, the WLE EJB container may passivate an object at any time. When the bean is reactivated, it may be reactivated in the same server process or in another server process in the same group.

If a client application creates or invokes a stateful or entity bean within a transaction, the bean will never be passivated while it is participating in the transaction. If the client invocation is nontransactional, the bean may be passivated at the end of the method invocation.

For concurrency control, the WLE system applies the following rules:

- For entity beans, for a given primary key, there is only one active instance of the bean at one time. This constraint is compatible with the activation policy provided by the WLE TP Framework.
- Although the client application can issue concurrent invocations on a bean, the WLE EJB container queues concurrent invocations on a bean so those invocations are performed one at a time. The WLE system enforces this rule by running each active object on its own thread. This rule is mandated by the EJB 1.1 Specification. Also, note that the EJB Specification discourages the direct use of threads by EJB programmers.

Note: If a passivated stateful bean is not removed due to application or system errors, its passivated state takes up disk space in the location specified by the `<persistence-store-directory-root>` element. This passivated state remains on disk until the temporary files containing that state are deleted by the System Administrator. These files can be identified by the syntax of their names, which include the following information:

- The server name, which includes the server group name and the group ID
- The server generation ID
- The bean name
- A string of several digits

The server name and bean name components of the file name are the most readily identifiable. To manage the number of unused bean state files that can potentially accumulate over time, System Administrators may choose to create scripts that delete those files whenever the WLE system is started or shut down.

EJBs As Client Applications

A bean may invoke the methods of another bean. When a bean behaves as a client application, the client rules still apply: the bean must obtain the reference to the other bean from that bean's home interface (factory), and references to the home interface must be obtained using JNDI.

The main differences with the client environment are the following:

- When the bean creates the `InitialContext` object, there is no authentication or connection setup because WLE Java servers run within the trusted server base.
- The WLE EJB container does not support reentrancy, and rejects loopback calls (a bean calling another bean that then calls the first bean) by throwing an exception (`java.rmi.RemoteException`) to the client application.

Note: The WLE EJB container does not propagate the security and transaction context on callbacks to a J2EE client.

Security, Transactions, and JDBC Connections

For additional EJB design considerations, see the following in the WebLogic Enterprise online documentation:

- *Using Security*
- *Using Transactions*
- *Using the JDBC Drivers*

Developing EJB Applications for the WLE System

This topics provides the following sections:

- Development Steps
- EJB Examples

Development Steps

The primary steps for developing an EJB are the following:

- Step 1: Write or obtain an EJB.
- Step 2: Create a deployment descriptor.
- Step 3. Package the EJB components into a Java Archive (JAR) file.

After you have created the EJB JAR file, you can then build and deploy the bean. The process for deploying an EJB in the WLE environment is described in Chapter 8, “Building and Deploying Enterprise JavaBeans (EJBs).”

Step 1: Write or obtain an EJB.

The EJB 1.1 Specification, published by Sun Microsystems, Inc., describes the different requirements of the EJB writer and the EJB framework; EJBs created for the WLE environment must conform to those requirements. When you write EJBs, pay close attention to these requirements.

When writing an EJB, you must implement the following:

- The business methods for the bean
- The `ejbCreate`, `ejbPostCreate`, and `ejbRemove` methods
- For session beans, the callbacks defined by the `SessionBean` interface and, optionally, the callbacks on the `SessionSynchronization` interface
- For entity beans, the callbacks defined on the `EntityBean` interface
- For bean-managed persistence, the `ejbLoad` and `ejbStore` callbacks
- The home interface, remote interface, and, for entity beans, the primary key classes

Note that the direct use of threads by Bean Providers is discouraged by the EJB 1.1 Specification. This constraint also applies to WLE server applications -- bean and RMI implementers should not attempt to manage, change properties, start, stop, suspend, or resume a thread or a thread group.

The `ejbc` command, which is provided with the WLE development software, includes a compliance checker utility that examines the packaged EJBs and determines if the EJBs conform to these requirements.

For examples of bean implementations, see the section “EJB Examples” on page 7-15.

Step 2: Create a deployment descriptor.

The deployment descriptor ties together the different classes and interfaces, and is used by the `ejbc` command to build the code-generated class files. You can also use the deployment descriptor to specify critical aspects of the EJB's deployment at run time.

Required Elements in the Deployment Descriptor

The elements that you, as the Bean Provider, need to specify in the deployment descriptor for each EJB is listed and described in the following table.

Element	Description	Purpose
`ejb-name`	EJB's name	Specifies the logical name you assign to each EJB in the EJB JAR file. There is no architected relationship between this name and the JNDI name that the Deployer assigns to the EJB.
`ejb-class`	EJB's class	Specifies the fully qualified name of the Java class that implements the EJB's business methods.
`home`	EJB's home interface	Specifies the fully qualified name of the EJB's home interface.
`remote`	EJB's remote interfaces	Specifies the fully qualified name of the EJB's remote interface.
`session | entity`	EJB's type	The EJB types are `session` and `entity`. Use the appropriate `session` or `entity` element to declare the EJB's structural information.
`session-type`	Session bean's state management type	Declares whether the session bean is stateful or stateless.
`transaction-type`	Session bean's transaction demarcation type	If the EJB is a session bean, declares whether transaction demarcation is performed by the enterprise bean or by the container.
`persistence-type`	Entity bean's persistence management	If the EJB is an entity bean, declares whether persistence management is performed by the EJB or by the container.
`prim-key-class`	Entity bean's primary key class	If the enterprise bean is an entity bean, specifies the fully qualified name of the entity bean's primary key class. You must specify the primary key class for an entity with bean-managed persistence, and you can optionally specify the primary key class for an entity with container-managed persistence.
`cmp-fields`	Container-managed fields	If the EJB is an entity bean with container-managed persistence, specifies the container-managed fields.

For complete details about the elements you specify in the standard deployment descriptor, see the *EJB XML Reference* in the WebLogic Enterprise online documentation.

How to Create the Deployment Descriptor

You can create the deployment descriptor by any of the following methods:

- Using the WebLogic EJB Deployer
- Using the `DDGenerator` command
- Creating the deployment descriptor XML file from scratch, or by copying one of the examples in this guide into a text editor and modifying it

The `DDGenerator` command is available from the WLE Developer Center. For information about locating and using this tool, see the *Release Notes*.

The deployment descriptor you create must:

- Be valid with respect to the Document Type Definition (DTD) documented in the EJB 1.1 Specification
- Conform to the semantics rules specified in the DTD comments and elsewhere in the EJB 1.1 Specification
- If you are creating the deployment descriptor from scratch, include the following reference to the deployment descriptor DTD at the beginning:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

The following is an example standard deployment descriptor for a container-managed entity bean. The specific class variables whose persistence is container-managed are `accountId` and `salary`. These fields are declared public in the bean's implementation, and are shown in boldface type in the following example:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>
        oracle
      </ejb-name>
      <home>
        samples.j2ee.ejb.sequence.oracle.OracleHome
      </home>
```

```
<remote>
  samples.j2ee.ejb.sequence.oracle.Oracle
</remote>
<ejb-class>
  samples.j2ee.ejb.sequence.oracle.OracleBean
</ejb-class>

<!-- Primary key class -->
<persistence-type>
  Container
</persistence-type>
<prim-key-class>
  samples.j2ee.ejb.sequence.oracle.OraclePK
</prim-key-class>
<!-- Specify type of persistence -->
<reentrant>
  false
</reentrant>

<!-- Container managed fields for this EJB -->
<cmp-field>
  <field-name>
    accountId
  </field-name>
</cmp-field>
<cmp-field>
  <field-name>
    salary
  </field-name>
</cmp-field>

<!-- Environment entries: Application specific and referred by
      OracleBeanBean
-->
<env-entry>
  <env-entry-name>
    sequenceName
  </env-entry-name>
  <env-entry-type>
    java.lang.String
  </env-entry-type>
  <env-entry-value>
    ejbSequence
  </env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>
    creationRange
  </env-entry-name>
```

```
            <env-entry-type>
              java.lang.Integer
            </env-entry-type>
            <env-entry-value>
              4
            </env-entry-value>
          </env-entry>

          <!-- VERBOSE is used to set tracing on or off in EJBean-->
          <env-entry>
            <env-entry-name>
              VERBOSE
            </env-entry-name>
            <env-entry-type>
              java.lang.Boolean
            </env-entry-type>
            <env-entry-value>
              true
            </env-entry-value>
          </env-entry>


        </entity>
      </enterprise-beans>

      <!-- Assembly information -->
      <assembly-descriptor>
        <container-transaction>
          <method>
            <ejb-name>
              oracle
            </ejb-name>
            <method-name>
              *
            </method-name>
          </method>
          <trans-attribute>
            Required
          </trans-attribute>
        </container-transaction>
      </assembly-descriptor>
    </ejb-jar>
```

Step 3. Package the EJB components into a Java Archive (JAR) file.

In this step, you package the deployment descriptor, the source files for the EJB classes, and any additional required classes into a special kind of JAR file called the EJB JAR file. You can package multiple beans together, provided that there is a deployment descriptor for each bean.

You can use the WLE `ejbc` command to create the EJB JAR file, as in the following example:

```
java weblogic.ejbc -i ejb-jar.xml -x weblogic-ejb-extensions.xml ejb-jar-file
```

In the preceding command line, the `-i` option specifies the name of the deployment descriptor, and *`ejb-jar-file`* represents the name of the EJB JAR file. For more information about the `ejbc` command, see *Command Reference* in the WebLogic Enterprise online documentation. Note that using the `-validate` option is recommended.

EJB Examples

This section provides a general discussion about EJBs and persistence, and shows sample fragments of EJB implementation code and deployment descriptors to illustrate the following:

- Container-managed Entity Beans
- Bean-managed Entity Beans
- Stateful Session Beans
- Stateless Session Beans

The topics described in this section use code from the EJB sample applications that are installed with the WLE software.

Note: The code fragments shown in this section are taken from the EJB samples provided with the WLE 5.0 software. These samples include tracing code, which is turned on by the `VERBOSE` flag, that helps show what is happening when the samples are executed. These tracing statements are not required by the EJB 1.1 Specification; they are present for instructional purposes only.

Development Considerations for EJBs and Persistence

Persistence refers to a bean's state information, which may be contained in durable storage when the bean is not active. When the bean is activated, this state is read in from durable storage. As a Bean Provider, you basically have two choices for what kind of broad mechanism you want to use for handling a bean's persistence: either directly in the bean's logic, or by delegating to the EJB container the tasks of handling the bean's persistence.

A bean that delegates to the container all the logic for handling its persistent data has what is referred to as **container-managed persistence**. A bean that contains its own logic for handling its persistence data has what is referred to as **bean-managed persistence**. The choice you make for a bean must be specified in the bean's deployment descriptor.

Container-managed Entity Beans

If your entity bean uses container-managed persistence, you need to do the following:

- In the bean's standard deployment descriptor, define the elements described in the section "Required Deployment Descriptor Elements for Container-managed Beans" on page 7-17.
- In the bean's implementation, declare as public the class variables whose persistence is container-managed, as shown in the code fragment in the section "Declaring Container-managed Fields as Public Variables" on page 7-17.

  Note: The EJB 1.1 Specification requires that class variables whose persistence is container-managed have public access.

The subsections that follow also provide code fragments that show the use of the `ejbCreate`, `ejbStore`, and `ejbRemove` methods in such beans.

Required Deployment Descriptor Elements for Container-managed Beans

The required deployment descriptor elements for container-managed beans are listed and described in the following table.

Deployment Descriptor Element	Description
`cmp-fields`	This element specifies the container-managed fields. This is a standard property that lists the public nontransient instance variables that the EJB expects will be made automatically persistent. Even if there are no managed fields, the bean's object reference and the primary key are remembered by the EJB container.
`persistence-type`	If the EJB is an entity bean, this element declares whether the persistence management is performed by the container or by the bean. For container-managed persistence, this element should specify `container`.
`persistence-store-jdbc`	This element specifies that the type of persistent store is `jdbc`. (Note that in WLE, file-based persistence for entity beans is not supported.)
`primary-key-class`	The deployment descriptor for any entity bean must set this element, which identifies the primary key class for the bean.

Declaring Container-managed Fields as Public Variables

The following code fragment shows a container-managed EJB declaring its container-managed fields as public class variables. These variables need to be public so that the container can manage them, and they are also specified in the sample deployment descriptor shown in the section "How to Create the Deployment Descriptor" on page 7-12.

```
// public container managed variables
public           String      accountId; // also the primary Key
public           double      balance;
public           String      type; // "Checking"
```

The ejbCreate Method

A container-managed entity bean needs to implement the `ejbCreate` method. Note that the `accountId` and `initialBalance` parameters in this method are managed by the container. The following code fragment shows a container-managed bean setting the values of the public class variables shown in the code fragment in the section "Declaring Container-managed Fields as Public Variables" on page 7-17.

```
public void ejbCreate(String accountId, double initialBalance,
                      String type) {
    if (VERBOSE)
      System.out.println("ejbCreate( id = " + id() +
                         ", initial balance = $" + initialBalance +
                          ", type: " + type + ")");
    this.accountId = accountId;
    this.balance   = initialBalance;
    this.type      = type;
  }
```

The ejbStore Method

The EJB 1.1 Specification states that implementing the `ejbStore` method in a container-managed entity bean is required, even if the method does not provide any specific functionality. One advantage to having this method in your container-managed bean is to provide a tracing capability for debugging purposes, as in the following example:

```
public void ejbStore() {
    if (VERBOSE)
      System.out.println("ejbStore (" + id() + ")");
  }
```

The ejbRemove Method

As with the `ejbStore` method in the preceding section, the `ejbRemove` method is not a functional component of a container-managed bean implementation; however, the EJB 1.1 Specification requires this method to be present, as in the following example:

```
public void ejbRemove() throws RemoveException {
    if (VERBOSE)
      System.out.println("ejbRemove (" + id() + ")");
  }
```

Bean-managed Entity Beans

If you are implementing a bean with bean-managed persistence, you need to do the following:

- Declare the bean's persistence type in the persistence-type element in the standard deployment descriptor.
- Implement code in the bean that accesses the JDBC connection pool.
- Implement the `ejbCreate` and `ejbStore` methods to create a database entity that contains the bean's persistent data, and store that data.

The code fragments provided in this section illustrate performing these tasks, as well as using the `ejbRemove` method to remove a bean's persistent data from a database.

Accessing the JDBC Pool

The following code example shows a bean-managed entity bean using static initialization to establish access to the JDBC pool, which is defined in the EJB application's `UBBCONFIG` file:

```
static {
    try{
      Context ctx = new InitialContext();
      pool = (DataSource)ctx.lookup("jdbc/pool1");
    } catch(Exception e) {
        System.out.println("problem with datasource.");
    }
  }
```

The ejbCreate Method

A bean-managed entity bean uses the `ejbCreate` method to create the bean and update the table in the database that contains the entity bean's value. The following code fragment shows creating a row in the table in that database, using a JBDC connection from the pool:

```
public AccountPK ejbCreate(String account_id, double initial_balance)
    throws CreateException,
  {
    if (VERBOSE) {
      System.out.println("AccountBean.ejbCreate( id = " +
                         System.identityHashCode(this) + ", PK = " +
                         account_id + ", " + "initial balance = $ " +
```

```
                        initial_balance + ")");
    }
    AccountId = account_id;
    Balance = initial_balance;

    Connection connection = null;
    PreparedStatement prep_stmt = null;


    try {
      connection = pool.getConnection();
      prep_stmt = connection.prepareStatement("insert into ejbAccounts "+
                                              "(id, bal) values (?, ?)");
      prep_stmt.setString(1, AccountId);
      prep_stmt.setDouble(2, Balance);
      if (prep_stmt.executeUpdate() != 1) {
        throw new CreateException ("JDBC did not create any row");
      }
      AccountPK primary_key = new AccountPK();
      primary_key.AccountId = AccountId;
      return primary_key;
    } catch (CreateException ce) {
       throw ce;
    } catch (SQLException sqe) {
       throw new CreateException (sqe.getMessage());
    } finally {
       try {
        prep_stmt.close();
        connection.close();
       } catch (Exception e) {
       }
    } // end of finally
  } // end of ejbCreate(..)
```

Updating the Database

The following code fragment shows updating the database with the values. Since this bean uses bean-managed persistence, updating the database is done manually. Whereas the code in the previous example created the database rows, the code in the following fragment specifies the values in those rows.

```
ejbStore()
public void ejbStore() throws EJBException {

    if (VERBOSE) {
      System.out.println("ejbStore (" + id() + ")");
    }
```

```
    Connection connection = null;
    PreparedStatement prep_stmt = null;
    try {
      connection = pool.getConnection();
      prep_stmt = connection.prepareStatement("update ejbAccounts set bal = "+
                                              "? where id = ?");
      prep_stmt.setDouble(1, Balance);
      prep_stmt.setString(2, AccountId);
      int i = prep_stmt.executeUpdate();
      if (i == 0) {
        throw new RemoteException ("ejbStore: AccountBean (" + AccountId +
                                   ") not updated");
      }
    } catch (RemoteException re) {
      throw re;
    } catch (SQLException sqe) {
      throw new EJBException (sqe.getMessage());
    } finally {
      try {
        prep_stmt.close();
        connection.close();
      } catch (Exception e) {
         throw new EJBException (e.getMessage());
      }
    } // end of finally
  } // end of ejbStore()
```

Removing Values from the Database

The following code fragment shows using the `ejbRemove` method to remove rows from the database that were created and set in the preceding code examples.

```
public void ejbRemove()
    throws RemoveException,
  {
    if (VERBOSE) {
      System.out.println("ejbRemove (" + id() + ")");
    }
    // we need to get the primary key from the context because
    // it is possible to do a remove right after a find, and
    // ejbLoad may not have been called.

    Connection connection = null;
    PreparedStatement prep_stmt = null;
    try {
      connection = getConnection();
      AccountPK pk = (AccountPK) ctx.getPrimaryKey();
      prep_stmt = connection.prepareStatement("delete from ejbAccounts where "+
```

```
                                              "id = ?");
      prep_stmt.setString(1, pk.AccountId);
      int i = prep_stmt.executeUpdate();
      if (i == 0) {
        throw new EJBException ("AccountBean ("
                                + pk.AccountId + " not found");
      }
    } catch (RemoteException re) {
       throw re;
    } catch (SQLException sqe) {
       throw new RemoteException (sqe.getMessage());
    } finally {
       try {
         prep_stmt.close();
         connection.close();
       } catch (Exception e) {
          throw new EJBException (e.getMessage());
       }
    } // end of finally
  } // end of ejbRemove()
```

Stateful Session Beans

This section shows the following examples of stateful session beans:

- Required standard deployment descriptor elements
- Code fragments showing two stateful session beans: one in which the client keeps track of the bean's state, and one in which the bean keeps track of its state

The code examples shown here are from the EJB Samples directory, which is available with the WLE software.

Example Deployment Descriptor

A deployment descriptor for a stateful session bean can optionally define the `persistentDirectoryRoot` element. The default file is `/pstore/`*`bean_name`*`.dat`, where the directory `pstore` represents the directory from which the WLE application was started, and *`bean_name`* is the fully qualified name of the EJB with underscores (_) replacing the periods (.) in the name.

If the `persistentStoreType` element is defined as `jdbc`, the container looks for additional values to determine the appropriate values for the JDBC connection. Note that if the bean's persistence is stored in a database via a JDBC connection, the system

administrator needs to add this information to the UBBCONFIG file as well. For more information, see *Using the JDBC Drivers* in the WebLogic Enterprise online documentation.

The following deployment descriptor fragment shows the location of the persistent store for a stateful session bean:

```
<persistence-store-descriptor>
        <persistence-store-file>
          <persistence-store-directory-root>
            c:\mystore
          </persistence-store-directory-root>
        </persistence-store-file>
      </persistence-store-descriptor>
```

Client Application Maintaining a Bean's State Information

The following code example shows an EJB client application keeping track of a bean's state information. In stateful session beans, you need to provide a one-to-one mapping between the client and the bean in the server, represented by a key. This key provides the map between the bean's instance and the client, because the bean instance cannot be shared with other clients.

The following code fragment shows the client application code creating the stateful session bean using the primary class key:

```
// Give this trader a name
      Trader trader = brokerage.create("Terry");
      System.out.println("Creating trader " + trader.getTraderName() + "\n");

      String stockName;
      int    numberOfShares;

      for (int i = 1 ; i <= 5; i++) {
        System.out.println("Start of Transaction " + i + " for " + customerName);

        // Buying
        stockName      = "WEBL";
        numberOfShares = 100 * i;
        System.out.println("Buying " + numberOfShares + " of " + stockName);
        TradeResult tr = trader.buy(customerName, stockName, numberOfShares);
        System.out.println("...Bought " + tr.numberTraded + " at $" +
                           tr.priceSoldAt);
        // Selling
        stockName      = "INTL";
        numberOfShares = 100 * (i+1);
```

```
        System.out.println("Selling " + numberOfShares + " of " + stockName);
        tr = trader.sell(customerName, stockName, numberOfShares);
        System.out.println("...Sold " + tr.numberTraded + " at $" +
                           tr.priceSoldAt);

        // Get change in Cash Account from EJBean
        System.out.println("Change in Cash Account: $" + trader.getBalance());

        System.out.println("End of Transaction " + i + "\n");
      }
     System.out.println("Change in Cash Account: $" + trader.getBalance() + "\n");
      System.out.println("Removing trader " + trader.getTraderName());
      trader.remove();
    }
    catch (ProcessingErrorException pe) {
      System.out.println("Processing Error: " + pe);
      pe.printStackTrace();
    }
    catch (Exception e) {
      System.out.println(":::::::::::::: Error ::::::::::::::::");
      e.printStackTrace();
    }
```

Bean Keeping Track of Its Own State

The following code example shows a stateful session bean keeping track of its state, and its mapping to a specific client. For example, the balance is kept on the EJB rather than on the client.

```
  // The reason the following attribute is public is to test
  // passivation into a persistent store, because the deployment descriptor
  // says it should be a stateful session bean.
  // This and the ejbCreate method in this file are the differences
  // between the examples in the stateful and stateless directories.
  public  String                  traderName;
  public  double                  tradingBalance;

  // ---------------------------------------------------------------
.
.
.

public TradeResult buy(String customerName, String stockSymbol,
                       int shares)
    throws ProcessingErrorException
  {
    if (VERBOSE && shares >= 0) {
      System.out.println("buy (" + customerName + ", " +
```

```
                        stockSymbol + ", " +
                        shares + ")");
    }
    try {
      int tradeLimit = getTradeLimit();
      if (shares > tradeLimit)       // limit for buying
        shares = tradeLimit;
      else if (shares < -tradeLimit) // limit for selling
        shares = -tradeLimit;

      double price = getStockPrice(stockSymbol);
      tradingBalance = tradingBalance - (shares * price); // subtract purchases
from cash account

      if (shares < 0)
        shares = -shares;
      return new TradeResult(shares, price);
    }
    catch (Exception e) {
      throw new ProcessingErrorException("Trader error: " + e);
    }
  }
```

Stateless Session Beans

This section provides the following two code examples:

- An EJB client application keeping track of a stateless session bean's state
- A stateless bean that keeps track of its own state data

Client Maintaining Bean's State

The following example shows a client application keeping track of the `cashBalance` variable, which is manipulated by the stateless bean. This example also shows the client invoking the `ejbCreate` method without any arguments and without any specific data.

```
 try {
      String    customerName  = "Erin";   // Default name for the customer
      Context   ctx           = null;     // To hold JNDI context
      Object    objref        = null;     // to hold object reference

      String    stockName       = null;   // Name of a stock
      int       numberOfShares  = 0;      // No. of shares to trade
      double    cashBalance     = 0.0;    // To hold balance between sessions
```

```
TraderHome  brokerage   = null;     // To hold home interface
Trader      trader      = null;     // To hold trader object
TradeResult tradeResult = null;     // To hold results from a trade

// Create a new initial context based on the url, user, and password
ctx = newInitialContext();

if ( ctx == null ){
  System.out.println("Initial context is null");
  exit(-1);
}
// do a JNDI lookup for the EJB;defined in the deployment descriptor
objref = ctx.lookup("statelessSession.TraderHome");
printTrace("Looked up home:");

/* Create a trader object, who'll later help us execute trades
 * The lookup has resulted in an Object.  We know
 * this object is actually a reference of type TraderHome,
 * so the reference is narrowed and cast to that type:
*/
brokerage = (TraderHome) PortableRemoteObject.narrow(objref,
                                                     TraderHome.class);
printTrace("Narrowed home.");

/* Create the EJB on the WLE server.
 * Unlike the statefulSession example,we don't give this trader a key
*/
printTrace("Creating trader.");
trader = brokerage.create();

/* Unlike the statefulSession example,
 * we have to keep track of the balance over the
 * life of our use of the session bean
 */

for (int i = 1 ; i <= maxTransaction; i++) {
  System.out.println("Start of Transaction " + i + " for " +
                     customerName);

  /* Buying
   * Stock symbol must be found in the deployment descriptor's environment
   * properties section. TraderBean EJB will check the validity of the
   * symbol, and its price using JNDI lookup on the environment
   * properties.
   */

  stockName      = "BEAS";
  numberOfShares = 100 * i;
```

```
        System.out.println("Buying " + numberOfShares + " of " + stockName);

        // buy() is executed on the TraderBean EJB in the WLE Server
        tradeResult = trader.buy(customerName, stockName, numberOfShares);
        System.out.println("...Bought " + tradeResult.numberTraded + " at $" +
                           tradeResult.priceSoldAt);

        // Keep track of the change in the Cash Account
        cashBalance = cashBalance - (tradeResult.numberTraded *
                                     tradeResult.priceSoldAt);

        // Selling
        stockName      = "INTL";
        numberOfShares = 100 * (i+1);
        System.out.println("Selling " + numberOfShares + " of " + stockName);

        // sell() is executed on the TraderBean EJB in the WLE Server
        tradeResult = trader.sell(customerName, stockName, numberOfShares);
        System.out.println("...Sold " + tradeResult.numberTraded + " at $" +
                           tradeResult.priceSoldAt);

        // Keep track of the change in the Cash Account
        cashBalance = cashBalance + (tradeResult.numberTraded *
tradeResult.priceSoldAt);

        // Print change in Cash Account
        System.out.println("Change in Cash Account: $" + cashBalance);
        System.out.println("End of Transaction " + i + "\n");
      }
      System.out.println("Change in Cash Account: $" + cashBalance + "\n");
      System.out.println("Removing trader");

      // Remove TraderBean EJB from the WLE server.
      trader.remove();
    }
    catch (ProcessingErrorException pe) {
      System.out.println("Processing Error: " + pe);
      pe.printStackTrace();
    }
    catch (Exception e) {
      System.out.println(":::::::::::::: Error ::::::::::::::::");
      e.printStackTrace();
    }
```

Stateless Bean Tracking Its Own State

The following code fragment shows a business method from the `TraderBean` example, available in the EJB Samples directory provided with the WLE software. In this example, the bean does not preserve any state. The bean's `buy` method performs simple calculations on data provided by the client application.

```
getStockPrice() and getTradeLimit() methods use DD environment properties to
access constant values using JNDI lookup() - prevents hardcoding data.

public TradeResult buy(String customerName, String stockSymbol,
                       int shares)
    throws ProcessingErrorException
  {

    if (shares >= 0) {
      printTrace("buy (" + customerName + ", " +
                         stockSymbol + ", " +
                         shares + ")");
    }
    try {
      int tradeLimit = getTradeLimit();
      if (shares > tradeLimit)
        shares = tradeLimit;
      else if (shares < -tradeLimit) // limit for selling
        shares = -tradeLimit;

      double price = getStockPrice(stockSymbol);
      printTrace("Executing buy...");
      if (shares < 0)
        shares = -shares;
      return new TradeResult(shares, price);
    }
    catch (Exception e) {
      throw new ProcessingErrorException("Trader error: " + e);
    }
  }
```

8 Building and Deploying Enterprise JavaBeans (EJBs)

This topic includes the following sections:

- Overview of the EJB Building and Deploying Process
- Steps for Building and Deploying EJBs

Overview of the EJB Building and Deploying Process

Building and deploying EJBs in the WebLogic Enterprise environment requires careful planning to define how to locate these EJBs in the WLE distributed environment.

After the Bean Provider has implemented an EJB's business logic and has produced an initial deployment descriptor, the process for building and deploying that EJB in the WLE environment includes the following steps:

- Step 1: Obtain the EJB JAR file from the bean provider.
- Step 2: Modify the deployment descriptor.

- Step 3: Create the WebLogic EJB extensions to the deployment descriptor DTD.
- Step 4: Produce the deployable EJB JAR file.
- Step 5: Configure the EJB application.
- Step 6: Specify the module initializer object in the WebLogic EJB extensions to the deployment descriptor DTD.

When you build the EJB that has been produced by the Bean Provider, the end result is an EJB JAR file. The WLE system allows you to build two kinds of EJB JAR files:

Standard EJB JAR file	An EJB that has been built, but lacks the specific deployment information on any specific system. You typically build a standard EJB with the goal of being able to distribute that EJB to a variety of deployment environments. If you are only creating a standard EJB JAR file, you only need to perform steps 1 and 2 in this topic.
Deployable EJB JAR file	An EJB that has been built with deployment descriptor information specific to a particular deployment environment. The steps described in this chapter for building a deployable EJB JAR file specifically create an EJB that can be deployed on a WLE system, and require you to perform steps 1 through 4 in this topic.

The remainder of this topic discusses each of these steps in detail.

Steps for Building and Deploying EJBs

This section describes the steps for building and deploying EJBs in the WLE environment and also provides the following sections:

- Scaling an EJB Application
- For More Information

Step 1: Obtain the EJB JAR file from the bean provider.

The first step in deploying an EJB is to obtain the EJB JAR file from the Bean Provider. In addition to the class files contained in the EJB JAR file, the EJB JAR file also has a deployment descriptor for each bean in that file. The steps for producing the Bean Provider's EJB JAR file are described in the section "Development Steps" on page 7-9.

Because multiple EJBs can be joined together in a single, deployable unit, part of the assembly process is joining the EJB JAR files for each of the beans.

Step 2: Modify the deployment descriptor.

As stated in Chapter 7, "Designing and Developing Enterprise JavaBeans for the WLE System," the deployment descriptor ties together the different classes and interfaces, and is used to build the code-generated class files. It also allows you to specify some aspects of the EJB's deployment at run time.

The Bean Provider specifies some initial deployment information in the deployment descriptor. The deployer typically needs to add to or modify that information, such as shown in the following table:

The EJB's name	You may change the enterprise bean's name defined in the `ejb-name` element.
Values of environment entries	You may change existing values or define new values for the environment properties.
Description fields	You may change existing or create new description elements.
Binding of enterprise bean references	You may link an enterprise bean reference to another enterprise bean in the EJB JAR file. You create the link by adding the `ejb-link` element to the referencing bean.

Security roles	You may define one or more security roles. The security roles define the recommended security roles for the clients of the enterprise beans. You define the security roles using the `security-role` elements. For more information about EJB security, see *Using Security* in the WebLogic Enterprise online documentation.
Method permissions	You may define method permissions, which are binary relationships between the security roles and the methods of the remote and home interfaces of the EJBs. You define method permissions using the `method-permission` elements.
Linking of security role references	If you define security roles in the deployment descriptor, you must link the security role references declared by the Bean Provider to the security roles. You define these links using the `role-link` element. For more information about EJB security, see *Using Security* in the WebLogic Enterprise online documentation.
Changing persistent storage information, if necessary	The deployer can change the type of persistent storage used by a bean. If the `persistentStoreType` is `file`, the serialized files are created in this directory. The default file is */pstore/bean_name*`.dat`, where the directory *`pstore`* represents the directory from which the WLE application was started, and *`bean_name`* is the fully qualified name of the EJB with underscores (_) replacing the periods (`.`) in the name. If the `persistentStoreType` is `jdbc`, the container looks for additional values to determine the appropriate values for the JDBC connection. Note that if the bean's persistence is stored in a database via a JDBC connection, the system administrator needs to add this information to the `UBBCONFIG` file as well. For more information, see *Using the JDBC Drivers* in the WebLogic Enterprise online documentation. Note that persistence information is specified in the WebLogic EJB extensions to the deployment descriptor DTD file, as described in the section "Specifying Persistence Information" on page 8-8.

To modify a deployment descriptor, you can use either of the following two methods:

- The WebLogic EJB Deployer
- The manual modification of the deployment descriptor in a text editor

The type of deployment descriptor you produce depends on whether you are creating a standard EJB JAR file or a deployable EJB JAR file.

Creating a Standard EJB JAR File

If you are creating a standard EJB JAR file, you only need to modify the single deployment descriptor contained in the EJB JAR file produced by the Bean Provider. When modifying this deployment descriptor, use the syntax described in the EJB 1.1 Specification produced by Sun Microsystems, Inc.

If you are creating a standard EJB JAR file:

1. Make the appropriate modifications to the bean's deployment descriptor file.
2. Run the `ejbc` command, specifying the `-nodeploy` option.

For more information about the `ejbc` command, see the *Command Reference* in the WebLogic Enterprise online documentation.

Creating a Deployable EJB JAR File

If you are creating a deployable EJB JAR file, you need to do the following:

- Modify the deployment descriptor, as described in the section "Creating a Standard EJB JAR File" on page 8-5.
- Create a WebLogic Enterprise extended deployment descriptor file, which is also created as an XML file, and which specifies the information described in the section "Step 3: Create the WebLogic EJB extensions to the deployment descriptor DTD." on page 8-6.

Step 3: Create the WebLogic EJB extensions to the deployment descriptor DTD.

For an EJB application to be deployable in the WLE environment, you need to create a file containing the WebLogic EJB extensions to the deployment descriptor DTD. This file specifies the following run time and configuration information for the EJB application:

- Custom application startup and shutdown properties
- Registration of the application's home interfaces
- Persistence information

Specifying the WebLogic EJB Extensions DTD

The file that includes the WebLogic EJB extensions to the deployment descriptor DTD must specify the following DTD reference at the beginning:

```
<!DOCTYPE weblogic-ejb-extensions SYSTEM "weblogic-ejb-extensions.dtd" >
```

Specifying the Module Initializer Class

Server application startup and shutdown can be handled automatically by the EJB container. However, if you have special server initialization or shutdown requirements, use the `module-initializer-class-name` element in the WebLogic EJB extensions to the deployment descriptor DTD to specify the name of the **module initializer object**.

The syntax for specifying a module initializer object for handling the `initialize` and `release` methods is similar to that for CORBA and RMI. However, with the CORBA and RMI models, the startup and shutdown information is specified in an XML file and is serialized by the `buildjavaserver` command. For EJB deployment, XML elements for startup and shutdown procedures are specified together with the other elements in the WebLogic EJB extensions to the deployment descriptor DTD, and you process them using the `ejbc` command. The WLE EJB container parses the XML at run time and performs the startup and shutdown processing. See the section "Step 5: Configure the EJB application." on page 8-14 for a complete description of startup and shutdown handling in WLE.

XML DTD Syntax

```
<!ELEMENT module-initializer-class (module-initializer-class-name*))>
<!ELEMENT module-initializer-class-name (#PCDATA)>
```

Example

```
<weblogic-ejb-extensions>
. . .
       <module-initializer-class>
              <moduleinitializer-class-name>ServerImpl
              </moduleinitializer-class-name>
       </module-initializer-class>
</weblogic-ejb-extensions>
```

Registering Names for the EJB Home Classes

A name for the EJB home class must be registered in the global WLE JNDI namespace. This allows Java clients to perform a lookup on the JNDI name for the EJB home and gain access to the object. The name for the EJB home class can be different than the `<ejb-name>` element specified in the standard EJB XML. The `<ejb-name>` in the standard deployment descriptor must be unique only among the names of the EJBs in the same EJB JAR file. However, the JNDI name must be unique among all global home or factory names in a WLE domain; this includes EJB homes, CORBA factories, and RMI named objects.

XML DTD Syntax

```
<!ELEMENT jndi-name (#PCDATA)>
```

Example

```
<weblogic-ejb-extensions>
  <weblogic-version>
    WebLogic Enterprise Server 5.0
  </weblogic-version>
  <weblogic-enterprise-bean>
    <ejb-name>
      oracle
    </ejb-name>
    <weblogic-deployment-params>

      <max-beans-in-free-pool>
        20
```

```
            </max-beans-in-free-pool>
            <max-beans-in-cache>
              1000
            </max-beans-in-cache>
            <idle-timeout-seconds>
              5
            </idle-timeout-seconds>
            <!-- JNDI name that is associated with this EJB;
                 used for lookup -->
            <jndi-name>
              oracle.OracleHome
            </jndi-name>
                    .
                    .
                    .
</weblogic-ejb-extensions>
```

Specifying Persistence Information

The WLE EJB container provides container-managed persistence. The code for implementing the persistence is generated by the `ejbc` command based on the deployment descriptors. The persistence store can be a flat file or it can be a database managed with a JDBC connection pool. For the EJB state to fully cooperate in a WLE global transaction, configure the EJB to use the JDBC-managed database store provided in WLE. Use file-based persistence only during development and prototyping.

The standard deployment descriptor created by the Bean Provider normally specifies:

- The fields in the EJB that are to be persistent, via the `cmp-field` element
- Information about the primary key

However, you, as the deployer, need to specify additional information for mapping an EJB to its persistent store via the WebLogic EJB extensions to the deployment descriptor DTD.

File-based Persistence

The following code shows the WebLogic EJB extensions to the deployment descriptor DTD for specifying file-based persistence:

```
<!--
Persistence store descriptor. Specifies what type of persistence store
EJB container should use to store state of bean.
```

```
-->
<!ELEMENT persistence-store-descriptor (description?,
(persistence-store-file |
persistence-store-jdbc)?)>
<!--
Persistence store using file. Bean is serialized to a file.
Mainly used to store state of Stateful Session Beans.
-->
<!ELEMENT persistence-store-file (description?,
persistence-store-directory-root
?)>
<!--
Root directory on File system for storing files per bean.
-->
<!ELEMENT persistence-store-directory-root (#PCDATA)>
```

The information supplied for the `persistence-store-directory-root` element is used by the EJB container to store all instances of the EJB, with the `ejb-name` element converted to a directory name.

Database-stored Persistence

The following code shows the WebLogic EJB extensions to the deployment descriptor DTD for specifying a JDBC connection for database-stored persistence:

```
<!--
Persistence store is any RDBMS. JDBC driver is used to talk to database.
Required for CMP.
-->
<!ELEMENT persistence-store-jdbc (description?, pool-name, user?, password?,
driver-url?, driver-class-name?, table-name, attribute-map,
finder-descriptor*)>

<!-- Required for CMP -->
<!ELEMENT pool-name (#PCDATA)>

<!-- Ignored in WebLogic Enterprise Server as this is part of connection
pool
setup at startup -->
<!ELEMENT user (#PCDATA)>

<!-- Ignored in WebLogic Enterprise Server as this is part of connection
pool
setup at startup -->
<!ELEMENT password (#PCDATA)>

<!-- Ignored in WebLogic Enterprise Server as this is part of connection
```

```
pool
setup at startup -->
<!ELEMENT driver-url (#PCDATA)>

<!-- Ignored in WebLogic Enterprise Server as this is part of connection
pool
setup at startup -->
<!ELEMENT driver-class-name (#PCDATA)>

<!-- Required for CMP -->
<!ELEMENT table-name (#PCDATA)>

<!-- Required for CMP -->
<!ELEMENT attribute-map (description?, attribute-map-entry+)>

<!-- Required for CMP -->
<!ELEMENT attribute-map-entry (bean-field-name, table-column-name)>

<!-- Required for CMP -->
<!ELEMENT bean-field-name (#PCDATA)>

<!-- Required for CMP -->
<!ELEMENT table-column-name (#PCDATA)>

<!-- Required for CMP -->
<!ELEMENT finder-descriptor (description?, method?, query-grammar?)>

<!-- Required for CMP -->
<!ELEMENT query-grammar (#PCDATA)>
```

The EJB instances are stored in a database that has been previously set up with a JDBC connection pool, which is identified by the `pool-name` element. The `table-name` and `attribute-map` elements map the EJB fields to the appropriate table columns in the database.

Finder descriptors are the WLE implementation of the EJB `find` methods. The `finder-descriptor` elements are pairs of method signatures and expressions. You specify a method signature in the `EJBHome` interface, and you specify the method's expression in the deployment descriptor via the `query-grammar` element. The finder methods return an enumeration of EJBs. For more information about specifying finder descriptors, see the *EJB XML Reference* in the WebLogic Enterprise online documentation.

Example

The following WebLogic EJB extensions to the deployment descriptor DTD fragment specify automatic saving using JDBC persistence of two fields of an entity bean (`accountId` and `salary`) to a database table (`emp`) using a connection pool (`pool1`), shown in boldface type:

```
<weblogic-ejb-extensions>
  <weblogic-version>
    WebLogic Enterprise Server 5.0
  </weblogic-version>
  <weblogic-enterprise-bean>
    <ejb-name>
      oracle
    </ejb-name>
    <weblogic-deployment-params>

      <max-beans-in-free-pool>
        20
      </max-beans-in-free-pool>
      <max-beans-in-cache>
        1000
      </max-beans-in-cache>
      <idle-timeout-seconds>
        5
      </idle-timeout-seconds>
     <!-- JNDI name that is associated with this EJB;used for lookup
-->
      <jndi-name>
        oracle.OracleHome
      </jndi-name>

      <!-- This is CMP EJB. Specify persistence information -->
      <persistence-store-descriptor>
        <persistence-store-jdbc>
          <!-- Pool name is looked up by the EJB source -->
          <pool-name>
            jdbc/pool1
          </pool-name>
          <!-- *** DATABASE INFORMATION SPECIFIC TO
                   INSTALLATION SITE *** -->
          <!-- Default user URL is for Oracle 8i-->
          <user>
            scott
          </user>

          <!-- Default password URL is for Oracle 8i-->
          <password>
```

```
      tiger
    </password>
    <!-- Default driver URL is for Oracle 8i,
         and default instance Beq-Local is used
    -->
    <driver-url>
      jdbc:weblogic:oracle:Beq-Local
    </driver-url>
    <table-name>
      emp
    </table-name>

    <!-- CMP Fields and database table column mapping-->
    <attribute-map>
      <attribute-map-entry>
        <bean-field-name>
          accountId
        </bean-field-name>
        <table-column-name>
          empno
        </table-column-name>
      </attribute-map-entry>
      <attribute-map-entry>
        <bean-field-name>
          salary
        </bean-field-name>
        <table-column-name>
          sal
        </table-column-name>
      </attribute-map-entry>
    </attribute-map>

    <!-- Finder Specifications -->
    <finder-descriptor>
      <method>
        <ejb-name>
        oracle
        </ejb-name>
        <method-name>
          findAccount(double salaryEqual)
        </method-name>
      </method>
      <query-grammar>
        (= salary $salaryEqual)
      </query-grammar>
    </finder-descriptor>
  </persistence-store-jdbc>
</persistence-store-descriptor>
</weblogic-deployment-params>
```

```
  </weblogic-enterprise-bean>
</weblogic-ejb-extensions>
```

For more information about JDBC connections, see *Using the JDBC Drivers* in the WebLogic Enterprise online documentation.

Step 4: Produce the deployable EJB JAR file.

In this step, you package the deployment descriptor, the compiled files for the EJB classes, the WLE extensions to the deployment descriptor DTD, and any additional required classes into a deployable EJB JAR file. You can do this using the `ejbc` command, as in the following example:

```
java com.beasys.ejb.utils.ejbc -validate -i DDfile -x WLEXfile archive-file
```

The `ejbc` command performs the following steps:

1. Parses the standard EJB deployment descriptor and WebLogic Enterprise extended deployment descriptor XML files, which are represented, respectively, as `DDfile` and `WLEXfile` in the preceding command.
2. Checks the deployment descriptors for semantic consistency, and writes any inconsistencies to standard output.
3. Generates the wrapper Java classes and compiles them. This is performed for each EJB in the deployment descriptor.
4. Packages the XML deployment descriptors and the generated class files into a deployable EJB JAR file. The command-line argument `archive-files` specifies the files that are archived into the EJB JAR file.

If you have multiple bean packages meant to be assembled as a deployable unit, the bean packages must be specified in a single deployment descriptor. For more information about the `ejbc` command, see the *Command Reference* in the WebLogic Enterprise online documentation.

Step 5: Configure the EJB application.

In the `SERVERS` section of the `UBBCONFIG` file, the administrator uses the `MODULES` keyword to identify the deployed EJB JAR files. Optionally, a related set of startup arguments can be specified for each EJB JAR file. For information about configuring the EJB container, configuring the WLE EJB server process, and specifying values in the `UBBCONFIG` file, see *Creating a Configuration File* in the WebLogic Enterprise online documentation.

Step 6: Specify the module initializer object in the WebLogic EJB extensions to the deployment descriptor DTD.

Server application startup and shutdown can be handled automatically by the EJB container. However, if you have special server initialization or shutdown requirements, you need to implement an application entity called the **module initializer object**.

The module initializer object implements operations that execute the following tasks:

- Performing basic module initialization (or EJB JAR file deployment) operations, which may include allocating resources needed by the EJB JAR file.
- Performing basic server application initialization operations, which may include registering homes or factories managed by the server application and allocating resources needed by the server application.
- Performing server process shutdown and cleanup procedures when the server application has finished servicing requests.

Note: For EJBs, the scope of the module initializer object is at the EJB JAR file level and not of the entire server application, as with the Server object and WLE CORBA applications.

You implement this module initializer object by creating a module initializer class that derives from `com.beasys.Tobj.Server` and by implementing the following two methods on that class:

- `initialize`

  The `initialize` method is invoked when the EJB JAR file loaded (generally when the WLE server process is booted).

- `release`

  The `release` method is invoked when the WLE server process is shut down or when the EJB JAR file is redeployed to another server process.

In the module initializer object application code, you can also write a public default constructor. You create the module initializer object class from scratch using a text editor.

If you have created a module initializer object, the EJB container parses the WebLogic EJB extensions to the deployment descriptor DTD in each deployed EJB JAR file (specified in the `UBBCONFIG` file) during startup.

The `module-initializer-class-name` element in the WebLogic EJB extensions to the deployment descriptor DTD identifies the module initializer object to be used at server initialization and shutdown. When the server process is booted, the EJB container instantiates this module initializer object and invoke its `initialize` method, passing in any startup arguments specified in the `UBBCONFIG` file. When the server process is shut down, the EJB container invokes the module initializer object's `release` method.

For information about the `com.beasys.Tobj.Server` base class, see the *WLE Javadoc* in the WebLogic Enterprise online documentation.

Scaling an EJB Application

For information about scaling an EJB application in the WLE environment, see the topic "Scaling Tasks for EJB Providers" in *Tuning and Scaling Applications* in the WebLogic Enterprise online documentation.

For More Information

For complete details on administering an EJB application, see the following in the WebLogic Enterprise online documentation:

- *Creating a Configuration File*
- *Starting and Stopping a WLE Application*

Index

A

B

C

D

E

F

G

H

T

U

W