BEA WebLogic Collaborate

Developer Guide

BEA WebLogic Collaborate 1.0.1
Document Edition 1.0.1
March 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA WebLogic Collaborate Developer Guide

Document Edition	Date	Software Version
1.0.1	March 2001	1.0.1

Contents

3. Using XOCP C-Enabler Applications to Exchange Business Messages

4. Developing Logic Plug-Ins

5. Developing Management Applications

6. Writing to the Log

Index

About This Document

This document describes how to develop applications to exchange business messages and monitor run-time activities in c-hubs and c-enablers in the BEA WebLogic Collaborate™ system.

This document is organized as follows:

- Chapter 1, “Introduction,” provides an introduction to developing applications for the BEA WebLogic Collaborate environment.
- Chapter 2, “Using Workflows to Exchange Business Messages,” describes how to exchange business messages using WebLogic Process Integrator workflows.
- Chapter 3, “Using XOCP C-Enabler Applications to Exchange Business Messages,” describes how to exchange business messages using c-enabler applications that implement the c-enabler API.
- Chapter 4, “Developing Logic Plug-Ins,” describes how to manipulate business messages as they travel through the c-hub.
- Chapter 5, “Developing Management Applications,” describes how to monitor run-time activities in the c-hub and c-enabler by developing management applications that implement the BEA WebLogic Collaborate Managed Beans (MBeans).
- Chapter 6, “Writing to the Log,” describes how to write messages to the log in any BEA WebLogic Collaborate application.

What You Need to Know

This document is intended primarily for:

- Business process designers who will use WebLogic Process Integrator studio to design workflows that integrate with the BEA WebLogic Collaborate environment.
- Application developers who will write Java applications that manage the exchange of business messages or monitor run-time statistics in the BEA WebLogic Collaborate environment.
- System administrators who will set up and administer BEA WebLogic Collaborate applications.

For an overview of the BEA WebLogic Collaborate architecture, see Overview in the *BEA WebLogic Collaborate Getting Started* document.

How to Print this 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 BEA WebLogic Collaborate documentation CD. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

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

Related Information

For more information about Java 2 Enterprise Edition (J2EE), Extended Markup Language (XML), and Java programming, see the *Bibliography* in the BEA WebLogic Collaborate online documentation.

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Collaborate 1.0 release.

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 filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

Convention	Item
{ }	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.

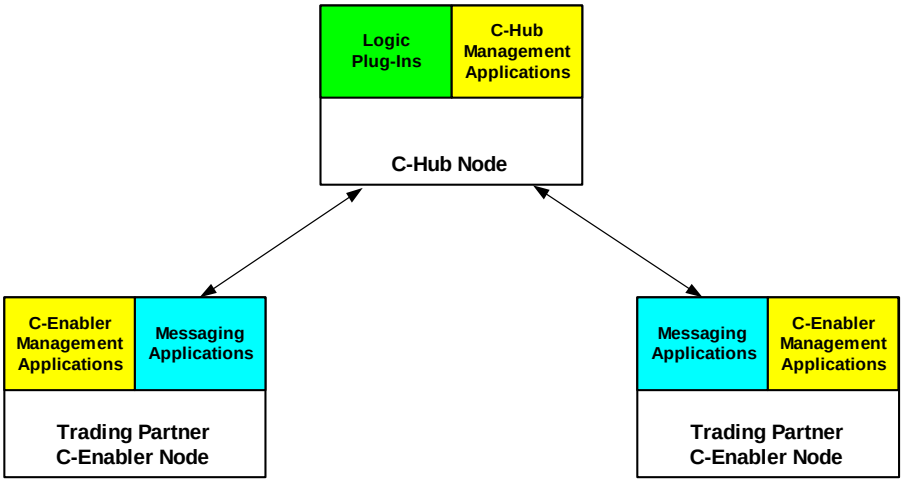
1 Introduction

The following sections introduce the different kinds of WebLogic Collaborate applications:

- Messaging Applications
- Management Applications
- Logic Plug-Ins

The following figure shows where these types of applications reside in the WebLogic Collaborate system.

Figure 1-1 Types of WebLogic Collaborate Applications



For an introduction to the WebLogic Collaborate system, see Overview in *BEA WebLogic Collaborate Getting Started*.

Messaging Applications

WebLogic Collaborate messaging applications handle the exchange of business messages among trading partners in a c-space. Messaging applications run on c-enabler nodes and exchange business messages via the c-hub. Developers can implement messaging applications in two different ways.

Table 1-1 Different Implementations of Messaging Applications

Implementation Option	Description
Workflow applications	Use WebLogic Process Integrator workflows to exchange business messages with other trading partners. The WLPI Verifier application (in `\examples\wlpiverifier`) is an example of a workflow application.
C-Enabler applications	Use the flexible and powerful c-enabler API to exchange business messages with other trading partners. The Verifier application (in `\examples\verifier`) is an example of a c-enabler application.

For any given messaging application, the best implementation approach depends on the particular needs of that application. Workflow applications provide design and run-time tools to expedite application development. C-Enabler applications provide greater programmatic control using Java APIs. Developers need to determine the best approach based on the relative advantages of each in relation to their specific application requirements.

A conversation usually has multiple, interoperating messaging applications, each tailored to a particular role in that conversation. For example, the trading partner who initiates the conversation uses an *initiating* messaging application, and trading partners who participate in the conversation use a different *participating* messaging application that interacts with the initiating messaging application at run time.

For more information about messaging applications, see Chapter 2, "Using Workflows to Exchange Business Messages," and Chapter 3, "Using XOCP C-Enabler Applications to Exchange Business Messages."

Management Applications

WebLogic Collaborate management applications monitor run-time activities, such as message traffic and conversation statistics, on c-hub and c-enabler nodes. WebLogic Collaborate provides two administrative tools, the C-Hub Administration Console and the C-Enabler Administration Console, that monitor run-time. In addition to these system tools, developers can create custom management applications that provide comparable monitoring functionality.

Developers can implement two kinds of management applications.

Table 1-2 Types of Managing Applications

Component	Description
C-Hub management applications	Monitor activities on the c-hub node and provide run-time statistics for c-hubs, c-spaces, conversation definitions, trading partners, and business messages.
C-Enabler management applications	Monitor activities on c-enabler nodes and provide run-time statistics for c-enablers, c-enabler sessions, conversations, and business messages.

For c-hub and c-enabler management applications, WebLogic Collaborate provides a set of Managed Beans, or *MBeans*, which are special JavaBeans with attributes and methods for management operations. These MBeans are BEA implementations of the Java Management Extensions (JMX) Managed Beans API, which is defined in the Java Management Extensions Specification published by Sun Microsystems, Inc.

For more information about WebLogic Collaborate management applications, see Chapter 5, “Developing Management Applications.”

Logic Plug-Ins

Logic plug-ins are Java classes that perform specialized processing of business messages as they pass through the c-hub. Logic plug-ins insert rules and business logic at strategic locations along the path that business messages travel as they make their way through the c-hub. WebLogic Collaborate provides XOCP and RosettaNet router and filter logic plug-ins. A c-hub provider or trading partner can develop and install custom logic plug-ins on the c-hub to provide additional value for c-hub management and for trading partners who use that c-hub.

Logic plug-ins are stored and executed on the c-hub node and are defined in the c-hub repository. Logic plug-ins are transparent to c-enabler users.

For more information about logic plug-ins, see Chapter 4, “Developing Logic Plug-Ins.”

CHAPTER

2 Using Workflows to Exchange Business Messages

You can use WebLogic Process Integrator workflows to exchange XOCP business messages in the WebLogic Collaborate environment. WebLogic Process Integrator accelerates application development by providing a visual design tool for designing workflows (process models); a run-time Process Engine for executing workflows; and process monitoring capabilities. Using WebLogic Process Integrator in the WebLogic Collaborate environment involves a combination of design, programming, and administrative tasks.

The following sections describe how to exchange business messages in WebLogic Collaborate by using WebLogic Process Integrator workflows:

- About Using Workflows
- Designing Workflows for Exchanging Business Messages
- Working with Business Messages
- Developing Applications That Start Conversation Initiator Workflows

The WebLogic Process Integrator Verifier program provides an example of using WebLogic Process Integrator workflows to exchange business messages in WebLogic Collaborate. For more information, see Running the WebLogic Process Integrator Verifier Example in *BEA WebLogic Collaborate Getting Started*.

About Using Workflows

The following sections describe key concepts for using WebLogic Process Integrator workflows in WebLogic Collaborate applications:

- About This WebLogic Process Integrator Version
- Architectural Overview
- Key Concepts
- Run-Time Prerequisites
- Summary of Workflow Integration Tasks

About This WebLogic Process Integrator Version

The version of WebLogic Process Integrator that is bundled with WebLogic Collaborate provides all of the functionality of WebLogic Process Integrator version 1.2, which ships separately. It also provides additional functionality for integrating with the WebLogic Collaborate environment, including:

- Specialized workflow properties for manipulating business messages, specifying the message delivery Quality of Service, handling message tokens, and so on
- A Java application programming interface (API) for use in WebLogic Collaborate workflow applications

For more information about the WebLogic Process Integrator application, see the following documents:

- *BEA WebLogic Process Integrator Studio User Guide*
- *BEA WebLogic Process Integrator Tutorial*
- *BEA WebLogic Process Integrator Worklist Guide*

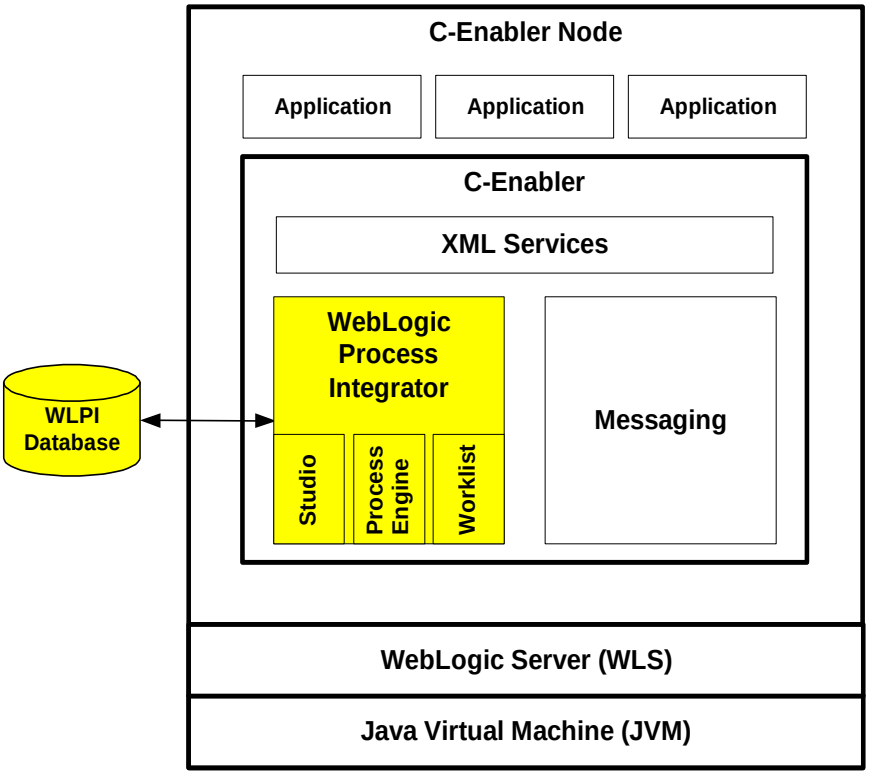
Architectural Overview

This section describes how WebLogic Process Integrator integrates with the WebLogic Collaborate architecture.

Architecture Diagram

The following figure shows the WebLogic Collaborate c-enabler architecture with WebLogic Process Integrator components.

Figure 2-1 C-Enabler Architecture



WebLogic Process Integrator is started automatically upon c-enabler startup.

WebLogic Process Integrator Components in WebLogic Collaborate

WebLogic Collaborate provides the following WebLogic Process Integrator components:

Table 2-1 WebLogic Process Integrator Components

Component	Description
WebLogic Process Integrator Studio	Client application that is used at design time to define workflows and at run time to monitor running workflows.
WebLogic Process Integrator Process Engine	Run-time controller and workflow engine that executes and manages workflows and tracks workflow instances.
WebLogic Process Integrator Database	Database in which workflow templates are stored. This database can reside locally on the c-enabler node, or it can reside on a different node that is network accessible to the c-enabler. The database can be deployed so that it is accessible to a single c-enabler only, to multiple c-enablers within the same trading partner organization, or to multiple c-enablers across trading partners in different organizations.
WebLogic Process Integrator Worklist	Application that is used to view and perform tasks that are currently assigned to a user or to roles to which the user belongs, such as reassigning tasks to other users, marking tasks as done, unmarking tasks done, viewing a workflow status, manually starting a workflow, and so on.

For an introduction to these WebLogic Process Integrator components, see WebLogic Process Integrator Overview in the *BEA WebLogic Process Integrator Studio User Guide*.

Key Concepts

This section describes key concepts that you need to understand before using WebLogic Process Integrator workflows in WebLogic Collaborate applications.

Workflows, Workflow Templates, and Workflow Template Definitions

This section describes the following key WebLogic Process Integrator concepts:

- WebLogic Process Integrator is a workflow automation tool. A *workflow* is a business process. Workflow automation is the automation of a business process, in whole or in part, during which information of any type is passed to the right participant at the right time according to a set of intelligent business rules that allow computers to perform most of the work while humans only have to deal with exceptions.
- In WebLogic Process Integrator, a *workflow template* is a folder (or a container) in the WebLogic Process Integrator Studio. This workflow template represents a workflow and is given a meaningful workflow name, such as Order Processing or Billing. The workflow template aggregates various definitions (or "versions") of its implementation; these are referred to as workflow template definitions. Further, a workflow template is responsible for controlling which organizations can use the "contained" workflow template definitions.
- A *workflow template definition* is a definition (or "version") of the workflow, distinguished by its Effective and Expiry dates. At design time, you use the WebLogic Process Integrator Studio to link a workflow template definition to a particular conversation and WebLogic Collaborate role (such as *buyer* or *seller*) in a WebLogic Collaborate conversation definition. At run time, WebLogic Process Integrator starts an instance (or session) of a workflow template definition, selecting the most effective (or current and active) definition.

For detailed information about these concepts, see WebLogic Process Integrator Overview in the *BEA WebLogic Process Integrator Studio User Guide*.

Conversations, Conversation Definitions, and Business Messages

This section defines the following key WebLogic Collaborate concepts:

- In WebLogic Collaborate, a *conversation* is a series of message exchanges between trading partners that takes place in a collaboration space and that is predefined according to a conversation definition. Each message in the conversation can cause any number of back-end transactions.
- A *conversation definition* consists of a unique conversation name, conversation version, document definitions, trading partner IDs, and trading partner roles for one conversation. At design time, you use the WebLogic Process Integrator Studio to link a workflow template definition to a particular role (such as *buyer* or *seller*) in a WebLogic Collaborate conversation definition.
- An *XOCP business message* is the basic unit of communication exchanged between trading partners in an XOCP conversation. An XOCP business message is a multi-part MIME message that consists of business documents and attachments. A *business document* is the XML-based payload part of a business message. An *attachment* is the non-XML-based payload part of a business message. To construct outgoing business messages or to process incoming business messages, a workflow uses the Manipulate Business Message action, which invokes a Java class that implements the `com.bea.b2b.wlpi.MessageManipulator` interface.

For detailed information about these concepts, see Overview in *BEA WebLogic Collaborate Getting Started*.

Initiators and Participants

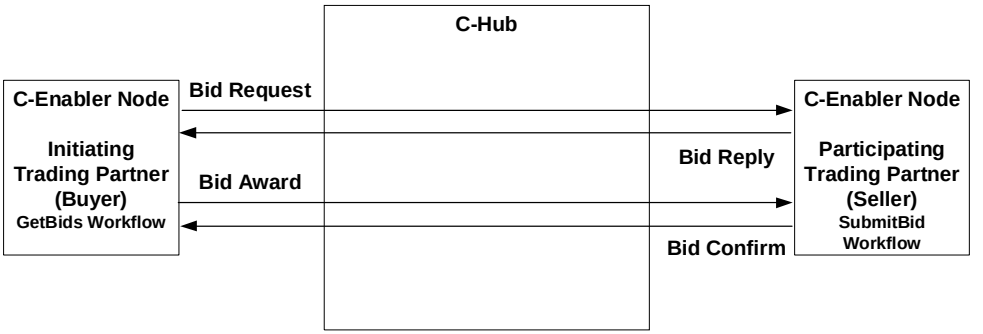
A conversation involves an *initiator* who starts the conversation and *participants* who participate in the conversation once it has started. Each perspective requires a different kind of workflow.

Table 2-2 Types of Workflows

Workflow Type	Description
Conversation initiator workflow	Defined to have conversation properties and a non-Business Message start property. This type of workflow initiates and terminates the conversation.
Conversation participant workflow	Defined to have conversation properties and a Business Message start property. This type of workflow can join and exit the conversation but cannot initiate or terminate it.

In the context of a business process, these two types of workflows are interlocking. For example, suppose a buyer wanted to obtain bids from various sellers. This business process could be described as follows.

Figure 2-2 Sample Business Process with Two Workflows



1. In WebLogic Process Integrator Studio, the buyer (the initiating trading partner) starts a workflow named `GetBids` (the conversation initiator workflow). The `GetBids` workflow constructs and sends a business message (containing a bid request in the form of an XML document) by way of the c-hub to qualified sellers and awaits a reply.

Note: The `GetBids` workflow is defined with conversation properties and a Manual start property. It is started programmatically by using a Java application.

2. Each qualified seller (a participant trading partner) receives the business message, which triggers the start of an instance of a workflow named `SubmitBid` (the conversation participant workflow) on each seller's c-enabler node.

   The `SubmitBid` workflow processes the incoming bid request, determines whether to submit a bid or not and, if so, constructs and sends a business message (containing a bid reply in the form of an XML document), and awaits the results of the bid selection.

   Note: The `SubmitBid` workflow is defined with conversation properties and a Message Start property.

3. On the buyer side, the `GetBids` workflow receives bid replies from all qualified sellers, determines which seller to award the bid, and then notifies all sellers of the results by:

   - Constructing and sending a business message (containing a bid award in the form of an XML document) to the winning bidder
   - Constructing and sending a different business message (containing a bid rejection in the form of an XML document) to all other sellers

   The `GetBids` workflow then awaits a bid confirmation from the winning seller.

4. On the seller side, the `SubmitBid` workflow receives and processes the results of the bid.

   - If the seller was awarded the winning bid, the seller constructs and sends a business message (containing a bid confirmation in the form of an XML document) to the seller.

     Alternatively, the seller could return a different business message (containing another XML document, such as a purchase order request) to the seller that would continue the conversation.

   - If the seller did not receive the bid award, that seller's `SubmitBid` workflow ends and the seller exits the conversation.

5. On the buyer side, the `GetBids` workflow receives and processes the bid confirmation from the seller and then terminates the conversation.

Sent and Received Business Messages

When trading partners exchange business messages, initiator and participant workflows both typically send and receive business messages.

It is important to keep in mind which parts of the workflow send business messages and which parts receive them. For example, a buyer might submit a bid request (a business message) to a seller. In this case, the buyer workflow is sending the business message and the seller workflow is receiving it. When the seller replies to the request with a bid (another business message), then the roles are reversed: the seller workflow is the sender and the buyer is the recipient workflow.

The design and programming tasks differ for sending and receiving business messages. However, in both cases, you must define certain properties in the workflow template definition and write application code (that implements the `com.bea.b2b.wlpi.MessageManipulator` interface) to manipulate the business message.

For more information about working with business messages, see "Working with Business Messages" on page 2-41.

Run-Time Prerequisites

Before exchanging messages at run time, the following prerequisites must be met:

- Install and configure WebLogic Process Integrator and WebLogic Collaborate, as described in "Administrative Tasks" on page 2-10.
- Define and link workflows to WebLogic Collaborate conversations, as described in "Design Tasks" on page 2-11.
- Write and test application code for manipulating messages and for starting the conversation initiator workflow, as described in "Programming Tasks" on page 2-13.
- For all trading partners, WebLogic Process Integrator is automatically loaded and running upon c-enabler startup.
- For each trading partner, all relevant workflows are active and stored in the WebLogic Process Integrator database.

- For an initiating trading partner, the conversation initiator workflow is active and defined with the non-Business Message start property. It awaits the invocation of the application that starts the workflow.
- For all participating trading partners, the conversation participant workflow is active and defined with a Business Message start property. It awaits the receipt of the initial business message in the conversation.

Summary of Workflow Integration Tasks

Using WebLogic Process Integrator workflows to exchange business messages in WebLogic Collaborate requires a combination of administrative, design, and programming tasks.

Administrative Tasks

Integrating WebLogic Process Integrator workflows requires the following administrative tasks:

1. Install WebLogic Collaborate, and configure WebLogic Process Integrator according to the instructions in Setting Up the WebLogic Process Integrator Environment in *BEA WebLogic Collaborate Getting Started*.
   - Create the tables for the WebLogic Process Integrator database by running the appropriate SQL script for the database management you will use.
   - Configure the JDBC connection pool for the WebLogic Process Integrator database in the using the Administrative Console.
   - Run the WebLogic Process Integrator Verifier program to validate your installation and configuration.
2. For c-enabler nodes, specify the workflow c-enabler session names you want to use in the c-enabler XML configuration file. For more information, see Configuring C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide*.

3. Using the C-Hub Administration Console, create and configure the necessary entities in the c-hub repository, including c-spaces, trading partners, conversation definitions, document definitions, and so on. For more information, see the *BEA WebLogic Collaborate C-Hub Administration Guide*.

   Note: Every WebLogic Process Integrator workflow template definition requires a conversation definition.

4. Using WebLogic Process Integrator Studio, specify the organizations, users, and roles in the WebLogic Process Integrator database, as described in Administering Data within WebLogic Process Integrator in the *BEA WebLogic Process Integrator Studio User Guide*.

Design Tasks

Integrating WebLogic Process Integrator workflows requires the following design tasks that you perform in WebLogic Process Integrator Studio:

1. Create and design workflows that automate business processes.

   You can create workflows from scratch, as described in Defining and Maintaining Workflows in the *BEA WebLogic Process Integrator Studio User Guide*.

   Alternatively, you can import workflows created in other versions of WebLogic Process Integrator, as described in "Using Workflow Templates Created in Other WebLogic Process Integrator Versions" on page 2-15.

   In addition to defining the standard workflow properties, you must also define properties that link the workflow to the WebLogic Collaborate messaging system. The remaining tasks in this sequence apply to integrating workflows into WebLogic Collaborate.

2. For each workflow template definition, specify conversation properties as follows:

   - Explicitly link the workflow template definition to a role in a conversation definition in the c-hub repository, as described in "Linking Workflows to Conversations" on page 2-19.
   - Associate at least one c-enabler session name to the workflow template definition, as described in "Linking C-Enabler Session Names to a Workflow Template Definition" on page 2-24.

- Optionally, specify other conversation properties, as described in "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20.

3. For each workflow template definition, define start actions depending on the type of workflow.
   - For conversation initiator workflows, define a non-Business Message start property, as described in "Defining the Start for a Conversation Initiator Workflow" on page 2-27.
   - For conversation participant workflows, define a Business Message start property, as described in "Defining the Start for a Conversation Participant Workflow" on page 2-29.
4. For each workflow template definition, define how the workflow will end.
   - For conversation initiator workflows, add a done shape and define its properties, as described in "Defining the Termination of Conversation Initiator Workflows" on page 2-31.
   - For conversation participant workflows, optionally define a Conversation Terminate Event property and status, as described in "Defining the End of Conversation Participant Workflows" on page 2-33.
5. For any input or output variables used in the workflow, define them in the workflow template definition, as described in "Defining WebLogic Process Integrator Variables for Workflows" on page 2-35.
6. For each workflow template definition, define how business messages are processed and exchanged.
   - For all workflows, define the WebLogic Process Integrator variables that are used to store business messages, as described in "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.
   - For all workflows, define business operations that manipulate business messages, either creating the business messages to send or processing business messages that are received, as described in "Defining Manipulate Business Message Actions" on page 2-44.
   - For workflows that send business messages, define the Send Business Message action, as described in "Defining Send Business Message Actions" on page 2-57. In addition, you add an associated Manipulate Business Message action to create the business message to send.

You can also assign information from the message token that is returned from a sent message to WebLogic Process Integrator variables, as described in "Assigning Message Token Information to WebLogic Process Integrator Variables" on page 2-63.

- For conversation participant workflows, define the Start node as a Business Message start so that the workflow is started upon receipt of the initial business message from the conversation initiator workflow, as described in "Defining the Business Message Start for Conversation Participant Workflows" on page 2-67. In addition, you add an associated Manipulate Business Message action to process the incoming business message.
- For non-initial business messages received by conversation initiator or conversation participant workflows, define a Business Message Receive event, as described in "Defining Business Message Receive Events" on page 2-70. In addition, you add an associated Manipulate Business Message action to process the incoming business message.

For comprehensive information about workflow design tasks, see "Designing Workflows for Exchanging Business Messages" on page 2-14.

Note: You can run WebLogic Process Integrator workflows in the WebLogic Collaborate environment even if they are not integrated with WebLogic Collaborate features. For example, you can run workflows created in WebLogic Process Integrator version 1.2 (shipped separately from WebLogic Collaborate) without specifically adapting them to integrate with WebLogic Collaborate.

Programming Tasks

Programming tasks depend on the specific needs of each application that makes use of a workflow. The following tasks are required:

- For conversation initiator workflows, write an application that performs the following tasks: creates a workflow c-enabler session; constructs a business message; starts the workflow; sends a business message; and, optionally, awaits a reply. This application can do other tasks, but it must at least perform these tasks. For more information, see "Developing Applications That Start Conversation Initiator Workflows" on page 2-76.
- For both conversation initiator workflows and conversation participant workflows, write application code that manipulates the business messages that a

workflow sends and receives. This code constructs a business message before it is sent and processes a business message that has been received. This code is a class that implements the `com.bea.b2b.wlpi.MessageManipulator` interface. For more information, see "Writing Business Operations to Manipulate Business Messages" on page 2-51.

Designing Workflows for Exchanging Business Messages

To use workflows to exchange business messages in WebLogic Collaborate, design workflow template definitions by using WebLogic Process Integrator Studio. In addition to the standard properties described in Defining and Maintaining Workflows in the *BEA WebLogic Process Integrator Studio User Guide*, you must define additional workflow properties, not described in that document, that allows the workflow to be used in the WebLogic Collaborate environment.

For example, you link a workflow template definition to a particular role in a conversation definition in the c-hub repository. You also define some additional attributes, including the message delivery Quality of Service, message token handling, and conversation termination.

The following sections describe how to design workflows to exchange business messages in the WebLogic Collaborate environment:

- Using Workflow Templates Created in Other WebLogic Process Integrator Versions
- Defining Conversation Properties
- Defining Start Actions
- Defining Conversation Termination

Using Workflow Templates Created in Other WebLogic Process Integrator Versions

The version of WebLogic Process Integrator that ships with WebLogic Collaborate is designed to work seamlessly within the WebLogic Collaborate environment. If you have workflows that were designed in other versions of WebLogic Process Integrator, you can use still these workflows in WebLogic Collaborate, but you must complete the following additional tasks to adapt them for use in WebLogic Collaborate:

1. Export the associated workflow template definition from the earlier version of WebLogic Process Integrator, as described in "Exporting Workflow Template Definitions" on page 2-15.
2. Import the workflow template definition that you previously exported into the WebLogic Process Integrator that ships with WebLogic Collaborate, as described in "Importing Workflow Template Definitions" on page 2-16.
3. Modify the workflow template design to work with WebLogic Collaborate, as described in the section that begins with "Designing Workflows for Exchanging Business Messages" on page 2-14.

Note: Standalone versions of WebLogic Process Integrator cannot use workflows that were created or modified using the version of WebLogic Process Integrator Studio that ships with WebLogic Collaborate.

Exporting Workflow Template Definitions

To export a workflow template definition from a WebLogic Process Integrator version that shipped *separately from* WebLogic Collaborate:

1. In the folder tree, right-click the workflow template definition you want to export.
2. Choose Export from the pop-up menu.
3. In the Save dialog box, select the location (drive and directory) where you want to save the exported workflow template definition.
4. Specify the file name of the exported workflow template definition.
5. Click Save.

Importing Workflow Template Definitions

To import a previously exported workflow template definition (see "Exporting Workflow Template Definitions" on page 2-15) into the version of WebLogic Process Integrator that ships with WebLogic Collaborate:

1. In the WebLogic Process Integrator folder tree, right-click the workflow template into which you will import the workflow template definition.
2. From the pop-up menu, choose Import Template Definition.
3. In the Save dialog box, select the current location (drive and directory) of the exported workflow template definition file that you want to import.
4. Click Save.
5. After the file is read, an import confirmation dialog box appears. Click Yes to confirm importing the workflow template definition.
6. Imported workflow template definitions are always marked "inactive." Before an imported workflow template definition can be instantiated, you must change its definition to "active" in the Template Definition dialog box. For more information, see Defining and Maintaining Workflows in the *BEA WebLogic Process Integrator Studio User Guide*.

Defining Conversation Properties

Before you use a WebLogic Process Integrator workflow to exchange business messages in WebLogic Collaborate, you define certain conversation properties that are specific to the WebLogic Collaborate environment. For detailed information about defining workflow templates, see Defining and Maintaining Workflows in the *BEA WebLogic Process Integrator Studio User Guide*.

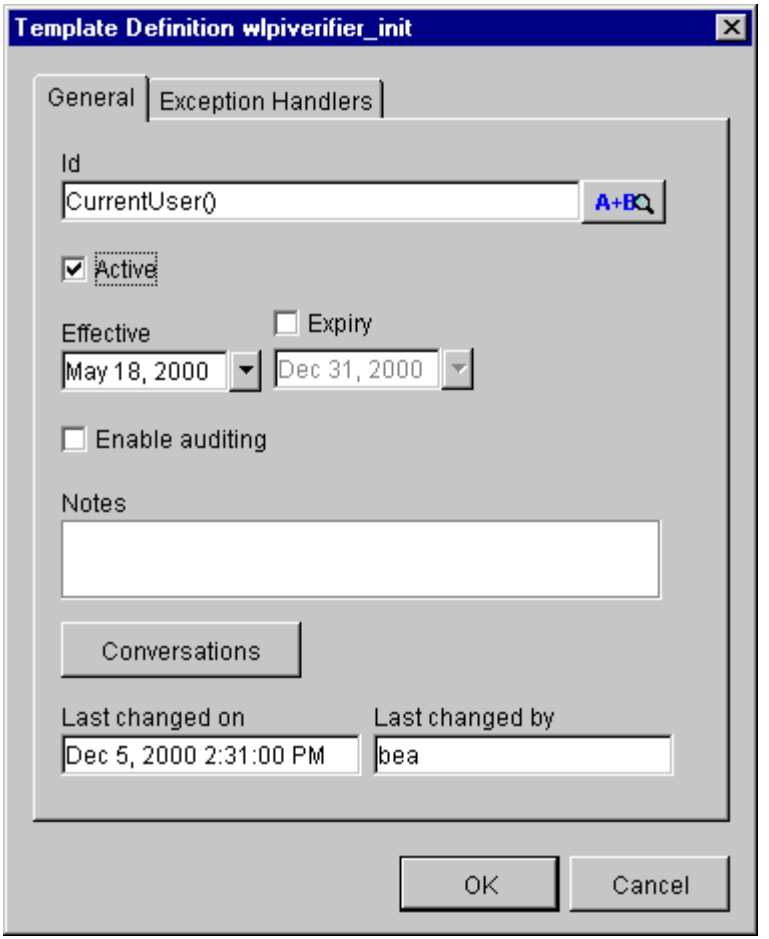
Opening Workflow Template Definitions

To define a workflow template definition in WebLogic Process Integrator Studio:

1. Do one of the following:
   - To create a new template definition, right-click the template that will contain the new template definition in the folder tree, and choose New Template Definition from the pop-up menu.
   - To open an existing template definition, right-click the template definition in the folder tree and choose Open from the pop-up menu.

2. Right-click the template definition and choose Properties from the pop-up menu to display the Template Definition dialog box.

Figure 2-3 Template Definition Dialog Box



3. Complete the fields in the Template Definition dialog box, as described in Defining and Maintaining Workflows in the *BEA WebLogic Process Integrator Studio User Guide*.
4. To define conversation properties, click Conversations.
5. Click OK to save your changes.

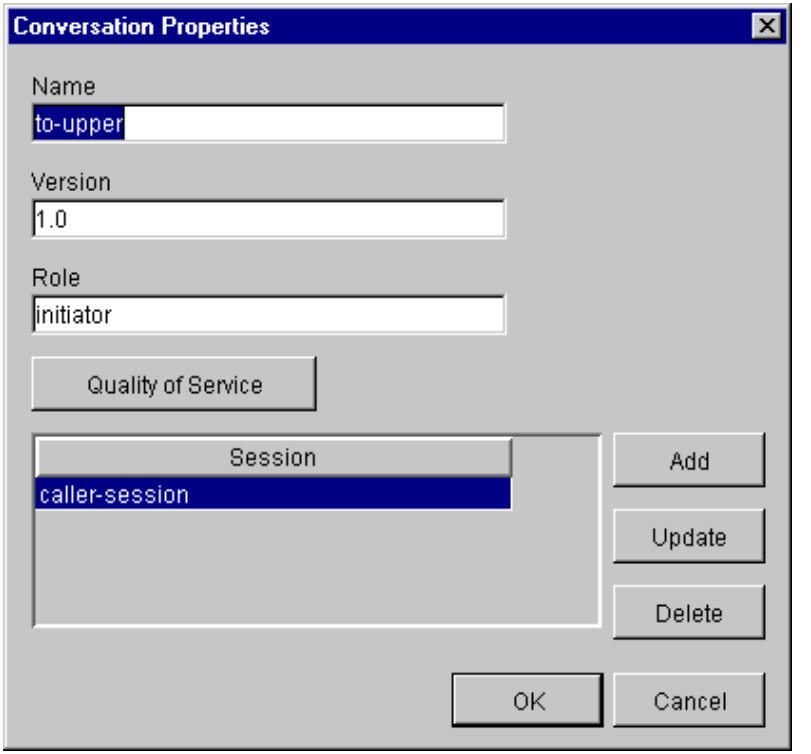
Linking Workflows to Conversations

Before you use a WebLogic Process Integrator workflow to exchange business messages in WebLogic Collaborate, you first link the workflow template definition in WebLogic Process Integrator with a particular conversation type (a conversation name, version, and WebLogic Collaborate role) in the WebLogic Collaborate c-hub repository.

To link a workflow template definition with conversation type:

1. Open the Template Definition dialog box, as described in "Opening Workflow Template Definitions" on page 2-17.
2. In the Template Definition dialog box, click the Conversations button to display the Conversation Properties dialog box.

Figure 2-4 Conversation Properties Dialog Box



3. Complete the following fields in the Conversation Properties dialog box.

Table 2-3 Fields in the Conversation Properties Dialog Box

Field	Description
Name	Name of the WebLogic Collaborate conversation definition in the c-hub repository to link with this workflow template definition.
Version	Version number of the conversation definition in the c-hub repository to link with this workflow template definition.
Role	Role in the conversation definition to link with this workflow template definition. In order for a trading partner to receive messages in this conversation, it must be registered in this role in the conversation at run time.
Quality of Service	Message delivery quality of service, as described in "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20.
Session	C-enabler session name(s) for which this workflow template should be used, as described in "Linking C-Enabler Session Names to a Workflow Template Definition" on page 2-24.

4. Click OK to save your changes.

Defining the Quality of Service for Message Delivery at the Template Level

The Quality of Service (QoS) is a set of attributes that are defined for reliable business message publishing. In WebLogic Process Integrator, you can define the QoS at the following levels:

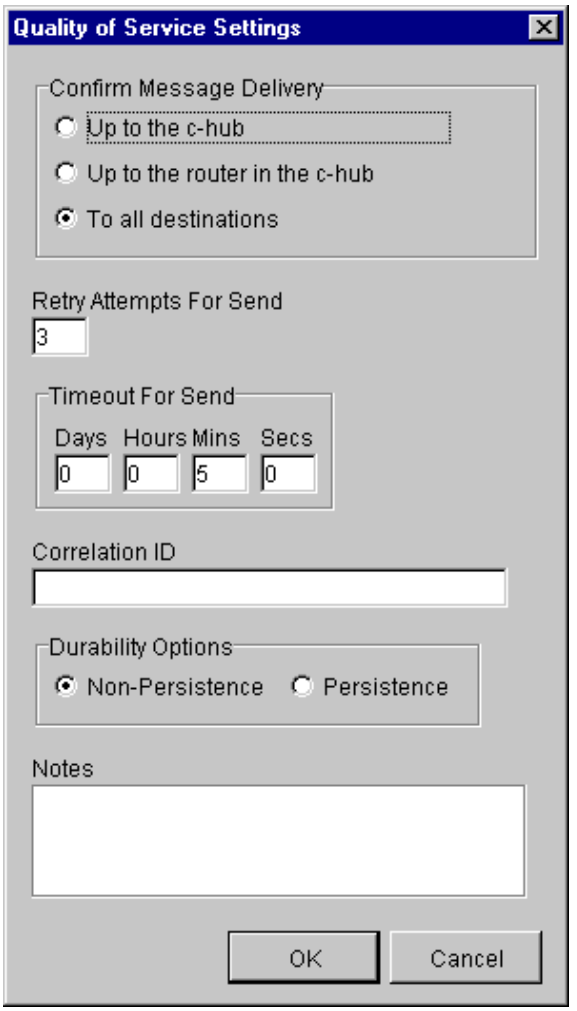
- At the workflow template definition level, where the settings apply to all Send Business Message actions, unless it is specifically overridden by the definition of the action.
- At the Send Business Message action level, where the settings apply to the specific action only but override the settings specified at the template level. For

more information, see "Defining the Quality of Service for Message Delivery for a Send Business Message Action" on page 2-62.

To specify the Quality of Service at the workflow template definition level:

1. Open the Conversation Properties dialog box for a workflow template definition, as described in "Linking Workflows to Conversations" on page 2-19.
2. In the Conversation Properties dialog box, click the Quality of Service button to display the Quality of Service Settings dialog box.

Figure 2-5 Quality of Service Settings Dialog Box



3. Complete the following fields in the Quality of Service Settings dialog box.

Table 2-4 Fields in the Quality of Service Dialog Box

Field	Description
Confirm Message Delivery	Degree to which message delivery confirmation is required: up to the c-hub (the default), up to the router in the c-hub, or to all destinations. Your selection determines which options are available in the Message Token Assignments dialog box, as described in "Assigning Message Token Information to WebLogic Process Integrator Variables" on page 2-63.
■ Up to the c-hub	Delivery confirmation is required when a message reaches the c-hub (default). Select this option to provide basic delivery confirmation with maximum run-time performance.
■ Up to the router in the c-hub	Delivery confirmation is required when a message reaches the router in the c-hub. This option provides the list of trading partners selected by the c-hub router to receive the message.
■ To all destinations	Delivery confirmation is required from all destinations. Select this option to provide the maximum delivery confirmation details. May affect run-time performance.
Retry Attempts for Send	Maximum number of retries for sending a message (default is 0). The WebLogic Process Integrator Process Engine will repeatedly attempt to send a message until it either successfully sends the message or it exceeds the maximum number of retries. A WebLogic Process Integrator exception will be thrown if the maximum retries are exceeded.
Timeout for Send	Timeout value for sending a message (default is 0, which means no timeout). The WebLogic Process Integrator Process Engine will wait until either a delivery confirmation is received or the timeout period has been exceeded.
Correlation ID	Message identification string that can be used to correlate the message with other business messages in the application (default is none). For example, a trading partner might want to specify a correlation ID in a request so that replies to that request can be matched to the original request. The WLC messaging system includes this property with the message.

Table 2-4 Fields in the Quality of Service Dialog Box (Continued)

Field	Description
Durability Options	Durability options for messaging: Persistence or Non-persistence (default). *Overrides* the default setting (if specified) for the associated conversation definition in the c-hub repository.
Non-Persistence	Messages are not to be saved in a persistent state. This option improves run-time performance but will reduce the likelihood of recovery from a system failure.
Persistence	Messages are to be saved in a persistent state. This option increases the likelihood of recovery from a system failure but requires additional processing that might affect run-time performance.
Notes	Optional descriptive text.

4. Click OK to save your settings.

Linking C-Enabler Session Names to a Workflow Template Definition

You can associate a workflow template definition with one or more workflow c-enabler session names. At a minimum, you must link at least one c-enabler session name. The WebLogic Process Integrator Studio allows you to add, update, and delete linked session names.

A workflow template definition can have more than one defined session name. This allows the same workflow template definition (different instances) to be used by different c-enablers in the same conversation. For example, a workflow template definition could be defined with three different session names (such as `sessionA`, `sessionB`, and `sessionC`). Each trading partner in the conversation can then use the appropriate session when providing their own implementation (manipulate message) to process the request.

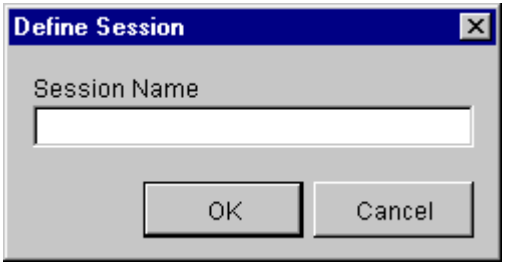
Session names are defined in the c-enabler XML configuration file. Each session name refers to one session entry in a c-enabler configuration file (which is known only at run time). Each session entry in the c-enabler XML configuration file refers to a specific c-hub, c-space, and trading partner.

Adding Sessions

To add a session:

1. In the Conversation Properties dialog box, click Add to display the Define Session dialog box.

Figure 2-6 Define Session dialog box



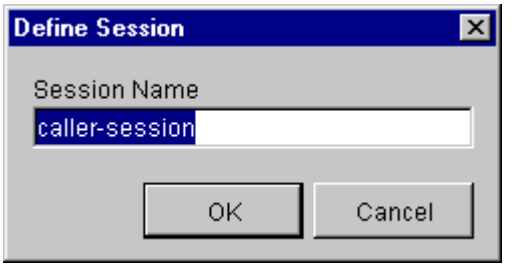
2. Enter a unique session name.
3. Click OK to save your changes.

Updating Sessions

To update a session:

1. In the Conversation Properties dialog box, select the session you want to update and then click Update to display the Define Session dialog box.

Figure 2-7 Define Session dialog box



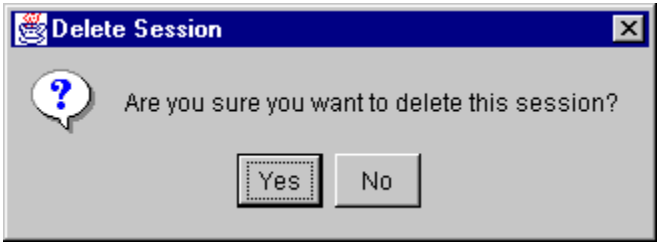
2. Edit the session name. It must be a unique session name.
3. Click OK to save your changes.

Deleting Sessions

To delete a session:

1. In the Conversation Properties dialog box, select the session you want to delete and then click Delete to display the Delete Session dialog box.

Figure 2-8 Delete Session dialog box



2. Click Yes.

Defining Start Actions

You define a start action based on the type of workflow and according to the following rules:

- To instantiate a workflow, the workflow template definition must be active and not expired.
- For a conversation initiator workflow that is started programmatically, you specify a Manual start property.
- For a conversation initiator workflow that is *not* started programmatically, you can specify any start property (Event, Timed, Manual, or Called) *except* the Business Message start state.
- For a conversation participant workflow that is started upon receiving a business message, you define a Business Message start state.

Defining the Start for a Conversation Initiator Workflow

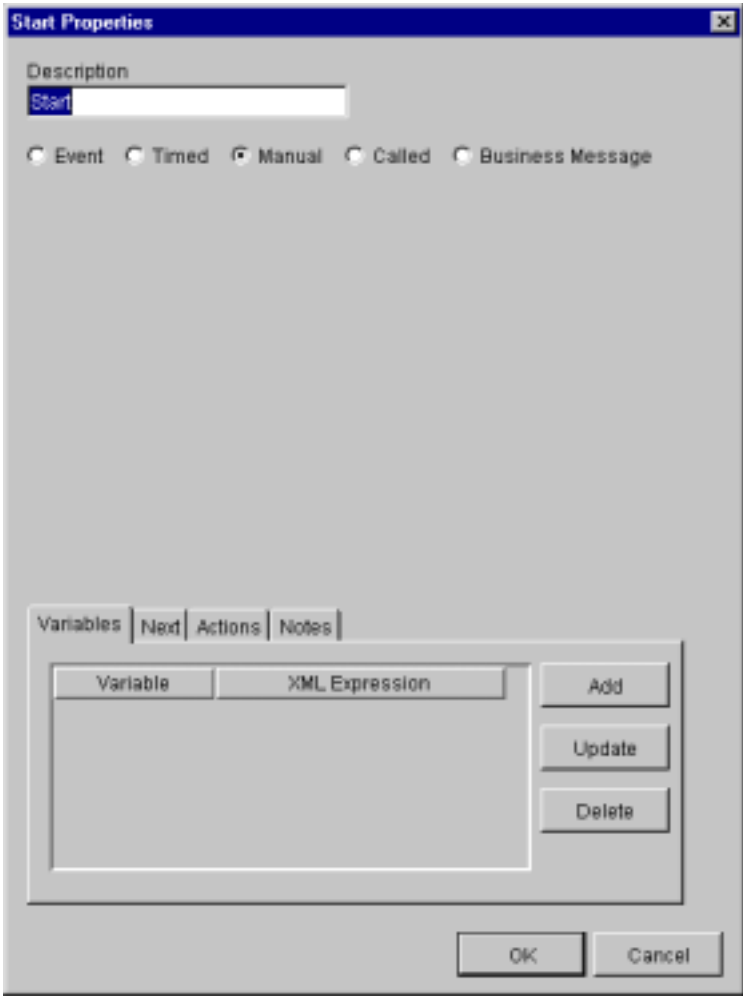
A conversation initiator workflow is started programmatically and must therefore have a Manual start property. For more information, see "Developing Applications That Start Conversation Initiator Workflows" on page 2-76.

To define the Manual start property for a conversation initiator workflow:

1. Display or add the start shape, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.

2. Double-click the start shape to display the Start Properties dialog box.

Figure 2-9 Start Properties Dialog Box: Manual Start



3. Change the text in the Description field to a unique, identifiable name.
4. If the workflow will be started programmatically, select Manual.

   Otherwise, select any other option *except* Business Message as appropriate. For more information about these options, see Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.
5. Click OK.

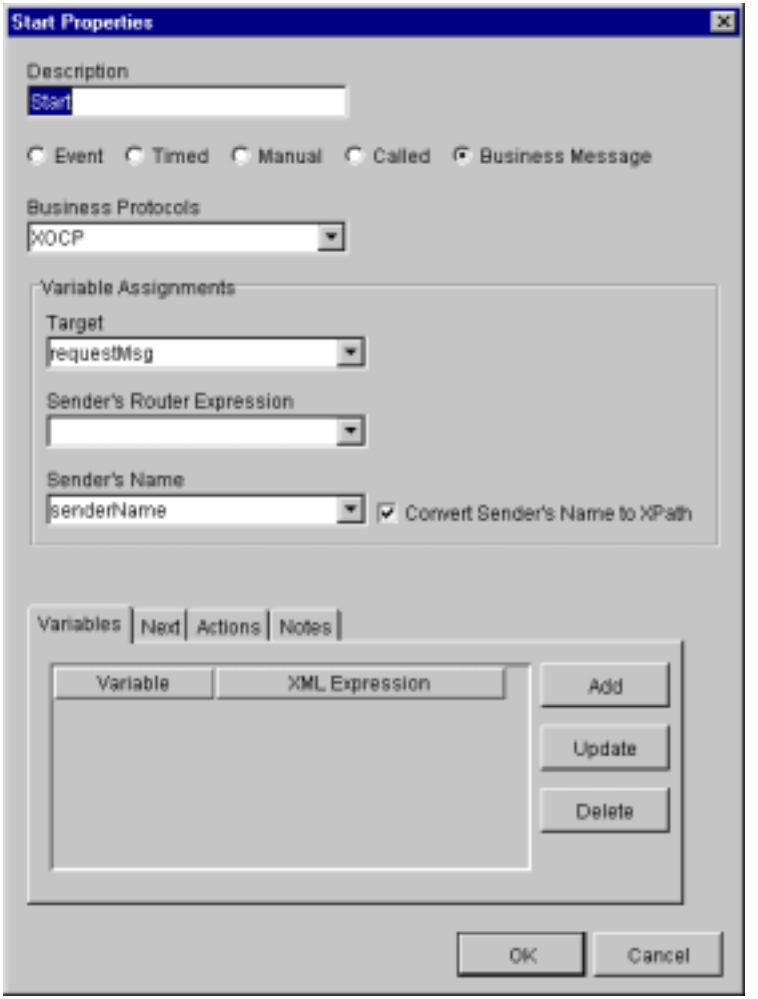
Defining the Start for a Conversation Participant Workflow

A conversation participant workflow is started when it receives an initial business message from a conversation initiator workflow. You must define a Business Message start state for such workflows.

To define the Business Message start state for a conversation participant workflow:

1. Display or add the start shape, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide.*
2. Double-click the start shape to display the Start Properties dialog box.

Figure 2-10 Start Properties Dialog Box: Business Message Start



3. Change the text in the Description field to a unique, identifiable name.
4. Select Business Message.
5. Select the Business Protocol.
6. Specify the target variable, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.
7. Specify the sender filter variable, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.
8. Click OK.

Defining Conversation Termination

A conversation is terminated when the conversation initiator workflow reaches a done state. Conversation participant workflows can end their participation in a conversation before the conversation is terminated.

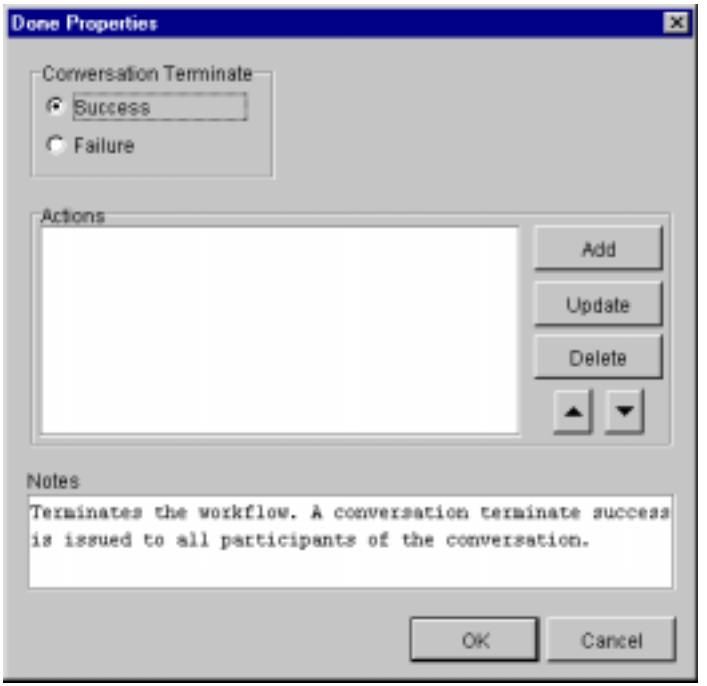
Defining the Termination of Conversation Initiator Workflows

For a conversation initiator workflow, you define the conversation termination property (terminate with success or failure) for any done node in the workflow. Once a done node is reached in the workflow, the running instance of the workflow is marked done, regardless of whether the active workflow has reach all the done nodes. A conversation initiator workflow can terminate a conversation, but other participants in the conversation cannot.

To define the termination for a conversation initiator workflow:

1. Add or view a done shape, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide.*
2. Double-click the done shape or right-click it in the folder tree and choose Properties to display the Done Properties dialog box.

Figure 2-11 Done Properties Dialog Box



3. Select a conversation termination option in the Done Properties dialog box.

Table 2-5 Conversation Termination Options in the Done Properties Dialog Box

Graphic	Field	Description
	Success	The conversation should be terminated with a SUCCESS result (default). The conversation is terminated after the actions for this state are done. The SUCCESS result indicates that the workflow instance completed successfully. The participants of the conversation will be notified (if possible) that the conversation is being terminated.
	Failure	The conversation should be terminated with a FAILURE result. The conversation is terminated after the actions for this state are done. The FAILURE result indicates that the workflow instance encountered conversation-specific or application-specific errors. The participants of the conversation are notified (if possible) that the conversation is being terminated.

Defining the End of Conversation Participant Workflows

A conversation participant workflow has defined conversation properties, a Business Message start property, and (optionally) a Conversation Terminate event. The Conversation Terminate event is used in a participant workflow to wait for a conversation termination signal from the conversation initiator. It allows a participant workflow to perform additional processing (such as housekeeping operations) based on the status of the conversation termination.

Note: The use of this event is optional. A workflow that does not wait for this event can leave the conversation by simply ending the workflow (a Done node).

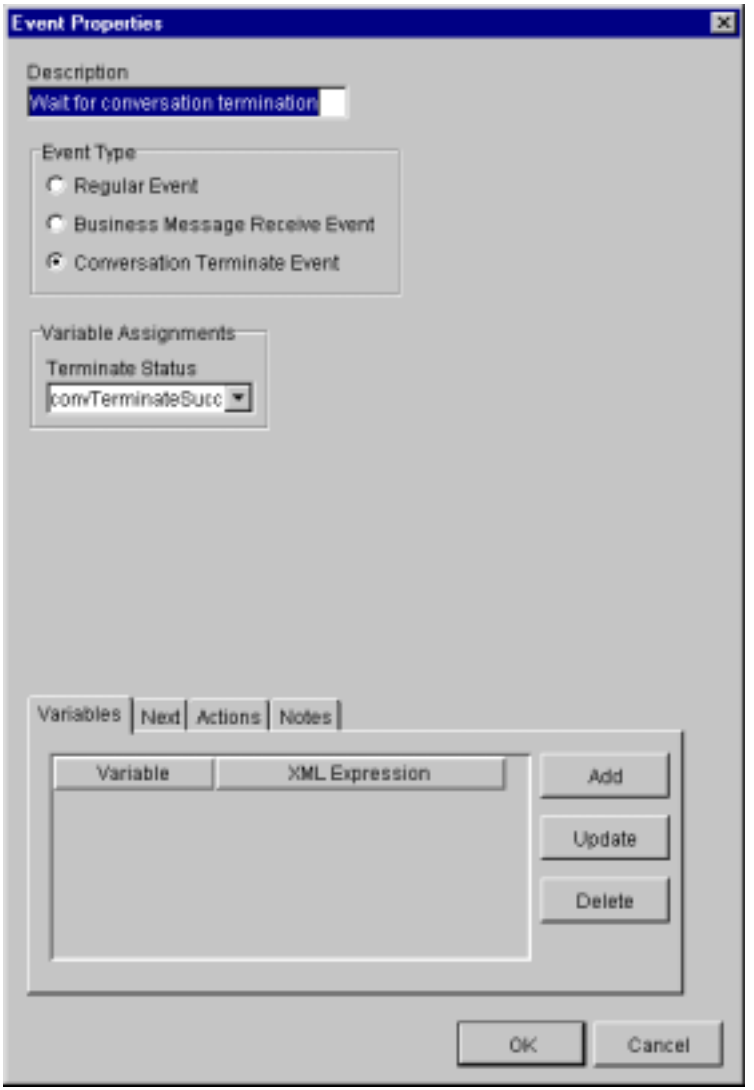
A workflow event shape represents a notification node. The workflow waits for a conversation terminate to trigger the event. Upon that trigger, actions defined within the event can be executed and/or workflow variables can be set.

To add a Conversation Terminate event to a conversation participant workflow:

1. Display or add any task as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.

2. Double-click the event shape or right-click it in the folder tree and choose the Properties command to display the Event Properties dialog box.

Figure 2-12 Event Properties Dialog Box



3. Select Conversation Terminate Event.
4. Select a WebLogic Process Integrator Boolean variable to store the terminate status, which will be set to one of the values in the following table.

Table 2-6 Terminate Status Options in the Event Properties Dialog Box

Option	Description
True	Indicates that the conversation was terminated with a `SUCCESS` value.
False	Indicates that the initiator has terminated the conversation with a `FAILURE` value.

Note: You must explicitly create this Boolean variable before selecting it in this dialog box. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.

5. Click OK to save your changes.

WebLogic Process Integrator assigns the conversation terminate status value to a WebLogic Process Integrator Boolean variable, which can be accessed by the workflow or passed to a business operation. The developer of the workflow should take appropriate actions based on this value.

Defining WebLogic Process Integrator Variables for Workflows

A WebLogic Process Integrator variable is typically used to store application-specific information required by the workflow at run time. Variables are created and assigned values largely to "control" the logical path through a workflow instance; the same workflow template definition is instantiated multiple times and can be traversed in different ways if the flow contains decision nodes, which evaluate workflow variable values and branch to either the next True or next False within the workflow, as appropriate.

You must define WebLogic Process Integrator variables for a workflow template definition that contains processes that require variables during run time. During workflow execution, you can access a WebLogic Process Integrator variable in the following ways:

- Within the workflow instance, such as in a decision node
- Within a business operation, such as by executing an Enterprise Java Bean (EJB) or Java class

Associations Between WebLogic Process Integrator Variables and Java Data Types

If you access a WebLogic Process Integrator variable within a business operation, you need to know how WebLogic Process Integrator variable types correspond to Java data types. The following table shows how they are related.

Table 2-7 WebLogic Process Integrator Variables and Java Data Types

WLPI Variable Type	Java Data Type
`String`	`java.lang.String`
`Integer`	`Java.lang.Integer`
`Long`	`Java.lang.Long`
`Double`	`java.lang.Double`
`Date`	`java.util.Date`
`Boolean`	`java.lang.Boolean`
`Complex Object`	`java.lang.Object` (must implement `Serializable`)
`XML`	`org.w3c.dom.Node`

Rules for Defining WebLogic Process Integrator Variables

When defining WebLogic Process Integrator variables, comply with the following rules:

- You can access workflow variables programmatically in the following situations:
  - After the instance is created but before it is started, as described in "Step 2: Initialize Input Variables" on page 2-79.
  - During the execution of a Manipulate Business Message action, as described in "Defining Manipulate Business Message Actions" on page 2-44.
  - After the instance completed, as described in "Step 7: Handle Results in Output Variables" on page 2-85.
- If you define an input variable (a variable that is passed into the workflow), it must have a value assigned to it at run time before the workflow instance starts. For an example of how this is done in the WebLogic Process Integrator Verifier application, see "Step 2: Initialize Input Variables" on page 2-79.
- If you define an output variable (a variable that is passed out of the workflow), you can access its value after the workflow has finished.
- If you define a variable as neither input or output, you can access it only while the workflow is running.
- If you want to save a variable in a conversation participant workflow, or if you want to access it after the workflow starts, you must define it as an output variable and you need to access is using a business operation. For more information about defining business operations, see in Administering Data within WebLogic Process Integrator in the *BEA WebLogic Process Integrator Studio User Guide*.
- Actions for accessing business messages use Java object variables for manipulating, sending, or receiving business messages. However, the objects stored in the variables through these actions belong to an internal class that encapsulates the business message. As a consequence, these variables should only be accessed through message manipulators. Directly accessing these variables will result in undefined behavior. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.

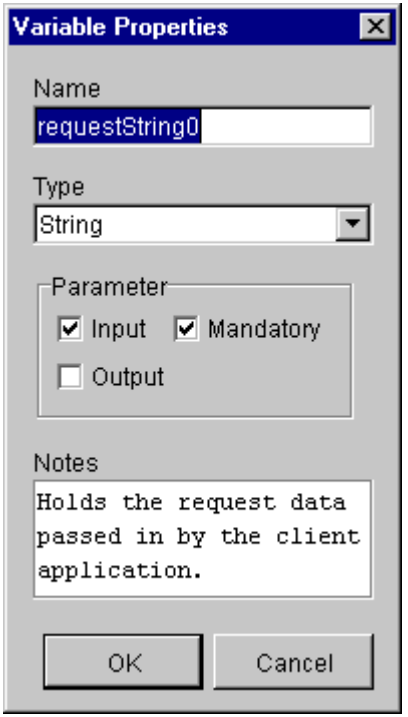
Defining Input Variables

Before a WebLogic Collaborate application can set an input variable, you must define it in WebLogic Process Integrator Studio.

To define an input variable:

1. Do one of the following:
   - In the folder tree, right-click Variables under the appropriate workflow template definition, and choose New Variable to display the Variable Properties dialog box.
   - Right-click an existing variable in the folder tree and choose Properties from the pop-up menu to display the Variables Properties dialog box.

Figure 2-13 Variables Properties Dialog Box for an Input Variable



2. Complete the fields in the following Variable Properties dialog box.

Table 2-8 Fields in the Variable Properties Dialog Box

Field	Description
Name	Meaningful name for the variable, such as ItemNumber.
Type	Variable type: Boolean, Date, Double, Entity EJB, Integer, Java Object, Session EJB, String, Long, or XML.
Parameter	Input or Output. For Input, choose whether the parameter is a mandatory one.
Notes	Optional descriptive text.

3. Select Input.
4. Click OK.

In the preceding example, the variable named `requestString0` is declared as an input variable. It is also declared as mandatory, which means that the workflow instance will start only if the WebLogic Collaborate application explicitly sets its value before attempting to start the workflow instance. For more information, see "Developing Applications That Start Conversation Initiator Workflows" on page 2-76.

Defining Output Variables

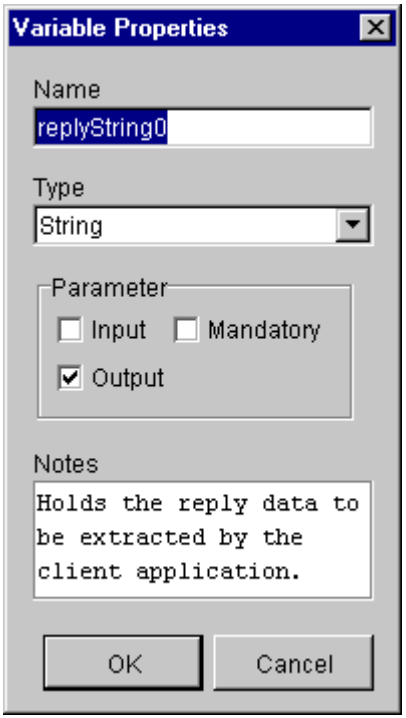
Variables that must be retrieved after the workflow completes must be declared as output variables. Otherwise, their value will not be preserved.

To define an output variable:

1. Do one of the following:
   - In the folder tree, right-click Variables under the appropriate workflow template definition, and choose New Variable to display the Variable Properties dialog box.

- Right-click an existing variable in the folder tree and choose Properties from the pop-up menu to display the Variables Properties dialog.

Figure 2-14 Variable Properties Dialog Box for an Output Variable



2. Complete the fields in the Variable Properties dialog box, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.
3. Select Output.
4. Click OK.

Working with Business Messages

You use WebLogic Process Integrator in conjunction with WebLogic Collaborate to exchange business messages between trading partners. The following sections describe how to work with business messages exchanged by using workflows:

- About Business Messages
- Summary of Prerequisite Tasks for Exchanging Business Messages
- Defining Variables and Manipulating Business Messages
- Creating and Defining Messages to Send
- Defining the Workflow to Receive Business Messages

About Business Messages

A business message is the basic unit of communication exchanged between trading partners in a conversation. A business message is a multi-part MIME message that consists of:

- A *business document*, which represents the XML-based payload part of a business message. The payload is the business content of a business message.
- An *attachment*, which represents a non-XML payload part of the business message.

You can access the contents of a business message programmatically using XOCP messaging objects, as described in "Step 1: Create the Business Message" on page 3-30 and "Receiving an XOCP Business Message" on page 3-53.

Summary of Prerequisite Tasks for Exchanging Business Messages

You must perform the following tasks before you can send and receive business messages. Subsequent sections describe these tasks in detail.

- Define the business message in the workflow template using WebLogic Process Integrator Studio.
  - For sending a business message, this involves defining a Manipulate Business Message action to construct the business message and the Send Business Message action to send the message.
  - For receiving a business message, this involves defining a Manipulate Business Message action to process an incoming business message.
- Write the Java application to process the business message by implementing the `com.bea.b2b.wlpi.MessageManipulator` interface and using the `manipulate` method on that object.

  This task involves writing the code associated with the Manipulate Business Message action.
  - For sending a business message, this code constructs the business message to send.
  - For receiving a business message, this code processes an incoming business message.

  For more information, see “Writing Business Operations to Manipulate Business Messages” on page 2-51.

Defining Variables and Manipulating Business Messages

The following sections describe procedures you perform in regard to both sent and received messages:

- Defining WebLogic Process Integrator Variables for Business Messages
- Defining Manipulate Business Message Actions
- Writing Business Operations to Manipulate Business Messages

Defining WebLogic Process Integrator Variables for Business Messages

At run time, a business message is stored in a WebLogic Process Integrator variable (of type Java Object) when it is ready to be sent or when it has been received. When a business message is ready to be sent, the application code associated with the Manipulate Business Message action constructs the business message and returns it in this variable to the workflow instance. When a business message has been received, the application code associated with the Manipulate Business Message action obtains this variable from the workflow instance and uses it to process the incoming business message.

Note: In WebLogic Collaborate, XOCP business messages are *not* stored in WebLogic Process Integrator variables of type XML Document.

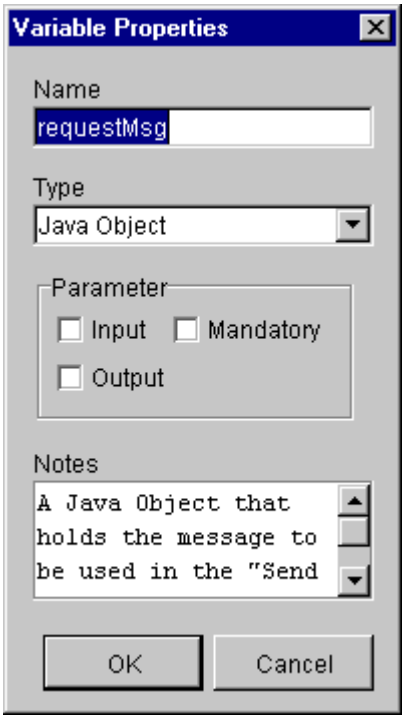
In WebLogic Process Integrator Studio, you must define the Java Object variables used to store business messages *before* you define any actions that refer to them, as described in "Defining Manipulate Business Message Actions" on page 2-44.

For each workflow template definition, you must define a separate variable for each business message that the workflow sends or receives. For example, if a workflow sends a request and receives a reply, you must define variables for both in its workflow template definition.

To define a variable for a business document in WebLogic Process Integrator Studio:

1. In the folder tree, right-click Variables under the appropriate workflow template definition and choose New Variable to display the Variable Properties dialog box.

Figure 2-15 Variable Properties Dialog Box



2. Specify a unique name for this variable.
3. Select the Java Object variable type.
4. Click OK.

Defining Manipulate Business Message Actions

At run time, the Manipulate Business Message action is invoked to manipulate a business message. If the workflow is sending a business message (such as request), the Manipulate Business Message action runs the associated application code to create the business message and save it in an output variable that is sent subsequently in a Send

Business Message action. If the workflow is receiving a business message (such as a reply), the Manipulate Business Message action captures the incoming business message in an input variable and passes it onto the associated application code for processing.

The Manipulate Business Message action can be associated with any of the following nodes: task, decision, event, and start. You must explicitly add the Manipulate Business Message action to the workflow template definition.

Adding a Manipulate Business Message Action

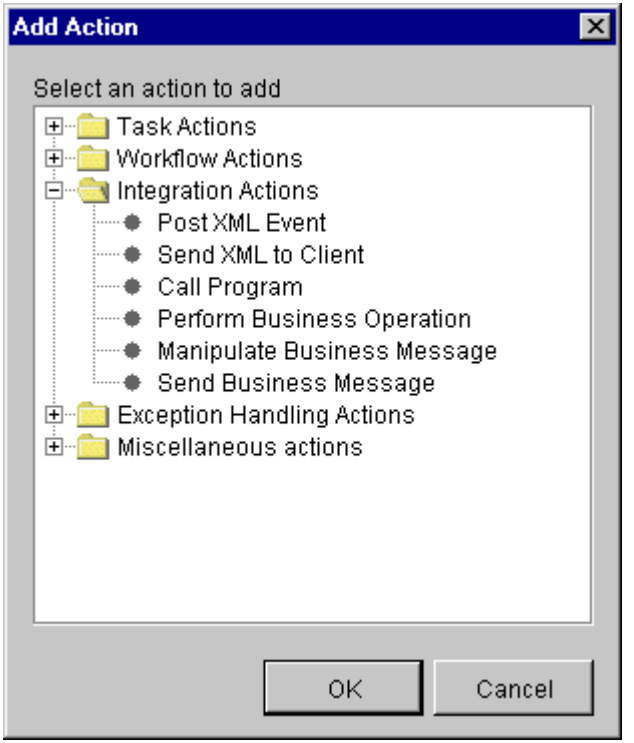
To define the Manipulate Business Message action for a workflow in WebLogic Process Integrator Studio:

1. In any dialog box where you can specify an action (such as the Task, Decision, Event, or Start Properties dialog box), click Add to display the Add Action dialog box.

Figure 2-16 Add Action Dialog Box

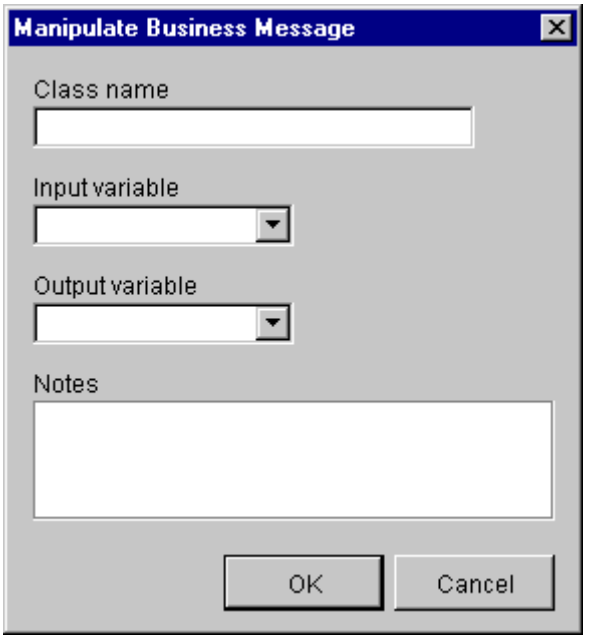
2. Click the Integration Actions folder to expand it.

Figure 2-17 Add Action Dialog Box With Integration Actions



3. Select Manipulate Business Message.
4. Click OK to display the Manipulate Business Message dialog box.

Figure 2-18 Manipulate Business Message Dialog Box



5. Complete the following fields in the Manipulate Business Message dialog box.

Table 2-9 Fields in the Manipulate Business Message Dialog Box

Field	Description
Class Name	Required. Name of a Java class that implements the `com.bea.b2b.wlpi.MessageManipulator` interface. For more information, see "Writing Business Operations to Manipulate Business Messages" on page 2-51.
Input variable	Name of a WebLogic Process Integrator variable that contains an existing business message, such as a message that has been received through a Receive Business Message action. The contents of this variable will be passed as the `in` parameter to the `manipulate` operation in the specified Java class that implements the `com.bea.b2b.wlpi.MessageManipulator` interface. If no variable name is specified, the value of the `in` parameter will be `null`. The specified variable must correspond to an existing WebLogic Process Integrator variable of type Java Object. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.
Output variable	Name of a WebLogic Process Integrator variable that will contain the business message returned by the `manipulate` operation in the specified Java class that implements the `com.bea.b2b.wlpi.MessageManipulator` interface. The specified variable must correspond to an existing WebLogic Process Integrator variable of type Java Object. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43. If no variable name is specified, then the return value of the `manipulate` operation will be ignored.
Notes	Optional descriptive text.

When specifying input or output variables, follow these guidelines:

- If the action receives a business message, then you must specify an input variable.
- If the action sends a business message, then you must specify an output variable.
- If the action receives a business message, modifies it, and then sends it, you must specify *both* an input and an output variable.

6. Click OK to save your changes.

Example of a Manipulate Business Message Action

For example, suppose you specify the following settings in the Manipulate Business Message dialog box.

Table 2-10 Sample Settings in the Manipulate Business Message Dialog Box

Field	Description
Class name	`examples.wlpiverifier.ProcessRequest`
Input variable	`requestMsg`
Output variable	`replyMsg`

At run time, when the WebLogic Process Integrator Process Engine executes the action with the specified settings, the following events occur:

1. An object of class `examples.wlpiverifier.ProcessRequest` is created using reflection and the default constructor.
2. The value of the `in` parameter (`requestMsg`) is retrieved.
3. The `manipulate` operation is invoked on the object.
4. The return value of the `manipulate` operation is stored in the WebLogic Process Integrator output variable (`replyMsg`).

Writing Business Operations to Manipulate Business Messages

You write business operations that use WebLogic Process Integrator variables and Java code to manipulate business messages that are exchanged between trading partners. The Manipulate Business Message action invokes a special WebLogic Collaborate business operation, a *message manipulator*, to create a business message to send or to process a business message that has been received. A message manipulator is a Java class that implements the `com.bea.b2b.wlpi.MessageManipulator` interface.

For more information about defining the message manipulator class and input and output variables for the Manipulate Business Message action, see "Defining Manipulate Business Message Actions" on page 2-44. For more information about the `com.bea.b2b.wlpi.MessageManipulator` interface, see the WebLogic Collaborate *Javadoc*.

Supported Operations

Message manipulators support the following operations for processing business messages:

- Creating business messages before sending them. (See "Steps for Creating Business Messages" on page 2-53.) A workflow *must* send messages to participate in conversations.
  - At run time, the Manipulate Business Message action is invoked. The Manipulate Business Message action creates a business message, based on the contents of other WebLogic Process Integrator variables, and returns the business message for storage in a variable. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.
  - The Send Business Message action retrieves this variable and sends the business message. For more information, see "Defining Send Business Message Actions" on page 2-57.
- Processing business messages after receiving them. After a business message has been received, an invoked business message manipulator extracts the contents of the message and stores any required message parts in WebLogic Process Integrator variables for use by other actions.

MessageManipulator Interface

To process business messages that are exchanged between roles in a conversation, workflow applications use Java classes that implement the `com.bea.b2b.wlpi.MessageManipulator` interface. This interface contains a single operation, `manipulate`, with the following signature:

```
XOCPMessage manipulate(WorkflowInstance instance, XOCPMessage in)
throws WLPIException;
```

When calling the `manipulate` operation, a workflow specifies the following parameters.

Table 2-11 Parameters in the Manipulate Operation

Parameter	Description
`instance`	Current workflow instance, which can be used to get or set variables. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.
`in`	XOCP message stored in the WebLogic Process Integrator variable specified as an input variable in the associated Manipulate Business Message action. If no input variable was specified in the Manipulate Business Message action or if the variable is empty, then `null` is passed.

The `manipulate` operation returns an XOCP message generated by the message manipulator. At run time, this XOCP message is stored in the output variable specified in the associated WebLogic Process Integrator Manipulate Business Message action. If this output variable was not specified, then the return value is ignored.

Note: Classes that implement the message manipulator interface *must* have a public default constructor (a constructor without arguments). The Process Engine uses Java reflection to create objects of that class and therefore invokes the default constructor.

Creating and Defining Messages to Send

The following sections describe how to prepare messages to be sent:

- Steps for Creating Business Messages
- Defining Send Business Message Actions
- Defining the Quality of Service for Message Delivery for a Send Business Message Action
- Assigning Message Token Information to WebLogic Process Integrator Variables

Steps for Creating Business Messages

The `PrepareQuery` class in the WebLogic Process Integrator Verifier program is an example of a message manipulator that constructs a business message before it is sent. It is called by the Manipulate Business Message action that occurs in the workflow. It returns a reply message (replyMsg variable) that is passed back to the workflow as the business message to send.

Step 1: Import the Necessary Packages

The following listing shows the packages that the `PrepareQuery` class imports, which includes the XOCP messaging objects that are used to create the XOCP message.

Listing 2-1 Importing the Necessary Packages

```
package examples.wlpiverifier;

import java.io.*;

import org.apache.xerces.dom.*;
import org.w3c.dom.*;

import com.bea.eci.logging.*;
import com.bea.b2b.wlpi.MessageManipulator;
import com.bea.b2b.wlpi.WorkflowInstance;
import com.bea.b2b.wlpi.WLPIException;

import com.bea.b2b.protocol.conversation.ConversationType;
import com.bea.b2b.enabler.*;
```

```
import com.bea.b2b.enabler.xocp.*;
import com.bea.b2b.protocol.messaging.*;
import com.bea.b2b.protocol.xocp.conversation.local.*;
import com.bea.b2b.protocol.xocp.messaging.*;
```

Step 2: Implement the MessageManipulator Interface

The following listing shows the `PrepareQuery` class declaration that implements the `MessageManipulator` interface.

Listing 2-2 Implementing the MessageManipulator Interface

```
public class PrepareQuery implements MessageManipulator{
...
}
```

Step 3: Call the Manipulate Method

The code in the following listing calls the manipulate method, which retrieves the current workflow instance object as well as the incoming business message.

Listing 2-3 Calling the manipulate Method

```
  public XOCPMessage manipulate(WorkflowInstance instance,
                                        XOCPMessage in)
    throws WLPIException{
```

Step 4: Get the Input Variables from the Current Workflow Instance

The code in the following listing gets the input variables associated with the current workflow instance by calling the `getVariable` method on the workflow instance.

Note: The `WlpiVerifierConstants` class contains constant values.

Listing 2-4 Getting the Input Variables

```
    int current =
((Integer)instance.getVariable(WlpiVerifierConstants.CURRENT)).in
tValue();

    String req =
(String)instance.getVariable(WlpiVerifierConstants.REQUESTSTR +
current);
    boolean last;
    if (current < 4){
      try{
        String v =
(String)instance.getVariable(WlpiVerifierConstants.REQUESTSTR +
(current + 1));
        if (v == null || v.length() == 0)
          last = true;
        else
          last = false;
      }catch (WLPIException e){
        last = true;
      }
    }else
      last = true;

    if (last)
      instance.setVariable("last", new Integer(1));
```

The code in the following listing creates the request message. For more information about creating XOCP business messages, see “Step 1: Create the Business Message” on page 3-30.

Listing 2-5 Creating the Request Message

```
    XOCPMessage xocpmsg = null;
    try{
      DOMImplementationImpl domi = new DOMImplementationImpl();

      // "request" - (param1) The qualified name of the document
type to be created.
      // "request" - The document type public identifier.
      // "upper-request.dtd" - The document type system identifier
      DocumentType dType = domi.createDocumentType("request",
"request", "upper-request.dtd");

      org.w3c.dom.Document rq = new DocumentImpl(dType);
      Element root = rq.createElement("request");
      rq.appendChild(root);
      Text t = rq.createTextNode(req);
      root.appendChild(t);
      root.setAttribute("last", last ? "true" : "false");

      xocpmsg = new XOCPMessage("");
      xocpmsg.addPayloadPart(new BusinessDocument(rq));

    }catch(Exception e){
      e.printStackTrace();
     throw new WLPIException("PrepareQuery raised exception:" + e);
    }
```

Step 5: Return the Request Message

The code in the following listing returns the request message in the variable `xocpmsg` (of type `XOCPMessage`) The return value is then assigned to an output variable (of type Java Object) in the workflow in preparation for sending the business message.

Listing 2-6 Returning the Request Message

```
return xocpmsg;
```

Defining Send Business Message Actions

After you create a business message in WebLogic Process Integrator using a Manipulate Business Message action and a message manipulator, you send the business message using the Send Business Message action.

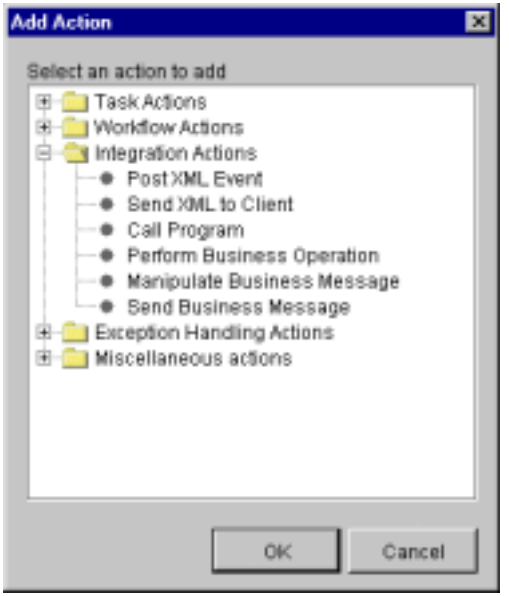
To define a Send Business Message action:

1. In any dialog box where you can specify an action (such as the Task, Decision, Event, or Start Properties dialog box), click Add to display the Add Action dialog box.

Figure 2-19 Add Action Dialog Box

2. Click the Integration Actions folder to expand it.

Figure 2-20 Add Action Dialog Box With Integration Actions



3. Select Send Business Message, and then click OK to display the Send Business Message dialog box.

Figure 2-21 Send Business Message Dialog Box

4. Complete the following fields in the Send Business Message dialog box.

Table 2-12 Fields in the Send Business Message Dialog Box

Field	Description
Source Variable	Name of a WebLogic Process Integrator Java Object variable that contains an `XOCPMessage`, probably created by an earlier call to a message manipulator. Required field. For more information, see "Defining WebLogic Process Integrator Variables for Business Messages" on page 2-43.
Router Expression Contains	Contents of the Router Expression field: a trading partner name or an XPath expression. Router expressions might be overridden by router expressions specified in the c-hub repository. For more information about routers, see Routing and Filtering XOCP Business Messages in the *BEA WebLogic Collaborate C-Hub Administration Guide*.
■ Trading Partner Name	The Router Expression field contains a single trading partner name.
■ XPath Expression	The Router Expression field contains an XPath expression.
■ Variable	The Router Expression field contains a WebLogic Process Integrator variable (of type String) with the contents of the Xpath expression. The variable is selected from a drop down list and may have been assigned by the Receive Business Message event.
Router Expression	Router expression that will be used when the message is sent. ■ If Trading Partner name is selected, the message will be sent to the specified trading partner. ■ If XPath Expression is selected, the message will be sent based on the specified XPath expression. If this field is left blank, a null filter will be used. For more information about router expressions, see Routing and Filtering XOCP Business Messages in the *BEA WebLogic Collaborate C-Hub Administration Guide*.
Target Role	The role in the conversation to which the message will be sent. Required field.

Table 2-12 Fields in the Send Business Message Dialog Box (Continued)

Field	Description
Use QoS Defined for Conversation	Specifies whether to use the Quality of Service defined at the template level or at this Send Business Message action level. ■ If selected, WebLogic Collaborate uses the QoS information that was defined at the workflow template definition level, as described in "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20. ■ If not selected, WebLogic Collaborate uses QoS information defined at this Send Business Message action level, as described in "Defining the Quality of Service for Message Delivery for a Send Business Message Action" on page 2-62.
Quality Of Service	Appears only if Use QoS Defined for Conversation is not selected. Click this button to specify the quality of service at this Send Business Message action level. For more information, see "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20.
Message Token	Click this button to assign the message token information to WebLogic Process Integrator variables. For more information, see "Assigning Message Token Information to WebLogic Process Integrator Variables" on page 2-63.
Notes	Optional descriptive text.

5. Click OK to save your changes.

Defining the Quality of Service for Message Delivery for a Send Business Message Action

The Quality of Service (QoS) is a set of attributes that are defined for reliable business message publishing. In WebLogic Process Integrator, you can define the QoS at the following levels:

- At the template level, where the settings apply to all Send Business Message actions, unless it is specifically overridden by the definition of the action. For

more information, see "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20.

- At the Send Business Message action level, where the settings apply to the specific action only but override the settings specified at the template level.

To define QoS at the Send Business Message level:

1. Click the Quality of Service button.
2. The Quality of Service Settings dialog box appears.
3. Complete the fields in the Quality of Service Settings dialog box, as described in Table 2-4.
4. Click OK.

Note: The definitions specified here will apply to this Send Message action only, not to all send actions within this conversation.

Assigning Message Token Information to WebLogic Process Integrator Variables

When a business message is sent by the WebLogic Collaborate messaging service, a message token is returned as a Java object at the programming level. The message token provides the information about the message, such as the message ID, conversation ID, send success/failure, the delivery status, and the number of recipient destinations after final selection (router and filter evaluations) at the c-hub. Applications call the `getVariable` method to get access to this variable.

This variable could be defined as an output variable that gets processed after the workflow ends. A message token is represented by the `com.bea.b2b.protocol.messaging.MessageToken` class, which is described in the Javadoc and in "Message Tokens" on page 3-48.

You can configure WebLogic Process Integrator workflows to get access to the message token by assigning the token and its associated information to WebLogic Process Integrator variables. At run time, values are assigned to the workflow instance variables after the Send Business Message action has completed. For more information about message tokens, see "Message Tokens" on page 3-48.

To assign the message token and its associated information to WebLogic Process Integrator variables:

1. Open the Send Business Message dialog box, as described in "Defining Send Business Message Actions" on page 2-57.
2. In the Send Business Message dialog box, click the Message Token button to display the Message Token dialog box.

Figure 2-22 Message Token Assignments Dialog Box

Note: The available options in this dialog box depend on the selected Quality of Service settings, as described in "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20.

3. Complete the following fields in the Message Token Assignments dialog box.

Table 2-13 Fields in the Message Token Assignments Dialog Box

Field	Description
QoS Messaging Confirmation set to:	Display only. Shows the QoS Message confirmation setting, which is described in "Defining the Quality of Service for Message Delivery at the Template Level" on page 2-20. The three possible settings are: ■ Confirm message delivery up to the c-hub ■ Confirm message delivery up to the router in the c-hub ■ Confirm message delivery to all destinations
Message Token	Assigns the returned message token to a WebLogic Process Integrator Java Object variable. This object can then only be passed to a business operation for processing.
Status of Send Message	Indicates whether the message was sent successfully (true for success and false for failure). Assigns the value to a WebLogic Process Integrator Boolean variable that can be accessed by the workflow or passed to a business operation.
Number of Initial Recipients	Number of recipients assigned by the c-hub after the message has traversed the router. Assigns the value to a WebLogic Process Integrator Integer variable that can be accessed by the workflow or passed to a business operation. This field appears only if the QoS Messaging Confirmation setting is either is one of the following selections: ■ To the router in the c-hub ■ To all destinations
Number of Actual Recipients	Actual number of recipients (number of c-enabler sessions that received the message). Assigns the value to a WLPI Integer variable that can be accessed by the workflow or passed to a business operation. This field is shown only if the QoS Messaging Confirmation setting is "To all destinations".

Table 2-13 Fields in the Message Token Assignments Dialog Box (Continued)

Field	Description
Time Elapsed for Acks (ms)	Time taken, in milliseconds, for acknowledgments from all recipients. This assigns the value to a WLPI Long variable that can be accessed by the workflow or passed to a business operation. This field is shown only if the QoS Messaging Confirmation setting is "Confirm message delivery to all destinations."
Notes	Optional descriptive text.

Defining the Workflow to Receive Business Messages

A workflow can receive a business message in the following circumstances:

- When a conversation participant workflow is waiting for the initial business message sent by the conversation initiator workflow. The first business message triggers the start node of the conversation, which is defined as a Business Message start. For more information, see "Defining the Business Message Start for Conversation Participant Workflows" on page 2-67.
- When a conversation initiator workflow or conversation participant workflow is waiting for another message, such as a reply to a request, as described in "Defining Business Message Receive Events" on page 2-70.

The following sections describe procedures for setting up your workflow to receive business messages:

- Defining the Business Message Start for Conversation Participant Workflows
- Defining Business Message Receive Events
- Steps for Receiving Business Messages

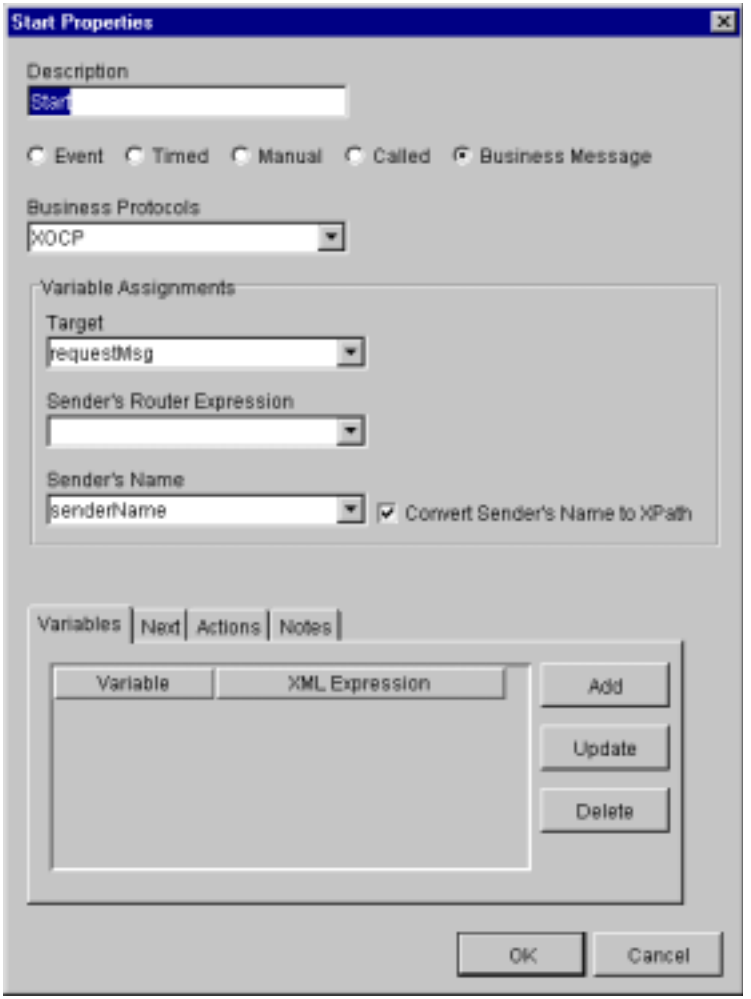
Defining the Business Message Start for Conversation Participant Workflows

To define the Business Message start property for a conversation participant workflow:

1. Display or add the start shape, as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.

2. Double-click the start shape to display the Start Properties dialog box.

Figure 2-23 Start Properties Dialog Box



3. Complete the following fields in the Start Properties dialog box.

Table 2-14 Fields in the Start Properties Dialog Box

Variable	Description
Start Property	Select `Business Message`.
Target	Name of a target WebLogic Process Integrator variable (of type Java Object) in which to store the business message. Required field.
Sender's Router Expression	Name of a WebLogic Process Integrator variable (of type String) in which to store an XPath expression. The value represents the XPath expression that was used by the sender to send the message. This XPath expression can be used later as part of a router expression to publish a reply to the current message back to the sender. Optional field.
Sender's Name	Name of a WebLogic Process Integrator variable (of type String) in which to store the name of the Trading Partner that sent the message. If the Convert Sender's Name to XPath check box is selected, then this name is converted to an XPath expression.
Convert Sender's Name to XPath	If selected, the contents of the variable specified in the Sender's Name field will be converted to an XPath expression suitable for use in the Send Business Message action. If not selected, the Sender's Name variable will be the actual name of the sending Trading Partner.

4. Click OK.

When a workflow instance is started for the conversation participant workflow, the target variable contains the business message that triggered the conversation. If a router variable is specified, it contains an XPath expression that can be used to reply to the sender. For more information, see "Defining Send Business Message Actions" on page 2-57.

Defining Business Message Receive Events

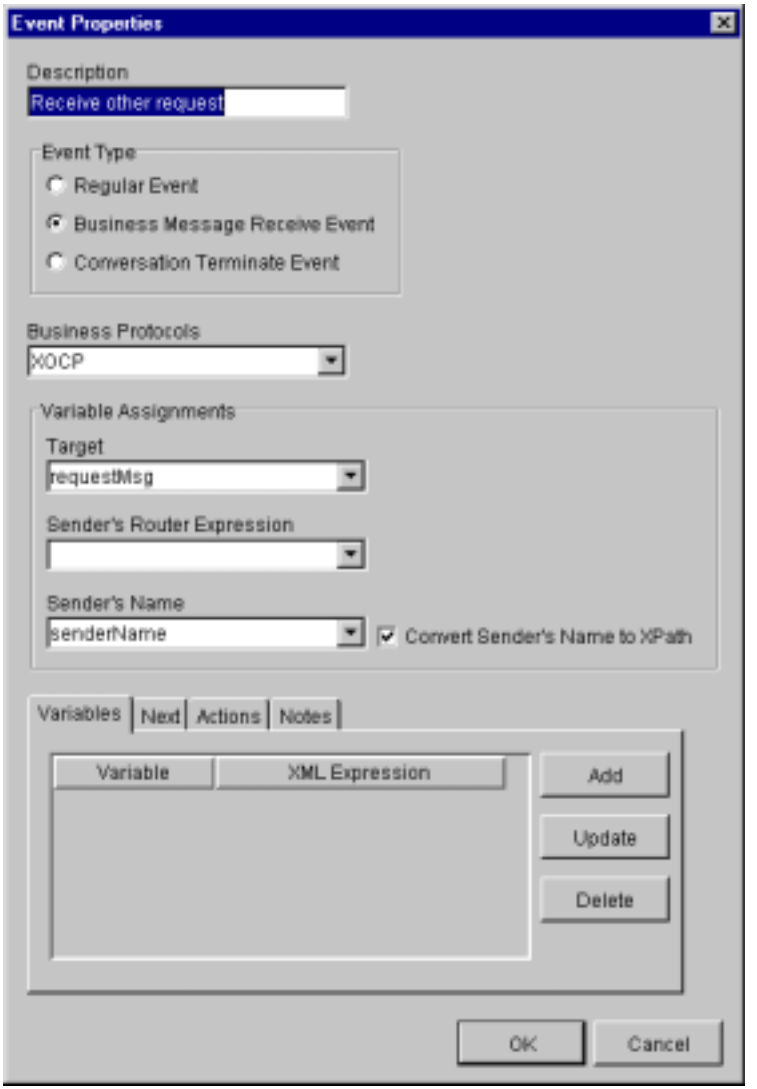
If a workflow waits to receive a business message, such as reply to a request or a subsequent (not an initial) request), you must define a Business Message Receive event. This event is triggered at run time when the appropriate business message is received in the conversation.

To define a Business Message Receive event:

1. Display or add a task as described in Working with Workflow Components in the *BEA WebLogic Process Integrator Studio User Guide*.

2. Double-click the event shape or right-click it in the folder tree and choose the Properties command to display the Event Properties dialog box.

Figure 2-24 Event Properties Dialog Box



3. Complete the following fields in the Event Properties dialog box.

Table 2-15 Fields in the Event Properties Dialog Box

Variable	Description
Event Type	Select `Business Message Receive Event`.
Target	Name of a target WebLogic Process Integrator variable (of type Java Object) in which to store the business message. Required field.
Sender's Router Expression	Name of a WebLogic Process Integrator variable (of type String) in which to store an XPath expression. The value represents the XPath expression that was used by the sender to send the message. This XPath expression can be used later as part of a router expression to publish a reply to the current message back to the sender. Optional field.
Sender's Name	Name of a WebLogic Process Integrator variable (of type String) in which to store the name of the Trading Partner that sent the message. If the Convert Sender's Name to XPath checkbox is selected, then this name is converted to an XPath expression.
Convert Sender's Name to XPath	If selected, the contents of the variable specified in the Sender's Name field will be converted to an XPath expression suitable for use in the Send Business Message action. If not selected, the Sender's Name variable will be the actual name of the sending Trading Partner.

At run time, when the business message is received, the event is triggered and the target variable is set to the business message that was just received. If a router variable is specified, it contains an XPath expression that can be used to reply to the sender. For more information, see "Defining Send Business Message Actions" on page 2-57.

Steps for Receiving Business Messages

The `PrepareReply` class in the WebLogic Process Integrator Verifier program is an example of a message manipulator that receives and processes a business message. It is called by the Manipulate Business Message action associated with the Start event (defined as Business Message start event) that is triggered when the initial business

message is received from the conversation initiator workflow. It returns a reply message (`replyMsg` variable) that is passed back to the workflow as the business message to send.

Step 1: Import the Necessary Packages

The code in the following listing shows the packages that the `PrepareReply` class imports, which includes the XOCP messaging objects that are used to create the XOCP business message.

Listing 2-7 Importing the Necessary Packages

```
import java.util.*;

import org.apache.xerces.dom.*;
import org.w3c.dom.*;

import com.bea.eci.logging.*;
import com.bea.b2b.wlpi.MessageManipulator;
import com.bea.b2b.wlpi.WorkflowInstance;
import com.bea.b2b.wlpi.WLPIException;
import com.bea.b2b.protocol.messaging.*;
import com.bea.b2b.protocol.xocp.conversation.local.*;
import com.bea.b2b.protocol.xocp.messaging.*;
```

Step 2: Implement the MessageManipulator Interface

The following listing shows the `PrepareReply` class declaration that implements the `MessageManipulator` interface.

Listing 2-8 Implementing the MessageManipulator Interface

```
public class PrepareReply implements MessageManipulator
```

Step 3: Call the Manipulate Method

The code in the following listing calls the `manipulate` method, which retrieves the current workflow instance object as well as the incoming business message (request).

Listing 2-9 Calling the manipulate Method

```
public XOCPMessage manipulate(WorkflowInstance instance,
                                        XOCPMessage in)
  throws WLPIException{
```

Step 4: Process the Request Message

The code in the following listing processes the request message.

Listing 2-10 Processing the Request Message

```
PayloadPart[] payload = in.getPayloadParts();

Document rq = null;
if (payload != null && payload.length > 0){
  BusinessDocument bd = (BusinessDocument)payload[0];
  rq = bd.getDocument();
}

if (rq == null){
  throw new WLPIException("Did not get a request document");
}

Element root = rq.getDocumentElement();
String name = root.getNodeName();
if (!name.equals("reply")){
  String msg = "Expected reply, found " + name;
  throw new WLPIException(msg);
}
if (!root.hasChildNodes()){
  String msg = "No data in reply";
  throw new WLPIException(msg);
}
Node value = root.getFirstChild();
if (value == null){
```

```
        String msg = "No text inside request";
        throw new WLPIException(msg);
      }
      if (value.getNodeType() != Node.TEXT_NODE){
        String msg = "Note inside request is not text node";
        throw new WLPIException(msg);
      }
      Text t = (Text)value;
      String data = t.getData();
```

Step 5: Process the Input Variables Associated with the Current Workflow Instance

The code in the following listing processes the input variables associated with the current workflow instance.

Listing 2-11 Changing the Input Variables

```
      // get the 'current' variable from the workflow
      int current =

((Integer)instance.getVariable(WlpiVerifierConstants.CURRENT)).in
tValue();
      // assign it to a variable that we can extract later in the
client servlet
      instance.setVariable(WlpiVerifierConstants.REPLYSTR +
current, data);
      // get the no of recipients that got the message
      Integer noOfRecipients =
(Integer)instance.getVariable(WlpiVerifierConstants.RECIPIENTSHOL
DER);
      // assign it to a variable that we can extract later in the
client servlet
      instance.setVariable(
          WlpiVerifierConstants.RECIPIENTS + current,
(Integer)noOfRecipients);

      return null;
  }
```

Developing Applications That Start Conversation Initiator Workflows

For conversation initiator workflows, you can start the workflow at run time by using WebLogic Process Integrator Worklist or by starting it in a Java application. The following sections describe how to start a conversation initiator workflow programmatically:

- WebLogic Process Integrator Integration API
- Creating Workflow C-Enabler Sessions
- Programming Steps for Accessing Conversation Initiator Workflows

To start a conversation initiator workflow programmatically, the start node for the workflow template must have a Manual start property, as described in "Defining the Start for a Conversation Initiator Workflow" on page 2-27.

WebLogic Process Integrator Integration API

WebLogic Collaborate applications use the `com.bea.b2b.wlpi` package to start WebLogic Process Integrator workflows. This package provides the following interface and classes.

Table 2-16 Components of the com.bea.b2b.wlpi Package

Object	Description
`MessageManipulator` interface	Implemented by all classes that are used in a Manipulate Message action inside a WebLogic Process Integrator task
`WorkflowEnablerSession` class	Represents a workflow c-enabler session that is coupled with WebLogic Process Integrator workflows
`WorkflowEnablerSessionManager` class	Controls workflow c-enabler sessions

Table 2-16 Components of the com.bea.b2b.wlpi Package (Continued)

Object	Description
`WorkflowInstance` class	Represents a running workflow instance
`WLPIException` class	Thrown if an error occurs with WebLogic Process Integrator processing

For details about this package, see the WebLogic Collaborate *Javadoc*.

Creating Workflow C-Enabler Sessions

Before you start the workflow, the application must first create a *workflow c-enabler session* with the c-hub. A workflow c-enabler session is a logical session between one c-enabler node and a c-hub that connects the c-enabler node to a c-space.

You can create workflow c-enabler sessions in the following ways:

- When the WebLogic Server starts up, as described in this section.
- At run time in a WebLogic Collaborate application, as described in "Creating a New Workflow C-Enabler Session Programmatically" on page 2-82.

It is generally more convenient to create workflow c-enabler sessions as part of the bootstrap sequence of the WebLogic Server hosting the c-enabler. To create workflow c-enabler sessions upon WebLogic Server startup, you must specify the `com.bea.b2b.wlpi.Start` startup class and c-enabler session information while using the WebLogic Server Administration Console to configure the c-enabler node.

The following listing shows a fragment of a sample `config.xml` file that specifies the startup class and defines a workflow c-enabler session named `caller-session`.

Listing 2-12 Starting a Workflow C-Enabler Session in the config.xml File

```
<<StartupClass
Arguments="ConfigFile=xml/enablers.xml,SessionName=caller-session
,User=bea,Password=12345678,OrgName=BEA"
      ClassName="com.bea.b2b.wlpi.Start"
      Name="WlpiVerifierCaller"
```

```
    Targets="myserver"
/>
```

For more information about using the WebLogic Server Administration Console, see see the *BEA WebLogic Server Administration Guide*.

Programming Steps for Accessing Conversation Initiator Workflows

To access a WebLogic Process Integrator conversation initiator workflow, a WebLogic Collaborate workflow application completes the following steps:

- Step 1: Import the Necessary Packages
- Step 2: Initialize Input Variables
- Step 3: Establish a Workflow C-Enabler Session
- Step 4: Create a Workflow Instance for a Specific Workflow Template
- Step 5: Start a Workflow Instance
- Step 6: Wait for the Workflow Instance to Complete
- Step 7: Handle Results in Output Variables
- Step 8: Handling Exceptions

The `WlpiVerifierServlet` in the WebLogic Process Integrator Verifier application provides the sample code for this section. For more information about the WebLogic Process Integrator Verifier application, see Running the WebLogic Process Integrator Verifier Example in *BEA WebLogic Collaborate Getting Started*.

Step 1: Import the Necessary Packages

To access a WebLogic Process Integrator workflow, a WebLogic Collaborate application begins by importing the necessary package(s). At a minimum, the WebLogic Collaborate application must import the `com.bea.b2b.wlpi` package, as shown in the following listing.

Listing 2-13 Importing the com.bea.b2b.wlpi Package

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import com.bea.wlpi.server.*;
import com.bea.wlpi.common.*;
import com.bea.wlpi.server.worklist.*;
import com.bea.wlpi.server.wlcollaborate.*;
import com.bea.b2b.wlpi.*;
import com.bea.eci.logging.*;
```

Step 2: Initialize Input Variables

If a conversation initiator workflow has input variables defined, you must initialize and assign variables to them in the WebLogic Collaborate application before starting the workflow instance. These input variables must first be declared in the template definition in WebLogic Process Integrator Studio, as described in "Defining Input Variables" on page 2-38.

Retrieving Values from a Submitted HTML Form

The `WlpiVerifierServlet` in the WebLogic Process Integrator Verifier application initializes the input variables and retrieves the values that a Web user has entered and submitted in an HTML form, as shown in the following listing from the WebLogic Process Integrator Verifier application.

Listing 2-14 Setting the Value of an Input Variable

```
// servlet parameters
  private static final String sendParam = "sendstring";
  private static final String idParam = "recipientId";

  // vars to hold parameter values
  private String[] sendStr = new String[5]; // strings to send
  // contains the replies for each sent string
  private String[] reply = new String[sendStr.length];
  // the no of actual recipients for each reply
  private Integer[] noOfRecipients = new Integer[sendStr.length];
```

```
  public void doPost(HttpServletRequest req, HttpServletResponse
res)
      throws IOException, ServletException{
    try{
      // get the parameters for this servlet
      Enumeration  e = req.getParameterNames();
      String name = null;
      while (e.hasMoreElements()){
        name = (String)e.nextElement();
        String lowName = name.toLowerCase();
        if (lowName.startsWith(sendParam)){
          int index =
Integer.parseInt(lowName.substring(sendParam.length()));
          sendStr[index] = req.getParameter(name);
        }
      }//while
```

Assigning Values by Using the setVariable Method

A WebLogic Collaborate application sets instance variables by calling the `setVariable` method on the workflow instance, passing the name of the variable and its value. The `setVariable` method requires the following parameters.

Table 2-17 Parameters for the setVariable Method

Parameter	Description
`name`	Name of the variable to be set.
`value`	Value for the variable. The value must be represented as a Java object, as described in "Associations Between WebLogic Process Integrator Variables and Java Data Types" on page 2-36.

The `WlpiVerifierServlet` of the WebLogic Process Integrator Verifier application uses the `setVariable` method to specify values for the request string, as shown in the following listing.

Listing 2-15 Setting the Value of Input Variables

```
for (int i = 0 ; i < data.length ; i++){
     wi.setVariable(WlpiVerifierConstants.REQUESTSTR + i, data[i]);
```

Step 3: Establish a Workflow C-Enabler Session

You establish a workflow c-enabler session by accessing the Workflow C-Enabler Session Manager and then retrieving an existing workflow c-enabler session or creating a new one.

Accessing the Workflow C-Enabler Session Manager

WebLogic Collaborate provides a Workflow C-Enabler Session Manager that manages all workflow c-enabler sessions. One Workflow C-Enabler Session Manager exists per WebLogic Server instance. A WebLogic Collaborate application must get access to this Workflow C-Enabler Session Manager by calling the `WorkflowEnablerSessionManager.get()` method, as shown in the following listing:

Listing 2-16 Accessing the Workflow C-Enabler Session Manager

```
WorkflowEnablerSessionManager wesm =
WorkflowEnablerSessionManager.get();
```

After accessing the Workflow C-Enabler Session Manager, a WebLogic Collaborate application either creates a new workflow c-enabler session or retrieves an existing one.

Retrieving an Existing C-Enabler Session

If the required workflow c-enabler session already exists (for example, the workflow c-enabler session was created upon WebLogic Server startup, as described in “Creating Workflow C-Enabler Sessions” on page 2-77), a WebLogic Collaborate

application can obtain access to it by calling the `getExistingEnablerSession` method and passing its workflow c-enabler session name. The workflow c-enabler session name is specified in the c-enabler XML configuration file.

The code in the following listing retrieves an existing workflow c-enabler session in a WebLogic Collaborate application by specifying its name (`caller-session`):

Listing 2-17 Retrieving an Existing Workflow C-Enabler Session

```
WorkflowEnablerSession wes = wesm.getExistingEnablerSession(
                  "caller-session");
```

Creating a New Workflow C-Enabler Session Programmatically

If you do not create workflow c-enabler sessions upon WebLogic Server startup, as described in "Creating Workflow C-Enabler Sessions" on page 2-77, the WebLogic Collaborate application must do so programmatically. Creating a workflow c-enabler session includes registering a workflow conversation handler for all of the conversation types that are active for the workflow c-enabler session.

Note: A `WLPIException` will be thrown if the specified workflow c-enabler session was already created in a startup class or by a servlet initialized during the WebLogic server bootstrap.

The code in the following listing creates a workflow c-enabler session in a WebLogic Collaborate application.

Listing 2-18 Creating a Workflow C-Enabler Session

```
// Define the parameters for the workflow c-enabler session
String configFile = "xml/enablers.xml";
String sessionName = "caller-session";
String orgName = "WAC1";
String user = "bea";
String password = "12345678";

// Get the only WorkflowEnablerSessionManager
WorkflowEnablerSessionManager wesm =
WorkflowEnablerSessionManager.get();
```

```
// Create a workflow c-enabler session with specified parameters
WorkflowEnablerSession wes = wesm.getEnablerSession(configFile,
sessionName, orgName, user, password);
```

The WebLogic Collaborate application passes the following parameters to the `getEnablerSession` method.

Table 2-18 Parameters for getEnablerSession()

Parameter	Description
configFile	Name of the c-enabler XML configuration file where the session is defined. This parameter is identical to the corresponding parameter in the `EnablerSessionFactory.getEnablerSession` method.
sessionName	Name of a session in the c-enabler XML configuration file. This parameter is identical to the corresponding parameter in the `EnablerSessionFactory.getEnablerSession` method.
orgName	Name of the WebLogic Process Integrator organization associated with the workflow template definition(s) to use.
user	Login username of a WebLogic user that has access to the workflow template definitions.
password	Login password of a user that has access to the template definitions.

Step 4: Create a Workflow Instance for a Specific Workflow Template

After establishing a workflow c-enabler session, a WebLogic Collaborate application creates a workflow instance by calling the `createInstance` method, passing the name of the WebLogic Process Integrator workflow template to use. Calling this method automatically creates a corresponding WebLogic Collaborate conversation associated with this workflow c-enabler session.

The specified workflow template must:

- Be active and not expired
- Have a Manual start state

Note: A WebLogic Collaborate application can call the `createInstance` method to to create any valid workflow instance, regardless of whether the workflow template is linked to a role in a WebLogic Collaborate conversation definition.

The code in the following listing creates a workflow instance.

Listing 2-19 Create a Workflow Instance for the Specified Workflow Template

```
WorkflowInstance wi =
wes.createInstance(WlpiVerifierConstants.INITIAL_TEMPLATEID);
```

Note: The `WlpiVerifierConstants` class contains constant values.

Step 5: Start a Workflow Instance

After creating a workflow instance and initializing input variables, a WebLogic Collaborate application starts the workflow instance by calling the `start` method on the instance, as shown in the following listing.

Listing 2-20 Start a Workflow Instance

```
wi.start();
```

Step 6: Wait for the Workflow Instance to Complete

Once a workflow instance has been started, a WebLogic Collaborate application can wait for its completion by calling the `waitForCompletion` method on the workflow instance. The operation blocks until the workflow instance has completed.

Listing 2-21 Waiting for Completion of the Workflow Instance

```
private void waitForWorkFlowToEnd(WorkflowInstance wi)
    throws Exception{

    wi.waitForCompletion();
...
}
```

While waiting for the workflow instance to complete, a WebLogic Collaborate application can determine the completion state of the workflow instance by calling the `isCompleted` method on the workflow instance. This method returns a Boolean `true` if the workflow execution completed, or `false` if not.

Step 7: Handle Results in Output Variables

After a workflow instance has completed, a WebLogic Collaborate application can handle the results of the workflow instance by retrieving the information stored in output variables. These output variables must first be declared in the template definition in WebLogic Process Integrator Studio, as described in “Defining Output Variables” on page 2-39.

A WebLogic Collaborate application retrieves the value of an instance variable by calling the `getVariable` method on the workflow instance, passing the name of the variable to retrieve, as shown in the following sample listing.

Listing 2-22 Retrieving the Results in Output Variables

```
for (int i = 0 ; i < reply.length ; i++){
      try{
        reply[i] =
(String)wi.getVariable(WlpiVerifierConstants.REPLYSTR + i);
```

```
        noOfRecipients[i] =
(Integer)wi.getVariable(WlpiVerifierConstants.RECIPIENTS + i);
```

The `getVariable` method returns a Java object that should be cast to the appropriate Java data type, as described in "Associations Between WebLogic Process Integrator Variables and Java Data Types" on page 2-36.

Step 8: Handling Exceptions

If an error occurs while running a workflow application, a `com.bea.b2b.wlpi.WLPIException` is thrown. Workflow applications can catch this exception and process it as appropriate, as shown in the following listing.

Listing 2-23 Handling WLPIExceptions in Workflow Applications

```
catch (WLPIException we){
      String msg = "Exception in Workflow: " + we;
      throw new Exception(msg);
```

3 Using XOCP C-Enabler Applications to Exchange Business Messages

The following sections describe how to develop c-enabler applications that exchange business messages by using the eXtensible Open Collaboration Protocol (XOCP) in the WebLogic Collaborate messaging system:

- About XOCP C-Enabler Applications
- Programming Steps for C-Enabler Applications
- Sending XOCP Business Messages
- Receiving XOCP Business Messages

Many of the code samples in this chapter derive from the installation verification example, which resides in the `/examples/verifier` subdirectory of the WebLogic Collaborate application directory. For more information, see the *BEA WebLogic Collaborate Installation Guide*.

Developers also design and implement workflows by using the WebLogic Process Integrator Studio. For more information, see Chapter 2, "Using Workflows to Exchange Business Messages."

About XOCP C-Enabler Applications

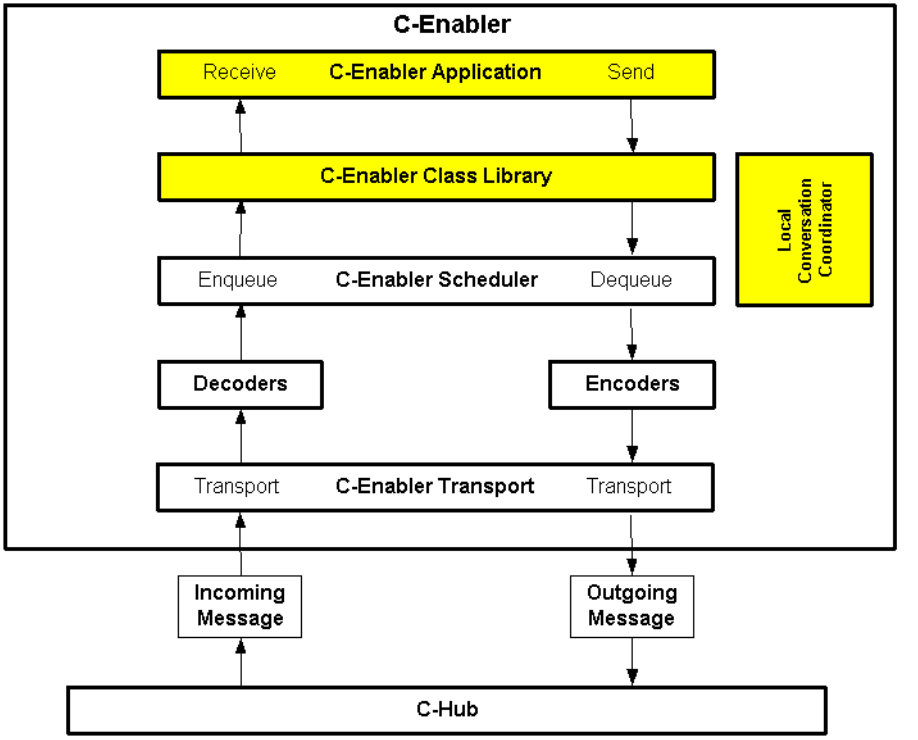
The following sections introduce XOCP c-enabler applications and related concepts:

- Architectural Overview
- Key Concepts
- Run-Time Information Flow
- Key Tasks for C-Enabler Applications

Architectural Overview

The following diagram shows how c-enabler applications fit into the c-enabler architecture.

Figure 3-1 C-Enabler Applications in the C-Enabler Architecture



The following table describes the key components related to c-enabler applications.

Table 3-1 Key C-Enabler Components for C-Enabler Applications

Component	Description
C-Enabler application	Java applications that exchange XOCP business messages with other trading partners. For more information, see "XOCP C-Enabler Applications" on page 3-5.
C-Enabler Class Library	Provides Java APIs for exchanging XOCP business messages. For more information, see "C-Enabler Class Library" on page 3-5.
Local Conversation Coordinator	Coordinates conversation activity between the c-enabler and the c-hub. For more information, see "Conversation Coordinators" on page 3-11.

For more information about other c-enabler architectural components, see Introduction to C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide.*

Key Concepts

This section describes the following key concepts associated with c-enabler applications:

- XOCP C-Enabler Applications
- C-Enabler Class Library
- Conversations and Conversation Definitions
- XOCP Business Messages and Message Envelopes
- Conversation Initiators and Participants
- Conversation Coordinators
- Trading Partner States
- Secure Messaging

XOCP C-Enabler Applications

XOCP C-enabler applications are Java applications that run on c-enabler nodes and use the C-Enabler Class Library to join and leave c-spaces; initiate or participate in conversations; terminate or leave conversations; and exchange XOCP business messages with other trading partners in the c-space. A c-enabler node can host many XOCP c-enabler applications.

C-Enabler Class Library

The C-Enabler Class Library provides APIs for exchanging XOCP business messages and consists of the packages in the following table.

Table 3-2 C-Enabler Class Library Packages

Package Name	Description
`com.bea.b2b.enabler`	Used for working with c-enabler nodes and c-enabler sessions.
`com.bea.b2b.enabler.xocp`	Used for working with c-enabler sessions for the XML Open Collaboration Protocol (XOCP).
`com.bea.b2b.protocol.xocp.conversation.local`	Used for working with conversations that use the XML Open Collaboration Protocol (XOCP).
`com.bea.b2b.protocol.messaging`	Used for working with messages in a conversation.
`com.bea.b2b.protocol.xocp.messaging`	Used for working with messages in conversations that use the XML Open Collaboration Protocol (XOCP).

For detailed information about these packages, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

Conversations and Conversation Definitions

In WebLogic Collaborate, a *conversation* is a series of message exchanges between trading partners that take place in a collaboration space and that are predefined according to a conversation definition. Each message in the conversation may cause any number of back-end transactions.

A *conversation definition* consists of a unique conversation name, conversation version, message definitions, trading partner IDs, and trading partner roles for one conversation. At design time, you use the WebLogic Process Integrator Studio to link a workflow template definition to a particular role (such as *buyer* or *seller*) in a WebLogic Collaborate conversation definition.

XOCP Business Messages and Message Envelopes

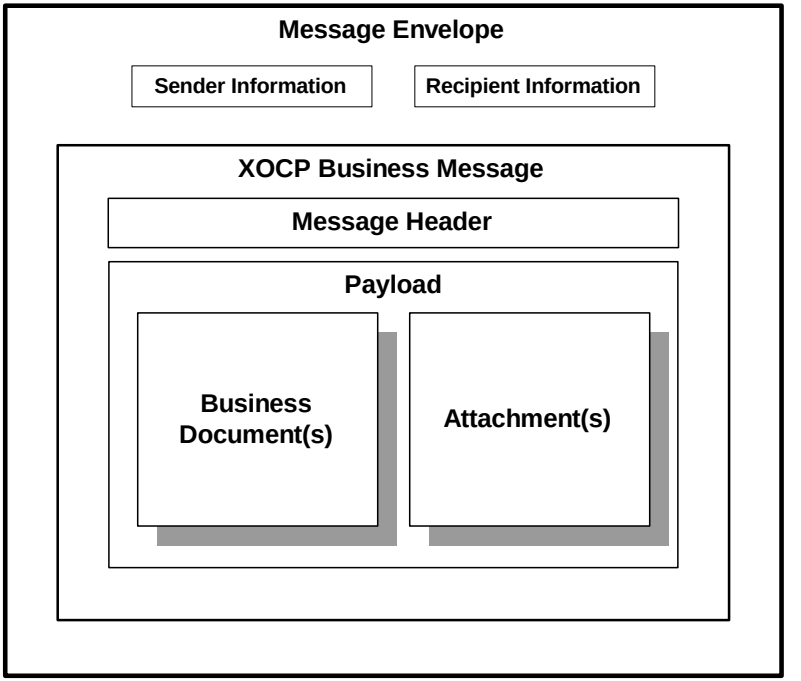
An *XOCP business message* is the basic unit of communication exchanged between trading partners in an XOCP conversation. An XOCP business message is represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.xocp.messaging.XOCPMessage` class.

A *message envelope* is a container for a business message. A message envelope contains information about the sender (such as the sender URL) and recipient (such as the destination URL). A message envelope is represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.MessageEnvelope` class. However, only logic plug-ins (not c-enabler applications) have programmatic access to message envelopes. For more information, see “Information Flow for Message Envelopes” on page 3-9 and Chapter 4, “Developing Logic Plug-Ins.”

Diagram of an XOCP Business Message

The following figure shows a message envelope and the components of an XOCP business message.

Figure 3-2 Components of an XOCP Business Message



Components of an XOCP Business Message

An XOCP business message is a multi-part MIME (Multipurpose Internet Mail Extensions) message. It consists of the following components.

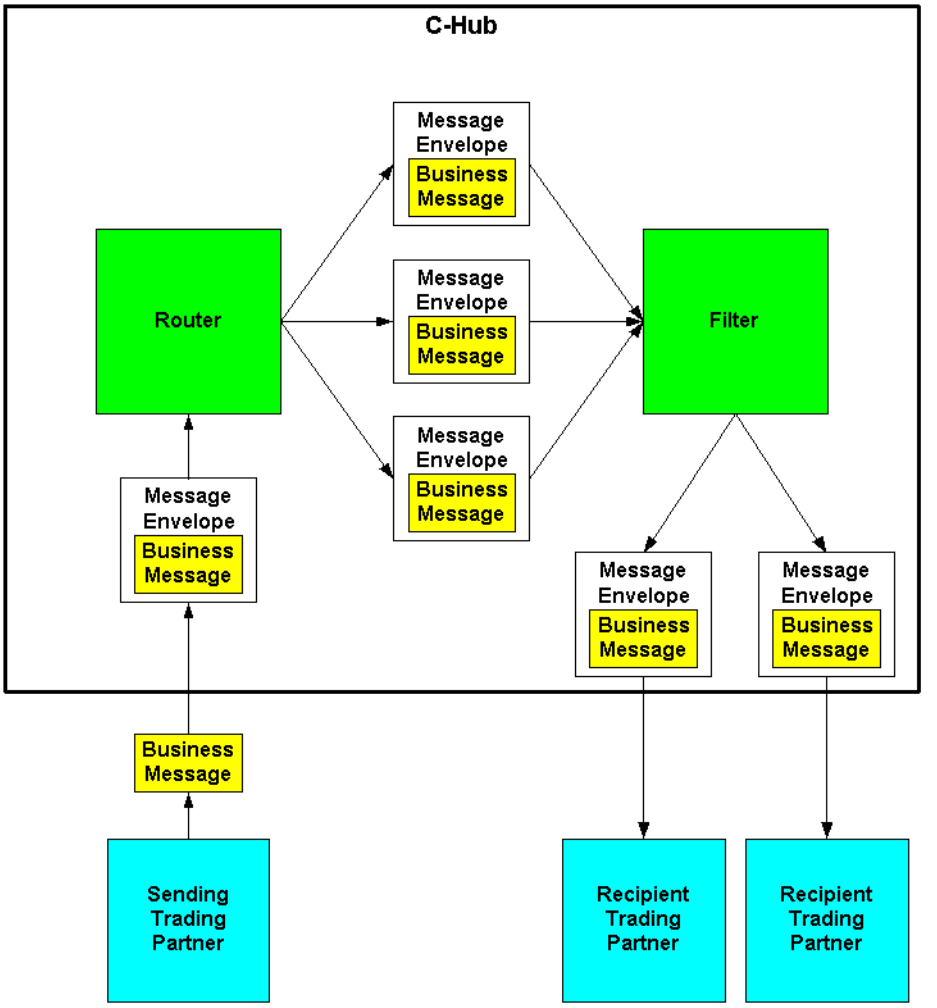
Table 3-3 Components of an XOCP Business Message

Component	Description
Message header	Message attributes, including the sender and recipient information, conversation information, Qualites of Service information, and so on.
Payload	Container for business document(s) and attachment(s) in this business message. The payload container has one or more business documents, one or more attachments, or a combination of both. A payload part is represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.PayloadPart` interface.
Business document(s)	XML-based payload part of a business message. Represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.BusinessDocument` class.
Attachment(s)	Non-XML-based payload part of a business message. Binary content. Represented in the C-Enabler Class Library by the `com.bea.b2b.protocol.messaging.Attachment` class.

Information Flow for Message Envelopes

The following figure shows an example of how message envelopes are processed in the c-hub.

Figure 3-3 Message Envelope Processing in the C-Hub



Message envelope processing occurs in the following sequence:

1. The sending c-enabler application creates and sends the business message to the c-hub.
2. The c-hub receives the business message and wraps it into a message envelope, extracting certain sender and recipient information from the business message.
3. The router processes the business message and then validates and finalizes the list of recipients.
4. The router creates a separate message envelope for each recipient in the recipients list, inserts a logical copy of the business message in the message envelope, and then forwards all message envelopes onto the filter.

   In the diagram, the router creates message envelopes for three recipients.
5. Within the filter, the applicable protocol-specific filter for each recipient trading partner evaluates each business message to determine whether it will be sent to the recipient. The filter forwards accepted messages onto the next processing step in the c-hub.

   In the diagram, the three business messages are evaluated in the filter. Two are accepted and one is rejected.
6. The c-hub validates the recipient, and then sends the business message (in its message envelope) to the recipient trading partner.
7. The recipient trading partner receives the business message.

Conversation Initiators and Participants

In any XOCP conversation, there are two types of trading partner roles:

- *Conversation initiator* is the trading partner who creates the conversation and sends the first business message (such as a request) to one or more recipient trading partners. The conversation initiator usually awaits a reply from each trading partner and might exchange subsequent business messages. When finished, the conversation initiator terminates the conversation (unless the conversation has timed out).
- *Conversation participant* is a trading partner who is enlisted in the conversation when it receives the first business message from the conversation initiator. The conversation participant usually sends a reply to the conversation initiator and,

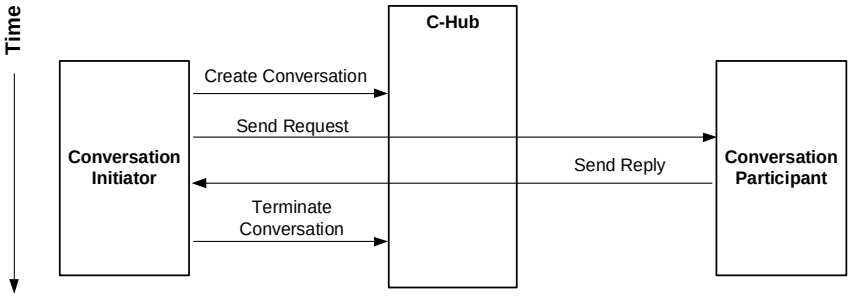
optionally, might exchange subsequent business messages. When finished, the conversation participant either leaves the conversation or waits until the conversation terminates.

Each conversation definition in the repository includes at least both of these types of roles. A trading partner must be subscribed to the appropriate role in the conversation in order to initiate or participate in conversations associated with that conversation definition.

The initiator of a conversation is usually determined by the role in which a trading partner is registered. For example, in a `GetQuote` conversation, the trading partner who is in the role of buyer would normally initiate a `GetQuote` conversation. Any trading partner who is in the role of seller would normally be a conversation participant in the `GetQuote` conversation.

The following figure shows some of the tasks that conversation initiators and conversation participants perform.

Figure 3-4 Conversation Initiators and Participants

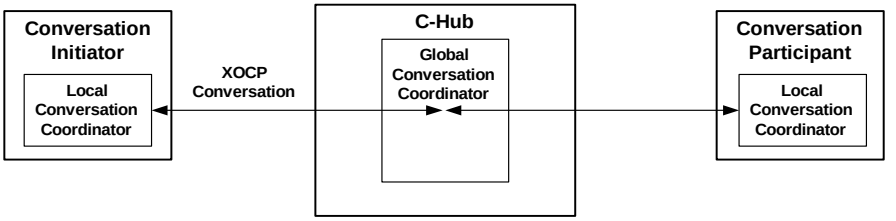


Conversation Coordinators

WebLogic Collaborate has two types of conversation coordinators that coordinate conversations at run time: a *global conversation coordinator* coordinates active conversations on the c-hub, and *local conversation coordinators* in c-enablers help the global coordinator coordinate active conversations locally.

The following figure shows global and local conversation coordinators in the WebLogic Collaborate architecture.

Figure 3-5 Global and Local Conversation Coordinators



Global Conversation Coordinator

A global conversation coordinator is a c-hub based service that coordinates conversation life cycles according to the rules of XOCP and supports long-living, durable conversations that span multiple organizational boundaries. The global conversation coordinator maintains a list of active conversations in the c-hub.

The global conversation coordinator performs the following services:

- Enlists and delists trading partners in a conversation
- Enforces the XOCP conversation termination protocol
- Maintains status information about conversations
- Provides the conversational context for the execution of the business protocol

Local Conversation Coordinators

A local conversation coordinator is a c-enabler based service that coordinates conversations in which the c-enabler node is participating. The local conversation coordinator maintains a list of active conversations in which the c-enabler node is participating. Each c-enabler session has a separate local conversation coordinator.

The local conversation coordinator performs the following tasks:

- Locally enlists in a conversation when the initial business message in a conversation is received from the c-hub
- Locally delists from a conversation when the terminate conversation system message is received from the c-hub

Trading Partner States

The following table describes the states assigned to trading partners as they perform tasks related to c-space and conversation participation.

Table 3-4 Trading Partner States

State	Description
`CONNECTED`	Trading partner has joined a c-space.
`REGISTERED`	Connected trading partner has registered for roles in conversations and is ready to initiate or participate in conversations.
`ACTIVE`	Registered trading partner has participated (sent or received a business message) in at least one conversation.
`DROPPEDOUT`	Trading partner has left a conversation.
`DISCONNECTED`	Trading partner has left a c-space.

Some of these trading partner states are visible in the C-Hub Administration Console and the C-Enabler Administration Console. For more information, see Using the C-Hub Administration Console in the *BEA WebLogic Collaborate C-Hub Administration Guide* and Working with C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide*.

Secure Messaging

Communication between the c-hub and c-enablers is secured via the Secure Sockets Layer (SSL). Before allowing the trading partner to exchange business messages, the c-hub must authenticate the identity of the trading partner using the trading partner's certificate. Once authenticated, business messages are exchanged securely among

trading partners by way of the c-hub. For more information about WebLogic Collaborate security, see Configuring Security in the *BEA WebLogic Collaborate C-Hub Administration Guide*.

Key Tasks for C-Enabler Applications

This section introduces the key tasks that c-enabler applications perform:

- Joining a C-Space
- Registering for a Role in a Conversation
- Engaging in Conversations with Trading Partners
- Shutting Down a C-Enabler Session to Leave a C-Space

Joining a C-Space

Before exchanging business messages, a c-enabler application must join a c-space. To join a c-space, the c-enabler application must create a *c-enabler session*, which is a logical session between a c-enabler node and one c-hub for one particular c-space.

Before a trading partner (c-enabler application) can create a c-enabler session to join a c-space:

- The c-space and trading partner configuration information must be defined in the WebLogic Collaborate repository on the c-hub that hosts the c-space.
- The session's configuration information (as well as the c-space, c-hub and c-enabler URL, trading partner name, and c-enabler session name) must be defined in the c-enabler XML configuration file. For more information, see Configuring C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide*. For an introduction to c-enabler sessions, see Introducing C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide.*
- The trading partner must be authorized to join the c-space.

When a c-enabler session is created, the c-enabler sends a system message to the c-hub with a request to join the c-space using the configuration settings specified in the c-enabler XML configuration file. This message acts as an authentication request to join the WebLogic Collaborate system. The c-hub validates the registration of the

trading partner in the requested c-space and, if valid, allows that trading partner to join that particular c-space. At this point, the trading partner is in a `CONNECTED` state but it cannot yet participate in conversations.

Note: If the c-enabler node crashes after joining a c-space, the c-enabler application can rejoin the c-space upon normal startup. The previous c-enabler session is discarded and new resources are assigned to the new c-enabler session. However, the c-hub will not be able to deliver business messages while the c-enabler node is down. Undelivered business messages will be discarded if the number of retry attempts is exceeded or if the business message or conversation times out.

When a trading partner wants to leave a c-space, the c-enabler application shuts down the associated c-enabler session, as described in "Shutting Down a C-Enabler Session to Leave a C-Space" on page 3-17.

Registering for a Role in a Conversation

Once connected, a trading partner needs to register a conversation handler for a particular role in a specific conversation definition in a given c-space. The conversation handler must be registered for the conversation type that will define how the trading partner participates in the conversation.

Role registration requires the following information in the c-hub repository:

- The *conversation type* is a subset of a conversation definition that defines a conversation for one trading partner based on the role in the conversation definition to which the trading partner subscribed.
- A *message definition* consists of ordered message parts. A message part contains a content type (XML or binary) and can contain a document definition. If the content type is XML, then the document definition is required for that part. For type binary, no other information is required.

For an introduction to these concepts, see Introducing C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide.*

Before registering for a conversation type, the trading partner must first be authorized to register. Authorization is configured by the c-hub administrator and is based on the trading partner's subscription to a role in a conversation definition.

When a c-enabler session attempts to register a conversation handler for a specific conversation type, the c-enabler sends an XOCP system message, "register for conversation," to the c-hub. The c-hub validates the role of the trading partner for the requested conversation type in the associated c-space. If the registration is valid, the trading partner is then allowed to initiate and participate in conversations associated with the registered conversation type. At this point, the trading partner is in a `REGISTERED` state and is ready to initiate or participate in conversations.

Engaging in Conversations with Trading Partners

Once registered for a role in a conversation, a trading partner can engage in conversations in accordance with that role. Conversation initiation and participation occurs on the c-hub itself. However, the c-enabler session maintains some state information about the conversations in which it is involved.

The overall tasks for conversation initiator c-enabler applications and conversation participant c-enabler applications are very similar. However, conversation initiator c-enabler applications can terminate conversations while conversation participant c-enabler applications cannot. Conversation participant c-enabler applications can only leave a conversation.

Initiating a Conversation and Sending a Business Message

To initiate a conversation, a conversation initiator c-enabler application creates the conversation. Optionally, the conversation initiator c-enabler application can specify a timeout value, after which the conversation will automatically terminate; this value overrides the timeout value that is specified in the associated conversation definition in the repository.

The local conversation coordinator on the c-enabler node sends an XOCP system message, "create conversation," to the c-hub. The global conversation coordinator in the c-hub creates a conversation in the appropriate c-space and enlists the trading partner as the conversation initiator. After the conversation is created, the conversation initiator c-enabler application creates and sends a business message, as described in "Sending XOCP Business Messages" on page 3-29.

Participating in a Conversation

The global conversation coordinator in the c-hub handles all business messages that the c-hub receives for a given conversation. After the c-hub delivers the initial business message to recipient trading partners, the global conversation coordinator enlists those

trading partners in that conversation. Once a trading partner is enlisted in a conversation, the trading partner is in an `ACTIVE` state and can send and receive business messages in that conversation.

When the c-enabler session on a target c-enabler node receives the initial business message in a conversation, it performs the necessary housekeeping (such as registering the conversation in the local list) before invoking the `onMessage` callback on the conversation handler. For more information, see "Receiving XOCP Business Messages" on page 3-52.

Once a registered trading partner is enlisted in a conversation, the trading partner is in an `ACTIVE` state and can send and receive business messages in that conversation.

Leaving a Conversation

When finished participating in a conversation, a conversation participant trading partner can leave the conversation. When a trading partner leaves a conversation, the conversation coordinator removes it from the list of participating trading partners. Subsequent business messages in that conversation will *not* be sent to that trading partner. After a trading partner leaves, it is in a `DROPPEDOUT` state for that conversation.

Terminating Conversations

A conversation terminates when the initiating trading partner explicitly terminates the conversation, or when the conversation times out, which ever occurs first. A trading partner who has initiated a conversation must terminate that conversation at the appropriate time in a business process.

Note: Only the conversation initiator can terminate a conversation.

When a conversation is terminated, the conversation coordinator sends all of the participating trading partners an XOCP system message, "terminate message," which is propagated as the callback `onTerminate` on registered conversation handlers in c-enabler sessions at respective c-enabler nodes.

Shutting Down a C-Enabler Session to Leave a C-Space

When a trading partner has finished its activities in a c-space, the c-enabler application should leave the c-space by shutting down the c-enabler session. When a c-enabler application shuts down a c-enabler session, the c-enabler sends an XOCP system message, "leave c-space," to the c-hub. When the c-hub receives this system message,

the conversation coordinator automatically terminates all of the conversations that the trading partner has initiated in the c-space and delists the trading partner from all other conversations in which it was participating in the c-space.

Leaving a c-space:

- Stops the c-hub from sending any further messages to the trading partner associated with the shutdown c-enabler session.
- Terminates all conversations that were initiated by the trading partner.
- Causes the trading partner to leave any conversations in which it was participating.
- Reclaims resources allocated in the c-hub for that c-enabler session.

At this point, the trading partner is in a `DISCONNECTED` state in that c-space.

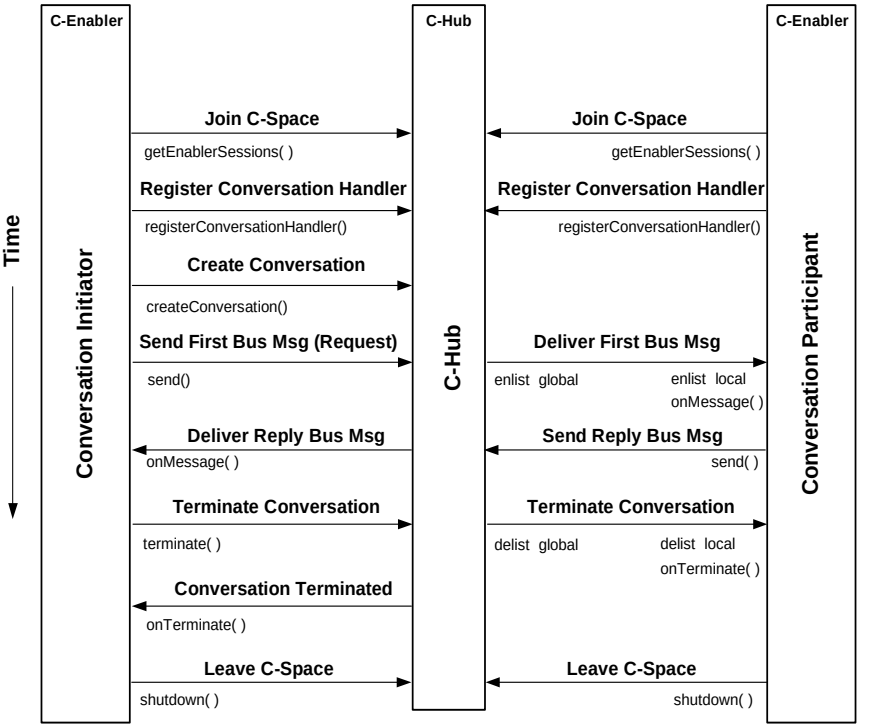
Run-Time Information Flow

At run time, all c-enabler applications perform certain tasks identically: joining a c-space, registering conversation handlers, and leaving the c-space. During individual conversations, however, conversation initiators and conversation participants perform a series of distinct, interweaving tasks.

Information Flow Diagram

The following figure shows the run-time information flow between conversation initiators and participants.

Figure 3-6 Information Flow Between Conversation Initiator and Participant



This is a simplified example that uses a single conversation and a minimal exchange of business messages (request and reply). In practice, a trading partner may participate in multiple conversations after registering a conversation handler and before leaving the c-space. In addition, within a single conversation, trading partners might exchange many business messages, not just a single request and a single reply.

Steps in the Information Flow

At run time, the flow of information between trading partners (via c-enabler applications communicating through the c-hub) proceeds in the following sequence:

1. Trading partner c-enabler applications join the c-space.
2. Each trading partner c-enabler application registers a conversation handler with the c-enabler session, which in turn (with the help of the local conversation coordinator) registers that trading partner for a given role in a given conversation at the c-hub.
3. The conversation starts when the conversation initiator c-enabler application creates a conversation.
4. The global conversation coordinator adds the conversation instance to its global conversation list and marks the trading partner as the initiator.
5. The local conversation coordinator in the conversation initiator c-enabler node adds the conversation instance to its local conversation list.
6. The conversation initiator's c-enabler application creates and sends a business message (such as a request).
7. The conversation initiator's c-enabler session delivers the business message to the c-hub.
8. The c-hub delivers the business message to the conversation participant's c-enabler node.
9. The global conversation coordinator in the c-hub enlists the participating trading partner in the conversation, adding the participating trading partner to the conversation instance entry in the global conversation list.
10. The local conversation coordinator receives the business message and enlists the trading partner in the conversation locally, adding the conversation instance to the local conversation list.
11. The `onMessage` implementation in the conversation participant c-enabler application is invoked, and the `onMessage` implementation processes the business message.
12. The conversation participant c-enabler application creates and sends a business message (such as a reply) back to the conversation initiator.

13. The c-enabler session on the conversation participant c-enabler node delivers the business message to the c-hub.
14. The c-hub receives the business message and delivers it to the conversation initiator c-enabler node.
15. The conversation initiator c-enabler node receives the business message.
16. The `onMessage` implementation in the conversation initiator c-enabler application is invoked, and the `onMessage` implementation processes the business message.
17. To end the conversation, the conversation initiator c-enabler application terminates the conversation.

    Note: A conversation might terminate automatically if the conversation timeout is exceeded.

18. The local conversation coordinator in the conversation initiator c-enabler node delivers notification of termination to the global conversation coordinator in the c-hub.
19. The global conversation coordinator in the c-hub delists the conversation participant in the global conversation list and delivers notification of termination to the local conversation coordinator on the conversation participant c-enabler node.
20. The local conversation coordinator on the conversation participant c-enabler node receives the termination notification and delists the conversation in the local conversation list.
21. The `onTerminate` implementation in the conversation participation c-enabler application is invoked.
22. The global conversation coordinator in the c-hub marks the conversation terminated and informs the conversation initiator by sending a conversation termination confirmation.
23. The conversation initiator c-enabler node receives the conversation termination confirmation.
24. The local conversation coordinator on the conversation initiator c-enabler node receives the termination notification and delists the conversation in the local conversation list.

25. The `onTerminate` implementation in the conversation initiator c-enabler application is invoked.
26. Trading partner c-enabler applications leave the c-space.

For more information about these steps, see "Key Tasks for C-Enabler Applications" on page 3-14.

Programming Steps for C-Enabler Applications

The previous section, "Run-Time Information Flow" on page 3-18, provided an end-to-end look at the flow of messages between c-enabler applications and the c-hub. The following steps describe the sequence of tasks that a developer usually provides in a c-enabler application:

- Step 1: Import Packages
- Step 2: Implement the ConversationHandler Interface
- Step 3: Create a C-Enabler Session
- Step 4: Register a Conversation Handler
- Step 5: Initiate or Participate in a Conversation
- Step 6: Exchange Business Messages
- Step 7: End the Conversation
- Step 8: Shut Down the C-Enabler Session

This section describes these steps using sample code.

Note: You must provide a c-enabler XML configuration file that contains the configuration information that the c-enabler application requires at run time. Only one c-enabler XML configuration file exists per c-enabler node.

However, the c-enabler XML configuration file can specify configuration information for multiple c-enabler sessions, one for each c-space that the associated trading partner joins.

For more information, see Configuring C-Enablers in the *BEA WebLogic Collaborate C-Enabler Administration Guide*. In addition, for help in defining the c-enabler XML configuration file, see the comments in the `EnablerConfig.dtd` file in the `dtd` subdirectory of your WebLogic Collaborate installation.

Step 1: Import Packages

C-enabler applications import the required packages from the C-Enabler Class Library. For a description of these packages, see "C-Enabler Class Library" on page 3-5.

The following listing is an example of the packages to import.

Listing 3-1 Importing Packages

```
import org.w3c.dom.*;
import org.apache.html.dom.*;
import org.apache.xml.serialize.*;
import org.apache.xerces.dom.*;

import com.bea.b2b.protocol.conversation.ConversationType;
import com.bea.b2b.enabler.*;
import com.bea.b2b.enabler.xocp.*;
import com.bea.b2b.protocol.messaging.*;
import com.bea.b2b.protocol.xocp.conversation.local.*;
import com.bea.b2b.protocol.xocp.messaging.*;

import com.bea.eci.logging.*;
```

Step 2: Implement the ConversationHandler Interface

To receive messages, a c-enabler application must implement the following interface:

`com.bea.b2b.protocol.xocp.conversation.local.ConversationHandler`

This interface provides the `onMessage` and `onTerminate` methods that are used to handle incoming business messages and conversation termination notifications, respectively. The `onMessage` method is invoked when the c-enabler receives a business message. The `onTerminate` method is invoked when the c-enabler receives a conversation termination.

The conversation handler is required in order for the trading partner to receive business messages in a conversation. A conversation handler must support at least one conversation type (`com.bea.b2b.protocol.conversation.ConversationType`), which represents a role in a conversation. A c-enabler session supports one conversation handler per conversation type.

Listing 3-2 Implementation of the ConversationHandler Interface

```
public class MyConversationHandler
    implements ConversationHandler{

   private String collaboratorId;

    MyConversationHandler(String collaboratorId){
      this.collaboratorId = collaboratorId;
    }

    public void onMessage(XOCPMessage msg){
System.out.println("onMessage: received for collaborator:" +
collaboratorId );
       Conversation conv = msg.getConversation();
     QualityOfService qos = msg.getQoS();
…
    }

    public void onTerminate(Conversation conv, int result) {
     System.out.println("onTerminate: received for collaborator:"
+ collaboratorId);
    }
}
```

For detailed information about the `ConversationHandler` interface, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

Step 3: Create a C-Enabler Session

To initiate or participate in conversations, a trading partner creates a c-enabler session on a c-enabler node. Each c-enabler session enables the trading partner to exchange messages with other trading partners in one c-space.

To create a new c-enabler session or to get an existing one, use the `com.bea.b2b.enabler.Enabler` class. The following listing is an example of getting the `session1` c-enabler session based on the information defined in the c-enabler XML configuration file. Alternatively, an application could get all the c-enabler session definitions from the c-enabler XML configuration file and then create c-enabler sessions as needed.

Listing 3-3 Obtaining the C-Enabler Session

```
Enabler enabler = Enabler.getEnabler("enabler.xml");
EnablerSession es = enabler.getEnablerSession("session1");
// Create all enabler session(s) defined in "enabler.xml"
// EnablerSession[] ess = enabler.getEnablerSessions();
// Optionally, get names of Enabler Sessions
// and use name to create enabler session individually
// String[] sessionNames = enabler.getSessionNames();
// EnablerSession es = null;
```

Step 4: Register a Conversation Handler

To participate in a conversation, a c-enabler application must register a conversation handler. A conversation handler can be associated with multiple conversation types (each type has conversation name, version and role). A conversation handler can also be shared among multiple conversations. As conversation handler is implemented by the application, and it is up to the developer to use it as needed.

To register a conversation handler, a c-enabler application calls the `registerConversationHandler` method on the `XOCPEnablerSession` instance, passing the conversation type and the conversation handler object as parameters.

The following listing is an example of registering a conversation handler for a buyer role (generally a conversation initiator) in the `BuyProcessor` conversation. Note that the specified conversation definition and role must be defined in the c-hub repository.

Listing 3-4 Registering a Conversation Handler

```
XOCPEnablerSession session = null;
if(es instanceof XOCPEnablerSession)
   session = (XOCPEnablerSession)es;
MyConversationHandler ch = new
MyConversationHandler(session.getTradingPartner());

ConversationType ctype = new ConversationType("BuyProcessor",
"1.0", "buyer");
ConversationType[] types = { ctype };
session.registerConversationHandler(types, ch);
```

Step 5: Initiate or Participate in a Conversation

A conversation initiator application explicitly starts a conversation. To initiate a conversation, the initiating trading partner calls the `createConversation` method on the `com.bea.b2b.enabler.xocp.XOCPEnablerSession` instance, passing the conversation type and, optionally, the conversation timeout value, in seconds (or zero, the default, for no timeout if the configured timeout is also zero in the conversation definition in the c-hub repository). The trading partner must be registered in the initiator role in the conversation definition.

The following listing is an example of initiating a conversation:

Listing 3-5 Initiating a Conversation

```
ConversationType ctype = new ConversationType("BuyProcessor",
"1.0", "buyer");
Conversation conv = session.createConversation(ctype, 0);
```

Step 6: Exchange Business Messages

After the conversation initiator application has created the conversation, it can begin exchanging business messages with other trading partners in the c-space.

Initially, the conversation initiator application creates and sends a business message (such as a request) to one or more trading partners in the c-space. When a trading partner receives the business message, its conversation participant application processes the business message and (usually) creates and sends a reply business message. The trading partners may send and receive several business messages in the conversation. For more information about exchanging business messages, see "Sending XOCP Business Messages" on page 3-29 and "Receiving XOCP Business Messages" on page 3-52.

Step 7: End the Conversation

A conversation can end after trading partners have finished exchanging business messages in that conversation. The way a trading partner ends involvement in a conversation depends on its role in the conversation.

Participant Leaves a Conversation

Participant trading partners can *leave* a conversation. To leave a conversation, a participant c-enabler application calls the `leave` method on the `Conversation` instance, passing `false`. No messages will be retained on the c-hub while the participant is not participating.

Note: In this release, only the `false` argument is supported.

The following listing shows an example of a participant leaving a conversation.

Listing 3-6 Leaving a Conversation

```
c.leave(false);
```

Initiator Terminates a Conversation

Conversation initiators can explicitly *terminate* the conversation or wait until the conversation times out (the conversation initiator can specify a timeout value at the time it creates the conversation, or they can specify zero to use the timeout value defined for the conversation in the c-hub repository). When a conversation terminates, the conversation initiator and all participating trading partners are delisted from the conversation, any undelivered business messages are discarded, and associated system resources are released.

To terminate a conversation explicitly, the initiating c-enabler application calls the `terminate` method in its implementation of the `Conversation` interface, as shown in the following listing.

Listing 3-7 Terminating a Conversation

```
c.terminate(Conversation.SUCCESS);
```

Step 8: Shut Down the C-Enabler Session

To shut down a c-enabler session and leave the c-space, an application uses the `shutDown` method in its implementation of the `EnablerSession` interface, always passing `false`. The following listing shows an example of shutting down a c-enabler session.

Listing 3-8 Shutting Down a C-Enabler Session

```
es.shutDown(false);
```

If a c-enabler application shuts down a c-enabler session, the trading partner leaves the c-space automatically and permanently.

Sending XOCP Business Messages

The following sections describe how a c-enabler application sends XOCP business messages to one or more trading partners in a c-space:

- Step 1: Create the Business Message
- Step 2: Specify the Recipients of the Business Message
- Step 3: Specify the Quality of Service for Message Delivery
- Step 4: Send the XOCP Business Message
- Step 5: Check the Delivery Status of the Business Message

To send an XOCP business message, a c-enabler application constructs the business document, creates the business message, specifies the message routing criteria and Quality of Service delivery options, and sends the business message to the c-hub for processing. The c-enabler application can also check the delivery status of the business

message, including whether it was successfully delivered. For an introduction to XOCP business messages, see "XOCP Business Messages and Message Envelopes" on page 3-6.

Step 1: Create the Business Message

To create a business message, a c-enabler application first creates the message payload, which consists of any business documents and attachments that the business message will contain. For an introduction to the components of a business message, see "XOCP Business Messages and Message Envelopes" on page 3-6.

Importing the Required Packages

To create a business message, a c-enabler application imports the necessary packages, as shown in the following listing.

Listing 3-9 Importing Packages for Business Message Creation

```
import org.w3c.dom.*;
import org.apache.html.dom.*;
import org.apache.xml.serialize.*;
import org.apache.xerces.dom.*;
import com.bea.b2b.protocol.conversation.ConversationType;
```

Creating Payload Parts

A c-enabler application next creates the message payload, which can include business documents and attachments.

Creating XML Documents

A business message can contain one or more business documents. A business document is the XML-based payload part of a business message. A business document is an instance of the `com.bea.b2b.protocol.messaging.BusinessDocument` class.

A `BusinessDocument` object contains an XML document, which is an instance of the `org.w3c.dom.Document` class in the `org.w3c.dom` package published by the World Wide Web Consortium (`www.w3.org`). A c-enabler application can also use a third-party implementation of that package, such as the `org.apache.xerces.dom` package provided by The Apache XML Project (`www.apache.org`), which is what the Verifier application uses to create and process XML documents.

Note: The specified document type parameters must map to a part content type of message definition associated with the conversation definition in the repository.

The following listing from the `Partner1Servlet` of the Verifier application creates a request in the form of an XML document.

Listing 3-10 Creating an XML Document

```
// Create a request document
DOMImplementationImpl domi = new DOMImplementationImpl();
DocumentType dType = domi.createDocumentType("request", null,
"request.dtd");
org.w3c.dom.Document rq = new DocumentImpl(dType);
Element root = rq.createElement("request");
// the actual string data to be processed by the other partner
String sendStr = "ABCDEFGHI";
root.appendChild(rq.createTextNode(sendStr));
rq.appendChild(root);
```

After creating the XML document, a c-enabler application creates a `BusinessDocument` object, passing the XML document (request) as a parameter to the constructor, as shown in the following listing.

Listing 3-11 Creating a BusinessDocument

```
BusinessDocument bdoc = new BusinessDocument(rq);
```

Creating Attachments

A business message can contain one or more attachments. An attachment is a non-XML-based payload part of a business message that contains binary content. An attachment is an instance of the `com.bea.b2b.protocol.messaging.Attachment` class. For more information, see the WebLogic Collaborate Javadoc.

The following listing shows creating an attachment.

Listing 3-12 Creating an Attachment

```
FileInputStream fis = new FileInputStream("somefile");
Attachment att = new Attachment (fis);
```

Creating the XOCP Business Message and Adding Payload Parts

After creating the message payload, a c-enabler application creates the XOCP business message and adds the payload parts to it. The `com.bea.b2b.protocol.xocp.messaging.XOCPMessage` class represents an XOCP business message. For more information, see the WebLogic Collaborate Javadoc.

To construct the business message, a c-enabler application:

1. Creates an instance of the `XOCPMessage` class.
2. Adds the payload parts to the business message by calling either of the following methods on the `XOCPMessage` message object:
   - `addPayLoadPart` adds a single business document or attachment to the business message.
   - `addPayLoadParts` adds multiple business documents or attachments to the business message.

The following listing creates an XOCP business message and adds payload parts to it.

Listing 3-13 Creating a Business Message and Adding Payload Parts

```
XOCPMessage smsg = new XOCPMessage("");
smsg.addPayloadPart(bdoc);
smsg.addPayloadPart(att);
```

Note: The c-Enabler application clones `XOCPMessage` content (except its payload parts) before sending it to the c-hub. Therefore, a payload part must not be changed after invoking the `send` or `sendAndWait` methods on the `XOCPMessage`.

Step 2: Specify the Recipients of the Business Message

After creating a business message, a c-enabler application optionally specifies the trading partner to which it will be sent. A c-enabler application might send the business message to a specific trading partner (a point-to-point exchange), such as when it replies to a request received from a conversation initiator. Alternatively, a c-enabler application might send the business message to a set of trading partners (multicasting) based on business criteria (c-enabler XPath expressions), such as when a buyer sends a bid request to multiple sellers of a particular product.

Either way, the set of eligible trading partners is constrained by those who are subscribed to the appropriate role in the conversation definition. In addition, router and filter expressions defined in the c-hub repository may also affect message delivery to particular trading partners. For more information, see Routing and Filtering XOCP Business Messages in the *BEA WebLogic Collaborate C-Hub Administration Guide*.

Specifying a Particular Trading Partner

If an XOCP business message is being sent to a single, known trading partner, a c-enabler application can call the `setRecipient` method on the `XOCPMessage` object, passing the trading partner name as the parameter. The specified trading partner must be defined in the c-hub repository.

The following listing shows specifying the trading partner named `ChipMaker` as the recipient of the business message.

Listing 3-14 Specifying a Single Trading Partner

```
String tradingPartnerName = "ChipMaker";
XOCPMessage msg = new XOCPMessage();
msg.setRecipient(tradingPartnerName);
```

Using `setRecipient` for a business message expedites message delivery because the c-hub does not perform the usual router processing, such as evaluating trading partner or c-hub XPath expressions. However, the business message is still subject to applicable filtering in the c-hub. For more information, see Routing and Filtering XOCP Business Messages in the *BEA* WebLogic Collaborate *C-Hub Administration Guide.*

Using C-Enabler XPath Expressions to Specify Message Recipient Criteria

A c-enabler application can use XPath expressions to specify the criteria for the set of trading partners that are to receive the business message. C-enabler XPath expressions are used to address parts of an XML document. For more information, see Routing and Filtering XOCP Business Messages in the *BEA WebLogic Collaborate C-Hub Administration Guide*.

The XPath expression should be specific to the document format of the c-hub repository and should define a node set of trading-partner elements. The XPath expression selects recipient trading partners based on the following attributes, which are defined in the c-hub repository:

- Standard attributes, such the trading partner name or a postal code
- Extended properties, which are custom attributes, elements, and text defined by the c-hub administrator

The XPath expression is passed as part of the message header in the business message from the c-enabler to the c-hub. The c-hub uses this XPath expression, along with other XPath expressions defined in the c-hub repository, to determine the set of message recipients for the business message.

If applicable trading partner and c-hub XPath expressions are defined in the c-hub repository, the c-hub evaluates these expressions after it receives the business message. Depending on how they are configured, these XPath expressions might override or append the c-enabler XPath expression that the c-enabler application specifies. For more information, see Routing and Filtering XOCP Business Messages in the *BEA WebLogic Collaborate C-Hub Administration Guide*.

To specify a c-enabler XPath expression for an XOCP business message, the c-enabler application calls the `setExpression` method on the `XOCPMessage` object, passing the XPath expression as the parameter.

Notes: The version of Apache Xalan (v 1.0.1) supports single quotes, but not double quotes, to delimit string literals.

Before the business message is delivered, it is still subject to applicable router and filter processing in the c-hub.

Specifying Standard Trading Partner Attributes

The following listing shows a c-enabler XPath expression that selects the trading partner with the specified name:

Listing 3-15 C-Enabler XPath Expression Specifying a Trading Partner Name

```
msg.setExpression("//trading-partner[@name=\'"+
tradingPartnerName+"\']")
```

The following listing shows a c-enabler XPath expression that selects the trading partner whose address contains the string “San”:

Listing 3-16 C-Enabler XPath Expression Specifying a Trading Partner Name

```
msg.setExpression("//trading-partner[contains(address,\'San\')]")
;
```

Specifying a C-Enabler XPath Expression Using Extended Properties

Extended properties are user-defined elements, attributes, and text that can be associated with trading partners in the c-hub repository. These properties provide application extensions to the standard predefined attributes in the repository. The extended property sets are modeled in the repository so that they can be retrieved as subtrees within an XML document. Extended properties are configured in the Trading Partners tab in the C-Hub Administration Console. For more information, see Using the C-Hub Administration Console in the *BEA WebLogic Collaborate C-Hub Administration Console*.

C-enabler XPath expressions can refer to these extended properties to assist with business message routing. For example, suppose a c-hub administrator added an extended property called "Maximum Order Quantity" so that sellers could indicate in the c-hub repository the largest quantity that they could accommodate. With this property defined, a buyer with a large order could specify a c-enabler XPath expression that sends the business message only to the sellers that can process the order.

The following code listing shows an XML document generated from the repository with an extended property set for a given seller:

Listing 3-17 Extended Property Set in XML Document Generated from the Repository

```
<c-hub context="message-router">
…
<trading-partner name="ABC Seller"
email="orderprocessing@somedomain.com"
phone="999-999-9999">
<address>123 Main St., San Jose, CA 95131</address>
<extended-property-set name="Capacity">
       <max-order-quantity>1000</max-order-quantity>
</extended-property-set>
</trading-partner>
…
</c-hub>
```

The following listing shows a c-enabler XPath expression that selects trading partners that can accommodate orders larger than 500 units:

Listing 3-18 C-Enabler XPath Expression Specifying an Order Size

```
msg.setExpression("//trading-partner[extended-property-set/(@max-
order-qty > \'500\')]")
```

Because the seller can accommodate orders of up to 1000 units, the seller would be selected as a recipient of this business message.

Step 3: Specify the Quality of Service for Message Delivery

The WebLogic Collaborate messaging system allows c-enabler applications to define the *Quality of Service* (QoS), or level of reliability, to use when delivering a business message to recipient trading partners. The Quality of Service settings are stored in the message header of the business message. The messaging system supports the reliable delivery of messages in the event of network-link or node failures. The messaging system provides other capabilities to support reliable messaging, such as message logging and tracking, correlation of messages, delivery retry attempts, message timeouts, and choice of message delivery methods.

Automatic Quality of Service Features

The WebLogic Collaborate messaging system provides certain automatic Quality of Service features that do not require input from c-enabler applications:

- WebLogic Collaborate prevents duplicate message delivery.
- WebLogic Collaborate time stamps every business message when it arrives at the c-hub or a c-enabler node, which helps with taking performance measurements and with application debugging.

QualityOfService Class

The `com.bea.b2b.protocol.xocp.messaging.QualityOfService` class represents Quality of Service settings for business messages. The `QualityOfService` class defines the quality of service required from the WebLogic Collaborate messaging system to deliver a specific message and it identifies to the WebLogic Collaborate messaging system the c-enabler application's expectation for delivering the business message. A c-enabler application creates an instance of this class, calls methods on this instance to specify various Quality of Service settings, and then calls the `setQoS` method on the message instance, passing the `QualityOfService` object, to associate the settings with the message. If a c-enabler application does not specify Quality of Service settings, the WebLogic Collaborate messaging system uses the default values where applicable.

Quality of Service Settings, Options, and Default Values

The following table describes the available Quality of Service settings, options, and default values.

Table 3-5 Quality of Service Settings, Options, and Default Values

QoS Setting / Description	Options	Default Value(s)
`CONFIRMED_DELIVERY_TO_DESTINATION(S)` ■ Provides the complete delivery status from each destination, including receipt timestamp, router selected trading partners, final list of recipient trading partners, and so on. ■ Provides complete message tracking information (all potential locations) for the c-hub administrator and the sending c-enabler's administrator.	Not applicable	Not applicable
`CONFIRMED_ROUTING` ■ Provides information from the c-hub router about the trading partners selected to receive the business message. ■ Provides message tracking for the sending c-enabler's administrator (until the business message reaches the c-hub router),	Not applicable	Not applicable

Table 3-5 Quality of Service Settings, Options, and Default Values (Continued)

QoS Setting / Description	Options	Default Value(s)
CONFIRMED_DELIVERY_TO_HUB (Default) ■ Message reached the c-hub ■ No message tracking for sending c-enabler's administrator	Not applicable	Not applicable
DURABILITY	■ PERSISTENT ■ NON-PERSISTENT	NON-PERSISTENT
TIMEOUT	Timeout, in milliseconds, after send	Ignored
RETRY_ATTEMPTS	0-n	As defined in the c-hub configuration
CORRELATION_ID	Application-defined field	Ignored

How Quality of Service Settings Affect Message Tracking and Delivery Acknowledgments

The following table describes how the Quality of Service setting affects message tracking and delivery acknowledgments.

Table 3-6 QoS, Acknowledgment, and Message Tracking

Quality of Service Setting	Message Tracking (Y/N)?	Acknowledgment (Y/N)?
Confirmed Delivery to Destination(s)	Y	Y
Confirmed Delivery To Router	Y	N
Confirmed Delivery To C-Hub	N	N

If the Confirmed Delivery to Destination(s) setting is used, then complete message tracking is available and acknowledgments are used to reliably deliver the message to its destination(s). If the Confirmed Delivery to Hub setting is used, then no message tracking is available and no acknowledgments are sent from recipient trading partners..

Code Example

The following listing is an example of setting the Quality of Service for a business message.

Listing 3-19 Setting the Quality of Service for a Business Message

```
// Relevant imports
import com.bea.b2b.protocol.messaging.MessageToken;
import com.bea.b2b.protocol.messaging.DeliveryStatus;
import com.bea.b2b.protocol.messaging.BusinessDocument;
import com.bea.b2b.protocol.xocp.conversation.local.*;
import com.bea.b2b.protocol.xocp.messaging.*;
import com.bea.b2b.enabler.*;
import com.bea.b2b.enabler.xocp.*;

XOCPMessage msg = …
// Create QoS object
QualityOfService qos = new QualityOfService();
// Specify message to be persisted
qos.setPersistent(true);
// Specify confirmed delivery to destination(s)
qos.setConfirmedDeliveryToDestination(true);
msg.setQoS(qos);
```

Setting the Message Delivery Confirmation Level

To specify the level of message delivery confirmation, a c-enabler application calls one of the following methods on the `QualityOfService` instance, passing the Boolean `true` parameter to enable that option:

Table 3-7 Message Delivery Confirmation Levels

Durability	Description
`setConfirmedDeliveryToDestination`	Specifies whether to confirm message delivery up to its destination (true) or only up to the c-hub (false).
`setConfirmedDeliveryToHub`	Specifies whether to confirm message delivery up to the c-hub (true) or not (false).
`setConfirmedDeliveryToRouter`	Specifies whether to confirm message delivery up to the router in the c-hub (true) or only up to the c-hub (false).

The following listing is an example of setting the message confirmation level up to its destination.

Listing 3-20 Setting the Message Delivery Confirmation Level

```
qos.setConfirmedDeliveryToDestination(true);
```

For more information about confirming message delivery, see “Step 5: Check the Delivery Status of the Business Message” on page 3-47.

Setting Message Durability

In the WebLogic Collaborate messaging system, message durability is a Quality of Service option that determines whether a durable message store is used in order to guarantee delivery of message in case of node failures.

Message Durability Options

A c-enabler application has two message durability options, non-persistent (the default) and persistent, as described in the following table.

Table 3-8 Message Durability Options

Durability	Description
Non-persistent	For non-persistent QoS, the message is not stored anywhere in a durable data store in the WebLogic Collaborate system in due process of delivery to its destination. A non-persistent business message en route to its destination is not recoverable in case of whole or partial system failures. Using this option requires less system resources and improves throughput.
Persistent	For persistent QoS, message is persisted to a durable data store in due process of delivery to its destination. This quality of service increases the guarantee of delivery, as the message is stored, in reliable data store. The message delivery guarantee increases at the expense of throughput of the system. Such a message travels slower in the system and consumes more resources. The message is persisted to a data store chosen by the owner of the WebLogic Collaborate component or serialized to a file on disk based on size of the message.

Message and Conversation Durability

A c-enabler application can specify message durability on a per message basis. In addition, message durability can be defined on a per conversation basis in the c-hub repository.

How business messages are persisted on a per message or a per conversation basis depends on a combination of whether persistence is enabled or disabled in the c-hub, the conversation, or the message, as shown in the following table.

Table 3-9 Message Persistence

Persistent Object	Persistence Enabled?				
If persistence is enabled (Y) or disabled (N) for:					
■ C-Hub	Y	Y	Y	Y	N
■ Conversation	Y	N	Y	N	Y/N
■ Business Message	Y	N	N	Y	Y/N
Then the conversation or business message is persisted (Y) or not persisted (N):	**Persisted?**				
■ Conversation	Y	N	Y	N	N
■ Business Message	Y	N	Y	N	N

A business message is considered persistent if persistence (recovery) is enabled in the c-hub, if the conversation in which message propagates is persistent, and if the message QoS indicates persistence. Even if persistence is enabled for conversations or messages, if persistence is *not* enabled in the c-hub, then no conversations or messages are stored to a reliable data store.

Specifying Message Persistence

To enable message persistence, a c-enabler application calls the `setPersistent` method on the `QualityOfService` instance, passing the Boolean `true` parameter, as shown in the following listing.

Listing 3-21 Specifying Message Persistence

```
qos.setPersistent(true);
```

Setting the Message Timeout

If specified, the message timeout determines how long a sender waits for acknowledgments. If a business message expires (times out), the receiver of the business message does not process it, and all other processing of the business message, including acknowledgment processing and delivery retries, is abandoned.

Timeout Algorithm

WebLogic Collaborate does not synchronize the clocks used by its different components, which can reside in different machines at different locations. Instead, WebLogic Collaborate uses a relative time algorithm.

Based on this algorithm, the time left before the timeout of a business message (relative to the absolute time of the component processing the business message) is included in the business message when the business message is sent to the other component. On the receiving component, the timeout calculations are based on an absolute time (at the arrival of the business message) and a relative time (embedded in the incoming message) left to process the message. This algorithm at least ensures that the actual message timeout in the system will always occur after the original timeout specified by the application.

```
Message Timeout on the C-Hub = Message timeout specified by the
c-enabler application when sending a message
```

```
Message Timeout on the Sending C-Enabler = Message Timeout on the
C-Hub + N x Delta
```

Where

- *N* = a predefined number in the system, such as `10`
- *Delta* = Estimated round-trip time between the sending c-enabler and the c-hub

Setting the Number of Delivery Retry Attempts

If an attempt to deliver a business message fails due to intermittent network failures, the WebLogic Collaborate messaging system attempts to retry sending the business message repeatedly until one of the following occurs:

- The business message is delivered (delivery succeeded).
- The number of retry attempts is exceeded.

- The message times out.
- The conversation in which the business message is sent either terminates or times out.

The default values for message timeouts and retry intervals are defined in the c-hub repository and are retrieved by a c-enabler at the time the c-enabler session is created. The WebLogic Collaborate messaging system waits for the configured interval before attempting to resend a business message.

To override the default retry attempt limit, a c-enabler application calls the `setTimeout` method on the `QualityOfService` instance, passing the timeout value (number of milliseconds) as a parameter, as shown in the following listing.

Listing 3-22 Specifying the Message Timeout

```
qos.setTimeout(10000);
```

Setting the Correlation ID for a Business Message

A c-enabler application can specify a unique correlation ID for a business message so that it can correlate received business messages (such as replies to a request) from trading partners to a previously sent message (such as a request). The correlation ID accompanies the business message to its destination. The c-enabler application of the recipient trading partner can use this value to unambiguously identify the reply message sent back to the originating trading partner.

To specify the correlation ID, a c-enabler application calls the `setCorrelationId` method on the `QualityOfService` instance, passing a string representing the correlation ID as a parameter, as shown in the following listing.

Listing 3-23 Specifying the Correlation ID for a Business Message

```
qos.setCorrelationId("ABC123");
```

Step 4: Send the XOCP Business Message

After specifying the recipients of a business message and the Quality of Service, a c-enabler application sends the business message in one of the following ways:

- Synchronous message delivery
- Deferred synchronous message delivery

Synchronous Message Delivery

With synchronous message delivery, the application waits until the sent message is delivered to the destination(s). The WebLogic Collaborate messaging system returns control to the application once the outcome of the activity of sending the message is known. The application waits until any of the following events occurs:

- Acknowledgments are received from all potential destinations
- Message times out
- Conversation in which message was sent terminates

To send a business message synchronously, a c-enabler application calls the `sendAndWait` method on the `XOCPMessage` instance, passing the time to wait (number of milliseconds) as a parameter. If zero (0) is specified, the c-enabler application waits until the business message reaches its destination(s), as shown in the following listing.

Listing 3-24 Sending a Message Using Synchronous Message Delivery

```
MessageToken token = msg.sendAndWait(0);
```

Deferred Synchronous Message Delivery

With deferred synchronous message delivery, the WebLogic Collaborate messaging system returns control to the c-enabler application immediately after a message is sent, and returns a message token that the c-enabler application can use to check the status of message delivery. Once a message token is accessed, the application waits for a specified time or until any of the following events occurs:

- Acknowledgments are received from all potential destinations.
- The message times out.
- The conversation in which message was sent either terminates or times out.

To send a business message with deferred synchronous message delivery, a c-enabler application calls the `send` method on the `XOCPMessage` instance, continues executing business logic, and then checks the status by calling the `waitForACK` method on the `MessageToken` instance, as shown in the following listing.

Listing 3-25 Sending a Message Using Deferred Synchronous Message Delivery

```
token = msg.send();
...
token.waitForACK();
```

The `waitForAck` method will block until the status of the business message is available (if no timeout is specified) or until the specified timeout (in milliseconds) is exceeded.

Step 5: Check the Delivery Status of the Business Message

Both the `send` and `sendAndWait` methods on the `XOCPMessage` instance return a message token that a c-enabler application can query to check the delivery status of the associated business message.

Message Tokens

A message token is an instance of the `com.bea.b2b.protocol.xocp.messaging.XOCPMessageToken` class. A message token has the following attributes:

Table 3-10 Message Token Information

Attribute	Description
Message ID	Unique ID of the business message.
Exception	If applicable, any exception that occurred before the business message left the sending c-enabler. An exception is usually returned when the message is sent, but for deferred synchronous message delivery, the business message might be kept in an internal send queue temporarily before being delivered to the c-hub.
Elapsed Time	Time taken to deliver the business message to all destination(s). This information is available only after acknowledgments have been received from all message destinations. Availability is subject to the specified Qualify of Service delivery option.
Delivery Status	Delivery status from recipient destination(s). This information depends on the availability of such information. Availability is subject to the specified Qualify of Service delivery option.
Number of Recipients (Router)	Number of recipient trading partners after the business message has been processed by the XOCP router in the c-hub. Availability is subject to the specified Qualify of Service delivery option.
Number of Recipients (Filter)	Number of recipient trading partners after the business message has been processed by the XOCP filter in the c-hub. Availability is subject to the specified Qualify of Service delivery option.

If the business message was sent using the synchronous send delivery option, then the message token cannot be used to wait for acknowledgments and, if used, the method returns immediately.

Delivery Status Tracking

In the WebLogic Collaborate messaging system, when a business message reaches its destination (the receive queue of the destination c-enabler node), a system message is returned to the sender to acknowledge the message delivery if the Quality of Service setting requires it.

A c-enabler application can use either of the following methods to obtain the delivery status:

- `getAllDeliveryStatus` if the business message was sent to multiple recipients
- `getDeliveryStatus` if the business message was sent to a single recipient

Both methods return a `DeliveryStatus` object, an instance of the `com.bea.b2b.protocol.messaging.DeliveryStatus` class that provides the following information:

- Recipient (name of the recipient trading partner or message tracking location)
- Timestamp of the receipt of the business message
- Status code, which is one of the following values.

Table 3-11 Message Delivery Status Codes

Status Code	Description
`SUCCESS`	Business message was successfully delivered to the destination. No errors or exceptions occurred.
`FAILURE`	An error occurred while delivering the business message to this destination.
`RETRIES_EXHAUSTED`	All delivery retry attempts have been exhausted and the business message has been discarded.
`TIMEDOUT`	Timeout occurred before message delivery and the business message has been discarded.

Message Tracking Locations

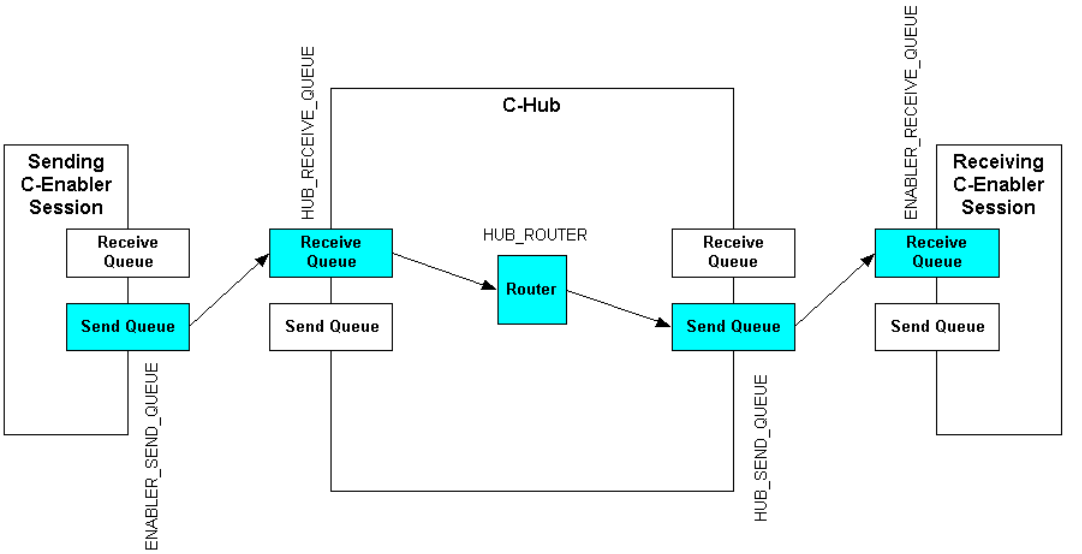
The WebLogic Collaborate messaging system provides message tracking features that allow c-hub and c-enabler administrators to check the progress of a business message as it moves through various predefined message tracking locations along the message path en route to its destination. The C-Hub Administration Console and the C-Enabler Administration Console can display status information if a business message passes through these tracking points. Administrators can use message tracking information for debugging and to identify bottlenecks in applications.

Note: The availability of message tracking locations depends on the configuration of the WebLogic Collaborate system and the specified Quality of Service for a given business message (such as `CONFIRMED_DELIVERY_TO_DESTINATION(S)`, which is described in Table 3-5). For example, if the c-enabler and c-hub are colocated on the same node, some locations are not available. Similarly, some locations may not be available for synchronous message delivery.

Diagram of Message Tracking Locations

The following figure shows the message tracking locations in the WebLogic Collaborate messaging system.

Figure 3-7 Message Tracking Locations



Description of Message Tracking Locations

The following message tracking locations are potentially visible in the C-Hub Administration Console or the C-Enabler Administration Console.

Table 3-12 Message Tracking Locations

Message Tracking Locations	Location	Activity Performed
ENABLER_SEND_QUEUE	Send queue in the c-enabler session of the sending trading partner.	Message is enqueued for sending.
HUB_RECEIVE_QUEUE	Receive queue for the sending trading partner in the c-hub.	Message is enqueued in the receive queue of the trading partner at the c-hub.

Table 3-12 Message Tracking Locations (Continued)

Message Tracking Locations	Location	Activity Performed
HUB_ROUTER	XOCP-Router in the c-hub.	Message has reached the router.
HUB_SEND_QUEUE	Send queue of the receiving trading partner in the c-hub.	Message has been enqueued for delivery in the target trading partner's queue at the c-hub.
ENABLER_RECEIVE_QUEUE	Receive queue in the c-enabler session of the receiving trading partner.	Message has been enqueued in queue of the listener thread of the target trading partner's c-enabler session.

Receiving XOCP Business Messages

The following sections describe how to receive XOCP business messages in a c-enabler application:

- About Receiving XOCP Business Messages
- Receiving an XOCP Business Message

About Receiving XOCP Business Messages

C-enabler applications must implement the `onMessage` method in the `ConversationHandler` interface to receive and process business messages. The `onMessage` method has the following signature.

Listing 3-26 Signature for onMessage Method

```
public void onMessage(XOCPMessage msg)
```

The c-enabler session invokes the `onMessage` method whenever a c-enabler receives a business message, passing the business message as an input parameter. The c-enabler application retrieves the `XOCPMessage` object containing the business message and then calls methods on that instance to process the message.

If a c-enabler application receives multiple business documents in a conversation, the `onMessage` implementation would first determine the type of document received (such as a bid request or bid reward), and then process that document accordingly.

In addition, the `onMessage` implementation might contain code that constructs and sends a business message. For example, a conversation participant c-enabler application might implement `onMessage` to receive a request, process the request, and then create and send the reply document.

Receiving an XOCP Business Message

Listing 3-27 describes the `onMessage` implementation in the `Partner2Servlet` of the Verifier application. This `onMessage` implementation processes the initial business document (a request) sent from the `Partner1Servlet`. It then creates and sends a reply document back to the Partner1 node.

Tasks Performed

The `onMessage` code performs the following tasks:

1. Retrieves the Quality of Service for the business message by calling the `getQoS` method on the `XOCPMessage` instance.

   The application will use the same Quality of Service settings to send the reply message.

2. Retrieves the payload parts of the business message by calling the `getPayloadParts` method on the `XOCPMessage` instance.

3. Retrieves the first (and only) business document in the `PayloadPart[]` array.

4. Extracts the associated XML document by calling the `getDocument` method on the `BusinessDocument` instance.

5. Retrieves and examines parts of the XML document using methods on the `Document` instance, which is an instance of the `org.w3c.dom.Document` class provided in the `org.w3c.dom` package published by the World Wide Web Consortium (www.w3.org).

   A c-enabler application can also use a third party implementation of that package, such as the `org.apache.xerces.dom` package provided by The Apache XML Project (www.apache.org), which is what the Verifier application uses to create and process business documents.

6. Retrieves the data string (`"ABCDEFGHI"`) embedded in the business document and converts it to all lowercase letters.

7. Constructs a reply document, specifies the same Quality of Service as the request document, and sends the document to Trading Partner 1.

Code Listing

The following listing is the `onMessage` implementation in the `Partner2Servlet` of the Verifier application.

Listing 3-27 The onMessage Implementation in Partner2Servlet

```
public void onMessage(XOCPMessage rmsg) {
      try{
        QualityOfService qos = rmsg.getQoS();

        PayloadPart[] payload = rmsg.getPayloadParts();
        Document rq = null;

        if (payload != null && payload.length > 0){
          BusinessDocument bd = (BusinessDocument)payload[0];
          rq = bd.getDocument();
        }
        if (rq == null){
          throw new Exception("Did not get a request document");
        }
        Conversation conv = rmsg.getConversation();

        Element root = rq.getDocumentElement();
        String name = root.getNodeName();
        if (!name.equals("request")){
          debug("Received "+name+" instead of a request");
          return;
```

```
    }
    Text revStr = (Text)root.getFirstChild();

    // Create the return document
    DOMImplementationImpl domi = new DOMImplementationImpl();
   DocumentType dType = domi.createDocumentType("reply", null, "reply.dtd");
    rq = new DocumentImpl(dType);
    root = rq.createElement("reply");
    String sendStr = new String(revStr.getData());
    root.appendChild(rq.createTextNode(sendStr.toLowerCase()));
    rq.appendChild(root);

    XOCPMessage smsg = new XOCPMessage("");
    smsg.addPayloadPart(new BusinessDocument(rq));
    smsg.setQoS(qos);
    smsg.setExpression("//trading-partner[@name=\'Partner1\']");

    smsg.setConversation(conv);
    smsg.sendAndWait(0);

  }catch(Exception e){
    e.printStackTrace();
  }
}
```

CHAPTER

4 Developing Logic Plug-Ins

The following sections describe how to develop logic plug-ins in WebLogic Collaborate:

- About Logic Plug-Ins
- Logic Plug-In API
- Rules and Guidelines for Logic Plug-Ins
- Creating and Adding Logic Plug-Ins

For sample applications that demonstrate the use of logic plug-ins, see the MessageCounter and CheckAccount logic plug-in samples, which are described in Using Logic Plug-Ins for Billing in *BEA WebLogic Collaborate Getting Started*.

About Logic Plug-Ins

The following sections describe logic plug-ins and related concepts:

- What Are Logic Plug-Ins?
- Logic Plug-In Architecture
- Chains
- Business Messages and Message Envelopes
- System and Custom Logic Plug-Ins
- Creating and Adding Logic Plug-Ins
- Creating and Adding Logic Plug-Ins

What Are Logic Plug-Ins?

Logic plug-ins are individual components that perform specialized processing of business messages that pass through the c-hub. A logic plug-in is a custom service that a c-hub provider or trading partner can develop and install on a c-hub to provide additional value for c-hub management and for trading partners who use that c-hub.

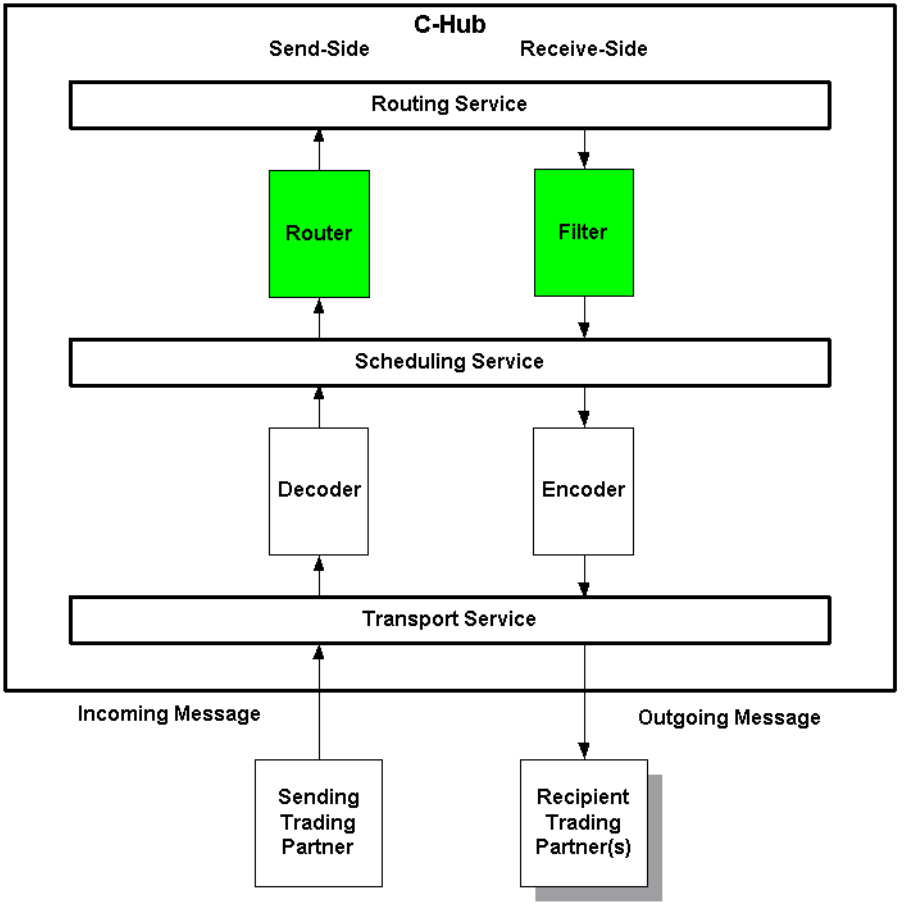
Logic plug-ins insert rules and business logic at strategic locations along the path that business messages travel as they make their way through the c-hub. Logic plug-ins are Java classes that are invoked when a c-space is started. At run time, logic plug-ins can intercept, process, and output business messages. When a business message passes through the location where a logic plug-in is configured, the logic plug-in processes the business message. Logic plug-in execution occurs on the c-hub and is transparent to c-enabler users.

Each logic plug-in is associated with a business protocol. A logic plug-in processes only the messages that are exchanged using that protocol. For example, if a particular plug-in is associated with the XOCP protocol, then it will process only XOCP business messages.

Logic Plug-In Architecture

Logic plug-ins can be installed at two processing locations in the c-hub: in the router and in the filter, which are shown in the following figure.

Figure 4-1 Logic Plug-In Locations in the C-Hub: Router and Filter



Logic Plug-In Processing Tasks

WebLogic Collaborate-provided XOCP router and XOCP filter plug-ins, as well as RosettaNet plug-ins, are directly involved in the processing of message recipients based on Xpath expressions in the repository. However, custom logic plug-ins can perform a wide range of services that are entirely unrelated to routing or filtering, as well as performing routing and filtering operations. For example, a custom logic plug-in might track the number of messages sent from each trading partner for billing purposes.

Logic plug-ins perform the types of tasks described in the following table.

Table 4-1 Tasks That Logic Plug-Ins Perform

Process	Description	Examples
Route Modification	Changes the list of target recipients for a business message. Subject to conversation and c-space validation of the recipient. (WebLogic Collaborate plug-ins and custom plug-ins.)	■ "If a computer chip order over $1M is placed, make sure that NewChipCo is one of the recipients." ■ "After January 1, 2000, no orders should be sent to OldChipCo."
Examination	Examines the contents of a business message and takes certain actions based on the results of the examination. (Custom plug-ins.) **Note:** Examination is usually performed on business messages *without* encrypted contents.	■ "Log all senders of messages for billing purposes." ■ "Sample 1 out of every N messages of type X for standards compliance." ■ "For messages of type X, how many are conversation version 1 versus conversation version 2?"
Content Modification	Changes the contents of a business message. **Note:** Content modification is *not* allowed in this release.	■ "If over N items are ordered, be sure to specify extra insurance."

Chains

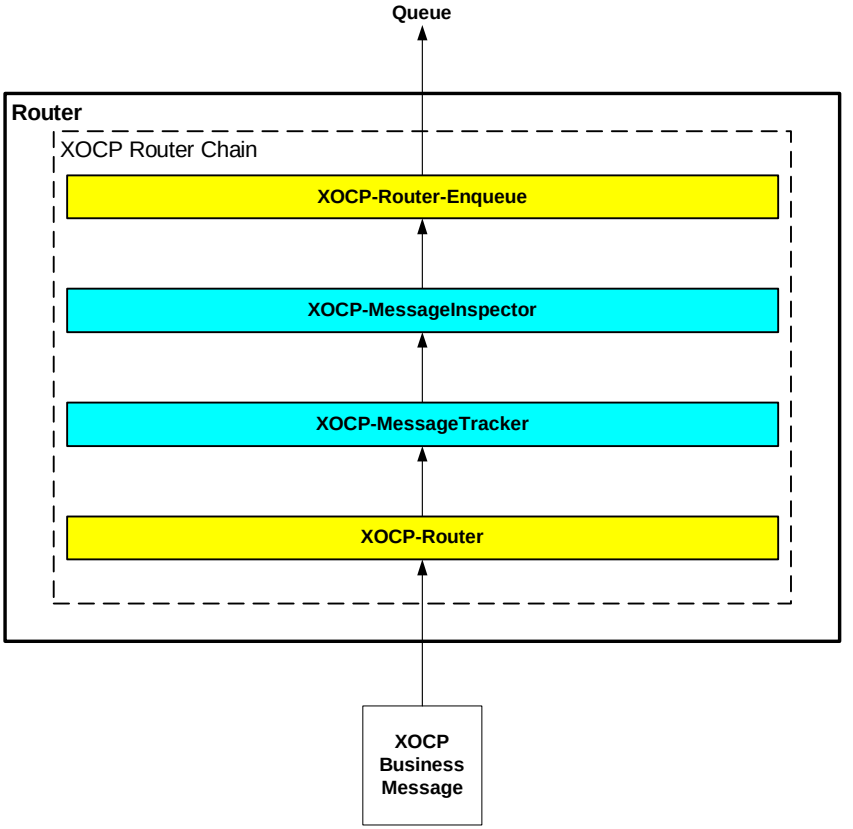
Both the router and filter modules can have multiple plug-ins that will be executed when a business message passes through that part of the c-hub. Multiple logic plug-ins that share the same protocol are sequenced as a logic plug-in *chain*.

In a chain, the logic plug-ins are processed sequentially at run time. After one plug-in has finished executing, the next sequential plug-in in the chain will normally be activated. Each successive plug-in can access any changes made previously to the shared message information as the business message passes throughout the c-hub.

Note: The position of a logic plug-in in a chain is configured in the repository using the C-Hub Administration Console, as described in Working with Logic Plug-ins in the *BEA WebLogic Collaborate C-Hub Administration Guide*.

The following figure shows an example of a chain of XOCP logic plug-ins in the router location in the c-hub.

Figure 4-2 Sample XOCP Router Chain



Note that even when custom logic plug-ins do not provide routing or filtering capability, they must still be part of an XOCP or RosettaNet router or filter chain. In this example, the chain contains four logic plug-ins that are processed in the order described in the following table.

Table 4-2 Logic Plug-Ins in the Sample XOCP Router Chain

Logic Plug-In	Description
XOCP router	WebLogic Collaborate provides this logic plug-in, which might modify the list of recipients for an XOCP business message based on XPath router expressions configured in the repository. This should be the first logic plug-in in the XOCP router chain.
XOCP-MessageTracker	Hypothetical logic plug-in. A c-hub owner or trading partner might provide such a custom logic plug-in to track the number of business messages sent from each trading partner for billing purposes.
XOCP-MessageInspector	Hypothetical logic plug-in. A c-hub owner or trading partner might provide such a custom logic plug-in to examine and maintain statistics for the types of business documents being exchanged on the c-hub (for example, purchase orders, invoices, and so on).
XOCP router enqueue	WebLogic Collaborate provides this logic plug-in, which enqueues the XOCP business message in an internal c-hub router message queue. This should be the last logic plug-in in the XOCP router chain.

In this example, only XOCP business messages will trigger the logic plug-ins in the XOCP router chain. Non-XOCP business messages (such as RosettaNet messages) are processed separately by the router chain associated with the protocol/c-space defined by the URL that received the business message.

Business Messages and Message Envelopes

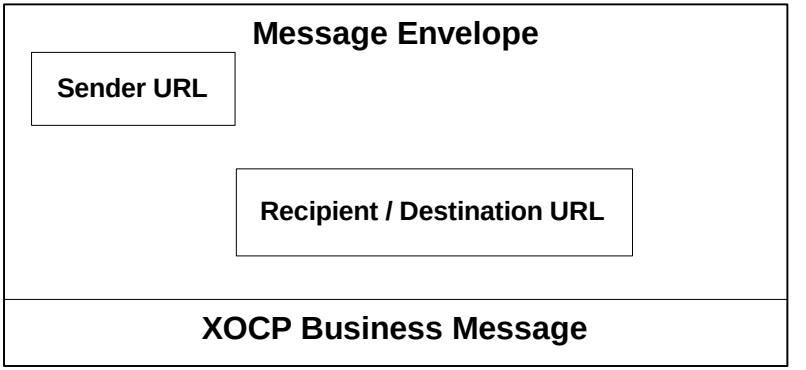
A *business message* is the basic unit of communication exchanged between trading partners in a conversation. The business message contains the list of message recipients. A business message is represented in the C-Hub API by the `com.bea.b2b.protocol.messaging.Message` interface. In addition, the following classes implement this interface and represent protocol-specific business messages:

- `com.bea.b2b.protocol.xocp.messaging.XOCPMessage`
- `com.bea.b2b.protocol.rosettanet.messaging.RNMessage`

When a business message enters the c-hub, the c-hub creates a *message envelope* that acts as a container for the business message as it travels through the c-hub. Message envelopes are instances of the `com.bea.b2b.protocol.messaging.MessageEnvelope` class.

The message envelope is used for routing purposes and is analogous to a paper envelope for a letter: the message envelope contains the business message plus addressing information, such as the identity of the sender (return address) and one recipient of the business message (destination address), as shown in the following figure.

Figure 4-3 Message Envelope Containing an XOCP Business Message



Message envelopes also contain other information about the business message. For detailed information about the `MessageEnvelope` class, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

For XOCP business messages, after the system XOCP router processes an XOCP business message and finalizes the list of intended message recipients, the c-hub validates the recipients and creates a separate message envelope (and a logical copy of the XOCP business message) for each recipient in the recipient list. These message envelopes are then forwarded to the XOCP filter for processing. For more information, see “Information Flow for Message Envelopes” on page 3-9.

System and Custom Logic Plug-Ins

WebLogic Collaborate provides the following logic plug-ins to provide standard services for processing business messages.

Table 4-3 System Logic Plug-Ins

Logic Plug-In	Description
XOCP router	Modifies the list of recipients for an XOCP business message based on XPATH router expressions configured in the repository. In general, this system logic plug-in should be first in the router chain so that custom logic plug-ins can subsequently process a business message after its list of intended recipients is known.
XOCP router enqueue	Enqueues the XOCP business message in the c-hub router message queue. In general, this system logic plug-in should be last in the XOCP router chain.
XOCP filter	Determines whether an XOCP business message is sent to a trading partner based on XPATH filter expressions configured in the repository. In general, this system logic plug-in should be first in the XOCP filter chain so that custom logic plug-ins can subsequently process a business message after rejected business messages have been filtered out.
RosettaNet router	Handles routing for RosettaNet business messages. In general, this system logic plug-in should be first in the RosettaNet router chain.

Table 4-3 System Logic Plug-Ins (Continued)

Logic Plug-In	Description
RosettaNet router enqueue	Enqueues the RosettaNet business message in the c-hub router message queue. In general, this system logic plug-in should be last in the RosettaNet router chain.
RosettaNet filter	Determines whether a RosettaNet business message is sent to a trading partner. In general, this system logic plug-in should be first in the RosettaNet filter chain.

In addition to using the system logic plug-ins, c-space owners and trading partners can develop their own custom logic plug-ins to provide specialized services on the c-hub. Each logic plug-in is a Java class that implements the Logic Plug-In API, as described in "Programming Steps for Logic Plug-Ins" on page 4-13.

Logic Plug-In API

WebLogic Collaborate provides a Logic Plug-In API that allows WebLogic Collaborate applications to:

- Add or remove target trading partners from the message recipient list. The c-hub validates the list of recipients before sending the business message.
- Retrieve, examine, and process parts of business messages. To ensure that the contents of business messages will not be altered or misrepresented programmatically, the logic plug-in API provides methods for examining business messages, but *not* for changing their contents.

The following table lists the components of the Logic Plug-in API. For more information, see the WebLogic Collaborate Javadoc.

Table 4-4 Logic Plug-In API

Class/Interface	Description
`com.bea.b2b.protocol.PlugIn`	Tagging interface that describes a generic logic plug-in, which represents pluggable code that can be inserted for execution at various places in the c-hub.
`com.bea.b2b.protocol.PlugInException`	Exception class that is thrown if an error occurs while executing a logic plug-in.
`com.bea.b2b.protocol.messaging.MessageEnvelope`	Represents the container ("envelope") for a business message passing through the c-hub. The `MessageEnvelope` contains the actual business message plus high level routing and processing information associated with the business message, such as the sender URL and the URL for one recipient (there is a single message envelope for each recipient). A Java `InputStream` is available in case access to the native message is needed (however, because message content modification is not allowed, no `OutputStream` is provided).
`com.bea.b2b.protocol.messaging.Message`	Represents a business message passing through the c-hub. It provides additional information to be used to properly route a message between trading partners. It also contains information specific to the particular business protocol being used for this business message. Depending on the protocol used, the `Message` class will usually be subclassed to provide additional protocol-specific information about the message.
`com.bea.b2b.protocol.messaging.PayloadPart`	Represents a component of the message payload. Specific implementing classes of this information are provided for some of the different types of parts of a business message, such as XML or non-XML parts, or to assist in accessing business protocol-specific information.
`com.bea.b2b.protocol.conversation.ConversationType`	Represents a single role in a specific conversation definition. It contains information such as the conversation name, conversation version, and trading partner role.

Rules and Guidelines for Logic Plug-Ins

Logic plug-ins should conform to the following rules and guidelines:

- Logic plug-ins must be thread-safe and, therefore, stateless. At run time, logic plug-in instances are cached and shared by multiple threads. Using instance variables is not recommended.
- If access to shared resources is required, then use the `synchronized` Java keyword to restrict access to the shared resource. Certain resources, such as instance variables within the class, shared objects, or external system resources (like files) might need shared access. Using the `synchronized` keyword can affect overall application performance, so use it only when necessary.
- Logic plug-ins can modify the message envelope and the list of recipients in the business message, but they *cannot* modify the message contents. Changing the business message invalidates the digital signature, if present. The Logic Plug-In API provides mutator methods for modifying the message envelope only.
- Logic plug-ins must be self-contained. They are not interdependent with other logic plug-ins; they cannot pass variables between them; and they do not return a variable. The message envelope is the only input and the only output. If the logic plug-in makes a change to the message envelope, it outputs the message envelope as modified.
- The main logic plug-in class must implement the `com.bea.b2b.protocol.PlugIn` interface.
- To ensure secure messaging, logic plug-ins are generally *not* able to inspect encrypted business messages. Examination is usually performed on business messages that do not have encrypted contents. To examine the encrypted contents of a business message, the logic plug-in would need to decrypt the message, inspect its contents, and then encrypt it again. Users would need to have their own public key infrastructure.
- It is the responsibility of the plug-in provider to ensure that any custom plug-ins that are installed on the c-hub are properly debugged and designed from a security perspective.
- A logic plug-in is always associated with at least one particular protocol in the repository. The logic plug-in is triggered only when a business message that

uses that protocol passes through the c-hub. For example, a RosettaNet business message does not trigger an XOCP-defined logic plug-in, and vice versa.

- A single logic plug-in can be associated with multiple protocols in the repository. For example, the same logic plug-in class named `SentMessages` could be associated with the XOCP and RosettaNet protocols. In the C-Hub Administration Console, you can define two separate logic plug-ins (such as `XOCP-SentMessages` and `RN-SentMessages`), although each would point to the same `SentMessages` class. Alternatively, the same logic plug-in can be used in two different protocol chains; they would share initialization parameters, but they would be separate instances.
- An efficient logic plug-in determines quickly whether a business message qualifies for processing and, if not, exits immediately.
- Logic plug-ins can call other modules, including shared methods in a utility library (for example, a module that accesses a database).
- Logic plug-ins are initialized one time, when the c-space is activated.
  - If the c-space is shut down (the `shutdown` method is called on the associated `com.bea.b2b.management.hub.runtime.CSpaceMBean`), then all protocol-specific logic plug-ins associated with that c-space are shut down as well. The c-space *must* be restarted for the logic plug-ins to be active.
  - If the c-hub is shut down (the `shutdown` method is called on the associated `com.bea.b2b.management.hub.runtime.CHubMBean`), then all logic plug-ins running on that c-hub are shut down as well. The c-hub and c-space must be restarted.
  - If the logic plug-in definitions change in the c-hub repository, such as when the chain is resequenced or when logic plug-in definitions are added, changed, or removed, then the c-space must be shut down and restarted to reflect the repository changes.
- The WebLogic Server instance *must* be restarted (and the Java Virtual Machine, or JVM, reloaded) if an upgraded version of the logic plug-in source code is installed on the c-hub.

Creating and Adding Logic Plug-Ins

Implementing a custom logic plug-in requires a combination of development and administrative tasks. The following steps describe those procedures:

- Programming Steps for Logic Plug-Ins
- Administrative Tasks

Programming Steps for Logic Plug-Ins

This section describes the programming steps that you must perform in the logic plug-in code. Although each logic plug-in processes business messages in its own way, all logic plug-ins must perform certain tasks.

To implement a logic plug-in, complete the following steps:

- Step 1: Import the Necessary Packages
- Step 2: Implement the PlugIn Interface
- Step 3: Specify the Exception Processing Model
- Step 4: Implement the Process Method
- Step 5: Get the Business Message from the Message Envelope
- Step 6: Validate the Business Message
- Step 7: Get Business Message Properties
- Step 8: Process the Business Message as Needed

This section uses code fragments from the `SentMsgCounter.java` file in the MessageCounter sample application. The `SentMsgCounter` class is a logic plug-in that:

- Intercepts a business message en route through the c-hub
- Obtains the names of the message sender, its target recipient, and its associated conversation definition
- Inserts a row with this information in the billing database.

The `CheckAccount.java` file in the CheckAccount sample application is another example of a logic plug-in. For more information about the MessageCounter and CheckAccount sample applications, see Using Logic Plug-Ins for Billing in *BEA WebLogic Collaborate Getting Started*.

Step 1: Import the Necessary Packages

At a minimum, a logic plug-in needs to import the following packages:

- `com.bea.b2b.protocol.*`
- `com.bea.b2b.protocol.messaging.*`

The following listing from the `SentMsgCounter.java` file shows importing the necessary packages.

Listing 4-1 Importing the Necessary Packages

```
import java.util.Hashtable;
import com.bea.b2b.protocol.*;
import com.bea.b2b.protocol.messaging.*;
import com.bea.eci.logging.*;
import javax.naming.*;
import javax.sql.DataSource;

// This package is needed to access the DB pool
import java.sql.*;
```

Step 2: Implement the PlugIn Interface

A logic plug-in needs to implement the `com.bea.b2b.protocol.PlugIn` interface, as shown in the following listing.

Listing 4-2 Implementing the PlugIn Interface

```
public class SentMsgCounter implements PlugIn
{ ...
}
```

Step 3: Specify the Exception Processing Model

A `PlugInException` is thrown if:

- A run-time exception (such as a `NullPointerException`) is thrown by a logic plug-in and caught by WebLogic Collaborate processing code.
- The logic plug-in throws an exception to indicate problems encountered during logic plug-in processing. The logic plug-in might handle the exception directly or it might notify the WebLogic Collaborate processing code.

The exception processing model specified in a logic plug-in determines what happens if an exception is thrown. Logic plug-ins must implement the `exceptionProcessingModel` method and specify one of the return values described in the following table.

Table 4-5 Options for the Exception Processing Model

Class/Interface	Description
`EXCEPTION_CONTINUE`	Indicates that processing should continue to the next logic plug-in in the chain if a `PlugInException` is thrown. Use this option to allow a business message to continue through the c-hub even if an error occurs during the execution of this logic plug-in.
`EXCEPTION_STOP`	Indicates that processing should stop at this logic plug-in if a `PlugInException` is thrown. The business message does not continue to the next logic plug-in in the chain. Use this option to cancel message processing and prevent its further progress through the c-hub. For example, if a logic plug-in might validate business documents and reject any that contain insufficient or incorrect data.
`EXCEPTION_UNWIND`	Indicates that processing should unwind if a `PlugInException` is thrown. The business message does not continue to the next logic plug-in in the chain. Use this option to reject a message; to prevent its further progress through the c-hub; and to undo any changes made by this plug-in, along with any changes made by previous plug-ins in the chain. If an exception is thrown and this is the exception processing model, then the `unwind` methods in all *previous* plug-ins in the chain (but not the current logic plug-in), are invoked in reverse order. In effect, unwinding cancels all changes made by the chain. For example, if a logic plug-in inserts a row in a database table, its `unwind` method should delete that row. **Note:** To use this exception processing model, all logic plug-ins in the chain must implement the `unwind` method, even if it does nothing.

If a business message is rejected, what happens next depends on the business protocol as well as the specified Quality of Service associated with the message. For example, the sending c-enabler application might be notified that message delivery failed and it might then attempt to send the business message again.

The following listing shows how the `SentMsgCounter` plug-in implements the `exceptionProcessingModel` method.

Listing 4-3 Specifying the Exception Processing Model

```
public int exceptionProcessingModel()
{
  return EXCEPTION_CONTINUE;
}
```

Step 4: Implement the Process Method

To process a business message, a logic plug-in must implement the `process` method, which accepts the message envelope of the business message as its only parameter. In the following listing, the `SentMsgCounter` class begins its implementation of the `process` method by defining the variables that it will later use to store message properties.

Listing 4-4 Implementing the Process Method

```
  public void process(MessageEnvelope mEnv) throws PlugInException
{

    String sender, conversation ;
    String tRecipient;
    Connection conn = null;
    Statement stmt = null;
    Message bMsg = null;
    ...
}
```

Note: When processing a business message, a logic plug-in is allowed to modify only the message envelope, not the business message.

Step 5: Get the Business Message from the Message Envelope

If a logic plug-in needs to inspect the contents of a business message, it must call the `getMessage` method on the `MessageEnvelope` instance, which retrieves the business message as a `Message` object.

In the following listing, the `SentMsgCounter` class gets the business message from the message envelope by calling the `getMessage` method.

Listing 4-5 Retrieving the Business Message from the Message Envelope

```
if((bMsg = mEnv.getMessage())== null)
{
    new Throwable("bMsg is NULL").printStackTrace();
}
```

Step 6: Validate the Business Message

Optionally, a logic plug-in can determine whether a message is a valid business message that should be processed, or a system message that should be ignored by the logic plug-in. To check a business message, the logic plug-in can call the `isBusinessMessage` method on the `Message` instance. In the following listing, the `SentMsgCounter` class uses the `isBusinessMessage` method to ensure that the message is a business message before processing it.

Listing 4-6 Validating the Business Message

```
if (bMsg.isBusinessMessage())
{
...
}
```

Step 7: Get Business Message Properties

Optionally, a logic plug-in can retrieve certain properties of the business message by calling methods on the `MessageEnvelope` or `Message` instance. In the following listing, the `SentMsgCounter` class gets the name of the conversation definition associated with the conversation in which this message was sent, the name of the sender of the business message, and the name of the recipient trading partner.

Listing 4-7 Retrieving Business Message Properties

```
conversation= bMsg.getConversationType().getName();
sender = mEnv.getSender();
tRecipient = mEnv.getRecipient();
```

Step 8: Process the Business Message as Needed

After a logic plug-in has obtained the necessary information from the business message, it processes this information as needed. For example, the `SentMsgCounter` plug-in updates the billing database with the message statistics it has collected.

Administrative Tasks

An administrator adds the logic plug-in definition to the repository by performing the following tasks from the Logic Plug-Ins tab of the C-Hub Administration Console:

1. Specify the following logic plug-in properties:
   - Name of the logic plug-in.
   - Java class that implements the `PlugIn` interface. This class can call auxiliary classes in the class library, but it must be the main point of entry for the logic plug-in. In addition, the Java class file must reside in a location specified by the WebLogic `CLASSPATH`.
   - Parameter name/value pairs to use when initializing the Java class.
2. Assign a logic plug-in to a business protocol.
3. Specify the position of the logic plug-in in the chain.

For more information about administrative tasks, see Working with Logic Plug-Ins and Using the C-Hub Administration Console in the *BEA WebLogic Collaborate C-Hub Administration Guide*.

5 Developing Management Applications

The following sections describe how to create WebLogic Collaborate management applications that monitor run-time activity on c-hub and c-enabler nodes:

- About Management Applications
- Programming Steps for Management Applications

The WebLogic Collaborate C-Hub Administration Console and the C-Enabler Administration Console tools provide run-time monitoring of c-hub and c-enabler activities. In addition to these standard tools, developers can create custom management applications that provide the same monitoring information that appears in the administration console tools.

These custom management applications can provide read-only access to real-time statistics, such as the number of messages exchanged in a particular conversation or the number of messages received by the c-hub. In addition, these custom applications can perform certain administrative tasks programmatically, such as shutting down a particular c-space or the c-hub (in c-hub management applications) or leaving or terminating a particular conversation (in c-enabler management applications).

Note: Configuring the c-hub repository requires the C-Hub Administration Console. Custom management applications cannot perform configuration tasks.

About Management Applications

The following sections describe management applications in WebLogic Collaborate:

- MBeans and the MBean Server
- C-Hub MBeans
- C-Enabler MBeans
- Configuration Requirements

MBeans and the MBean Server

WebLogic Collaborate provides developers with the application programming interfaces (APIs) needed to create custom management applications that monitor run-time activity on c-hub and c-enabler nodes. The C-Hub Administration Console and the C-Enabler Administration Console tools also use these APIs to provide real-time monitoring information.

These APIs consist of sets of Java Management Extensions (JMX) Managed Beans, or *MBeans*, which are special JavaBeans with attributes and methods for management operations. For more information about JMX, particularly the use of the JMX API (including the MBean Server and MBeans), see the Java Management Extensions Specification published by Sun Microsystems, Inc., at the following URL:

`http://www.java.sun.com/products/JavaManagement/index.html`

MBean Packages

WebLogic Collaborate provides the following packages for creating custom management applications.

Table 5-1 Packages for WebLogic Collaborate Management Applications

Package	Description
`com.bea.b2b.management`	Provides the `ManagementException` class for handling errors that occur when running a management application.
`com.bea.b2b.management.hub.runtime`	Provides c-hub MBeans used for creating management applications that monitor run-time activity on c-hub nodes.
`com.bea.b2b.management.enabler.runtime`	Provides c-enabler MBeans used for creating management applications that monitor run-time activity on c-enabler nodes.

For detailed information about these packages, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

Note: In this release, all MBeans are implemented as Standard MBeans, which is a class that implements its own MBean interface.

MBean Server Implementation

WebLogic Collaborate provides an implementation of the JMX MBean Server component that serves as a repository for MBeans.

- The c-hub MBeans are registered with the MBean Server running inside the c-hub. When c-hub MBeans are created, WebLogic Collaborate populates their attributes from settings in the c-hub repository.
- The c-enabler MBeans are registered with the MBean server running inside the c-enabler. When c-enabler MBeans are created, WebLogic Collaborate populates their attributes from settings in the c-enabler XML configuration file.

At run time, WebLogic Collaborate updates the MBean attributes to reflect the state of the running system.

Note: C-enablers that are co-located with the c-hub or with other c-enablers share the same MBean Server instance. If a c-enabler runs in WebLogic Server without a co-located hub, it will have its own local MBean server.

C-Hub MBeans

The `com.bea.b2b.management.hub.runtime` package contains the WebLogic Collaborate c-hub MBeans, which are described in the following table.

Table 5-2 Managed Beans in the C-Hub MBeans

Label	Description
`HubMBean`	Represents a c-hub. Used for monitoring a c-hub at run time.
`CSpaceMBean`	Represents a c-space. Used for monitoring c-spaces on the c-hub at run time.
`GlobalConversationMBean`	Represents an instance of a global conversation managed by the Conversation Manager on the c-hub. Used for monitoring active conversations within a c-space.
`CollaboratorMBean`	Represents a trading partner in a c-space. Used for monitoring trading partners in the c-space.
`MessageMBean`	Represents a message in a conversation. Used for monitoring messages in the c-space.

C-Enabler MBeans

The `com.bea.b2b.management.enabler.runtime` package contains the WebLogic Collaborate c-enabler MBeans, which are described in the following table.

Table 5-3 Managed Beans in the C-Enabler MBeans

Label	Description
`EnablerMBean`	Represents a c-enabler. Used for monitoring the c-enabler at run time.
`ConversationMBean`	Represents a conversation. Used for monitoring active conversations in which the c-enabler is involved.
`EnablerSessionMBean`	Represents a c-enabler session. Used for monitoring active c-enabler sessions on the c-enabler.
`MessageMBean`	Represents a message in a conversation. Used for monitoring messages in the c-enabler.

Configuration Requirements

To use the WebLogic Collaborate MBeans, make sure that the following file is included in the `CLASSPATH`:

```
lib\jmxri.jar
```

Note: Be sure to use the Javasoft implementation of this file.

Programming Steps for Management Applications

The steps for using MBeans to develop management applications for c-hubs and c-enablers are nearly identical. To access WebLogic Collaborate MBeans using the JMX API, a Java application must complete the following steps:

- Step 1: Import the Necessary Packages
- Step 2: Get a Reference to the MBean Server Object
- Step 3: Construct an ObjectName Object
- Step 4: Query the MBean Server
- Step 5: Read the Attributes of the MBean
- Step 6: Navigate Across MBeans
- Step 7: Handle Exceptions

The C-Hub and C-Enabler Administration Consoles use the JMX API and WebLogic Collaborate MBeans to monitor running c-hubs and c-enablers, respectively.

Step 1: Import the Necessary Packages

To work with MBeans, a management application must import the necessary packages. At a minimum, the application must import the packages described in the following table.

Table 5-4 Packages that Must Be Imported

Label	Description
`javax.management.*;`	Required for JMX MBeans, as mandated in the Java Management Extensions Specification published by Sun Microsystems, Inc.
`javax.naming.*;`	Required for retrieving the MBean server object using JNDI lookup. Only the following are required: ■ `javax.naming.Context` ■ `javax.naming.InitialContext`
`com.bea.b2b.management.ManagementException`	Required for handling exceptions in all management applications.
`com.bea.b2b.management.hub.runtime.*`	Required for c-hub management applications.
`com.bea.b2b.management.enabler.runtime.*`	Required for c-enabler management applications.

C-Hub Example

The code in the following listing imports the necessary packages for c-hub management applications.

Listing 5-1 Importing Packages for C-Hub Management Applications

```
import javax.management.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.bea.b2b.management.ManagementException;
import com.bea.b2b.management.hub.runtime.*;
```

C-Enabler Example

The code in the following listing imports the necessary packages for c-enabler management applications.

Listing 5-2 Importing Packages for C-Enabler Management Applications

```
import javax.management.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.bea.b2b.management.ManagementException;
import com.bea.b2b.management.enabler.runtime*;
```

Step 2: Get a Reference to the MBean Server Object

To get a reference the MBean server object, a management application calls the `findMBeanServer` method on the `MBeanServerFactory` object, as shown in the following listing.

Listing 5-3 Getting a Reference to the MBean Server Object

```
MBeanServer server = null;
ArrayList mbsList = MBeanServerFactory.findMBeanServer("WLC");
if (mbsList.size() > 0) {
   server = (MBeanServer)mbsList.get(0);
}
```

Step 3: Construct an ObjectName Object

MBeans are uniquely identified by object names inside the MBean server. The `ObjectName` class represents an object name.

Object Names

An object name consists of two parts:

- **Domain Name**. For all MBeans in WebLogic Collaborate, the domain name is the same:

  `WLC`

- **Key property list**. Enables you to assign unique names to the MBeans of a given domain. The WebLogic Collaborate MBeans have the following key properties:

Table 5-5 Key Properties

Property	Description
`type`	Name of the MBean interface. For c-hub management applications, one of the following values: ■ `HubMBean` ■ `CSpaceMBean` ■ `CollaboratorMBean` ■ `GlobalConversationMBean` ■ `MessageMBean` For c-enabler management applications, one of the following values: ■ `EnablerMBean` ■ `EnablerSessionMBean` ■ `ConversationMBean` ■ `MessageMBean`
`name`	Unique identifier of the MBean.

Table 5-5 Key Properties (Continued)

Property	Description

`subsystem`	Unique identifier of the subsystem.
	For c-hubs, use the following format:
	`hub.<hubName>`
	For c-enablers in conjunction with `EnablerMBeans` and `MessageMBeans`, use the following format:
	`enabler`.
	For c-enablers in conjunction with `EnablerSessionMBeans` and `ConversationMBeans`, use the following format:
	`enabler.<enablerName>`
	The *`enablerName`* extension uniquely identifies the specific c-enabler in the conversation. This is needed because a c-enabler can participate in the same conversation in two different roles using two different c-enabler sessions.

C-Hub Example

For example, a `CSpaceMBean` could have the following object name:

```
WLC:type=CSpaceMBean,*
```

C-Enabler Example

Similarly, an `EnablerMBean` could have either of the following object names:

```
WLC:type=EnablerMBean,*
```

```
WLC:type=EnablerMBean,subsystem=enabler,*
```

Object Name Expressions

For MBeans, object names can be also used for query operations that use object name expressions. The MBean server uses pattern matching on the object names of the registered MBeans. The matching syntax is consistent with file globing, which is described in the Java Management Extensions Specification published by Sun Microsystems, Inc.:

- An asterisk (`*`) matches any character sequence.
- A question mark (`?`) matches a single character.

C-Hub Example

For example, the following object name expression will match the object names of all registered `CSpaceMBeans`.

```
WLC:type=CSpaceMBean,name=*
```

C-Enabler Example

Similarly, the following object name expression will match the object names of all registered `EnablerMBeans`.

```
WLC:type=EnablerMBean,name=*
```

Step 4: Query the MBean Server

After constructing an object name expression, an application queries the MBean server by passing in the `ObjectName` object corresponding to the expression. To retrieve the set of registered MBeans whose names satisfy an object name expression, use the following method:

```
javax.management.MBeanServer.queryNames()
```

The MBean server returns a set of objects that satisfy the query criteria. Note that these are `ObjectName` objects that *represent* MBeans, *not* direct references to the MBeans themselves.

C-Hub Example

The code in the following listing retrieves a set of `ObjectName` objects that represent the `CSpaceMBeans` associated with registered c-spaces on the c-hub.

Listing 5-4 Retrieving Registered CSpaceMBeans

```
if (server != null) {
    ObjectName queryObjName = new ObjectName("WLC:type=HubMBean,*");
    // beans is a set of ObjectName objects
    beans= server.queryNames(queryObjName, null);
  }

  if (null == beans)
    noCsps = true;
  else {
    Iterator it = null;
    it = beans.iterator();
    csps = new ArrayList();
    while (it != null && it.hasNext()) {
      ObjectName objname = (ObjectName)it.next();
      hubObj = objname.toString();
      cspbeans = (CSpaceMBean[])server.getAttribute(objname , "CSpaces");
      for (int c=0; c < cspbeans.length; c++)
        csps.add(cspbeans[c]);
    }
```

C-Enabler Code Example

The code in the following listing retrieves a set of `ObjectName` objects that represent active `EnablerSessionMBeans` on the c-enabler.

Listing 5-5 Retrieving Registered EnablerSessionMBeans

```
if (server != null)
{
  ObjectName queryObjName = new ObjectName("WLC: subsystem=enabler." + enablerName
+ ",name=" + sessionName + ",type=EnablerSessionMBean" );
  // beans is a set of ObjectName objects
  beans= server.queryNames(queryObjName, null);
}
```

```
Iterator it = beans.iterator();
// Iterate through the EnablerSessionMBeans
while (it.hasNext()){
   ObjectName objName = (ObjectName)it.next();
   // do something
}
```

Step 5: Read the Attributes of the MBean

Use the `ObjectName` instance, obtained in the previous step, to access other MBeans, provided that the `ObjectName` has one or more attributes whose type is MBean. To read the attributes of an MBean, use the following method, passing the `ObjectName` object as a parameter:

`javax.management.MBeanServer.getAttribute()`

Once you call the `getAttribute` method by passing in the `ObjectName` object for the first MBean, you can get references directly to other MBean instances.

C-Hub Example

The code in the following listing retrieves a set of attributes associated with a global conversation.

Listing 5-6 Retrieving Conversation Attributes

```
while ((count < convsPerPage) && (idx < totalConvs)) {
   ObjectName objName = (ObjectName)convs.get(idx);
   String convId = (String) server.getAttribute(objName, "ConversationId");
   CSpaceMBean cspace = (CSpaceMBean) server.getAttribute(objName, "CSpace");
   Protocol protocol = (Protocol)server.getAttribute(objName, "Protocol");
   Date startTime = (Date) server.getAttribute(objName, "ActiveSince");
   Date lastMessage = (Date) server.getAttribute(objName, "LastMessageTime");
   String lastSender = (String) server.getAttribute(objName, "LastSender");
   CollaboratorMBean[] parts = (CollaboratorMBean[])
server.getAttribute(objName, "ActiveCollaborators");
   String checkBoxSuccess = "checkBoxSuccess" + idx;
   String checkBoxFailure = "checkBoxFailure" + idx;
   String regConvId = convId.replace('*','$');
   regConvId = regConvId.replace(':','$');
```

```
   regConvId = regConvId.replace('?','$');
   regConvId = regConvId.replace('=','$');
   String checkBoxValue = "WLC:subsystem=hub,name=" + regConvId + ",cspacename="
+ cspace.getName() + ",type=GlobalConversationMBean" ;
```

In this example, `server` is a reference to the MBean server and `objName` is a reference to the `ObjectName` object representing the `GlobalConversationMBean`.

The application can then iterate through and process the set of `GlobalConversationMBean` objects as needed. Because it now has direct references to the MBean, the application can use methods on the MBean to retrieve attributes, such as run-time monitoring information.

C-Enabler Example

The code in the following listing retrieves a set of attributes associated with a c-enabler session.

Listing 5-7 Retrieving C-Enabler Session Attributes

```
Iterator it = beans.iterator();
while (it.hasNext()){
   ObjectName obj = (ObjectName)it.next();
   hubUrl = (String)server.getAttribute(obj, "HubUrl");
   hubProxyHost = (String)server.getAttribute(obj, "ProxyHost");
   hubProxyPort = (String)server.getAttribute(obj, "ProxyPort");
   hubUser = (String)server.getAttribute(obj, "HubUser");
   hubCertField = (String)server.getAttribute(obj, "CertificateFieldName");
   hubCertValue = (String)server.getAttribute(obj, "CertificateFieldValue");
   hubServerCertField = (String)server.getAttribute(obj,
"ServerCertificateFieldName");
   hubServerCertValue = (String)server.getAttribute(obj,
"ServerCertificateFieldValue");
   enablerUrl = (String)server.getAttribute(obj, "EnablerUrl");
   cSpaceName = (String)server.getAttribute(obj, "CSpaceName");
   tradingPartner = (String)server.getAttribute(obj, "TradingPartnerName");
   certLocation = (String)server.getAttribute(obj, "CertificateLocation");
   privateKeyLoc = (String)server.getAttribute(obj, "PrivateKeyLocation");
}
```

In this example, `server` is a reference to the MBean server and `objName` is a reference to the `ObjectName` object representing the `EnablerSessionMBean`.

The application can then iterate through and process the set of `EnablerSessionMBean` objects as needed. Because it now has direct references to the MBean, the application can use methods on the MBean to retrieve attributes, such as run-time monitoring information.

Step 6: Navigate Across MBeans

MBeans that are logically related have accessor methods to retrieve references to each other. These methods are strongly typed and return an exact MBean type. For example, the `CSpaceMBean.getHub()` method returns a `HubMBean` that represents the c-hub associated with that c-space. Similarly, the `EnablerSessionMBean.getEnabler()` method returns a `EnablerMBean` that represents the associated c-enabler.

For detailed information about these methods, see the Javadoc on the WebLogic Collaborate documentation CD or in the `classdocs` subdirectory of your WebLogic Collaborate installation.

Step 7: Handle Exceptions

If an error occurs while running a WebLogic Collaborate management application, a `com.bea.b2b.management.ManagementException` is thrown. Management applications can catch this exception and process it as appropriate, as shown in the following listing.

Listing 5-8 Handling ManagementExceptions in Management Applications

```
catch (ManagementException me){
      String msg = "Exception in Management Application: " + me;
      debug(msg);
      throw new Exception(msg);
```

CHAPTER

6 Writing to the Log

The following sections describe how to write messages to the log in WebLogic Collaborate applications:

- About the Log
- Writing Messages to the Log

About the Log

WebLogic Collaborate applications can write messages (errors, warnings, and information) to a log file for subsequent examination. WebLogic Collaborate provides a Logging API that applications can use to write messages to the log.

Log Files

Logged messages are written to the following locations:

- WebLogic Collaborate system log file (`wlc.log`), the C-Hub Administration Console, and the C-Enabler Administration Console
- WebLogic Server log file (`weblogic.log`) and the WebLogic Server Console (if it is running)

The `wlc.log` system log file is created automatically when a c-hub or c-enabler starts up. The size of this file is limited to 1MB. When the file size is exceeded, the file is renamed with a numeric suffix (such as `wlc1.log`) and a new empty file is created.

Logging API

The com.bea.eci.logging package contains the WebLogic Collaborate logging API, which consists of the classes described in the following table.

Table 6-1 Logging API

Name	Description
Log	Defines severity levels for log messages.
UserLog	Represents a user log. Provides access to the log for users. The user log is defined as a User log output stream (with a <user> tag) in the system log.

For detailed information about these classes, see the Javadoc on the WebLogic Collaborate documentation CD or in the classdocs subdirectory of your WebLogic Collaborate installation.

Severity Levels

The Log class defines the severity levels described in the following table.

Table 6-2 Severity Levels Defined in Log Class

Level	Severity	Description
1	FATAL	Fatal error has occurred. A system component failed abnormally due to the exception that was detected.
2	ERROR	User level error has occurred. A critical error occurred that impacts system stability.
3	WARNING	Warning message. A minor exception occurred that does not impact system stability.
4	INFO	Informational message. Informational only. Not used in exception conditions. An example would be logging the successful startup of the c-hub.

Writing Messages to the Log

WebLogic Collaborate applications can write messages to the user log using the `log` method in the `UserLog` class. The log method has two versions: one version specifies the message text with an `INFO` severity level, and the other version specifies the message text and a particular severity level (`FATAL`, `ERROR`, or `WARNING`). In addition, applications have print stream access to the log via `userlog.out`.

Importing the Logging Package

To write to the log, WebLogic Collaborate applications must import the `com.bea.eci.logging` package, as shown in the following listing.

Listing 6-1 Importing the com.bea.eci.logging Package

```
import com.bea.eci.logging.*;
```

Writing a Log Message with an INFO Severity Level

To write a log message with an `INFO` severity level, an application can use the following version of the `log` method:

```
static void log(java.lang.String userMsg)
```

The following listing shows writing a log message with an `INFO` severity level:

Listing 6-2 Writing an INFO Message to the Log

```
private static void debug(String msg){
    if (DEBUG)
      UserLog.log("***Partner1Servlet: "+msg);
}
```

Writing a Message With a Specific Severity Level

To write a log message with a specific severity level, an application uses the following version of the `log` method:

```
static void log(int severity, java.lang.String userMsg)
```

The following listing shows writing a log message with a `WARNING` severity level:

Listing 6-3 Writing a WARNING Message to the Log

```
private static void debug(String msg){
   if (DEBUG)
      UserLog.log(log.WARNING, msg);
}

try {
   }catch(Exception e){
      debug("Partner1 exception errors");
      e.printStackTrace(UserLog.out);
}
```

Index

D

E

G

H

I

J

K

L

M

O

P

Q

R

S

T

V

W

X