BEA WebLogic Integration™

Implementing cXML for B2B Integration

Release 7.0
Document Date: June 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Implementing cXML for B2B Integration

Part Number	Date	Software Version
N/A	June 2002	7.0

Contents

4. Using Workflows with cXML

Index

About This Document

WebLogic Integration supports a routing architecture that allows it to manage and resolve XOCP, RosettaNet, and cXML messages. This architecture allows WebLogic Integration to engage in business-to-business conversations using any of these protocol standards.

This document describes the cXML capabilities of WebLogic Integration.

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

cXML on WebLogic Integration provides the ability to send and receive cXML messages as described in the *cXML User's Guide*, available at `http://www.cxml.org`.

This document is organized as follows:

- Chapter 1, "Introduction," provides an introduction to cXML on WebLogic Integration and the architecture used to implement cXML on WebLogic Integration.
- Chapter 2, "cXML Administration," describes cXML-specific administration and security issues for WebLogic Integration.
- Chapter 3, "Using the cXML API," describes the cXML API and how it is used.
- Chapter 4, "Using Workflows with cXML," describes how to use the WebLogic Integration Studio to create workflows for use with cXML.

What You Need to Know

This document is intended primarily for:

- Business process designers who use the WebLogic Integration Studio to design workflows that integrate with the WebLogic Integration environment, specifically focusing on cXML implementations.
- Application developers who use the cXML API to implement buyer or supplier applications using WebLogic Integration.
- System administrators who set up and administer WebLogic Integration applications in a cXML environment.

For an overview of the WebLogic Integration architecture, see "Overview" in the *Introducing B2B Integration* 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 WebLogic Integration 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 Javasoft Web site at the following URL: `http://java.sun.com`.

You will also find useful information at the BEA edocs Web site at the following URL: `http://edocs.bea.com`.

For more information about cXML, visit the cXML.org Web site at the following URL: `http://www.cxml.org`.

Contact Us!

Your feedback on the 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 Release 7.0.

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 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`*

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.

1 Introduction

Note: The cXML business protocol is deprecated as of this release of WebLogic Integration. For information about the features that are replacing it, see the *BEA WebLogic Integration Release Notes*.

This section introduces the cXML standard for electronic business transactions. cXML is an extensible e-commerce-oriented XML standard developed by Ariba and widely used for e-commerce purchasing transactions.

This section describes the following aspects of the cXML standard and its use with WebLogic Integration:

- WebLogic Integration Architecture and cXML
- cXML API
- Business Documents
- Digital Signatures and Shared Secrets
- Message Validation
- Limitations

WebLogic Integration Architecture and cXML

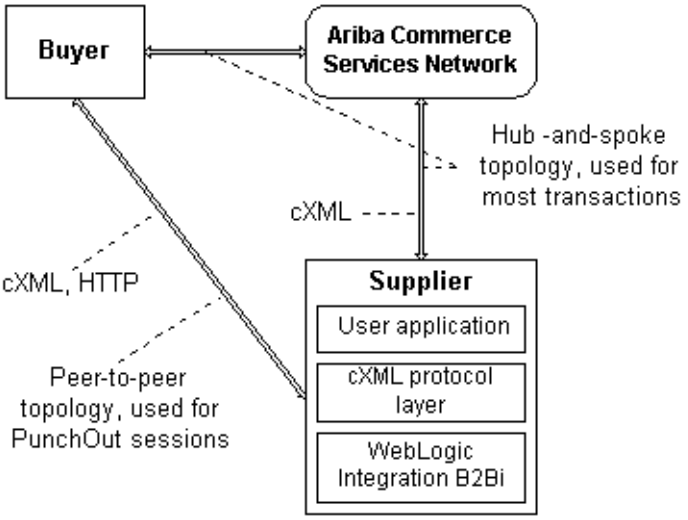
cXML support provided by WebLogic Integration consists of the following components:

- cXML protocol layer
- cXML API support

cXML integration is provided through the use of business process management (BPM) business operations and the cXML API. For more information, see Chapter 3, "Using the cXML API," and Chapter 4, "Using Workflows with cXML."

The following diagram illustrates the cXML architecture used by WebLogic Integration, and shows how WebLogic Integration interacts with other systems using cXML.

Figure 1-1 WebLogic Integration cXML Architecture



WebLogic Integration support for cXML is designed to allow seamless integration of cXML with the standard B2B integration infrastructure. For more information about the remainder of the B2B integration architecture, see *Introducing B2B Integration*.

Because of the design environment for cXML, support for hubs other than the Ariba Commerce Services Network is not provided.

cXML Protocol Layer

The cXML protocol layer provides the ability to send and receive messages via the Internet, according to the cXML specifications for transport, message packaging, and security. WebLogic Integration creates individual cXML sessions, each of which creates and manages a URL where the WebLogic Integration server can receive cXML

messages. You can configure cXML sessions, as needed, by using either the WebLogic Integration B2B Console or a configuration file. To use a WebLogic Integration configuration file, create a configuration file based on the `WLC.dtd` file to configure cXML sessions as needed. If you choose this approach, use the Bulk Loader to load the configuration file into the repository.

cXML API

WebLogic Integration includes comprehensive API support for the creation of cXML user applications. For more information about the cXML API, see Chapter 3, "Using the cXML API," and the *BEA WebLogic Integration Javadoc*.

Business Documents

Business document processing is performed in WebLogic Integration using a combination of *public* and *private* processes. *Public* processes are processes used to integrate and manage transactions between trading partners. *Private* processes are processes used internally by a trading partner; for example to communicate between a company's *public* processes and its internal ERP and CRM systems. *Private* processes are thus not directly exposed for trading partner consumption or use. For further explanation, see "Managing Business Processes" in "Overview" in *Introducing B2B Integration*.

cXML business documents are part of the public processes in which trading partners participate while performing e-business transactions. For example, a *PunchOut* is part of the process that a *Customer* trading partner performs with a *Product Supplier* trading partner to get information from a live repository on the price and availability of goods that the *Customer* wants to buy and the *Product Supplier* wants to sell. Trading partners planning to use *PunchOuts* must do the following:

- Implement the public process associated with the *PunchOut*
- Connect their internal systems, as well as their private processes and workflows, to the public process

WebLogic Integration implements all business documents available within cXML:

- Catalogs
- PunchOuts
- Purchase Orders
- Subscriptions

For further information on cXML business documents, go to the `cXML.org` Web site at the following URL:

`http://www.cxml.org`

Digital Signatures and Shared Secrets

The standard method of securing transactions in cXML is the *shared secret*. In cXML terms, a shared secret is typically a username/password combination, exchanged through secure transport before business communication begins.

WebLogic Integration includes full support for cXML shared secrets. For more information about implementing and configuring shared secrets, see the *Online Help for the WebLogic Integration B2B Console.* In addition, you may optionally use https transport for your messages.

In cXML v1.2, optional digital signatures based on the Base64-encoded X.509 V3 certificate model were introduced. These digital signatures are not the same as the RSA CertJ digital signatures implemented in WebLogic Integration. Currently WebLogic Integration does not support cXML digital signatures. For more information, visit the `cXML.org` Web site at the following URL:

`http://www.cxml.org`

Message Validation

The cXML standard requires all cXML documents to be valid and to refer to published cXML Document Type Definitions (DTDs). Validation is not required by the cXML standard, but it is provided by WebLogic Integration as a service.

Limitations

Several cXML-related features are not supported in this release of WebLogic Integration:

- Digital Certificates: As discussed earlier, cXML 1.2 digital certificates are not supported.
- cXML 1.2 attachments are not supported at this time.
- Non-ACSN hub support: No support is provided for any hub other than the Ariba Commerce Services Network. All hub-based transactions must be routed through the ACSN. Peer-to-peer support is provided.
- No Sample implementation is provided at this time.

CHAPTER

2 cXML Administration

Note: The cXML business protocol is deprecated as of this release of WebLogic Integration. For information about the features that are replacing it, see the *BEA WebLogic Integration Release Notes.*

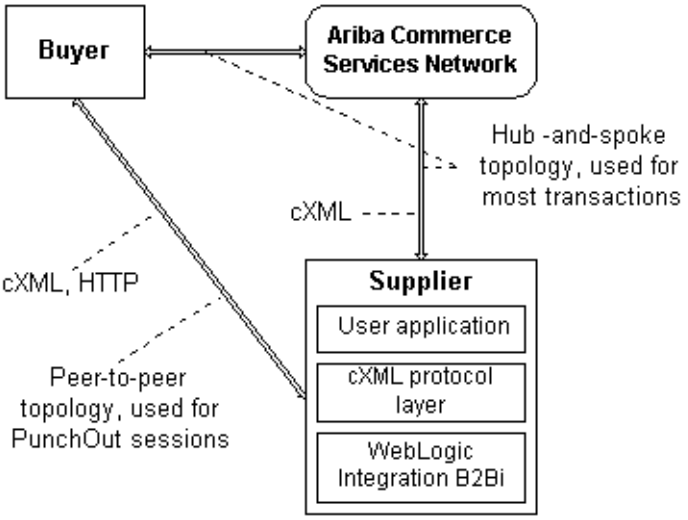
Administration of cXML transactions is performed using the WebLogic Integration B2B Console. The following sections describe the administrative work required to support cXML transactions:

- Connecting to Other cXML Trading Partners
- Collaboration Agreements
- Security

Connecting to Other cXML Trading Partners

Conversations between cXML trading partners use both peer-to-peer and hub-and-spoke configurations. While these configurations are discussed in *Introducing B2B Integration* as they apply to most situations, cXML varies slightly in its use of these configurations. The following illustration demonstrates how a routine cXML transaction uses both topologies simultaneously.

Figure 2-1 cXML Deployment Configurations



In this illustration, the Ariba Commerce Services Network (ACSN) is the hub. Most transactions are performed through this hub, with individual trading partners serving as the related spokes. However, when you browse a partner's catalog, a PunchOut trading session is created that connects directly to the remote system. In this case, the hub-and-spoke topology is bypassed in favor of a peer-to-peer configuration. Once the PunchOut session is finished, and the Buyer wants to send an Order or a Subscription, then the system topology reverts to a hub-and-spoke model, and the ACSN again acts as the hub.

It is important to note that when cXML is used with WebLogic Integration, the ACSN is the only authorized hub. WebLogic Integration does not provide support for any other hub-and-spoke deployments, and no other system is capable of acting as a hub for cXML-based transactions.

Collaboration Agreements

Collaboration agreements used with cXML are similar in scope and effect to collaboration agreements configured for other trading protocols. For more information about configuring collaboration agreements, see *Administering B2B Integration* and *Online Help for the WebLogic Integration B2B Console*.

The one significant difference in configuring collaboration agreements lies in how credentials are configured. Because cXML uses the Ariba Commerce Services Network as the authenticating hub, all credentials are configured in relation to the ACSN, not in relation to another trading partner.

In practice, this means that your shared secret is always registered with the ACSN, rather than one that is defined solely and specifically between you and a trading partner.

Security

WebLogic Integration provides support for the security model embodied in cXML 1.1, which uses the concept of shared secrets to verify message authenticity. Shared secrets are passwords or other text strings used to verify the identity of a given partner. Like a password, a given trading partner entity is linked to a specific shared secret, providing one-to-one identity mapping. There is no provision to prevent multiple trading partners from using identical shared secrets, however.

cXML 1.2 introduces a specific implementation of digital signatures based on the Base64-encoded X.509 V3 certificate model. Currently, WebLogic Integration does not support this implementation of digital signatures.

Configuring Shared Secrets

Use the WebLogic Integration B2B Console to configure shared secrets. For more information about this procedure, see *Online Help for the WebLogic Integration B2B Console*.

CHAPTER

3 Using the cXML API

Note: The cXML business protocol is deprecated as of this release of WebLogic Integration. For information about the features that are replacing it, see the *BEA WebLogic Integration Release Notes*.

The following sections describe some key programming issues for the cXML API:

- cXML Methods
- cXML Message Structure
- cXML DTDs
- Dealing with Shared Secrets
- Processing Incoming Messages
- Processing Outgoing Messages

For more information about programming business operations, see *Creating Workflows for B2B Integration*.

cXML Methods

The following table describes the methods available for cXML message manipulation.

Table 3-1 Public cXML Methods

Method	Package	Description
`onMessage`	`com.bea.b2b.protocol.cxml.CXMLListener`	Receives an incoming CXMLMessage
`deregister`	`com.bea.b2b.protocol.cxml.CXMLManager`	Deregisters the application with this CXMLManager. Uses a set of properties to select the registration.
`getInstance`	`com.bea.b2b.protocol.cxml.CXMLManager`	Gets an instance of the CXMLManager.
`getSharedSecret`	`com.bea.b2b.protocol.cxml.CXMLManager`	Gets the Shared Secret for this Trading Partner. Uses the Trading Partner name to find the Shared Secret.
`register`	`com.bea.b2b.protocol.cxml.CXMLManager`	Registers the application with this CXMLManager. Uses a set of properties to select the collaboration agreement for this Trading Partner. Use for sending cXML messages.
`getHttpStatusCode`	`com.bea.b2b.protocol.cxml.CXMLHttpStatusException`	Returns the HTTP Status code from the exception.
`getAsString`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the cXML part as a String.
`getDocument`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the associated XML Document.
`getFromCredentialDomains`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the From Credential Domains from the document header.
`getFromCredentialIdentities`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the From Credential Identities from the document header.

Table 3-1 Public cXML Methods

Method	Package	Description
`getIdentifier`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets either the document identifier or the message identifier, as appropriate.
`getNodeValue`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the value of a document node using the specified XPath expression to locate the node. If multiple nodes match the XPath expression, then only the first node is used.
`getSenderCredentialDomain`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the Sender Credential Domain from the document header.
`getSenderCredentialIdentity`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the Sender Credential Identity from the document header.
`getSenderSharedSecret`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the Sender Credential Shared Secret from the document header.
`getSenderUserAgent`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the Sender User Agent from the document header.
`getTimeStamp`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets either the document timestamp or the message timestamp, as appropriate.
`getToCredentialDomain`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the To Credential Domain from the document header.
`getToCredentialIdentity`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the To Credential Identity from the document header.
`getVersion`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Gets the document version.
`setDocument`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Sets the associated XML document.
`setNodeValue`	`com.bea.b2b.protocol.cxml.messaging.CXMLDocument`	Sets the value of a document node using the specified XPath expression to locate the node.

Table 3-1 Public cXML Methods

Method	Package	Description
`reply`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Replies to the request message. This method is only valid when this object is used as a parameter to `CXMLListener.onMessage()`. The reply may be sent asynchronously after the method has been called.
`getReplyDocument`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Gets the reply cXML document.
`getRequestDocument`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Gets the request cXML document.
`send`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Sends a request message. This method blocks until a reply is received. The reply can be accessed via `getReplyDocument()`.
`setCollaborationAgreement`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Sets the Collaboration Agreement ID for the collaboration agreement to which this message belongs. Uses a set of properties to select the Collaboration Agreement.
`setReplyDocument`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Sets the reply cXML document.
`setRequestDocument`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessage`	Sets the request cXML document.
`getHttpStatusCode`	`com.bea.b2b.protocol.cxml.messaging.CXMLMessageToken`	Gets the HTTP status code.

For more information about individual methods, see the *BEA WebLogic Integration Javadoc*.

Properties Used to Locate Collaboration Agreements

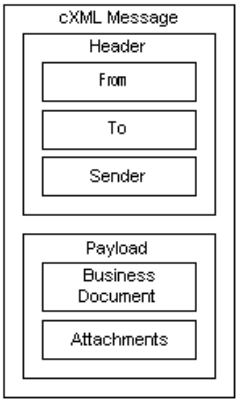
cXML uses a set of defined properties to locate unique collaboration agreements. When attempting to locate a specific collaboration agreement, you must supply values for all of the following properties:

- BusinessProcessName
- BusinessProcessVersion
- DeliveryChannel
- toRole
- fromTradingPartner
- toTradingPartner

cXML Message Structure

A cXML message is based on the message envelope. The message envelope includes the following data structures.

Figure 3-1 cXML Message Architecture



The message data structures are as follows:

- Header—Contains addressing and validation information, including the From, To, and Sender data arrays. These data arrays provide information on the various parties to the transaction, as well as validation information for each of the participants.
- Payload—All the business documents and attachments that make up the body of a message.

Your application should be able to deal with all of the messaging objects included in the payload:

- Business document—One or more cXML documents, each containing cXML data, each part of which can be validated using cXML DTDs.
- cXML document—A cXML-standard document containing data. cXML documents can be validated individually using cXML DTDs.
- Attachment—An optional MIME-encoded binary attachment. WebLogic Integration does not support cXML attachments. If you want to use them, however, you can configure a private process to resolve MIME-encoded attachments.

cXML DTDs

The DTDs you will need are available at the following locations:

- cXML DTDs are available from the `cXML.org` Web site at the following URL:

  `http://xml.cxml.org/schemas/cXML/`*`version`*`/cXML.dtd`

  Here, *`version`* represents the full cXML version number (such as 1.1, 1.2, and so on).

- The Confirmation and Ship Notice transactions are contained in a separate DTD, located at the following URL:

  `http://xml.cxml.org/schemas/cXML/`*`version`*`/Fulfill.dtd`

  Here, *`version`* represents the full cXML version number (such as 1.1, 1.2, and so on).

Validation using these DTDs is not required when you send a cXML message. However, one assumption of the cXML messaging structure is that any message you send has been validated. Therefore it is a good idea to validate your messages routinely against the DTDs, at least while you are testing interoperability with a new trading partner. Once you are comfortable with your trading partner, you may optionally turn off message validation to enhance performance.

Dealing with Shared Secrets

The cXML API provides access to the value of the shared secret stored in the repository. The `GetSharedSecret` method allows you to retrieve the shared secret from the repository for comparison to the shared secret stored in incoming documents, or for use in outgoing cXML documents.

For incoming documents, your business operation code must perform verification of the shared secret of an incoming message by matching its value with the value specified in the configuration stored in the repository.

In each outgoing cXML document, your business operation code must insert the shared secret in the Credential node.

Processing Incoming Messages

To process an incoming message, you must first initialize it. The registration function associates a collaboration agreement with either a listener or a sending application. The token returned by the initialization process is used in the cXML message when the message is sent or received.

Initialization

To initialize an incoming message:

1. Get a copy of the listener object.
2. Define the return token:
   `private static CXMLToken token;`
3. Retrieve an instance of the cXML Manager class `com.bea.b2b.protocol.cxml.CXMLManager`:

   ```
   private static CXMLManager cxmlm = CXMLManager.getInstance();
   ```

4. Define a set of properties to register this application and listener. The properties are used to locate and map a unique collaboration agreement. For more information about the properties needed to map a unique collaboration agreement, see "Properties Used to Locate Collaboration Agreements" on page 3-5.

   ```
   prop.setProperty("BusinessProcess", businessProcess);
   prop.setProperty("BusinessProcessVersion",
   businessProcessVersion);
   prop.setProperty("DeliveryChannel", deliveryChannel);
   prop.setProperty("thisTradingPartner", myTradingPartnerName);
   prop.setProperty("otherTradingPartner",
   otherTradingPartnerName);
   prop.setProperty("toRole", toRole);
   prop.setProperty("Party", "duns4");
   ```

5. Invoke the register method from the `CXMLManager` class:
   `token = cxmlm.register(prop);`

Processing the Message

Once you have initiated an incoming message, as described in the "Initialization" section, you can process it. To do so, your application must:

1. Get the request cXML document from the received cXML message using the `onMessage()` callback method. This method passes the received cXML message from the WebLogic Integration run time to your application code.
2. Get the XML DOM document from the cXML document:

   ```
   // Get the cXML document
     CXMLDocument reqMsgDoc = cmsg.getRequestDocument();

     // Get the XML DOM doc
     Document reqXMLDoc = reqMsgDoc.getDocument();
   ```

3. Process the request document based on the payload.
4. Retrieve the shared secret from the incoming message. The shared secret for the trading partner is defined in the message, in:
   `//cXML/Header/From/Credential.`

   ```
   String otherSharedSecret =
   cxmlm.getSharedSecret(otherTradingPartnerName);
   ```
5. Verify that the shared secret from the message matches the shared secret defined in the configuration for the trading partner:

   ```
   debug("Stored Shared Secret for " + otherTradingPartnerName + ":
   " + otherSharedSecret);
   ```

   If the transaction is peer-to-peer, then the trading partner is the buyer or supplier. If the transaction is being conducted through the hub, then the trading partner is the hub.

   The following comparison failure options may occur.

Table 3-2 Verification Failure Options

Result	Reason
No comparison was performed	The shared secret has not been configured for the trading partner.
Message was rejected with *400* (bad request) http status code.	Request message could not be parsed. This problem should be resolved in the WebLogic Integration run time.
Message was rejected with *401* (unauthorized access) http status code.	Shared secrets do not match.
Message was rejected with *500* (Unable to forward request) http status code.	The listener was not properly configured. This should be resolved in the WebLogic Integration run time.

6. Create the reply XML DOM implementation document:

   ```
   DOMImplementationImpl domi = new DOMImplementationImpl();

   DocumentType dType =
   domi.createDocumentType("request", null, "cXML.dtd");
   ```

```
org.w3c.dom.Document punchoutDoc = new DocumentImpl(dType);
CxmlElementFactory cf = new CxmlElementFactory(punchoutDoc);
```

7. Create the reply cXML document:

```
Element request = punchoutDoc.createElement("Request");
```

8. Create the header elements in the document:

```
// header
cf.createHeaderElement(
// from
cf.createFromElement(
cf.createCredentialElement(
"DUNS",
myTradingPartnerName,
null)),
// to
cf.createToElement(
cf.createCredentialElement(
"DUNS",
otherTradingPartnerName,
null)),
// sender
cf.createSenderElement(
cf.createCredentialElement(
"AribaNetworkUserId",
"admin@acme.com",
otherSharedSecret),
"Ariba ORMS 5.1P4")),
```

9. Set the XML document in the cXML document:

```
CXMLDocument replyMsgDoc = new CXMLDocument();
replyMsgDoc.setDocument(replyXMLDoc);
```

10. Set the cXML document in the reply cXML message:

```
cmsg.setReplyDocument(replyMsgDoc);
```

11. Set the collaboration agreement in the cXML message:

```
cmsg.setCollaborationAgreement(prop);
```

12. Send the reply message to dispatch the outgoing cXML message from your application to the WebLogic Integration run time:

```
cmsg.reply();
```

Processing Outgoing Messages

You must initialize outgoing messages before you send them. To do so:

1. Define the return token:

   ```
   private static CXMLToken token;
   ```

2. Retrieve an instance of the cXML Manager class: `com.bea.b2b.protocol.cxml.CXMLManager`.

   ```
   private static CXMLManager cxmlm = CXMLManager.getInstance();
   ```

3. Define a set of properties to register this application. The properties are used to locate and map a unique collaboration agreement. For more information about the properties needed to map a unique collaboration agreement, see "Properties Used to Locate Collaboration Agreements" on page 3-5.

   ```
   prop.setProperty("BusinessProcess", businessProcess);
   prop.setProperty("BusinessProcessVersion",
   businessProcessVersion);
   prop.setProperty("DeliveryChannel", deliveryChannel);
   prop.setProperty("thisTradingPartner", myTradingPartnerName);
   prop.setProperty("otherTradingPartner",
   otherTradingPartnerName);
   prop.setProperty("toRole", toRole);
   prop.setProperty("Party", "duns4");
   ```

4. Invoke the `register` method:

   ```
   token = cxmlm.register(prop);
   ```

Sending the Message

To send a message, your application must perform the following actions:

1. Create a cXML message:

   ```
   DOMImplementationImpl domi = new DOMImplementationImpl();

   DocumentType dType =
   domi.createDocumentType("request", null, "cXML.dtd");
   ```

```
org.w3c.dom.Document punchoutDoc = new DocumentImpl(dType);
CxmlElementFactory cf = new CxmlElementFactory(punchoutDoc);
```

2. Create the XML DOM request document:

```
Element request = punchoutDoc.createElement("request");
Element trans =
punchoutDoc.createElement("PunchoutSetupRequest");
request.appendchild(trans);
```

3. Create the header elements in the request document:

```
punchoutDoc.appendChild(
cf.createCxmlElement(
// header
cf.createHeaderElement(
// from
cf.createFromElement(
cf.createCredentialElement(
"DUNS",
myTradingPartnerName,
null)),
// to
cf.createToElement(
cf.createCredentialElement(
"DUNS",
otherTradingPartnerName,
null)),
// sender
cf.createSenderElement(
cf.createCredentialElement(
"AribaNetworkUserId",
"admin@acme.com",
otherSharedSecret),
"Ariba ORMS 5.1P4")),
```

4. Retrieve the receiving trading partner's shared secret from the appropriate trading partner profile. For peer-to-peer messages, this will be the actual receiving trading partner's shared secret. For messages routed through a hub, this will be the hub's shared secret.

   The value of the receiving trading partner's shared secret (defined in `//cXML/Header/To/Credential`) is updated to the sender's shared secret element (defined in `//cXML/Header/Sender/Credential`):

```
String otherSharedSecret =
cxmlm.getSharedSecret(otherTradingPartnerName);
debug("Stored Shared Secret for " + otherTradingPartnerName + ":
" + otherSharedSecret);
```

5. Create the cXML document:

```
CXMLDocument reqMsgDoc = new CXMLDocument();
```

6. Set the cXML document in the cXML message:

```
reqMsgDoc.setDocument(reqXMLDoc);
cmsg.setRequestDocument(reqMsgDoc);
```

7. Set the collaboration agreement in the cXML message:

```
cmsg.setCollaborationAgreement(prop);
```

8. Send the message:

```
CXMLMessageToken sendToken = (CXMLMessageToken) cmsg.send();
```

9. Get the reply document:

```
CXMLDocument replyMsgDoc = cmsg.getReplyDocument();
```

10. Extract the XML document:

```
org.w3c.dom.Document replyXMLDoc = replyMsgDoc.getDocument();
```

11. Verify the response.

Code Samples

This section shows examples of code used by buyers and suppliers to process messages. These examples are provided solely to illustrate the operation of the cXML classes; they are not intended for execution. The examples below are configured for peer-to-peer operation.

For more information about cXML classes, see *BEA WebLogic Integration Javadoc*.

Sample Buyer

Listing 3-1 Code for Sample Buyer

```
/*
 *        Copyright (c) 2001 BEA
 *        All rights reserved
*/
package examples.ibcxmlverifier;
```

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

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

import com.bea.b2b.protocol.cxml.messaging.*;
import com.bea.b2b.protocol.cxml.*;

import com.bea.eci.logging.*;

/**
 * This example provides a simple test that will verify message flow of cXML
 * peer-to-peer sending and receiving a cXML document.
 * The two peers (Partner1 and Partner2) are running on a single WLS.
 * Partner1 sends a PunchoutRequest to Partner2.  Partner2 generates a
 * PunchoutSetupResponse and returns it to Partner1.  Shared Secrets are verified
 * at both ends.
 */
public class Partner1Servlet extends HttpServlet
{
  static final boolean DEBUG = true;

  private final static String businessProcess = "PunchoutSetup";
  private final static String businessProcessVersion = "1.1.009";
  private final static String deliveryChannel = "CXMLPartnerVerifier1";
  private final static String myTradingPartnerName = "CXMLPartnerVerifier1";
  private final static String otherTradingPartnerName = "CXMLPartnerVerifier2";
  private final static String toRole = "Supplier";
  private final static String expectedURL = "http://xyz/abc?from=" +
myTradingPartnerName;
  private DocSerializer ds;

  // Create the token for this application
  private static CXMLToken token;

  // Get the manager instance
  private static CXMLManager cxmlm = CXMLManager.getInstance();

  private static Properties prop = new Properties();

  public void init(ServletConfig sc) {
    try {
      debug("Initializing servlet for Partner1");
```

```
      // Set the properties for finding the Collaboration Agreement
      prop.setProperty("BusinessProcess", businessProcess);
      prop.setProperty("BusinessProcessVersion", businessProcessVersion);
      prop.setProperty("DeliveryChannel", deliveryChannel);
      prop.setProperty("thisTradingPartner", myTradingPartnerName);
      prop.setProperty("otherTradingPartner", otherTradingPartnerName);
      prop.setProperty("toRole", toRole);
      prop.setProperty("Party", "duns4");

      // Register the buyer with the manager using properties
      token = cxmlm.register(prop);

    } catch (Exception e) {
      debug("CXMLPartnerVerifier1 init exception: " + e);
      e.printStackTrace();
    }
  }

  private org.w3c.dom.Document getBusinessDocument() {
    DOMImplementationImpl domi = new DOMImplementationImpl();

    DocumentType dType =
      domi.createDocumentType("request", null, "cXML.dtd");

    org.w3c.dom.Document punchoutDoc = new DocumentImpl(dType);
    CxmlElementFactory cf = new CxmlElementFactory(punchoutDoc);

    try {
      String otherSharedSecret = cxmlm.getSharedSecret(otherTradingPartnerName);
      debug("Stored Shared Secret for " + otherTradingPartnerName + ": " +
otherSharedSecret);

      // Header
      Element request = punchoutDoc.createElement("Request");
      Element trans = punchoutDoc.createElement("PunchoutSetupRequest");
      request.appendChild(trans);

      punchoutDoc.appendChild(
        cf.createCxmlElement(
      // payload
      "1233444-200@ariba.acme.com",

      // header
      cf.createHeaderElement(
      // from
      cf.createFromElement(
      cf.createCredentialElement(
        "DUNS",
```

```
        myTradingPartnerName,
        null)),
      // to
      cf.createToElement(
      cf.createCredentialElement(
        "DUNS",
        otherTradingPartnerName,
        null)),
      // sender
      cf.createSenderElement(
      cf.createCredentialElement(
        "AribaNetworkUserId",
        "admin@acme.com",
        otherSharedSecret),
        "Ariba ORMS 5.1P4")),
      // request
      request));

    }
    catch( Exception e )  {
      debug("MessageDeliveryException: " + e.toString());
      e.printStackTrace();
    }
    return punchoutDoc;
  }

  /**
   * The actual work is done in this routine.  Construct a message document,
   * publish the message, wait for a reply, terminate and report back.
   */
  public void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
  {
    try {

      // setup for the reply display to client
      res.setContentType("text/html");
      PrintWriter pw = res.getWriter();
      pw.println("<HTML><BODY BGCOLOR=#ff0000>");
      pw.println("<P><IMG SRC=logo.jpg WIDTH=185 HEIGHT=156"+
        " ALIGN=TOP BORDER=0 NATURALSIZEFLAG=3></P>");
      pw.println("<P><FONT SIZE=-1>Partner1 process flow:<BR>");
      pw.println("Starting Partner1...");

      debug("Starting Partner1: get Document...");

      CXMLMessage cmsg = new CXMLMessage();

      org.w3c.dom.Document reqXMLDoc = getBusinessDocument();
```

```
CXMLDocument reqMsgDoc = new CXMLDocument();
reqMsgDoc.setDocument(reqXMLDoc);
cmsg.setRequestDocument(reqMsgDoc);

DocSerializer ds = new DocSerializer();

debug("buyer: request document:\n" +
  ds.docToString(reqXMLDoc, true) + "\n");

// Set the CA with the properties
cmsg.setCollaborationAgreement(prop);

// Send the message and get the reply
CXMLMessageToken sendToken = (CXMLMessageToken) cmsg.send();
CXMLDocument replyMsgDoc = cmsg.getReplyDocument();

debug("Got document");
if (replyMsgDoc == null) {
  debug("replyMsgDoc bad");
}
org.w3c.dom.Document replyXMLDoc = replyMsgDoc.getDocument();

debug("buyer: reply document:\n" +
  ds.docToString(replyXMLDoc, true) + "\n");

// Verify we get the correct response
String punchoutURL = replyMsgDoc.getNodeValue(
  "//cXML/Response/PunchoutSetupResponse/StartPage/URL");
if (punchoutURL.equals(expectedURL)) {
  debug("Correct response received");
  pw.println("<P>Correct response received");
}
else  {
  debug("Unexpected response received");
  pw.println("<P>Unexpected response received");
}

// Verify that the shared secret is mine
String dss = replyMsgDoc.getSenderSharedSecret();
debug("Document Shared Secret for " + myTradingPartnerName + ": " + dss);

String sss = cxmlm.getSharedSecret(myTradingPartnerName);
debug("Stored Shared Secret for " + myTradingPartnerName + ": " + sss);

if (dss.equals(sss)) {
  debug("Shared Secret match");
  pw.println("<P>Shared Secret match");
} else {
```

```
        debug("Shared Secret mismatch");
        pw.println("<P>Shared Secret mismatch");
      }
    }
    catch( Exception e )  {
      debug("MessageDeliveryException: " + e.toString());
      e.printStackTrace();
    }
  }

  /**
   * A simple routine that writes to the wlc log
   */
  private static void debug(String msg){
    if (DEBUG)
      UserLog.log("***Partner1Servlet: " + msg);
  }
}
```

Sample Supplier

Listing 3-2 Code for Sample Supplier

```
/*
 *        Copyright (c) 20001 BEA
 *        All rights reserved
*/
package examples.ibcxmlverifier;

import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

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

import com.bea.b2b.protocol.messaging.*;
import com.bea.b2b.protocol.cxml.messaging.*;
import com.bea.b2b.protocol.cxml.CXMLListener;
import com.bea.b2b.protocol.cxml.*;

import com.bea.eci.logging.*;

/**
 * This example provides a simple test that will verify message flow of cXML
 * peer-to-peer sending and receiving a cXML document.
 * The two peers (Partner1 and Partner2) are running on a single WLS.
 * Partner1 sends a PunchoutRequest to Partner2.  Partner2 generates a
 * PunchoutSetupResponse and returns it to Partner1.  Shared Secrets are verified
 * at both ends.
 */
public class Partner2Servlet extends HttpServlet {

  static final boolean DEBUG = true;

  private final static String businessProcess = "PunchoutSetup";
  private final static String businessProcessVersion = "1.1.009";
  private final static String deliveryChannel = "CXMLPartnerVerifier2";
  private final static String myTradingPartnerName = "CXMLPartnerVerifier2";
  private final static String otherTradingPartnerName = "CXMLPartnerVerifier1";
  private final static String toRole = "Buyer";
```

```
// Create the token for this application
private static CXMLToken token;

// Get the manager instance
private static CXMLManager cxmlm = CXMLManager.getInstance();

private static Properties prop = new Properties();

public void init(ServletConfig sc) {
  try {
    debug("Initializing servlet for Partner2");

    // Set the properties for finding the Collaboration Agreement
    prop.setProperty("BusinessProcess", businessProcess);
    prop.setProperty("BusinessProcessVersion", businessProcessVersion);
    prop.setProperty("DeliveryChannel", deliveryChannel);
    prop.setProperty("thisTradingPartner", myTradingPartnerName);
    prop.setProperty("otherTradingPartner", otherTradingPartnerName);
    prop.setProperty("toRole", toRole);
    prop.setProperty("Party", "duns5");

    // Register the supplier listener with the manager using properties
    token = cxmlm.register(new Partner2MessageListener(), prop);

    debug("Partner2 waiting for message...");
  } catch (Exception e) {
    debug("CXMLPartnerVerifier2 init exception: " + e);
    e.printStackTrace();
  }
}

/**
 * This routine starts the peer
 */
public void service(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException{
  debug("Starting Partner2");
}

/**
 * A simple routine that writes to the wls log
 */
private static void debug(String msg){
  if (DEBUG)
    UserLog.log("***Partner2Servlet: " + msg);
}

public class Partner2MessageListener
  implements CXMLListener
```

```
  {
    public void onMessage(CXMLMessage cmsg) {
        XPathHelper xp = new XPathHelper();

      try {
        debug("Partner2 received message");
        // QualityOfService qos = cmsg.getQoS();

        CXMLDocument reqMsgDoc = cmsg.getRequestDocument();
        if (reqMsgDoc == null){
          throw new Exception("Did not get a request payload");
        }
        Document reqXMLDoc = reqMsgDoc.getDocument();
        if (reqXMLDoc == null){
          throw new Exception("Did not get a request document");
        }
        String from = reqMsgDoc.getNodeValue(
          "//cXML/Header/From/Credential/Identity" );
        if (from == null) {
      from = "nobody";
        }
        debug("Received request from " + from );

        DocSerializer ds = new DocSerializer();

        debug("supplier: request document:\n" +
          ds.docToString(reqXMLDoc, true) + "\n");

        debug("Building reply document");

        DOMImplementationImpl domi = new DOMImplementationImpl();
        DocumentType dType =
      domi.createDocumentType("response", null, "cXML.dtd");

        org.w3c.dom.Document replyXMLDoc = new DocumentImpl(dType);
        CxmlElementFactory cf = new CxmlElementFactory( replyXMLDoc );

       String otherSharedSecret = cxmlm.getSharedSecret(otherTradingPartnerName);
        debug("Stored Shared Secret for " + otherTradingPartnerName  + ": " +
otherSharedSecret);

        replyXMLDoc.appendChild(
          cf.createCxmlElement(
        // payload
          "1233444-200@ariba.acme.com",

        // header
      cf.createHeaderElement(
        // from
```

```
cf.createFromElement(
cf.createCredentialElement(
  "DUNS",
  myTradingPartnerName,
  null)),
  // to
cf.createToElement(
cf.createCredentialElement(
  "DUNS",
  otherTradingPartnerName,
  null)),
// sender
cf.createSenderElement(
cf.createCredentialElement(
  "AribaNetworkUserId",
  "admin@acme.com",
  otherSharedSecret),
  "Ariba ORMS 5.1P4")),
// body
cf.createResponseElement(
  "200",
  "ok",
cf.createPunchoutSetupResponseElement(
  "http://xyz/abc?from=" + from ))));

  CXMLDocument replyMsgDoc = new CXMLDocument();
  replyMsgDoc.setDocument(replyXMLDoc);

  cmsg.setReplyDocument(replyMsgDoc);

  debug("supplier: reply document:\n" +
    ds.docToString(replyXMLDoc, true) + "\n");

  // Verify that the shared secret is mine
  String dss = reqMsgDoc.getSenderSharedSecret();
 debug("Document Shared Secret for " + myTradingPartnerName + ": " + dss);

  String sss = cxmlm.getSharedSecret(myTradingPartnerName);
  debug("Stored Shared Secret for " + myTradingPartnerName  + ": " + sss);

  if (dss.equals(sss)) {
    debug("Shared Secret match");
  } else {
    debug("Shared Secret mismatch");
  }

  // Set the CA with the properties
  cmsg.setCollaborationAgreement(prop);
  cmsg.reply();
```

```
        debug("Partner2 sent reply");
      } catch(Exception e) {
        debug("Exception errors" + e);
        e.printStackTrace();
      }
    }

    public void onTerminate(Message msg) throws Exception {
      debug(" received terminate notification for " + msg.getConversationId());

      // Deregister with the manager
      cxmlm.deregister(prop);
    }
  }
}
```

4 Using Workflows with cXML

Note: The cXML business protocol is deprecated as of this release of WebLogic Integration. For information about the features that are replacing it, see the *BEA WebLogic Integration Release Notes*.

WebLogic Integration allows you to use business process management (BPM) workflows to exchange normal business messages. While there is no cXML plug-in for WebLogic Integration, you can nonetheless integrate cXML business documents through the use of business operations.

The following sections describe how to exchange cXML business messages in WebLogic Integration, using workflows and the cXML API-driven interface:

- Including cXML in Workflows
- Designing Workflows for Exchanging Business Messages
- Working with Business Messages

For more information about developing workflows using WebLogic Integration, see *Creating Workflows for B2B Integration*.

Including cXML in Workflows

Workflows intended to use cXML must make use of an externally-created business operation class to encapsulate the cXML API used by WebLogic Integration.

The result of this development process is a workflow that, when executed, allows the methods defined in the wrapper class to be invoked. These methods perform the defined cXML business operation.

Workflow Integration Tasks

Using cXML with BPM workflows requires a specific combination of administrative, design, and programming tasks to be performed.

Programming Task

Externally created business operation classes use the cXML API to perform a specific business operation. For example, you might create a class that implements the `PunchoutSetupRequest` functionality for a workflow. For more information, see the *cXML User's Guide* at:

`http://www.cxml.org`

If you plan to pass parameters using the workflow, you must first create a class that can accept such parameters. Next, you must pass parameters into the class using workflow variables. The parameters can then be used to set up your cXML output.

To configure the class, its methods, and any parameters that you have defined, open the WebLogic Integration Studio and select *Business Operations* from the *Configure* menu. For more information, see *Using the WebLogic Integration Studio*.

Within theWebLogic Integration Studio, you can then invoke the business operation used to invoke the cXML process operation as a workflow action. When you add an action, select *Perform Business Operations* from the *Integration Actions* folder of the *Add Actions* dialog box. This option allows you to map workflow variables to the method parameters used by the cXML wrapper class. For more information, see *Using the WebLogic Integration Studio*.

Administrative Tasks

Before you start using cXML with workflows, you must complete the following administrative tasks. These tasks are in addition to those that you normally perform while using the WebLogic Integration Studio to generate workflows for use with WebLogic Integration:

- Using the WebLogic Integration B2B Console, create and configure the entities that will be involved in your cXML transactions in the WebLogic Integration repository, including trading partners, collaboration agreements, and so on. For more information, see *Administering B2B Integration*.
- After you create a Business Operation class, create a Business Operation within the WebLogic Integration Studio to make use of the Business Operation class. For more information about creating a Business Operation class, see Chapter 3, "Using the cXML API."

Design Task

In addition to the design work required to create a workflow for use with WebLogic Integration, you must do some extra design work if you want to use cXML in your workflow. Specifically, you must design your workflow to use a Business Operation to execute all cXML functionality. For each cXML function you need to execute, you must create a separate Business Operation.

Designing Workflows for Exchanging Business Messages

To use workflows to exchange business messages in WebLogic Integration, design workflow template definitions by using the WebLogic Integration Studio. For information about creating workflows, see *Using the WebLogic Integration Studio* and *Creating Workflows for B2B Integration*.

As discussed previously, use of cXML in workflows requires the creation of business operation classes to implement the cXML API. In the previous section, we discuss the creation of these business operation classes. In this section, we discuss the use of business operation classes to manipulate cXML messages within the BPM component of WebLogic Integration.

Working with Business Messages

The WebLogic Integration Studio allows you to enable trading partners to exchange business messages. cXML is one method by which this task may be performed.

The following sections describe how to work with cXML business messages exchanged using workflows:

- About cXML Business Messages
- Prerequisite Tasks for Exchanging Business Messages

About cXML Business Messages

A cXML business message is the basic unit of communication exchanged by trading partners in a conversation. A cXML business message is a multipart MIME message that consists of the following:

- 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 the nonXML payload part of the business message. Attachments are optional entities within the cXML1.2 standard, and are not available with cXML 1.1 implementations.

As with other forms of business messages, you can access the contents programmatically, as described in *Creating Workflows for B2B Integration*. Unlike with XOCP and RosettaNet business messages, however, the WebLogic Integration implementation of cXML does not allow you to use any other method to access the contents of a business message when using cXML.

Prerequisite Tasks for Exchanging Business Messages

Before you can send and receive business messages, you must define the following actions in the workflow template, using the WebLogic Integration Studio:

- To define the sending of a business message, define a Manipulate Business Message action to construct the business message and a Send Business Message action to send the message.
- To define the reception of a business message, define a Manipulate Business Message action to process an incoming business message.

For more information, see *Creating Workflows for B2B Integration*.

Index