BEA WebLogic Integration™

Programming Messaging Applications for B2B Integration

Release 7.0
Document Date: June 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming Messaging Applications for B2B Integration

Part Number	Date	Software Version
N/A	June 2002	7.0

Contents

2. Programming Steps for XOCP Applications

3. Sending XOCP Business Messages

About This Document

This document describes how to use the Messaging API for BEA WebLogic Integration B2B integration to develop XOCP protocol messaging applications.

Note: The Messaging API and XOCP business protocol are deprecated as of this release of WebLogic Integration. For information about the features that are replacing the Messaging API and XOCP business protocol, see the *BEA WebLogic Integration Release Notes*.

This document includes the following topics:

- Chapter 1, "Developing XOCP Applications to Exchange Business Messages," discusses the steps required to develop applications that exchange business messages using the WebLogic Integration eXtensible Open Collaboration Protocol (XOCP).
- Chapter 2, "Programming Steps for XOCP Applications," discusses the steps required to program applications that exchange business messages using the XOCP protocol.
- Chapter 3, "Sending XOCP Business Messages," discusses the requirements for sending XOCP business messages.
- Chapter 4, "Receiving XOCP Business Messages," discusses requirements for receiving XOCP business messages.

What You Need to Know

This document is intended for independent software vendors (ISVs) who want to extend their WebLogic Integration environment. It is assumed that the reader has a familiarity with the BEA WebLogic Integration platform and Java programming.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

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

A PDF version of this document is available on the WebLogic Integration documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Integration documentation Home page, click the PDF files button and select the document you want to print.

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

Related Information

The following WebLogic Integration documents contain information that will help you understand how to write messaging applications that take advantage of the B2B integration functionality provided by WebLogic Integration:

- *Administering B2B Integration*
- *Programming Management Applications for B2B Integration*
- *Programming Logic Plug-Ins for B2B Integration*

For more information about BEA WebLogic Integration and Java, see the WebLogic Integration documentation available at `http://edocs.bea.com/`.

Contact Us!

Your feedback on the BEA WebLogic Integration 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 WebLogic Integration documentation.

In your e-mail message, please indicate that you are using the documentation for BEA WebLogic Integration 2.1 Service Pack 1.

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

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes

- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

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

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

CHAPTER

1 Developing XOCP Applications to Exchange Business Messages

Note: The Messaging API and XOCP business protocol are deprecated as of this release of WebLogic Integration. For information about the features that are replacing the Messaging API and XOCP business protocol, see the *BEA WebLogic Integration Release Notes*.

The eXtensible Open Collaboration Protocol (XOCP) is the default business protocol used by WebLogic Integration for exchanging business messages. This section includes the following topics:

Introduction

WebLogic Integration provides two means to implement trading partner conversations that are based on the XOCP protocol:

- Via business process management (BPM) collaborative workflows that you create using the WebLogic Integration Studio. These workflows define each role in a conversation, and they specify how business messages are handled and exchanged by trading partners. For information about creating collaborative workflows, see *Creating Workflows for B2B Integration*.
- Via XOCP applications that you create using the WebLogic Integration Messaging API. An XOCP application implements a trading partner role and interacts directly with the B2B engine to manage the conversation and handle business messages as appropriate.

This document explains how to use the Messaging API to create XOCP applications to conduct and manage conversations among trading partners.

Many of the code examples in this documentation derive from the Messaging API example. For more information, see the "Messaging API Sample" in *Running the B2B Integration Samples*.

Note: The C-Enabler API, which was formerly used for creating XOCP applications (in WebLogic Integration Release 2.0 and the WebLogic Collaborate product) is deprecated but still supported. Information about creating applications that use this deprecated API is available at the following URL:

`http://e-docs.bea.com/wlintegration/v2_0/collaborate/devxocp/index.htm`

XOCP applications, including those originally written with the WebLogic Collaborate C-Enabler API for WebLogic Integration Release 2.0, must be run in a separate Java Virtual Machine (JVM) in nonpersistent mode.

Key Concepts

This section describes the following key concepts associated with XOCP applications:

- XOCP Applications
- Messaging API Class Library
- XOCP Business Messages and Message Envelopes
- Conversation Initiators and Participants
- Conversation Coordinators
- Trading Partner States
- Secure Messaging

XOCP Applications

An *XOCP application* is a user-written Java application that runs on a WebLogic Integration node that is deployed in a hub-and-spoke configuration and that uses the *XOCP application class* to execute a specific role in a conversation definition. In a hub-and-spoke configuration, a trading partner XOCP application is associated with a spoke delivery channel, or *B2B spoke*. This XOCP application allows a trading partner to communicate with other trading partners at B2B spokes via an intermediary, or routing proxy, which is configured with a hub delivery channel.

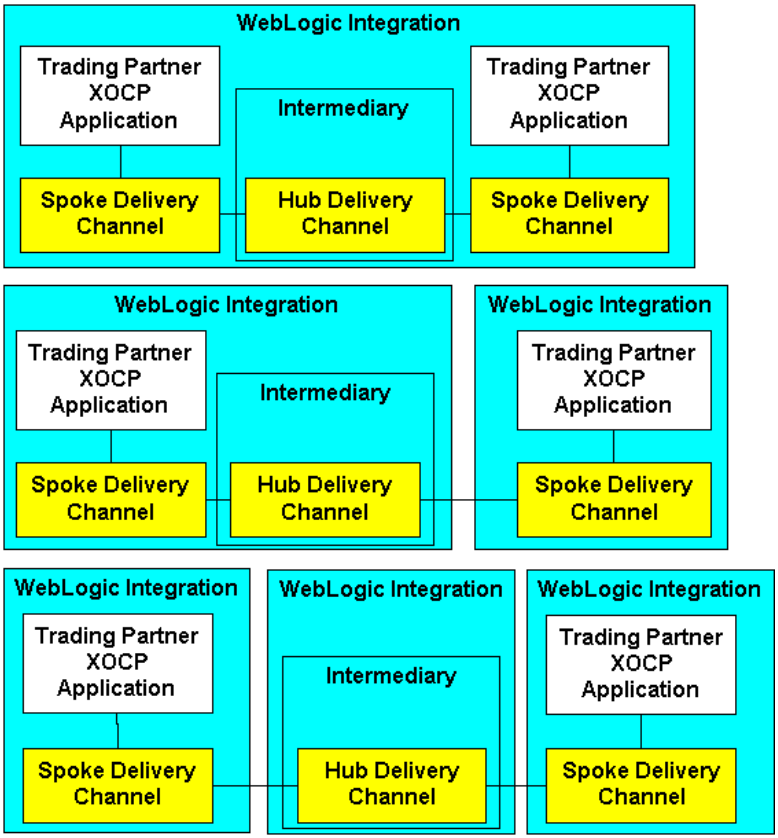
A user-written XOCP application executes the following tasks:

- Create and shut down XOCP application sessions
- Initiate or participate in conversations
- Exchange XOCP business messages with other trading partners
- Terminate or leave conversations

Note: For complete details on the XOCP application class, see the `com.bea.b2b.protocol.xocp.application` class in the *BEA WebLogic Integration Javadoc*.

The following figure shows three possible hub-and-spoke configurations for the delivery channels, the XOCP applications, and the instances of WebLogic Integration that host the XOCP applications.

Figure 1-1 Possible Hub-and-Spoke Configurations



A WebLogic Integration node can host many XOCP applications. For more information about configuring hub and spoke delivery channels used with XOCP applications, see "Configuration Requirements" in *Administering B2B Integration*.

XOCP Application Sessions

An XOCP application session is the means by which an XOCP application is associated with a collaboration agreement and a delivery channel. An XOCP application session is created by an application to communicate with a trading partner; and its scope is bounded by a delivery channel. XOCP applications create an XOCP application session by invoking the `getXOCPApplicationSession` method on the XOCP application class.

An XOCP application can be associated with multiple XOCP application sessions, enabling the application to participate in multiple conversations simultaneously.

Messaging API Class Library

The Messaging API class library includes the XOCP application class and provides APIs for exchanging XOCP business messages. It consists of the packages listed in the following table.

Table 1-1 Messaging API Class Library Packages

Package Name	Description
`com.bea.b2b.protocol.xocp.application`	Used for working with XOCP applications and XOCP application sessions. This package is designed for applications used by trading partners configured with a spoke delivery channel.
`com.bea.b2b.protocol.xocp.conversation.local`	Used for working with conversations based on XOCP.
`com.bea.b2b.protocol.messaging`	Used for working with messages in a conversation.
`com.bea.b2b.protocol.xocp.messaging`	Used for working with business messages in conversations based on XOCP.

For detailed information about these packages, see *BEA WebLogic Integration Javadoc* on the WebLogic Integration documentation CD or, on Windows systems, choose the BEA WebLogic e-Business Platform→WebLogic Integration 2.1→Javadocs from the Windows Start menu.

XOCP Business Messages and Message Envelopes

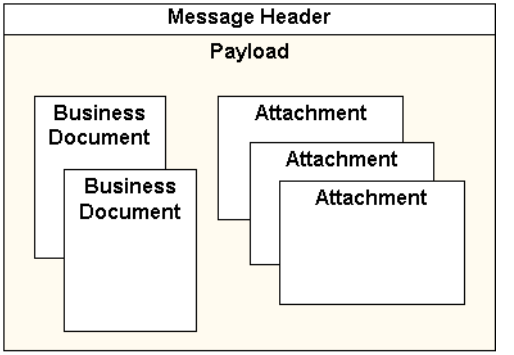
An *XOCP business message* is the basic unit of communication exchanged between trading partners in an XOCP conversation. An XOCP business message is represented in the Messaging API class library by the `com.bea.b2b.protocol.xocp.messaging.XOCPMessage` class.

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 Messaging API class library by the `com.bea.b2b.protocol.messaging.MessageEnvelope` class. However, only logic plug-ins (not XOCP applications) have programmatic access to message envelopes. For more information, see "Information Flow for Message Envelopes" on page 1-8 and "Routing and Filtering Business Messages" in *Programming Logic Plug-Ins for B2B Integration*.

Diagram of an XOCP Business Message

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

Figure 1-2 Components of an XOCP Business Message



Components of an XOCP Business Message

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

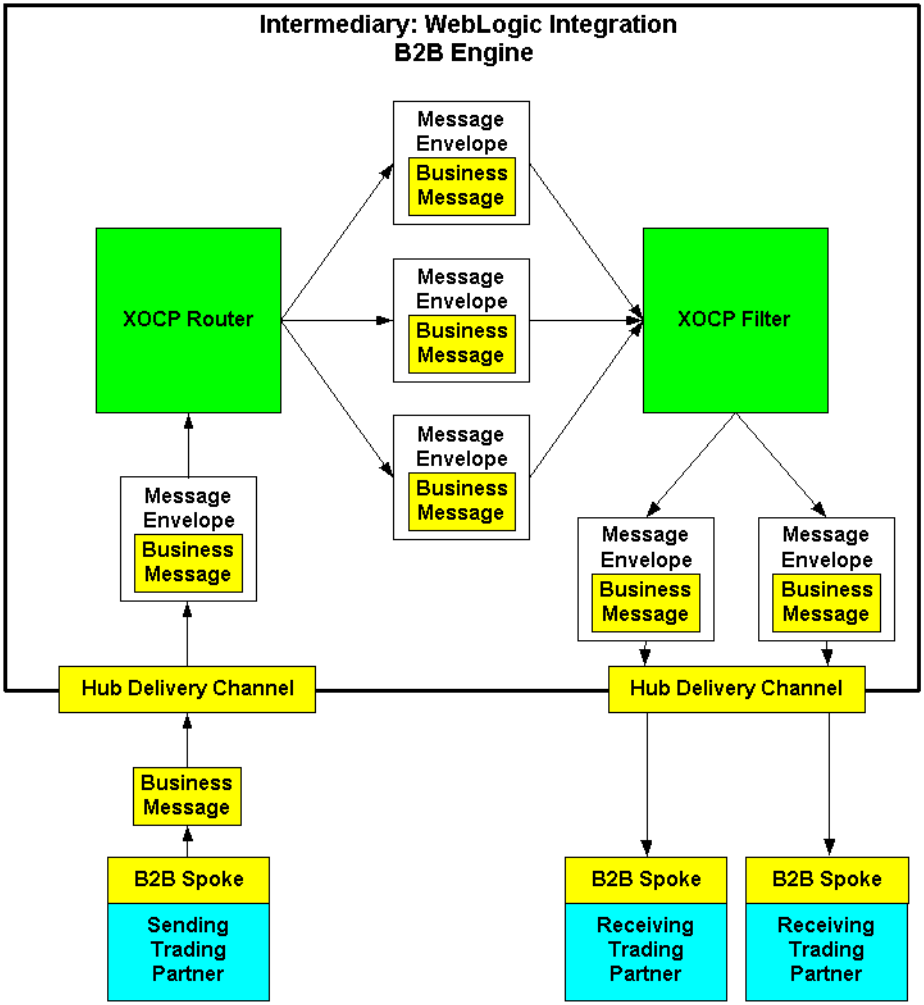
Table 1-2 Components of an XOCP Business Message

Component	Description
Message header	Message attributes, including information about the sender and recipient, the conversation, Qualities of Service, and so on.
Payload	Container for one or more business documents, one or more attachments, or a combination of both. The payload component is represented in the Messaging API class library by the `com.bea.b2b.protocol.messaging.PayloadPart` interface.
Business document(s)	XML files in the XML-based part of the payload. Represented in the Messaging API class library by the `com.bea.b2b.protocol.messaging.BusinessDocument` class.
Attachment(s)	NonXML files in the nonXML-based part of the payload. Binary content. Represented in the Messaging API 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 WebLogic Integration.

Figure 1-3 Message Envelope Processing in WebLogic Integration



Message envelope processing occurs in the following sequence:

1. A trading partner creates and sends a business message from its spoke delivery channel to the hub delivery channel at the intermediary. (This hub delivery channel can be configured on a B2B engine collocated with the sending trading partner, on a standalone machine, or on a B2B engine collocated with a recipient trading partner, as shown in Figure 1-1.)
2. The business message is received at the hub delivery channel. The B2B engine wraps the business message with a message envelope, extracting certain sender and recipient information from the business message.
3. The XOCP 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 list of recipients, inserts a logical copy of the business message in each message envelope, and then forwards all message envelopes to the XOCP filter.

   As shown in the example in Figure 1-3, the router creates message envelopes for three recipients.
5. Within the XOCP filter, the applicable 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 to the next processing step in the B2B engine.

   In Figure 1-3, the three business messages are evaluated in the filter. Two are accepted and one is rejected.
6. The B2B engine 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 that 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).

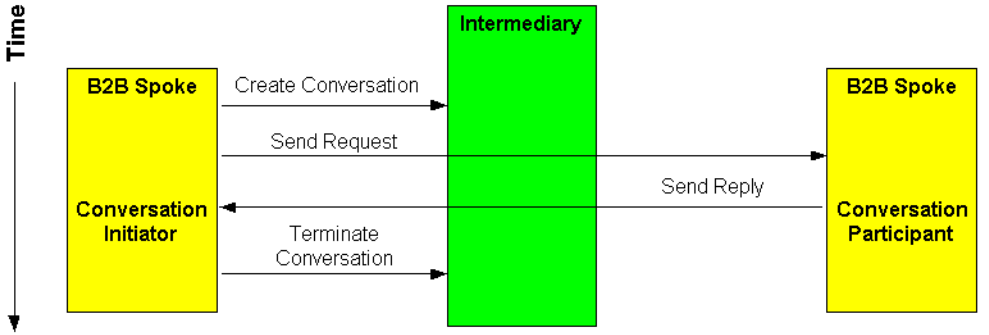
- *Conversation participant* is a trading partner that is enlisted in the conversation when it receives the first business message from the conversation initiator. The conversation participant usually sends a reply to the conversation initiator and, optionally, might exchange subsequent business messages. 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 to initiate or participate in conversations associated with the associated 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 in the role of the buyer normally initiates a `GetQuote` conversation. Any trading partner in the role of the seller normally acts as a conversation participant in the `GetQuote` conversation.

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

Figure 1-4 Conversation Initiators and Participants

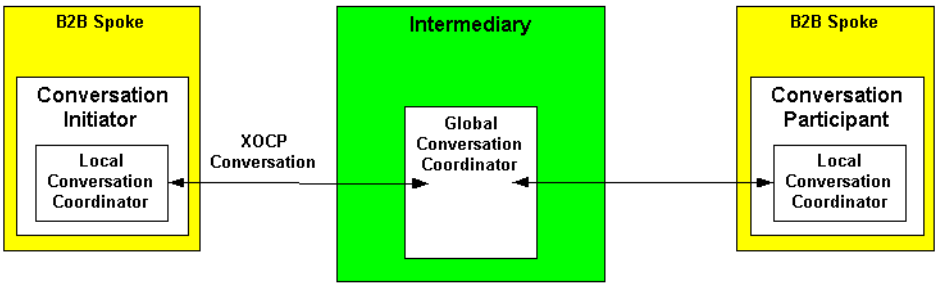


Conversation Coordinators

WebLogic Integration supports two types of conversation coordinators that manage conversations at run time: a *global conversation coordinator* manages active conversations on the B2B intermediary, and *local conversation coordinators* associated with B2B spokes help the global coordinator manage active conversations locally.

The following figure shows where global and local conversation coordinators work in the WebLogic Integration architecture.

Figure 1-5 Global and Local Conversation Coordinators



Global Conversation Coordinator

A global conversation coordinator is a service associated with the intermediary, which is configured with a hub delivery channel. The global conversation coordinator manages conversation lifecycles 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.

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 service associated with a B2B spoke. The local conversation coordinator manages conversations in which the local trading partner (configured with a spoke delivery channel) is participating and maintains a list of active conversations. Each XOCP application 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 intermediary
- Locally delists from a conversation when the system message that terminates the conversation is received from the intermediary

Trading Partner States

The following table describes the states assigned to trading partners as they perform tasks related to XOCP application sessions and conversation participation.

Table 1-3 Trading Partner States

State	Description
`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 (that is, has sent or received a business message) in at least one conversation.
`DROPPEDOUT`	Trading partner has left a conversation.

Some of these trading partner states are visible in the WebLogic Integration B2B Console.

Secure Messaging

Communication among trading partners is secured via the Secure Sockets Layer (SSL). Before allowing trading partners to exchange business messages, the WebLogic Integration node must authenticate the identity of each trading partner using the trading partner's certificate. Once these identities are authenticated, business messages are exchanged securely among trading partners. For more information about WebLogic Integration security, see *Implementing Security with B2B Integration.*

Key Tasks for XOCP Applications

This section introduces the key tasks that XOCP applications perform:

- Creating an XOCP Application Session
- Registering for a Role in a Conversation
- Engaging in Conversations with Trading Partners
- Shutting Down an XOCP Application Session

Creating an XOCP Application Session

Before exchanging business messages, an XOCP application must create an XOCP application session for the trading partner and its associated delivery channel.

Before a trading partner XOCP application can create an XOCP application session:

- Configuration information about the delivery channel and trading partner must be defined in the WebLogic Integration repositories for *both* the hub and spoke delivery channels associated with the collaboration agreement. For more information, see "Configuration Requirements" in *Administering B2B Integration*.
- The trading partner must be authorized to access the delivery channel.

Note: If the machine hosting the XOCP application associated with the spoke delivery channel crashes after connecting to a hub delivery channel, the XOCP application can reconnect to the hub delivery channel upon normal startup. The previous XOCP application session is discarded and new resources are assigned to the new XOCP application session. However, the intermediary cannot deliver business messages while the machine associated with the spoke delivery channel is down. Undelivered business messages are discarded if the number of retry attempts is exceeded or if the business message or conversation times out.

When a trading partner no longer wants to exchange business messages with other trading partners, the XOCP application shuts down the XOCP application session, as described in "Shutting Down an XOCP Application Session" on page 1-17.

Registering for a Role in a Conversation

After the XOCP application session has been created, a trading partner needs to register a message listener for the conversation type to which it is bound by the collaboration agreement. The message listener must be registered for a conversation type that defines how the trading partner participates in the conversation.

Role registration requires the following information in the repository associated with the hub delivery channel:

- The *conversation type*—is a subset of a conversation definition that defines a conversation for one trading partner based on the trading partner's role in the conversation definition to which it is 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 for a part is XML, then a document definition is required for that part. For the binary type, no other information is required.

For an introduction to these concepts, see "Overview" in *Introducing B2B Integration.*

Before registering a message listener for a conversation type in a collaboration agreement, the trading partner must first be authorized to register. Authorization is configured by the administrator of the intermediary and is based on the trading partner's subscription to a role in a conversation definition.

When an XOCP application session attempts to register a message listener for a specific conversation type in a collaboration agreement, the spoke delivery channel sends an XOCP system message, register for conversation, to the intermediary. The intermediary validates the role of the trading partner for the requested conversation type in the associated delivery channel. 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 intermediary itself. However, the XOCP application session maintains some state information about the conversations in which it is involved.

Conversation initiator XOCP applications and conversation participant XOCP applications are very similar. However, conversation initiator XOCP applications can terminate conversations while conversation participant XOCP applications cannot. Conversation participant XOCP applications can only leave a conversation.

Initiating a Conversation and Sending a Business Message

To initiate a conversation, a conversation initiator XOCP application first creates it. Optionally, the conversation initiator XOCP application can specify a timeout value, after which the conversation automatically terminates; this value overrides the timeout value that is specified in the associated conversation definition in the repository.

The local conversation coordinator on the B2B spoke sends an XOCP system message, create conversation of the specific collaboration agreement, to the intermediary. The global conversation coordinator in the intermediary creates a conversation using the appropriate delivery channel and enlists the trading partner as the conversation initiator. After the conversation is created, the conversation initiator XOCP application creates and sends a business message, as described in "Sending XOCP Business Messages" on page 3-1.

Participating in a Conversation

The global conversation coordinator in the intermediary handles all business messages that the intermediary receives for a given conversation. After the intermediary delivers an 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 XOCP application session on a target spoke delivery channel 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 message listener. For more information, see "Receiving XOCP Business Messages" on page 4-1.

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 it has finished participating in a conversation, a conversation participant trading partner can leave it. When a trading partner leaves a conversation, it is removed, by the conversation coordinator, from the list of participating trading partners. Subsequent business messages in that conversation are *not* sent to that trading partner. After a trading partner leaves, it is kept in a `DROPPEDOUT` state for the remainder of that conversation.

Terminating Conversations

A conversation terminates when the initiating trading partner explicitly terminates the conversation, or when the conversation times out; whichever occurs first. A trading partner who initiates 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. This message is propagated as the callback `onTerminate` on registered message listeners in XOCP application sessions on B2B spokes.

Shutting Down an XOCP Application Session

When a trading partner finishes all the activities in a conversation, the XOCP application shuts down the XOCP application session. When an XOCP application shuts down an XOCP application session, the B2B engine associated with the spoke unregisters with the intermediary all the collaboration agreements associated with this session. This causes the intermediary to unregister the associated conversation type. In response, the conversation coordinator automatically terminates all of the conversations that the trading partner has initiated in the XOCP application session and delists the trading partner from all other conversations in which it was participating.

When a trading partner shuts down an XOCP application session, the consequences are as follows:

- The intermediary is stopped from sending any further messages to the trading partner at the specified delivery channel.
- All conversations initiated by the trading partner are terminated.
- The trading partner leaves any conversations in which it is participating.
- The trading partner reclaims resources allocated in the intermediary for the shut down XOCP application session.

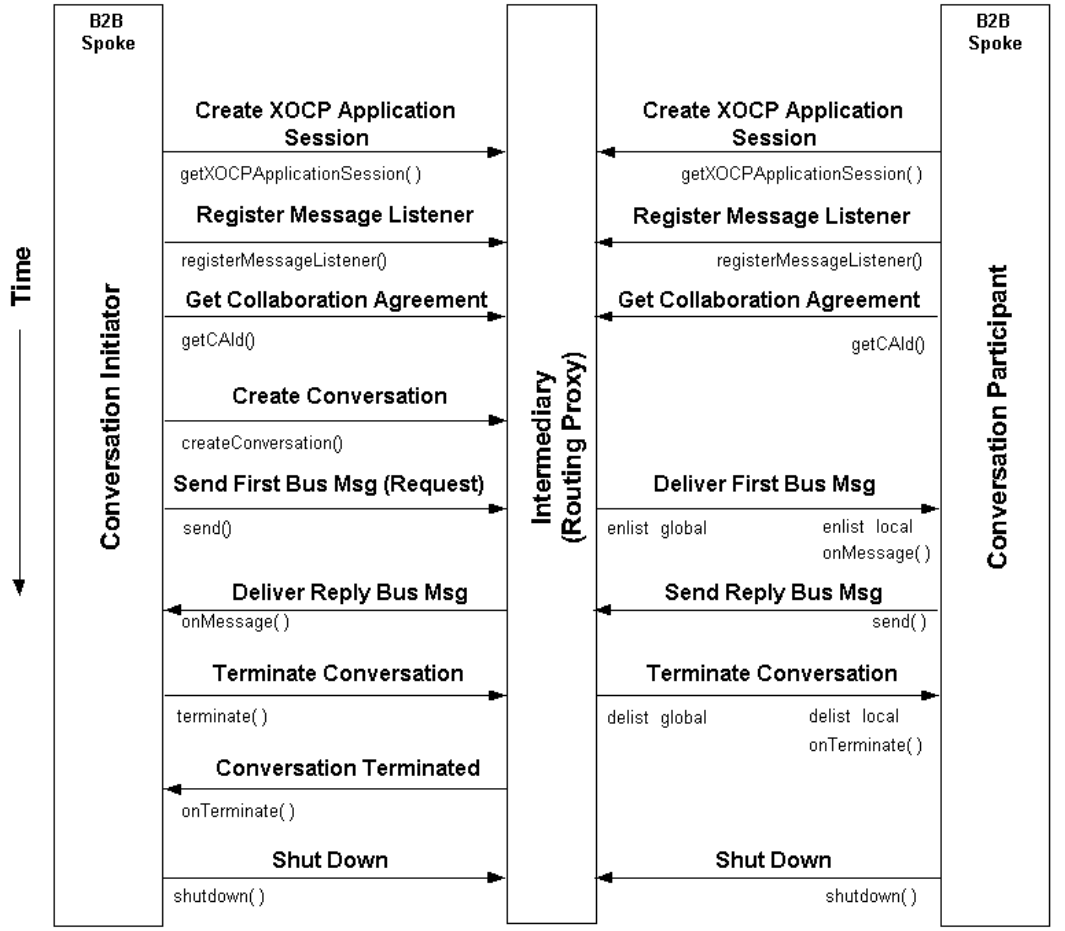
Run-Time Information Flow

At run time, all XOCP applications perform certain tasks identically: they connect to a delivery channel, register message listeners, and shut down the application session in the same way. 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 a conversation initiator and a participant.

Figure 1-6 Information Flow Between Conversation Initiator and Participant



This is a simplified example that involves 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 message listener and before shutting down an XOCP application session. 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 XOCP applications communicating through the intermediary) proceeds in the following sequence:

1. Each trading partner with a specific delivery channel creates an XOCP application session.
2. Each trading partner XOCP application registers a message listener with the XOCP application session, which, in turn (with the help of the local conversation coordinator), registers that trading partner for a given role in a conversation in a given collaboration agreement maintained by the intermediary.
3. Each trading partner XOCP application gets the collaboration agreement ID, if it is not known.
4. The conversation starts when it is created by the conversation initiator XOCP application.
5. The global conversation coordinator adds the conversation instance to its global conversation list and marks the trading partner as the initiator.
6. The local conversation coordinator in the conversation initiator adds the conversation instance to its local conversation list.
7. The conversation initiator's XOCP application creates and sends a business message (such as a request).
8. The conversation initiator's XOCP application session delivers the business message to the hub delivery channel in the intermediary.
9. The intermediary delivers the business message to the conversation participant's spoke delivery channel.

10. The global conversation coordinator in the intermediary enlists the participating trading partner in the conversation, adding it to the conversation instance entry in the global conversation list.
11. 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.
12. The `onMessage` implementation in the conversation participant XOCP application is invoked and processes the business message.
13. The conversation participant XOCP application creates and sends a business message (such as a reply) back to the conversation initiator.
14. The XOCP application session associated with the conversation participant delivers the business message to the intermediary.
15. The intermediary receives the business message and delivers it to the conversation initiator.
16. The conversation initiator receives the business message.
17. The `onMessage` implementation in the conversation initiator XOCP application is invoked and processes the business message.
18. To end the conversation, the conversation initiator XOCP application terminates it.

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

19. The local conversation coordinator in the conversation initiator delivers notification of termination to the global conversation coordinator in the intermediary.
20. The global conversation coordinator in the intermediary delists the conversation participant from the global conversation list and delivers notification of termination to the local conversation coordinator associated with the conversation participant.
21. The local conversation coordinator associated with the conversation participant receives the termination notification and delists the conversation from the local conversation list.

22. The `onTerminate` implementation in the conversation participation XOCP application is invoked.
23. The global conversation coordinator in the intermediary marks the conversation terminated and informs the conversation initiator by sending a conversation termination confirmation.
24. The conversation initiator receives the conversation termination confirmation.
25. The local conversation coordinator on the conversation initiator receives the termination notification and delists the conversation from the local conversation list.
26. The `onTerminate` implementation in the conversation initiator XOCP application is invoked.
27. Each trading partner XOCP application shuts down its respective XOCP application session.

For more information about these steps, see "Key Tasks for XOCP Applications" on page 1-13.

CHAPTER

2 Programming Steps for XOCP Applications

The following sections describe each step in the procedure that a developer usually provides in an XOCP application:

- Step 1: Import Packages
- Step 2: Implement the MessageListener Interface
- Step 3: Create an XOCP Application Session
- Step 4: Create and Register a Message Listener
- Step 5: Initiate or Participate in a Conversation
- Step 6: Exchange Business Messages
- Step 7: End the Conversation
- Step 8: Shut Down the XOCP Application Session

Each section includes example code from the Messaging API sample, which is fully described in *Running the B2B Integration Samples*.

Note: Before you can run an XOCP application, the administrator must specify a collaboration agreement for the trading partners who participate in the conversation associated with that XOCP application. For more information, see "Configuration Requirements" in *Administering B2B Integration*. For information about backward compatibility with XOCP applications created with earlier versions of WebLogic Integration, see *Migrating to BEA WebLogic Integration Release 2.1*.

Step 1: Import Packages

XOCP applications import the required packages from the Messaging API class library. For a description of these packages, see “Messaging API Class Library” on page 1-5.

The following example listing shows the type of packages that must be imported.

Listing 2-1 Importing Packages

```
import java.util.Properties;
import com.bea.b2b.protocol.xocp.application.*;
import com.bea.b2b.protocol.xocp.messaging.*;
import com.bea.b2b.protocol.conversation.ConversationType;
import com.bea.b2b.protocol.messaging.PayloadPart;
import com.bea.b2b.protocol.xocp.conversation.local.Conversation;
import com.bea.eci.logging.*;
```

Step 2: Implement the MessageListener Interface

To receive messages, an XOCP application must implement the following interface:

`com.bea.b2b.protocol.xocp.messaging.XOCPMessageListener`

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 a B2B spoke receives a business message. The `onTerminate` method is invoked when a B2B spoke receives notification of a conversation termination.

The message listener is required in order for the trading partner to receive business messages in a conversation. An XOCP application session supports one message listener per collaboration agreement.

Listing 2-2 Implementation of the MessageListener Interface

```
public class Partner1MessageListener
    implements XOCPMessageListener{

    public void onMessage(XOCPMessage rmsg){

      counter ++;

      QualityOfService qos = rmsg.getQoS();

      // Received reply, time to wake up waiter

      synchronized(waiter){

        debug("onMessage in waiter counter = " + counter);

        PayloadPart[] payload = rmsg.getPayloadParts();
        // we are using a single part document
        if (payload != null && payload.length > 0){
          BusinessDocument bd = (BusinessDocument)payload[0];
          waiter.reply = bd.getDocument();
        }
        waiter.done = true;
        waiter.notify();
      }
    }

    public void onTerminate(Conversation conv, int result)
    }
  }
```

For detailed information about the `XOCPMessageListener` interface, see *BEA WebLogic Integration Javadoc* on the WebLogic Integration documentation CD or, in Windows systems, choose the BEA WebLogic e-Business Platform→WebLogic Integration 2.1→Javadocs from the Windows Start menu.

Step 3: Create an XOCP Application Session

To initiate or participate in conversations, a trading partner creates an XOCP application session associated with a local B2B spoke delivery channel. Each XOCP application session enables the associated trading partner to exchange messages with other trading partners in a conversation.

To create a new XOCP application session or to get an existing one, use the `com.bea.b2b.protocol.xocp.application.XOCPApplication` class. The following listing is an example of getting the `MdmApp1` XOCP application session, based on the trading partner name (`Partner 1`) and the delivery channel (`Partner1-Channel0`).

Listing 2-3 Obtaining an XOCP Application Session

```
XOCPApplication app = XOCPApplication.getXOCPApplication("MdmApp1");
XOCPApplicationSession es = app.getXOCPApplicationSession("Partner1",
"Partner1-Channel0");
```

Step 4: Create and Register a Message Listener

To participate in a conversation, an XOCP application must register a message listener. A message listener is implemented by the application; the developer is responsible for using it as needed.

To register a message listener, an XOCP application calls the `registerMessageListener` method on the `XOCPApplicationSession` instance, passing the collaboration agreement ID, the conversation role of the trading partner, and the message listener object as parameters.

The following example listing shows how to register a message listener for a requestor role (generally a conversation initiator) in the `verifierConversation` conversation. Note that the required collaboration agreement ID and role must be specified in the repositories of the trading partner and the intermediary respectively.

Listing 2-4 Registering a Message Listener

```
Partner1MessageListener ml =  new Partner1MessageListener();

      Properties prop = new Properties();
      prop.setProperty("BusinessProcessName", "verifierConversation");
      prop.setProperty("BusinessProcessVersion", "1.0");
      prop.setProperty("otherTradingPartner", "Hub");
      prop.setProperty("toRole", "replier");
      String caId = es.getCAId(prop);

      String myRole = "requestor";

      es.registerMessageListener(caId, myRole, ml);
```

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.protocol.xocp.application.XOCPApplicationSession` instance, passing the collaboration agreement ID, the trading partner role, and, optionally, the conversation timeout value, which is specified in seconds. (The default value is zero, or no timeout, if the configured timeout is also zero in the conversation definition in both the trading partner's and intermediary's respective repositories.) The trading partner must be registered in the initiator role in the conversation definition.

The following example listing shows how a conversation is initiated.

Listing 2-5 Initiating a Conversation

```
      long timeout = 0;
     Conversation c = es.createConversation(caId, myRole, timeout);
```

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 conversation.

Initially, the conversation initiator application creates and sends a business message (such as a request) to one or more trading partners in the conversation. 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. Trading partners may send and receive several business messages in the course of a conversation. For more information about exchanging business messages, see "Sending XOCP Business Messages" on page 3-1 and "Receiving XOCP Business Messages" on page 4-1.

Step 7: End the Conversation

A conversation can end after trading partners finish exchanging business messages in that conversation. The way in which a trading partner ends its 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 XOCP application calls the `leave` method on the `Conversation` instance, passing `false`. No messages are retained in the intermediary while the participant is not participating.

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

The following example listing shows how a participant leaves a conversation.

Listing 2-6 Leaving a Conversation

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

Initiator Terminates a Conversation

Conversation initiators can explicitly *terminate* a conversation or wait until the conversation times out. (The conversation initiator can specify a timeout value when it creates the conversation, or it can specify zero to use the timeout value defined for the conversation in the trading partner's and intermediary's repositories.) 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 XOCP application calls the `terminate` method in its implementation of the `Conversation` interface, as shown in the following listing.

Listing 2-7 Terminating a Conversation

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

Step 8: Shut Down the XOCP Application Session

To shut down an XOCP application session and leave the conversation, an application uses the `shutDown` method in its implementation of the `XOCPApplicationSession` interface. The following example listing shows how an XOCP application session is shut down.

Listing 2-8 Shutting Down an XOCP Application Session

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

If an XOCP application shuts down an XOCP application session, the trading partner leaves the conversation automatically and permanently.

3 Sending XOCP Business Messages

The following sections describe how an XOCP application sends XOCP business messages to one or more trading partners in a conversation:

- Step 1: Create the Business Message
- Step 2: Specify the Recipients of the Business Message (Optional)
- 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, an XOCP application constructs a business document, creates a business message, specifies message routing criteria and Quality of Service delivery options, and sends the business message to the intermediary for processing. The XOCP 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 1-6.

Step 1: Create the Business Message

To create a business message, an XOCP application first creates a message payload, which consists of any business documents and attachments to be sent.

The creation of a payload involves three steps:

1. Importing the Required Packages
2. Creating Payload Parts
3. Creating the XOCP Business Message and Adding Payload Parts

This section describes these three steps. For an introduction to the components of a business message, see "XOCP Business Messages and Message Envelopes" on page 1-6.

Importing the Required Packages

To create a business message, an XOCP application imports the necessary packages, as shown in the following listing.

Listing 3-1 Importing Packages for Business Message Creation

```
class java.io.FileInputStream;
import org.apache.xerces.dom.*;
import com.bea.b2b.protocol.xocp.application.*;
import com.bea.b2b.protocol.xocp.messaging.*;
import com.bea.b2b.protocol.messaging.Attachment;
import com.bea.eci.logging.*;
```

Creating Payload Parts

An XOCP application next creates a message payload, which includes business documents and/or 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`). An XOCP 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 used by the Messaging API sample to create and process XML documents.

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

The following code from the `MdmTp1Servlet` of the Messaging API application creates a request in the form of an XML document.

Listing 3-2 Creating an XML Document

```
// Create a request document
DOMImplementationImpl domi = new DOMImplementationImpl();
DocumentType dType =
        domi.createDocumentType("request", null, "request.dtd");
org.w3c.dom.Document rq1 = new DocumentImpl(dType);
Element root1 = rq1.createElement("request");
String sendStr1 = "FIRST MESSAGE"; // the actual string data to be
                                   // processed by the other partner
root1.appendChild(rq1.createTextNode(sendStr1));
rq1.appendChild(root1);
```

After creating an XML document, an XOCP application creates a `BusinessDocument` object, passing the XML document (`payload[0]`) as a parameter to the constructor, as shown in the following listing.

Listing 3-3 Creating a BusinessDocument

```
BusinessDocument bd = (BusinessDocument)payload[0];
```

Creating Attachments

A business message can contain one or more attachments. An attachment is a nonXML-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 *BEA WebLogic Integration Javadoc*.

The following example listing shows how to create an attachment.

Listing 3-4 Creating an Attachment

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

Creating the XOCP Business Message and Adding Payload Parts

After creating a message payload, an XOCP application creates an 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 *BEA WebLogic Integration Javadoc*.

To construct the business message, an XOCP 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`, which adds a single business document or attachment to the business message

- `addPayLoadParts`, which adds multiple business documents or attachments to the business message

In the following listing an XOCP business message is created and a payload part is added to it.

Listing 3-5 Creating a Business Message and Adding Payload Parts

```
XOCPMessage smsg1 = new XOCPMessage("");
smsg1.addPayloadPart(new BusinessDocument(rq1));
```

Note: The XOCP application clones the `XOCPMessage` content (except its payload parts) before sending it to the intermediary. Therefore, a payload part must not be changed after the application invokes the `send` or `sendAndWait` method on the `XOCPMessage`.

Step 2: Specify the Recipients of the Business Message (Optional)

After creating a business message, an XOCP application may specify the trading partner to which the message will be sent. An XOCP 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, an XOCP application might send a business message to a set of trading partners (via multicasting) when certain business criteria (represented by XOCP XPath expressions) are met. For example, an application might send a message via multicasting when a buyer sends a bid request to multiple sellers of a particular product.

Either way, the set of eligible trading partners is limited to those who are subscribed to the appropriate role in the conversation definition. In addition, router and filter expressions defined in the intermediary repository may also affect message delivery to particular trading partners. For more information, see "Advanced Configuration Tasks" in *Administering B2B Integration*.

Specifying a Particular Trading Partner

If an XOCP business message is sent to a single, known trading partner, an XOCP 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 intermediary repository.

The following example listing shows how a trading partner named `ChipMaker` is specified as the recipient of a business message.

Listing 3-6 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 intermediary does not perform the usual router processing, such as the evaluation of trading partner or intermediary XPath expressions. However, the business message is still subject to applicable filtering in the intermediary. For more information, see "Advanced Configuration Tasks" in *Administering B2B Integration*.

Using XPath Expressions to Specify Message Recipient Criteria

An XOCP application can use XPath expressions to specify the criteria for a set of trading partners that are to receive a business message. XPath expressions are used to address parts of an XML document. For more information, see "Advanced Configuration Tasks" in *Administering B2B Integration*.

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

- Standard attributes, such as the trading partner name or a postal code
- Extended properties: custom attributes, elements, and text defined by the administrator of the intermediary

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

If applicable trading partner and intermediary XPath expressions are defined in the intermediary repository, the B2B engine hosting the intermediary evaluates these expressions after it receives the business message. Depending on how they are configured, these XPath expressions might override or append the XOCP XPath expression that the XOCP application specifies. For more information, see "Advanced Configuration Tasks" in *Administering B2B Integration*.

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

Note: Apache Xalan v 1.0.1 supports single quotes, but not double quotes, to delimit string literals.

Before the business message is delivered, it undergoes applicable router and filter processing in the intermediary.

Specifying Standard Trading Partner Attributes

The following listing shows an XOCP XPath expression that selects the trading partner with the specified name.

Listing 3-7 XOCP XPath Expression Specifying a Trading Partner Name

```
smsg1.setExpression("//trading-partner[@name=\'Partner2\']");
```

The following listing shows an XOCP XPath expression that selects the trading partner with an address that contains the string `San`.

Listing 3-8 XOCP XPath Expression Specifying a Trading Partner Address

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

Specifying an XOCP XPath Expression Using Extended Properties

Extended properties are user-defined elements, attributes, and text that can be associated with trading partners in the repository in the intermediary. These properties provide application extensions to the standard predefined attributes in the repository. The extended property sets are modeled in the repository such that they can be retrieved as subtrees within an XML document. Extended properties are configured on the Trading Partners tab in the WebLogic Integration B2B Console. For more information, see "Using Advanced Trading Partner Configuration Options" in "Configuring B2B Integration" in *Online Help for the WebLogic Integration B2B Console*.

XOCP XPath expressions can refer to these extended properties to assist with business message routing. For example, suppose the administrator for the intermediary adds an extended property called Maximum Order Quantity so that sellers can indicate, in the intermediary repository, the largest quantity that they can accommodate. With this property defined, a buyer with a large order can specify an XOCP XPath expression that sends the business message only to sellers that are qualified to 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 3-9 Extended Property Set in XML Document Generated from the Repository

```
<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>
```

```
. . .
</hub>
```

The following listing shows an XOCP XPath expression that selects trading partners that can accommodate orders larger than 500 units.

Listing 3-10 XOCP 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, it is selected as a recipient of this business message.

Step 3: Specify the Quality of Service for Message Delivery

The B2B engine messaging service allows XOCP applications to define the *Quality of Service* (QoS), or level of reliability, to enforce when delivering a business message to recipient trading partners. Quality of Service settings are stored in the message header of the business message. The messaging service supports the reliable delivery of messages in the event of network-link or node failures. The messaging service 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 B2B engine messaging service provides certain automatic Quality of Service features that do not require input from XOCP applications:

- The B2B engine prevents duplicate message delivery.
- The B2B engine affixes a timestamp to every business message when it arrives at an intermediary or an XOCP application node. Timestamps can be helpful for taking performance measurements and with debugging applications.

QualityOfService Class

The `com.bea.b2b.protocol.xocp.messaging.QualityOfService` class represents Quality of Service settings for business messages. The `QualityOfService` class defines the level of reliability required from the B2B engine messaging service when it delivers a specific message. It also identifies, to the B2B engine messaging service, the XOCP application's expectation of how the business message will be delivered.

An XOCP application creates an instance of this class and calls methods on this instance to specify various Quality of Service settings. It then calls the `setQoS` method on the message instance, passing the `QualityOfService` object as a parameter, to associate the settings with the message. If an XOCP application does not specify Quality of Service settings, the B2B engine messaging service uses 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 3-1 Quality of Service Settings, Options, and Default Values

QoS Setting / Description	Options	Default Value(s)
`CONFIRMED_DELIVERY_TO_APPLICATION` ■ Provides confirmation of application delivery up to the receiving application. ■ Provides complete status of delivery to each destination, including receipt timestamp, list of router-selected trading partners, final list of recipient trading partners, and so on. ■ Provides complete message tracking information (of all potential locations) for the intermediary administrator and the administrator of the trading partner sending the message.	Not Applicable	Not Applicable
`CONFIRMED_DELIVERY_TO_DESTINATION(S)` ■ Provides complete status of delivery to each destination, including receipt timestamp, list of router-selected trading partners, final list of recipient trading partners, and so on. ■ Provides complete message tracking information (of all potential locations) for the intermediary administrator and the administrator of the trading partner sending the message.	Not applicable	Not applicable
`CONFIRMED_ROUTING` ■ Provides information from the XOCP router on the intermediary about the trading partners selected to receive the business message. ■ Provides message tracking for the administrator of the trading partner sending the messages (until the business message reaches the intermediary's XOCP router).	Not applicable	Not applicable
`CONFIRMED_DELIVERY_TO_HUB` (Default) ■ Verifies that the message reached the intermediary. ■ No message tracking for the administrator of the trading partner sending the message.	Not applicable	Not applicable
`TIMEOUT`	Timeout, in milliseconds, after send	Ignored

Table 3-1 Quality of Service Settings, Options, and Default Values (Continued)

QoS Setting / Description	Options	Default Value(s)
RETRY_ATTEMPTS	0-n	As configured for the intermediary
CORRELATION_ID	Application-defined field	Ignored

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

Table 3-2 How QoS Settings Affect Message Tracking and Acknowledgment

Quality of Service Setting	Message Tracking (Y/N)?	Acknowledgment (Y/N)?
CONFIRMED_DELIVERY_TO_APPLICATIONS	Y	Y
CONFIRMED_DELIVERY_TO_DESINATION(S)	Y	Y
CONFIRMED_DELIVERY_TO_ROUTER	Y	N
CONFIRMED_DELIVERY_TO_HUB	N	N

If the CONFIRMED_DELIVERY_TO_DESINATION(S) setting is used, then complete message tracking is available and acknowledgments are used to make sure that the message is delivered reliably 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 example listing shows how to set the Quality of Service for a business message.

Listing 3-11 Setting the Quality of Service for a Business Message

```
// Relevant imports
import com.bea.b2b.protocol.xocp.messaging.XOCPMessage;
import com.bea.b2b.protocol.xocp.messaging.QualityOfService;

XOCPMessage msg = . . .
// Create QoS object
QualityOfService qos = new QualityOfService();
// 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, an XOCP application calls one of the following methods on the `QualityOfService` instance, passing the Boolean `true` parameter to enable the desired option.

Table 3-3 Message Delivery Confirmation Levels

Durability Level	Description
`setConfirmedDeliveryToDestination`	Specifies whether to confirm message delivery up to its destination (true) or only up to the intermediary (false).
`setConfirmedDeliveryToHub`	Specifies whether to confirm message delivery up to the intermediary (true) or not (false).
`setConfirmedDeliveryToRouter`	Specifies whether to confirm message delivery up to the XOCP router in the intermediary (true) or only up to the intermediary (false).
`setConfirmedDeliveryToApplication`	Sets the Quality of Service for this business message, specifying whether to confirm message delivery up to the target application (true) or only up to the intermediary (false).

The following example listing shows how to set the message confirmation level up to its destination.

Listing 3-12 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 3-19.

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 recipient does not process it, and all other processing of it, including acknowledgment processing and delivery retries, is abandoned.

Timeout Algorithm

The B2B engine does not synchronize the different clocks used by its components, which can reside in different machines at different locations. Instead, the B2B engine uses a relative time algorithm.

Based on this algorithm, the amount of time remaining before the timeout of a business message (relative to the absolute time at which the component finished processing the business message) is specified in the business message when that message is sent to another component.

In the receiving component, the timeout calculations are based on the amount of time remaining to process the message, expressed through both an absolute time (indicating the arrival of the message) and a relative time (embedded in the message itself). This algorithm at least ensures that the actual message timeout in the system always occurs after the original timeout specified by the application.

```
Message Timeout on the Hub = Message timeout specified by the XOCP
application when sending a message
```

```
Message Timeout on the Sending XOCP Application = Message Timeout
on the Hub + N x Delta
```

In these settings:

- *N* = A predefined number in the system, such as `10`
- *Delta* = Estimated amount of time required for a message to travel, round-trip, between the sending trading partner and the intermediary

Setting the Number of Delivery Retry Attempts

If an attempt to deliver a business message fails due to intermittent network failures, the B2B engine messaging service attempts to retry sending the business message repeatedly until one of the following occurs:

- The business message is delivered (that is, delivery succeeds).
- 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 repository in the intermediary and are retrieved by an XOCP application when the XOCP application session is created. The B2B engine messaging service waits for the configured interval before attempting to resend a business message.

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

Listing 3-13 Specifying the Message Timeout

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

Setting the Correlation ID for a Business Message

An XOCP application can specify a unique correlation ID for a business message so that it can correlate business messages (such as replies to a request) received from trading partners in response to a previously sent message (such as a request). The correlation ID accompanies the business message to its destination. 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, an XOCP application invokes the `setCorrelationId` method on the `QualityOfService` instance, passing a string representing the correlation ID as a parameter, as shown in the following listing.

Listing 3-14 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, an XOCP application sends the business message in one of the following ways:

- Synchronous message delivery
- Deferred synchronous message delivery

When sending an XOCP business message with either synchronous or deferred synchronous delivery, you need to set the following values:

- The collaboration agreement ID associated with that message
- The conversation

Synchronous Message Delivery

With synchronous message delivery, the application waits until the message is delivered to the destinations. The B2B engine messaging service returns control to the application once the outcome of the activity of sending the message is known. The application waits until one of the following events occurs:

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

To send a business message synchronously, an XOCP application invokes the following methods on the `XOCPMessage` instance:

- The `setCAId` method, which sets the collaboration agreement ID
- The `setConversation` method, which sets the conversation in which to send the business message
- The `sendAndWait` method, which specifies the amount of time to wait (in milliseconds) before timing out. If you specify zero (0), the XOCP application waits until the business message reaches its respective destinations.

The following example shows how to send an XOCP business message using synchronous message delivery.

Listing 3-15 Sending a Message Using Synchronous Message Delivery

```
smsg1.setCAId(caId);
smsg1.setConversation(c);
MessageToken token = msg.sendAndWait(0);
```

Deferred Synchronous Message Delivery

With deferred synchronous message delivery, the B2B engine messaging service returns control to the XOCP application immediately after a message is sent, and returns a message token that the XOCP 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 the message was sent either terminates or times out.

To send a business message asynchronously, an XOCP application invokes the following methods on the `XOCPMessage` instance:

- The `setCAId` method, which sets the collaboration agreement ID
- The `setConversation` method, which specifies the conversation in which to send the business message
- The `send` method, which sends the message

The XOCP application 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 3-16 Sending a Message Using Deferred Synchronous Message Delivery

```
smsg1.setCAId(caId);
smsg1.setConversation(c);
token = msg.send();
...
token.waitForACK();
```

The `waitForAck` method blocks 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 invoked on the `XOCPMessage` instance return a message token that an XOCP 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 3-4 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 XOCP application. 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 intermediary.
Elapsed Time	Amount of time taken to deliver the business message to all destinations. This information is available only after acknowledgments have been received from all message destinations. Availability is subject to the specified Quality of Service delivery option.
Delivery Status	Delivery status from recipient destinations. This information depends on the availability of such information. Availability is subject to the specified Quality of Service delivery option.

Table 3-4 Message Token Information (Continued)

Attribute	Description
Number of Recipients (Router)	Number of recipient trading partners after the business message has been processed by the XOCP router in the intermediary. Availability is subject to the specified Quality 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 intermediary. Availability is subject to the specified Quality of Service delivery option.

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

Delivery Status Tracking

When a business message reaches its destination (the receive queue of the destination trading partner node), a system message is returned to the sender (by the B2B engine messaging service) to acknowledge the message delivery if an acknowledgment is required by the Quality of Service setting.

An XOCP 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 for the receipt of the business message
- Status code, valid values for which are shown in the following table

Table 3-5 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 during delivery of the business message to the destination.
RETRIES_EXHAUSTED	All delivery retry attempts were exhausted and the business message was discarded.
TIMEDOUT	Timeout occurred before message delivery and the business message was discarded.

Message Tracking Locations

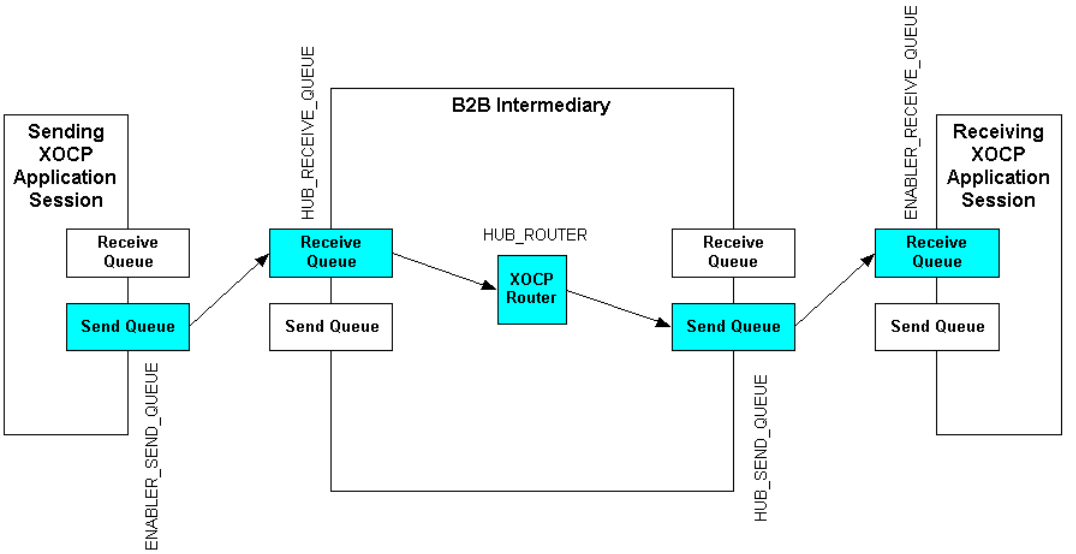
The B2B engine messaging service provides tracking features that allow administrators to check the progress of a business message as it moves through various predefined locations en route to its destination. The B2B Console can display status information as a business message passes through these tracking points. Administrators can use message tracking information for debugging and identifying bottlenecks in applications.

Note: The availability of message tracking locations depends on the configuration of the WebLogic Integration system and the specified Quality of Service for a given business message, such as CONFIRMED_DELIVERY_TO_DESTINATION(S). (For a description of Quality of Service settings, see Table 3-1). For example, if the XOCP application and the intermediary are collocated on the same node, some locations are not available. Similarly, some tracking locations may not be available for synchronous message delivery.

Diagram of Message Tracking Locations

The following figure shows the message tracking locations in the B2B engine messaging service.

Figure 3-1 Message Tracking Locations



Description of Message Tracking Locations

The following message tracking locations are potentially visible in the B2B Console.

Table 3-6 Message Tracking Locations

Location	Description of Location	Activity Performed
ENABLER_SEND_QUEUE	Send queue in the XOCP application session of the sending trading partner	Message is enqueued for sending.
HUB_RECEIVE_QUEUE	Receive queue for the sending trading partner in the intermediary	Message is enqueued in the receive queue of the sending trading partner in the intermediary.

Table 3-6 Message Tracking Locations (Continued)

Location	Description of Location	Activity Performed
HUB_ROUTER	XOCP router in the intermediary	Message has reached the XOCP router.
HUB_SEND_QUEUE	Send queue of the receiving trading partner in the intermediary	Message is enqueued for delivery in the target trading partner's queue in the intermediary.
ENABLER_RECEIVE_QUEUE	Receive queue in the XOCP application session of the receiving trading partner	Message is enqueued in the queue of the listener thread of the target trading partner's XOCP application session.

4 Receiving XOCP Business Messages

The following sections describe how to receive XOCP business messages in an XOCP application:

How XOCP Business Messages Are Received

XOCP applications must implement the `onMessage` method in the `MessageListener` interface to receive and process business messages. The signature for the `onMessage` method is as follows.

Listing 4-1 Signature for onMessage Method

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

Whenever a business message arrives, the XOCP application invokes the `onMessage` method, passing the business message as an input parameter. The XOCP application retrieves the `XOCPMessage` object containing the business message and then calls methods on that instance to process the message.

If an XOCP application receives multiple business documents in a conversation, the `onMessage` implementation first determines the type of document received (such as a bid request or bid reward), and then processes the document accordingly.

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

Receiving an XOCP Business Message

Listing 4-2 shows how the `onMessage` method is implemented in the `MdmTp2Servlet` of the Messaging API sample application. This `onMessage` implementation processes the initial business document (a request) sent from the `MsmTp1Servlet`. It then creates and sends a reply document back to the conversation initiator.

The following listing is the `onMessage` implementation in the `MdmTp2Servlet` of the Messaging API sample application.

Listing 4-2 onMessage Implementation in MdmTp2Servlet

```
public void onMessage(XOCPMessage rmsg) {
      try{

        QualityOfService qos = rmsg.getQoS();

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

        // we are using a single part document
        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();
    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());
    sendStr="Partner2 -- " + sendStr;
    root.appendChild(rq.createTextNode(sendStr.toLowerCase()));
    rq.appendChild(root);

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

    smsg.setQoS(qos);

    //TradingPartnerFilter filter = new TradingPartnerFilter("Partner1");
    smsg.setExpression("//trading-partner[@name=\'Partner1\']");
    smsg.setCAId(rmsg.getCAId());
    smsg.setConversation(conv);

    smsg.sendAndWait(0);

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

The `onMessage` code performs the following tasks:

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

   The application uses the same Quality of Service settings used in the original message 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 by using methods on the `Document` instance. The latter 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`).

   An XOCP 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`). This package is used by the Messaging API sample application 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 and specifies the same Quality of Service setting specified for the request document.

8. Sets the collaboration agreement ID and conversation, and sends the document to the trading partner called Partner 1.

Index

M

P

Q

R

S

T

X