XML PORTAL CONNECTOR

DEVELOPER GUIDE AND API REFERENCE

Version 4.0

Corporate Headquarters
4440 Rosewood Drive
Pleasanton, CA 94588-3050

www.commerceone.com

XML Portal Connector Developer Guide and API Reference, Version 4.0

Copyright © 2000, 2001 Commerce One, Inc. All rights reserved.

February 2001

COMMERCE ONE, Inc. Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted.

This documentation and the software described constitute proprietary and confidential information protected by copyright laws, trade secret, and other laws. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Commerce One, Inc.

Copyright © 2001. Commerce One, Many Markets. One Source. Global Trading Web, Commerce One.net, MarketSite, XML Common Business Library, XML Development Kit, eLink, Net Market Maker, RoundTrip, and SupplyOrder are either trademarks or registered trademarks of Commerce One, Inc. Enterprise Buyer and MarketSet are trademarks of Commerce One, Inc and SAPMarkets. All other company, product, and brand names are trademarks of their respective owners.

Contents

Preface

Purpose of this Guide

The XML Portal Connector (XPC) provides an API that can be used by trading partners to automate the processing of xCBL documents. The *XPC Developer Guide and API Reference* provides information necessary to use this API. It includes:

- An overview of XPC architecture
- An explanation of customizations used to integrate business document transactions with the trading partner's back office system
- Information about how to use the XPC Manager to configure the component execution flow in XPC Services
- A description of the Trading Partner Preconfiguration
- Descriptions of the XPC Component Library
- Instructions for using the Invoker to test an integrations
- Descriptions of XPC Java classes, interfaces, and methods
- Information about the Transmitter API
- Information about the Credential API

Documentation Conventions

All pathnames in this book are expressed relative to the root directory of the XPC installation.

Audience

This book is for individuals responsible for processing MarketSite business transactions for trading partner back-end integration or application development. It assumes that you have the following skills:

- Ability to read and write Java 2 code
- Familiarity with the trading partner's target back office system
- Ability to read SOX schemas

How to Use this Guide

The information in this book is logically organized for your development and testing of XPC customizations. The following table describes each chapter in this guide:

Chapter	Description
1	Provides an overview of XPC architecture and an introduction to some important concepts.
2	Describes how to use XPC Manager to add, remove, enable, and configure XPC Services and components.
3	Provides guidelines for developing custom integrations.
4	Describes the services in the Trading Partner Preconfiguration.
5	Provides descriptions, inputs, outputs, and configurations for the XPC Component Library.
6	Provides tips on using the Invoker to test your integration.
7	Is a reference for selected XPC APIs.

In addition, Appendixes A, B, and C, respectively, provide information on the Transmitter API, the Security Credential, and the components used to convert xCBL documents to EDI and vice versa.

Related Information

You may find this additional documentation helpful:

Documentation	Description
XPC Installation and Administration Guide	This guide, which is available on the XPC 4.0 product CD, describes how to install and configure XPC and how to perform common administrative tasks.
Javadoc for the Commerce One CCS, XDK, and XPC packages	These API descriptions are available in the *\docs\api\ccs*, *\docs\api\xdk* and *doc\api\xpc* directories of the XPC installation.
XDK Pro Developer Guide	This guide, which is available on the XPC 4.0 product CD, describes how to use the Commerce One XML Parser (CXP) and the SOX to Java compiler, and how to interface with CXP via SAX.
xCBL Online Reference Guide	This guide, which is available at www.xcbl.org, describes the structure of Commerce One's xCBL3.0 documents.
HotFS Installation and Configuration Guide	This guide describes how to install and configure HotFS including XCC configuration, sample scenarios, and client and service setup and execution.

If You Need Help

Commerce One Technical Support is available to all Commerce One customers who purchase XPC software directly from Commerce One. If you cannot resolve a problem by using the Commerce One manuals or on-line help, ask the designated person to contact Technical Support via email (csc@commerceone.com).

If you purchased or obtained XPC software from your Global MarketPlace, Systems Integration Partner, or any source other than Commerce One, please refer to that source for technical support. Please do not contact Commerce One directly.

1 Overview of XPC

In This Chapter

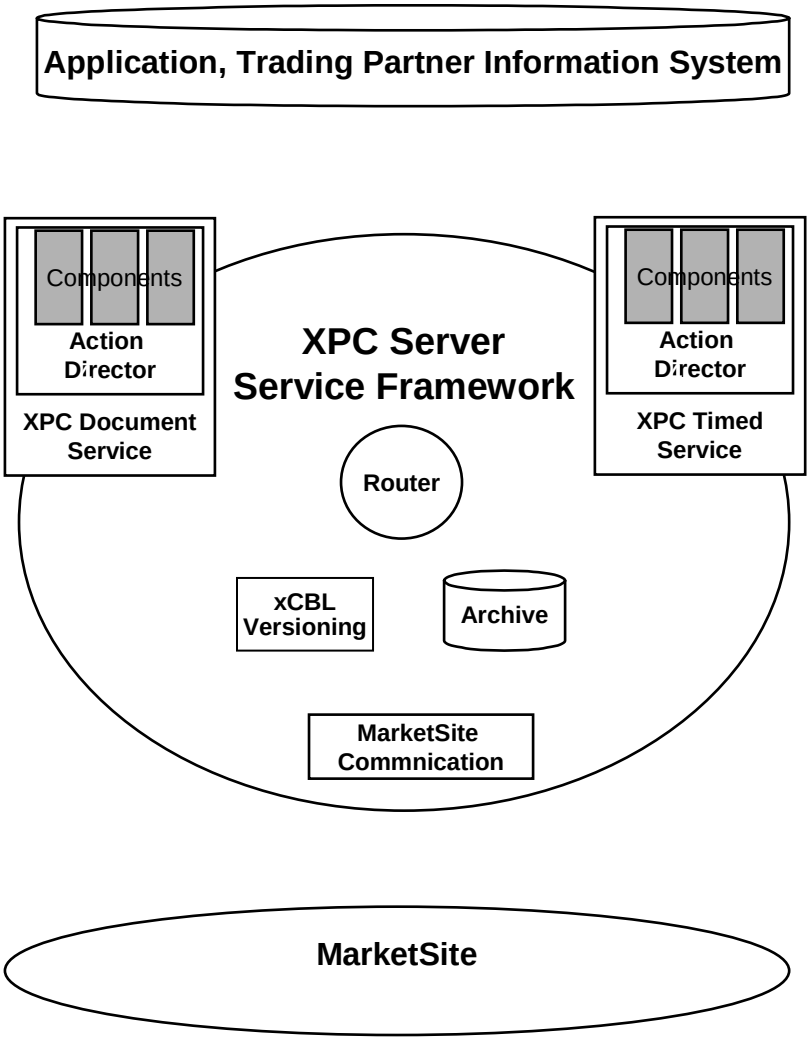
This chapter provides an overview of the Commerce One XML Portal Connector (hereafter referred to as XPC). It includes the following information:

XPC Concepts and Terminology

XPC facilitates integration between MarketSite and the back office systems of trading partners. XPC provides a set of components plus a Framework within which the components are designed to run. Trading partners and third party integrators can use the components out of the box, extend them, or replace them with new components tailored to their specific needs.

XPC components are Java methods designed to facilitate the flow of information through xCBL documents. Components perform such functions as reading xCBL request documents, archiving documents and their envelopes, preparing default response documents, modifying the default response documents with data from the back office system, and sending MessageAcknowledgements.

The following diagram illustrates the major pieces that play a role in transmitting information between the trading partner's back office system and MarketSite.



The following paragraphs describe various parts of this diagram and introduce important XPC concepts and terminology:

- Service Framework

  XPC contains a framework to deploy special Java objects known as services. XPC handles two types of services: Document Services and Timed Services.

Previous Commerce One XML Commerce Connector based technology exposed an XCC Service programming API. With XPC 4.0, this API is deprecated in favor of XPC component APIs that run in Action Director enabled services.

- XPC Document Services

  These services are activated when a document of the specified type is received. Document Services generally perform actions to process the incoming document.

- XPC Timed Services

  These services wake up at user-defined intervals to handle outgoing documents of the specified document_type.

- XPC Components

  A component is a Java class responsible for a particular type of functionality. All components extend the XPCAbstractComponent class. XPC includes a Component Library, a set of predefined components designed to assist with xCBL choreography, interface with the file system, and perform other functions.

- Action Director

  An **action** is defined as the execution of a component's public method. Each XPC Service has an Action Director that determines which components it executes and in what order. **Standard components** provide only a single method, the process method. Components that provide multiple methods are known as **extended components**.

- XPC Manager

  XPC Manager is a user interface that runs in your browser. You can use it to add, enable, disable, remove and configure services and to configure components.

- Communication with MarketSite

  XPC uses either SonicMQ or HTTPS transport protocols to communicate with MarketSite.

- Archive

  XPC automatically archives both inbound and outbound peer-to-peer and one- way documents.

- xCBL Versioning

  XPC is configured to define the xCBL version supported by the Document Services. If necessary, XPC transforms an envelope to the expected xCBL version before sending it to the router to ensure that the Document Service receives documents in the specified xCBL version.

- Router

  The Router sends each xCBL document to the correct service -- either a document service for incoming message or the transmitter service for outgoing messages.

- Stand-alone transmitter

  The Transmitter Java class can be wrapped in an external application to communicate with MarketSite outside the server.

Help In Customizing

Building an integration consists of:

- Creating a logical list of actions to execute for each incoming document type using standard and custom XPC components
- Creating custom components containing the specific business logic required by the back office system

XPC comes with a “head-start” to customization. The Trading Partner Preconfiguration sets up an out-of-the-box XPC integration that serves as a useful starting point for developing an integration. The sample components, such as the XPCDefaultPriceCheckResponseBuilder component, demonstrate how to build these custom components.

XPC also includes the XPCDocHandle Java class, which allows you to access specific portions of an xCBL document using XML query strings based on the XPath standard.

xCBL Document Exchange Choreography

XPC manages the exchange of xCBL documents from the sending party to the receiving party. Each type of document exchange is governed by a set of rules, sometimes referred to as a contract. The contract determines what happens when a particular type of document is received--how the receiving party acknowledges receipt, what types of documents are sent in reply, and when they are sent. XPC handles multiple versions of xCBL-- xCBL 2.0, 2.2, and 3.0--each with its own document types and choreographies.

XPC handles the following types of document exchanges:

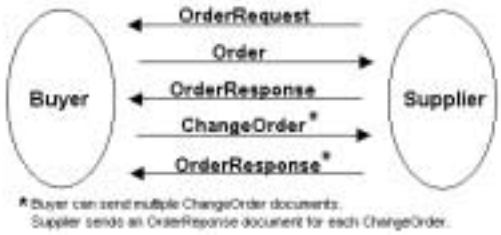
- **One-way** exchanges for request documents that do not require a response. When a one-way document is received, the recipient sends a MessageAcknowledgement

or, if the request cannot be understood or cannot be processed, an Error document. One-way exchanges include the AdvanceShipmentNotice, Invoice, PlanningSchedule, ProductCatalog, and RemittanceAdvice xCBL documents.

- **Synchronous** exchanges for request documents that require an immediate response by XPC. These exchanges hold the XPC connection open until the response document has been sent. Synchronous exchanges include the AvailabilityCheckRequest, PriceCheckRequest, and OrderStatus xCBL documents.
- **Peer-to-peer asynchronous** exchanges for request documents that are request documents that are responded to in a separate session. This choreography introduces the concept of correlation across sessions. The responder may be generating the response and receiving the request in different sessions. The sender has to remember the state it was in when it made the request. Order Management and Time Series are examples of transactions that use asynchronous exchanges.

Some document exchanges are simple, involving just a single request document, a single response document, and a MessageAcknowledgement for each. The diagram below illustrates such an exchange. The buyer sends a RequestForQuotation and the supplier responds with a Quote.

Other exchanges are more complex, involving the exchange of multiple documents. The following OrderManagement diagram illustrates a complex exchange. The supplier sends an Order Request, the buyer responds with an Order, the supplier sends a ChangeOrder, and so forth.



2 XPC Manager

In This Chapter

XPC Manager is the XPC configuration environment It is a browser based user interface that runs on the XPC's built-in web server. You use this tool to define the runtime configuration for processing business data and to define the sequences of actions, or methods on a Java objects, that will be invoked under specified conditions.

This chapter provides some conceptual material about XPC Services, Action Directors, and components. It also explains how to use XPC Manager to enable or disable XPC services, to configure their Action Directors and components, to create additional services to be run within XPC, and to delete services from XPC.

Overview of Framework and Terminology

At runtime, the XPC server has any number of services running. The core functions of XPC--communication, routing, archiving, version transformation--are all performed by services.There are two types of services available to the XPC developer to process document exchanges:

- **Document services** are invoked when an envelope is received that adheres to the subscription of the service. Typically, a **subscription** defines a document type, sender id, and receiver id. A document service can be defined as **synchronous** or **asynchronous**. Synchronous document services reply to invocation with a business document. Asynchronous document services assume the reply is initiated in a separate session, generally via a timed service.
- **Timed services** are invoked when a timer expires. The timer can be set to fire at any time interval and is specified in milliseconds. A timed service is responsible for processing an outgoing envelope. Timed services can also perform functions for out-of-band regular processes, server maintenance or archive processing for example.

These services run an **Action Director** which executes Action Lists. An **Action List** is a sequence of Java methods to be invoked sequentially. Java objects that provide such methods are known as XPC Components.

A Document Service may contain multiple subscriptions, each defining its own action list. A Timed Service defines an action list that is executed each time its timer expires. Both types of services have an Exception action list that specifrying the actions to invoke when an exception is thrown.

XPC Components are subclasses of *XPCAbstractComponent*, which serves as an API to define inputs, outputs, and configurations. XPC comes with a library of standard components and a developer can create custom components and deploy them in the framework.

XPC Manager allows the configuration of this framework.

Loading XPC Manager

Before running the XPC Manager, you must load XPC's self-signed certificate into your browser. This certificate is automatically created when you install XPC.

To load the self-signed certificate on Internet Explorer 5:

1. Select Tools->Internet Options->Content Tab->Certificates->Import
2. Select <install:root>/bin/client.p12

Load XPC Manager as follows:

1. Start the XPC server. For information about starting XPC, see the *XPC Installation and Administration Guide*.
2. Browse to https://localhost:4433/servlet/XPCManager (case matters!)

Note ..... If you use Configure XPC to change the port your XPC server is listening on from the default of 4433, you will need to modify this URL appropriately.If you cannot load XPC Manager, restart XPC from the command line and check the logs. For more information, see the **Troubleshooting** chapter of the *XPC Installation and Administration Guide*.

XPC comes with a set of predefined services that support xCBL 3.0. These can be loaded from the XPC Configuration user interface. Please see the **Trading Partner Preconfiguration** chapter for more information.

Note ..... Changes you make with XPC Manager will not take effect until the XPC Server has been stopped and restarted.

Enabling and Disabling Services

Enabling a service adds it to the list of services that start when the XPC Server is started. The **Services** screen of XPC Manager contains a list of all available XPC Document Services and Timed Services. Services are listed together in alphabetical order.

The **Enabled** column to the right of the service names contains a check box for each service. Services that have been enabled are checked; those not enabled are not checked. You can change the enablement status of any number of XPC services at the same time by toggling the check boxes.

Note ..... For information about which services to enable, see the **Trading Partner Preconfiguration** chapter.

To enable or disable services:

1. Click the **Enabled** column to the right of the service to change its status from disabled to enabled or vice versa. If you make a mistake, click the **Enabled** column again to undo the change.
2. When you are satisfied with the changes, click the **Save** button at the top or bottom of the screen. XPC Manager asks you to confirm that you want to save the current configuration.
3. Click **OK** to save the changes or **Cancel** to cancel without saving.

The changes will take effect the next time the XPC Server is started.

Configuring a Service

You must configure each service separately. To select a service for configuration:

1. Select the service from the **Services** page by setting the **Select** radio button to the right of its name and clicking the **Edit** button.
2. Click the **Service Level Configuration** link to configure the service.

Service Level Configuration

1. The **Service Name** cannot be the same as any other service name. By convention, document service names end with “Inbound” and timed service names end with “Outbound.”
2. **Service Type** is either **Document** or **Timed**.

3. Document services can be either **Synchronous** or **Asynchronous**. Synchronous document services reply to an invocation with a business document. In xCBL, the price check, availability check, and order status transactions require synchronous document services.
4. Specify the service's arguments in the **Add New Key**s area, one argument at a time. Type the argument's key in the **New Key** text box and its value in the **New Value** text box. Then click the **Add** button to add the argument to the **Service Configuration** area.

   If you add an argument by mistake, select the argument by clicking the button to its left, then click the **Remove** button. The argument is removed from the list.

   For Timed Services, you can configure the the following arguments:

   - **timeout** specifies, in milliseconds, the frequency with which the timer expires.
   - **inclusive** A **true** setting means that the time taken executing the service method called on each timeout is included when calculating when each timeout should occur; **false** means that the time taken executing the service method is not taken into consideration.
   - **messagestore**, which can be set to either **on** or **off**. This should be set to **on** for asynchronous documents, causing transmitted envelopes to be archived; when set to **off** , transmitted envelopes are not archived.

   For Document Services, you can configure the following arguments:

   - **maximumThreads** The default setting, 1, results in single-threaded services. Integrations that require concurrent processing should set maximumThreads to a high enough value to handle the volume of incoming documents queued to the service.
   - **initialThreads** Setting this to a non-zero value avoids the overhead of creating threads as documents are queued.
5. When the argument list is complete, click **Save** to save the changes or **Cancel** to cancel without saving.

The configuration will take effect the next time the XPC Server is started.

Configuring a Service's Action Director

You must configure each service separately. To select a service for configuration:

1. Select the service from the **Services** page by checking the **Select** button to the right of its name.

2. Click the **Edit** button to display the **Action Director Configuration Screen** for the service. The **Current Subscription List** lists the documents to which the service subscribes.

Adding a Subscription

Subscriptions determine which envelopes the service will process. Only Document services may create subscriptions. Each subscription within a service must be unique.

To subscribe to an additional document type:

1. Type the subscription in the **Create New Subscription** text box. If you make a mistake, click the **Clear** button to clear the text box.

   A subscription consists of an Sender MPID, Receiver MPID, Document Type separated by periods. The sender and receiver can be ommitted, or the plus sign (+) wildcard specified. For example, use the following subscription to process PaymentRequest documents from any sender and to any receiver:

   +.+.PaymentRequest

2. When you are satisfied with the subscription, click **Add** to add it to the **Current Subscription List.**
3. Click **Save** to save the changes or **Reset** to clear the changes from the screen.

The new subscription will take effect the next time the XPC Server is started.

Removing a Subscription

To remove a subscription:

1. Select the subscription from the **Current Subscription List**
2. Click **Remove**.
3. Click **Save** to save the changes or **Reset** to clear the changes from the screen.

The subscription will be removed the next time the XPC Server is started.

Editing a Subscription's Action List

Each entry in the list is a call to a method on an XPC component. To edit the Action List for a subscription:

1. Select the Subscription Name or Exception Action List from the **Current Subscription List**.

2. Click the **View** button to display the Action List in the bottom half of the screen. Actions are listed in the order they are executed. The list displays the component name, method name, inputs, and outputs for each action.

3. To remove an action, select the action from the list and click the **Remove** button.

Note ..... Each Timed Service must have at least one action. If a Timed Service contains only one action, you will not be allowed to remove it.

4. You can add an action either at the end of the list, before another action, or after another action.

   a) To add a new action at the end of the list, click the **Add** button.

   b) To add a new action before or after an existing action, select the action from the list and click the **Insert Before** or **Insert After** button.

   New actions are temporarily assigned the name of new_ followed by a sequence number, the default method, process, and the default creation mode, once. Use the **Edit** button to specify the actual name of the action's class and method, its component code, creation mode and any arguments.

Note ..... The **Save** buttons will be disabled until the new component has been defined with code which is in the server's class path at the start of the server and a method has been specified which exists in the specified class.

5. To edit an action, select the action from the list and click the **Edit button**. You can edit the following fields from the **Edit Component** pages:

   a) Component Name must be a Java class that XPC recognizes and must be unique for each component.

   b) Component Method must specify the name of a method in the specified Java class.

   c) Component Code is the fully qualified package name of the Java class and must be a subclass of *XPCAbstractComponent*.

Note ..... The component code specified must be in the server's classpath at the time of specification or the XPCManager will not allow the save.

   d) Creation is either **Once** or **Invocation**. A creation mode of **Once** indicates that a single instance of the component's class is shared by all threads within the service for a single action list. If the creation mode is **Invocation**, a new instance of the component's class is instantiated each time the Action Director is executed. For Document services, this causes instantiation each time a document is received. For Timed services, it causes instantiation whenever the timer expires. Typically, sharing a single instance of a component's class places

higher requirements for thread-safety than does the *invocation* creation mode. Standard XPC components are thread-safe, regardless of whether they are created once or on invocation.

6. Specify the component's arguments in the **Add New Keys** area one argument at a time.

Note ..... Please reference the **XPC Component Library** chapter for a list of available configurations for standard XPC components.

Type the argument's key in the **New Key** text box and its value in the **New Value** text box. Then click the **Add** button to add the argument to the **Component Configuration** area. If you add an argument by mistake, select the argument by clicking the button to its left, then click the **Remove** button to remove the argument from the list.

Action List Buttons

-Under Input and Output are a list of items and data types. These are the items that will be included in the DataManager object that is passed to the component upon action invocation.The input and output items only show class name and the name assigned to the output/input by the class. To display a tooltip with the fully qualified class name of the inputs/outputs, hold the cursor over the input/output names.

Components self-describe their inputs and outputs through the getInputList and getOutputList methods of XPCAbstractComponent.

Through the drop-down, a component's input can be assigned to a previous component's output.

This drop down list has available the following potential entries:

- -Last. The last previous component's output that is the same – or superclass – of the input datatype.
- requestEnvelope. The invoking envelope (only available for Document services)
- -requestDocument. The primary document from the invoking envelope (only available for Document services)
- -*component.method.output*. All previous components' output that is the same – or superclass – of the input datatype.
- **Use Generic**. Assign this subscription to the generic action list for the service. This allows multiple subscriptions to share the same action list. To display the generic action list for editing, push **View Generic**.

Timed Service Execution

Special considerations apply to the execution of an action list within a timed service. The first action within a timed service's action list is known as an event source. An event source typically retrieves an event from some external entity, and provides output arguments that represent the event which can then be used by subsequent actions in the action list. An example of an event source would be the readEnvelope action on the FileStore component. It retrieves an envelope from the file system, then provides the envelope as an output argument.

Once the event source action has completed, execution of the action list then proceeds sequentially as would be the case for an action list executing within a document service. However, when all actions in the list have been executed, the Action Director does not exit as would be the case with a document service. Instead it reinvokes the initial event source action. This allows the event source to pick up another event and have it processed by the rest of action list.

This looping execution of the action list continues until the event source action indicates to the Action Director that it has no more events. This indication is achieved through use of a non-zero result value for the XPC action. For more information about how non-zero results are handled, see **Error Handling** on page 11 of this chapter. The Action Director then not only halts the current sequential processing of the action list, but also exits its looping execution of the action list as a whole, ending processing of this timeout.

Sharing Components Across Action Lists

In a service, two actions may share the same java component, and call the same or different methods.

If two actions define a component of the same Component Name and Component Code and Creation is set as "once", they will share the same instance of the component.

If two actions share one component, and an action is edited to use another component, Save must also be selected on the Action List screen to save proper state.

Exception List Execution

The exception list is invoked when an exception is thrown by a component in an action list. This exception action list assumes that the action director will have two data elements: the name of the component that threw the exception and the Exception object.

Adding Services

To add a service:

1. From the **Services** page, select **Add New Service** button.
2. Type the name of the service in the **Service Name** text box. This cannot be the same as any existing service name. By convention, document service names end with “Inbound” and timed service names with “Outbound.”
3. Choose either **Document** or **Timed** from the **Service Type** drop down list.
4. For document services, choose either **Synchronous** or **Asynchronous** from the **Transaction Type** drop down list.**Transaction Type**.
5. Specify the service’s arguments in the **Add New Key**s area, one argument at a time. Type the argument’s key in the **New Key** text box and its value in the **New Value** text box. Then click the **Add** button to add the argument to the **Service Configuration** area.

   If you add an argument by mistake, select the argument by clicking the button to its left, then click the **Remove** button. The argument is removed from the list.
6. When the argument list is complete, click **Save** to save the changes or **Cancel** to cancel without saving.

The next time the server is started, the new service will appear in the list of services on the **Services** screen.

Before using the service, you must enable it For information about enabling services, see **Enabling and Disabling Services** on page 3.

You may also want to configure the service in order to change its subscription list, arguments, or action list. For information about configuring services, see **Configuring a Service’s Action Director** on page 4.

Note ..... Each Timed Service must have at least one action. If a Timed Service contains only one action, you will not be allowed to remove it. If a timed service is created and the action list is not filled in with at least one action, it will fail to initialize.

Removing Services

You may only select one service at a time for removal. To remove a service:

1. Select the service by clicking its radio button in the far right column.
2. Click the **Save** button at the top or bottom of the screen. XPC Manager asks you to confirm that you want to save the current configuration.
3. Click **Save** to save the changes or **Cancel** to cancel without saving.

The services will be removed after you stop and restart the XPC Server.

Action Director Runtime Processing

The Action Director manages the inputs and outputs of each action, keeping a chronological list of all outputs that have been generated by executed actions. For XPC document services, the list is initially populated with the received document and the envelope that contains it. For XPC timed services, the list is initially empty.

When a particular action is to be executed, the Action Director looks for the most recently generated outputs that match the types of input arguments defined for the component's method and passes them to the method. When the method's invocation is complete, the Action Director takes the method's outputs and adds them to the start of the list.

The Action Director maintains a linked list of all previously produced component outputs. It searches this list for output whose Class is assignable to the Class of the input argument. For a document service, this list is initialized with an Envelope object (requestEnvelope) and document object (requestDocument).

By default, the Action Director selects as input arguments the **most recently produced** output of the appropriate class. Using the XPC Manager, an input can link to a particular output argument of a previously executed component. The explicit linkage requires that the Class of the output argument be assignable to the Class of the input argument to which it is being linked.

Action Director Places Inputs into Data Manager

If the Action Director finds the required input in the linked list, it places the associated value into the Data Manager and “get-enables” the argument. Only arguments that have been get-enabled can subsequently be read by the component invoking the Data Manager’s get() method.

If the get() method tries to read an argument that has not been get-enabled, it throws an IllegalArgumentException, which can be caught by the component. If the Action Director cannot find a required input, it generates a NullPointerException, causing the execution of the Action Director's Exception action list.

Action Director Enables Component Outputs

The Action Director "set-enables" each argument in the component's output list. Only arguments that have been set-enabled can be written by the component invoking the Data Manager's set() method. If the set() method tries to write an argument that has not been set-enabled, it throws an IllegalArgumentException, which can be caught by the component.

Action Director Adds Component Outputs to the Linked List

Once the component has successfully executed, the Action Director adds its component outputs to the head of the linked list. These outputs will be the first ones to be searched for the next component's required inputs.

Note ..... If the component throws an exception, none of its outputs will be added to the linked list, regardless of whether they have been written by calls to the Data Manager's set() method.

Error Handling

The return value of all invocable methods of an XPC component is an object of type XPCResult. This class contains an integer value that allows the method to indicate to the Action Director the success or failure of the method's execution. A value of zero indicates successful execution; a non-zero value indicates failure, causing the Action Director to terminate processing of its action list.

3 Building Custom Integrations

In This Chapter

This chapter provides guidelines for general information about customizing XPC. It includes:

Overview of XPC Integration

XPC runs at a trading partner site exchanging xCBL messages with other trading partners over MarketSite. XPC is customized to integrate these xCBL messages with the trading partner's back office system.

This MarketSite message exchange must follow an xCBL choreography. An integration consists of configuring the XPC with appropriate document and timed services to process the incoming and outgoing messages corresponding to the trading partner's role in the choreography.

For example, in the Quote choreography, the buyer has a timed service that generates a RequestForQuotation message and a document service that receives the Quote reply. The seller has a document service that receives the RequestForQuotation message and a timed service that replies with a Quote message.

The development of an integration consists of defining the logic required for each of these services. Following are some examples of this logic:

- simple exchange of xCBL documents with the back office system through the file system
- transformation between xCBL and another format and file system exchange of non-xCBL documents
- direct back office API calls

The logic is defined using the XPC Manager to string together lists of actions, and configuring the components that perform the actions. For more information, see the **XPC Manager** chapter.

XPC comes with a set of standard components as defined in the XPC Component Library. For more information, see the **XPC Component Library** chapter. You may need to create new components to handle your custom business logic.

"Helper" features

To assist in representation of business logic, XPC provides a XPath-based mechanism to represent xCBL business data -- either programmatically or in component configurations. This is available via a helper class, XPCDocHandle, that provides XPath querying on xCBL documents. Also included is a browser based xCBL XPath querying tool (https://localhost:4433/servlet/XPathTester).

XPC provides the Trading Partner Preconfiguration (Chapter 4) which defines a complete set of service definitions for all inbound and outbound xCBL 3.0 documents. These service definitions implement a configurable file system document exchange scheme and will serve as a good starting point for an xCBL 3.0 based integration project.

XPC provides default response builder components for all xCBL 3.0 request/response pairs. These provide integration developers a starting point response. The integration developer then only has to update the provided response as opposed to building it from scratch.

File System Document Exchange

This method of integration assumes an agreed upon set of file system logic rules between the XPC and an external process (the ERP system). These logic rules include:

- Significance of directories. Example: incoming Orders in directory *inbound**order*.
- Significance of file naming schemes. Example: PaymentRequest documents begin with “pr” and have a “.cbl” extension.

These rules are defined by configuring the FileStore component using XPC Manager. Using FileStore, each element of an envelope -- document, attachments, header -- can have its own directory/file-naming definition. Please reference FileStore in Chapter 5 for a complete description of the functionality of this component.

An file system integration is inherently asynchronous. For both XPC and the external process, a document is received via a file system polling mechanism. This polling mechanism in the XPC can be implemented via a timed service initiated with the readDocument or readStream action of FileStore.

In the Trading Partner Preconfiguration, all asynchronous document exchanges use this scheme.

This integration scheme can be augmented with custom components. For example, if the documents passed to the file system are not xCBL, a custom component could perform a transformation before writing and after reading.

Multiple Transactions

Many non-xCBL formats may include multiple transactions per document, which are not supported by xCBL. If single transaction documents cannot be produced by the ERP system, a custom component can be developed to run as the initiating action in a timed service (see Chapter 2, Timed Service Execution).

This custom component would follow the requirements of the initiating action:

- Assume that it will be invoked repeatedly
- Stop repeat by generating an exception
- Issue only one xCBL transaction

The last point is because all envelopes exchanged with MarketSite must contain a single xCBL document.

Since a component can be configured to be invoked “once” (Chapter 2, Service Level Configuration), its state in memory can be retained between this looping invocation. The component can pick up the entire file and store in memory and remove and issue the first transaction. For subsequent invocations, it would remove the next transaction from memory. Throw exception when there’s nothing in memory or additional files.

API based integration

This method of integration uses direct calls between XPC and the back-office system.

Inbound Messages

For messages inbound from MarketSite, an XPC document service includes a custom component performing the API calls: These calls might load, extract, or select data from the back office system.

The Trading Partner Preconfiguration demonstrates making an API call using a sample custom component with the PriceCheck, AvailabilityCheck, and OrderStatus document services. These are the only xCBL 3.0 transactions that require synchronous processing.

The same scheme can be used when responding in a peer to peer (asynchronous) exchange. If you develop a component to extract business response data in real-time, it can be deployed in a synchronous document service (see Chapter 2, Service Level Configuration).

Outbound Messages

For outbound messages that are in response to a request, the custom component could be in the synchronous document service and the outbound message initiated by the Responder component.

For outbound messages that are initiating requests, the ERP system may make direct calls to MarketSite using the XPC Transmitter API, as detailed in Appendix A.

Or, a custom component can be developed to run as the initiating action in a timed service (see Chapter 2, Timed Service Execution). This custom component could perform a back office API call following the requirements of the initiating action:

- Assume that will be invoked repeatedly
- Stop repeat by generating an exception
- Only output a single xCBL transaction

XPath-based Document Programming Model

XPCDocHandle is a helper class that provides an easy way to interact with the contents of an xCBL document. A developer constructs an XPCDocHandle with an xCBL Java bean. The XPCDocHandle get() and set() methods use an XPath-like string variable to represent the xCBL nodes (elements or attributes) to be manipulated.

- The get() method takes an XPath String and returns a String or array of Strings. If the XPath String defines an element within a looping section, it returns an array of Strings.
- set() takes both the XPath String and a String or array of Strings to set in the bean.

XPath is a W3C recommended language for addressing parts of an XML document. XPCDocHandle implements simple node selection capabilities and a subset of XPath predicates. For a full description of the standard, visit the W3C website: http://www.w3.org/TR/xpath.html.

XPath Tester

XPC includes a browser tool to test and build XPath query strings. These query strings can then be cut-and-pasted into component source code or configurations.

1. Start XPC (XPath Tester runs off XPC's web server)

2. Browse to https://localhost:4433/servlet/XPathTester
3. Load the document whose Xpath you want to test.

   XPC provides sample documents in the < XPC Root>/sample/xpc/instances directory. Each document type has its own subdirectory.

   If you know where the document is located, type the path in the **Current File Path** text box. Otherwise use the **Browse** button to locate it. Once you have entered the path, click **Load** to load the document in the XML Source pane in the lower left portion of the screen.
4. Type the Xpath to the node in the **Enter Xpath** text box or build the XPath by selecting pieces of the xCBL document and using your browser's **Copy** and **Paste** commands to add them to the XPath string. Be sure to use a slash to separate nodes.
5. When you have constructed the path, click **Go**. The node referenced by the Xpath is displayed in the **Results** pane in the lower right portion of the screen.

The XPath string remains on the screen, in the **Enter XPath** text box, until you clear it with the **Reset** button. You can continue to modify the string until it displays the correct node in the **Results** pane. Once you are satisfied with the XPath string, use your browser's **Copy** command to copy it, then paste it into your code.

XPCDocHandle Functional Definition

An XPath string specifies the path to the desired node, beginning at the outermost node--the document itself--and working inward towards the leaf nodes. XPath notation uses the forward slash, /, between nodes and precedes attribute names with the at symbol, @. The entire XPath string must be enclosed within a pair of double quote characters, " and ". Any double quote characters within the string must be preceded by a backslash escape character, \, or they will be treated as the string terminator.

Following is an example of an XPath string that retrieves the Quantity element from a PriceCheckRequest document:

```
String XPATH_QUANTITY="PriceCheckRequest/
PriceCheckRequestDetail/ListOfPriceCheckRequestItemDetail/
PriceCheckRequestItemDetail/PriceCheckRequestBaseItemDetail/
TotalQuantity/Quantity";
<PriceCheckRequest>
...
```

The bold text below identifies which portion of the document is selected:

```
<PriceCheckRequestDetail>
       <ListOfPriceCheckRequestItemDetail>
              <PriceCheckRequestItemDetail>
                     <PriceCheckRequestBaseItemDetail>
                            <TotalQuantity>
                                   <Quantity>-1</Quantity>
                            <TotalQuantity>
                            ...
```

Using the Position() Predicate to Narrow Your Selection

XPCDocHandle knows from a document's SOX schema whether a particular element occurs only once or can be repeated any number of times. The <PriceCheckRequestItemDetail>, element, for example, is a repeating element; there is one instance for each item whose price is being checked. XPCDocHandle uses an array to represent repeating elements **even if the particular document you are accessing has only a single instance**.

The position() predicate allows you to retrieve elements based upon their position within an array. Like all XPath predicates, it must be enclosed within left and right square brackets, [and].

Adding the predicate below to an XPath string restricts the selection to the first element in the array--that is, the element whose position is equal to 1:

```
[position()=1]
```

You could use the "is not equal to"operator, !=, instead of the "is equal to" operator, to select every element **other than** the first, as below:

```
[position() !=1]
```

The syntax for the position() predicate is:

[position() *comparison_operator n*]

The *comparison_operator* can be =, <, <=, >, >=, or != and *n* can be any integer.

Testing for the Value or Presence of a Node

Other predicates allow you to restrict your selection based upon the existence of a particular element or attribute, or upon its having a specified value. Adding the following predicate to an XPath string restricts the selection to instances that have a Currency attribute:

```
[@Currency]
```

If you add the following predicate to an XPath string, you only retrieve instances in which the Currency attribute has a value of “USD”:

```
[@Currency=\”USD\”]
```

Similarly, the following predicate restricts your selection to instances in which the Currency attribute has a value **other than** “USD”:

```
[@Currency!=\”USD\”]
```

Note ..... This type of predicate allows only the = and != comparison operators.

Using Compound Predicates

You can also create predicates that combine these features. The following predicate, for example, selects the first item in an array provided its Currency attribute has a value of “USD”:

```
[position()=1 and @Currency=\”USD\”
```

XPCDocHandle Limitations

Note the following caveats when using XPCDocHandle to set values in the response documents:

- The get(String atPath) and set(String atPath, Object objParams) methods in XPCDocHandle cannot handle xpaths containing nodes where a choice must be made in order to get to the next node in your xpath.

The xpath below, for example, cannot be specified in the get method, because the Quantity node in this path contains a quantity choice. The choices are QuantityValue and QuantityRange:

```
String XPATH_QUANTITY=”PriceCheckRequest/PriceCheckRequestDetail/
ListOfPriceCheckRequestItemDetail/PriceCheckRequestItemDetail/
```

```
PriceCheckRequestBaseItemDetail/TotalQuantity/Quantity/QuantityValue”;
```

To work around this situation, use the get method to retrieve the node just before the choice occurs and then manually get the selected choice, as below:

```
String XPATH_QUANTITY="PriceCheckRequest/
PriceCheckRequestDetail/ListOfPriceCheckRequestItemDetail/
PriceCheckRequestItemDetail/PriceCheckRequestBaseItemDetail/
TotalQuantity/Quantity;
Object quantityChoice =
ourQuantityObject.getQuantityChoice().getChoice();

if (quantityChoice instanceof QuantityValue)  {
….
}
else if (quantityChoice instanceof QuantityRange) {
       ….
}
```

- XPC ignores any invalid values set by XPCDocHandle, returning instead the value specified by the service’s defaultResponseBuilder component class. Values specified in the service’s configuration file override these default values. Specifying an obviously incorrect value in the configuration file is a convenient way to alert your back office application to the presence of errors in the response document.

  The defaultAvailabilityCheckResponseBuilder class, for example, specifies an AvailableQuanity equal to the quantity that was requested. Be careful not to remove the following line, which changes the AvailableQuantity to -1, from the service’s default.prop file:

  AvResponseBuilder.config=Quantity= -1

- When the XPCDocHandlexDocHandle.get() method is called on a node that does not exist in an xCBL document’s bean, it returns NULL. The XPCDocHandlexDocHandle.set() method, however, **cannot** be used to set the value of such a node; it throws an IllegalAccessError exception. Before calling the set method on the node, you must first create the subtree that contains the node.

  To set error information into a response document, for example, you must first build the entire ErrorInfo branch of the xCBL document. The sample integrators

that ship with XPC call the XPCErrorInfo.buildErrorInfo() method to build the CBL_sox.ErrorInfo object before calling the hDocOutgoing.set(XPATH_ERRORINFO,vResults) method to set the value of the error information.

The sample components included in these services demonstrate use of the XPC's XPath based document programming model.

Correlation Across Sessions

For a file system document exchange approach for handling peer to peer exchanges., a request is handled by an asynchronous document service, the response by a timed service.

To respond to a request, the timed service must have both the response document and the original request envelope. This is the reply() signature in XPCAbstractComponent.

This can be managed by sharing a file name key from the request to the response. The document service configures a FileStore action to store the request envelope using a key. The responding time service determines the key to sends it to the lookupEnvelope action on FileStore.

The Trading Partner Preconfiguration uses a component GetCorrelationKey to define the naming scheme. By default, the key is the correlation id property of the envelope. However, this component can be configured with an XPath query string to denote specific data elements in the document.

The correlation key is defined by the document service. This means that the timed service needs to determine this key from the response data passed to it by the ERP system. This could be accomplished using a file naming scheme, or XPath query into the document.

To facilitate this retrieval, XPC provides this common correlation key for all files associated with an asynchronous the transaction—the request envelope, request document, its envelope, the message acknowledgment document, the reply document, and any attachments. XPC uses this correlation key, plus configurable prefixes, extensions, and directory names, to determine where these the files are stored. The ERP system uses the must use this same key to pass response documents back to XPC.

Example Inbound Flow: Order

The action director configuration for the Order document service specifies the following actions:

1. Build default response (DefaultOrderResponse30Builder)

   The default Action Taken is “NoAction”.

2. Determine file name key (GetCorrelationKey)

   The GetCorrelationKey component This extracts the desired file name key from the envelope. By default, this is the envelope’s Correlation ID, but you can use the XPath configuration to obtain other values from the xCBL document -- concatenating the Buyer Account Code with the PO Number, for example.

3. Store the request envelope and its contents

   The Filestore.storeEnvelope action stores the request envelope and each of its parts in a separate subdirectory of the /filestore directory. The common correlation key makes it easy to identify all files associated with a single request.

   The following table shows the default subdirectory and file names assigned to the request envelope, document, and any attachments. You can reconfigure the component to specify new locations and file names.

Part	Subdirectory	Prefix, extension	Description
Envelope	/envelope	E_*key*.env	The entire MIME envelope
Document	/request	R_*key*.xml	The xCBL document that is the body of the envelope
Attachments	/attachment	A_*#_key*.att	Each attachment has its own unique number
Attachment description	/meta	M_*key*.adf	File enumerating all the attachment files and their locations

4. Store the Default Response document

   The FileStore.storeDocument component stores the default response document in the filestore/default_response directory on the file system. The file naming convention is based on the correlation key of the request document:

   D_*key*.xml

5. Store request envelope (Filestore.storeEnvelope)

   This component takes the correlation key as an input, which comes from the output of GetCorrelationKey. By default, the request envelope and each of its parts are stored in a separate subdirectory of the /filestore directory:

   Configurations for the file store component allow you to change these default file locations and naming conventions.

6. Store default response document (FileStore.storeDocument)

   This stores the response document as generated by the first action.

Default Response Document	/default_response	D_*key*.xml	File containing the default PurchaseOrderResponse document
Response Document	/response	D_*key*.xml	File that the timed service must find

Example ERP flow

1. Pick up and load purchase order
2. Pick up the default response document from filestore/default_response directory, update it, and save it to the /response directory where the timed service can pick it up.

   Warning! The ERP system needs a queuing mechanism to prevent it from reprocessing a document. This could be implemented by moving processed files to a different directory.

   Ideally, the queuing mechanism should use the default response document (filestore/default_response/D_key.xml). If the queue uses the request envelope, that implies that the ERP archives this file -- and the timed service action would need to be reconfigured to look in this archive directory.

3. Store the Customized Response

   The ERP calls picks up the default response from the filestore/default_response directory and customizes it. It then stores the customized response in the /response directory with the same file name as that used for the default response.

   When the timed service wakes up, it looks for the customized response in the /response directory.

Example outbound flow: OrderResponse

When the Timed Service wakes up, it executes the following actions:

1. Reads the response document (FileStore.readDocument)

   This method picks up the next document found in the /filestore/response directory, determines its correlation key, and archive directory and moves it in to the /filestore/response_archive directory.

2. FileStore.lookupEnvelope to search the /request directory for the r_key.env file. This file contains the envelope of the original request document.

   This component captures the correlation key (the string between the configurable prefix and the extension) from the file read and uses it to look up the original request envelope. The ERP system must use the same file name key for persisting the updated response as was used by the default response (though it can use a different directory, prefix, and extension).

3. Looks up request envelope (FileStore.lookupEnvelope)

   This component takes the correlation key issued above, and searches in the /request directory for the r_key.env file (as persisted by the PurchaseOrder Action Director).

4. Transmits to MarketSite (Responder)

   Responder This uses the request envelope to transmit the customized publish response document to MarketSite.

Building a Custom Component

XPC components must subclass XPCAbstractComponent.

```
public class myComponent extends
com.commerceone.xpc.abs.XPCAbstractComponent
```

Please reference Chapter 7 for a full description of this abstract class. The following capabilities are available to components through this abstract class.

- Access configurations
- Define input names and types
- Define output names and types
- Transmit new message

- Respond to message
- Logging

Sample Components

XPC contains sample components for that perform synchronous processing. They each take a request envelope and default response document as input, and generate an updated response document as output. The source code for these components can be found:

- <install:root>\sample\com\commerceone\sample\xpc\my_integrators\myAvailabilityCheckIntegrator30.java
- <install:root>\sample\com\commerceone\sample\xpc\my_integrators\myPriceCheckIntegrator30.java
- <install:root>\sample\com\commerceone\sample\xpc\my_integrators\myOrderStatusIntegrator30.java

Helper Methods

Each of these sample components contains a “helper” method, doAvailabilityCheck, doPriceCheck, and doOrderStatus. The signature of this private method includes the most common data needed to understand the request and define the response. It is called once for each item in the request.

This helper method serves to clearly isolate the location of the API call and the values the call needs and generates. In most integrations of PriceCheck, AvailabilityCheck, OrderStatus, it is only necessary to edit the helper method.

When editing helper methods, you must insert your own business logic to access the database or file system in which you store information about item availability, price, and order status. For example, this code could open a JDBC connection, create a SQL SELECT statement, execute the SQL, and read the response into the result parameters.

Adding Fields to the Helper Method signature

Each of the sample components contains two methods:

- A “helper” method, with a name such as doAvailabilityCheck() or doPriceCheck(), which accepts as input the key fields of the request document and returns as output the key fields of the response document, including any error information.

- A process() method, which calls the helper method once for each line item to be processed.

Because the helper methods are private, you cannot add fields to the signature. If you require additional information that is not contained in the signature, you must modify the process() method to include this information.

The signature of the doAvailabilityCheck() method, for example, accepts the buyer's account code, the supplier's partID and extension, the requested quantity, and the unit of measurement. These key fields uniquely identify the item whose availability is being checked:

```
private XPCResult doAvailabilityCheck(
final String acctCode_Buyer, // [IN]Buyer Account Code
final String partID_Supplier, // [IN] Supplier's PartId
final String partExt_Supplier, // [IN] Supplier's PartId
extension
final String quantity, // [IN] Requested Quantity
final String uomCode, // [IN] Unit of Measure (UOM) code
StringBuffer resultQuantity, // [OUT] Available Quantity
StringBuffer resultUOM, // [OUT] Unit of Measure code
StringBuffer errorCode, // [OUT,OPTIONAL] Error code if any
StringBuffer errorMessage, // [OUT,OPTIONAL] Descriptive error
message
StringBuffer errorVendorMessage  // [OUT,OPTIONAL] vendor
specific error message
)
```

If the item is processed successfully, the doAvailabilityCheck() method returns two string buffers, one containing the available quantity and the other the unit of measurement:

```
private XPCResult doAvailabilityCheck(
final String acctCode_Buyer, // [IN]Buyer Account Code
final String partID_Supplier, // [IN] Supplier's PartId
final String partExt_Supplier, // [IN] Supplier's PartId
extension
final String quantity, // [IN] Requested Quantity
final String uomCode, // [IN] Unit of Measure (UOM) code
```

```
StringBuffer resultQuantity, // [OUT] Available Quantity
StringBuffer resultUOM, // [OUT] Unit of Measure code
StringBuffer errorCode, // [OUT,OPTIONAL] Error code if any
StringBuffer errorMessage, // [OUT,OPTIONAL] Descriptive error
message
StringBuffer errorVendorMessage  // [OUT,OPTIONAL] vendor
specific error message
)
```

If business errors, such as an invalid PartID, prevent the item from being processed, the doAvailabilityCheck() method returns three string buffers, one containing an error code, one an error message suitable for display to users, and one a more technical description of the error suitable for troubleshooting. These buffers are highlighted below:

```
private XPCResult doAvailabilityCheck(
final String acctCode_Buyer, // [IN]Buyer Account Code
final String partID_Supplier, // [IN] Supplier's PartId
final String partExt_Supplier, // [IN] Supplier's PartId
extension
final String quantity, // [IN] Requested Quantity
final String uomCode, // [IN] Unit of Measure (UOM) code
StringBuffer resultQuantity, // [OUT] Available Quantity
StringBuffer resultUOM, // [OUT] Unit of Measure code
StringBuffer errorCode, // [OUT,OPTIONAL] Error code if any
StringBuffer errorMessage, // [OUT,OPTIONAL] Descriptive error
message
StringBuffer errorVendorMessage  // [OUT,OPTION] vendor specific
error message
)
```

Error Handling

This section describes how XPC handles business errors. It includes information about:

- Setting the ErrorInfo element in a response document

- ErrorInfo coding conventions

Business errors occur when XPC is able to create a response document but is unable to fulfill the request. Business errors commonly result when the request document includes fatal errors such as invalid account codes, expired contracts, or invalid part numbers.

When a business error occurs, the component ceases to execute the action list and fills the response document's ErrorInfo element with information that describes the error.

Warning! Use of the ErrorInfo element indicates to the requesting application that the request cannot be fulfilled. Use this element only for **fatal** errors. Do **not** use ErrorInfo for warnings or informational messages.

Following is the SOX schema for the ErrorInfo element:

```
<elementtype name="ErrorInfo">
       <model>
          <sequence>
             <element type="string" name="CompletionCode"/>
             <element type="LangString" name="CompletionMsg"/>
             <element type="SeverityCode" name="Severity"/>
             <element type="ListOfParameter" occurs="?"/>
             <element type="int" name="MinRetrySecs" occurs="?"/>
             <element type="string" name="SwVendorErrorRef" occurs="?"/>
          </sequence>
       </model>
</elementtype>
```

You must specify values for the following required fields:

- CompletionCode, which specifies a standard error code.
- CompletionMsg, which provides a description of the error in language suitable for display to users of the application.
- Severity, which should be set to "Error".

The optional SwVendorErrorRef field can be used for system-specific error information that may help in troubleshooting.

The ErrorInfo schema does not have any enumerated values for CompletionCode. This document provides a list of recommended CompletionCode values. This same list will be available to various MarketSite buying solutions, giving trading partners the option of implementing specific processing for specific codes.

Setting ErrorInfo

To set the error information in a reply document, modify the helper method in your my_integrator component. To set the error information in the sample my_integrators supplied by XPC, you need to modify:

- doAvailabilityCheck( ) in the myAvailabilityCheckIntegrator class
- doPriceCheck( ) in the myPriceCheckIntegrator class
- doOrderStatus() in the myOrderStatusIntegrator class

The modification consists of:

- Setting the errorCode to the appropriate CompletionCode value, as indicated in the **ErrorInfo Coding Conventions** section below
- Setting the errorMessage to an error message description suitable for display to the user

ErrorInfo Coding Conventions

CompletionCode values are all uppercase. Each code begins with a prefix, indicating the type of document being processed, followed by a brief description of the problem. XPC CompletionCodes use the following prefix conventions:

- Price Check errors begin with PC_
- Availability Check errors begin with AC_
- Order Status errors begin with OS_
- Sales Order (also known as Purchase Order) errors begin with SO_
- Standard errors that apply to all document types begin with STD_

Recommended Price Check Completion Codes

The PriceCheckResult document, which is generated in response to a PriceCheckRequest, stores error information at both the detail and summary levels.

When an error related to an individual line item occurs, the error information is stored in the item's PriceCheckResult/ListOfPriceResultItem/PriceResultItem/ PriceErrorInfo element. The following table lists the recommended CompletionCodes for different types of line item errors:

Type of Error	CompletionCode	Severity
Part Number / SKU not found	PC_SKU_NOTFOUND	Error
Invalid Quantity specified	PC_QTY_INVALID	Error
General Error such as back-end or authentication failure	STD_ERROR_GENERIC	Error

If errors occur for one or more line items, the PriceCheckResult/PriceCheckSummary/PriceCheckSummaryErrorInfo/ CompletionCode element should be set to PC_ERROR_RESULTITEM. This element should also be used for other errors that apply to the entire PriceCheckRequest.

The following table lists the recommended CompletionCodes for different types of summary errors:

Type of Error	CompletionCode	Severity
Invalid Account code	STD_ACCOUNT_INVALID	Error
No valid Contract or Contract Expired	STD_CONTRACT_EXPIRED	Error
General errors such as back-end failure	STD_ERROR_GENERIC	Error
Errors occurred during price checks for one or more line items	PC_ERROR_RESULTITEM	Error

Recommended Availability Check Completion Codes

The AvailabilityCheckResult document, which is generated in response to an AvailabilityCheckRequest, stores error information at both the detail and summary levels.

When an error related to an individual line item occurs, the error information is stored in the item's AvailabilityCheckResult/ListOfAvailabilityResultItem/ AvailabilityResultItem/AvailabilityErrorInfo element. The following table lists the recommended CompletionCodes for different types of line item errors:

Type of Error	CompletionCode	Severity
Part Number / SKU not found	AC_SKU_NOTFOUND	Error
Invalid Quantity specified	AC_QTY_INVALID	Error
General error such as back-end failure	STD_ERROR_GENERIC	Error
No inventory for this part number	AC_NOINSTOCKQTY	Error

If errors occur for one or more line items, the AvailabilityCheckResult/ AvailabilityCheckSummary/SummaryErrorInfo/CompletionCode element should be set to AC_ERROR_RESULTITEM. This element should also be used for other errors that apply to the entire AvailabilityCheckRequest.

The following table lists the recommended CompletionCodes for different types of summary errors:

Type of Error	CompletionCode	Severity
Invalid Account code	STD_ACCOUNT_INVALID	Error
General error such as back-end failure	STD_ERROR_GENERIC	Error
Errors occurred during availability checks for one or more items	AC_ERROR_RESULTITEM	Error

Recommended Order Status Completion Codes

The OrderStatusResult document, which is generated in response to an OrderStatusRequest document, stores error information at both the detail and summary levels.

Note ..... In general, Enterprise Buyer ignores the line item status information transmitted during an OrderStatus transaction. It uses this information only when the Order as a whole has been rejected.

When an error occurs while checking the status of an individual line item, the information is stored in the OrderStatusResult/ListOfOrderStatusDetailResult/ OrderStatusDetail element of the OrderStatusCheckResult document. The following table lists the recommended CompletionCodes for different types of line item errors:

Type of Error	CompletionCode	Severity
Order not found	OS_ORDER_NOTFOUND	Error
Invalid account code	STD_ACCOUNT_INVALID	Error
General error such as back-end failure	STD_ERROR_GENERIC	Error

If errors occur for one or more line items, the OrderStatusResult/ OrderStatusCheckSummary/OrderStatusSummaryErrorInfo element should be set with a CompletionCode of OS_ERROR_RESULTITEM and the Severity set to Error.

Recommended Sales Order Completion Codes

The OrderResponse document, which is generated in response to an Order document, stores error information at both the detail and summary levels.

Line item errors related to price are stored in the OrderResponse/ ListOfOrderResponseDetail/OrderResponseDetail/OrderDetail/PriceErrorInfo element. The following table lists the recommended CompletionCodes for different types of line item errors related to price:

Type of Error	CompletionCode	Severity
Part / SKU not found	SO_ITEM_NOTFOUND	Error
LineNum/SubLine num not unique	SO_LINENUMSUBLINENUM_NOTUNIQUE	Error
Invalid PRICE, Price doesn’t match	SO_PRICE_INVALID	Error
Contract Expired	SO_PRICECONTRACT_EXPIRED	Error
General error such as back-end failure	STD_ERROR_GENERIC	Error

Line item errors related to availability are stored in PurchaseOrderResponse/ListOfOrderResponseDetail/OrderResponseDetail/OrderDetail/AvailabilityErrorInfo. The following table lists the recommended CompletionCode values for different types of line item errors related to availability:

Type of Error	CompletionCode	Severity
Part / SKU not found	SO_ITEM_NOTFOUND	Error
The LineNum and SubLineNum are not unique	SO_LINENUMSUBLINENUM_NOTUNIQUE	Error
Invalid Qty	SO_QTY_INVALID	Error
Insufficient Qty	SO_NOINSTOCKQTY	Error
General error such as back-end failure	STD_ERROR_GENERIC	Error

If errors occurred at one or more line items, set the PurchaseOrderResponse/OrderResponseSummary/OrderResponseErrorInfo to SO_ERROR_ORDERITEM. The following table lists the recommended CompletionCode values for different types of summary errors:

Type of Error	CompletionCode	Severity
Invalid PO, General error	SO_INVALID_DATA	Error
Invalid PO, with invalid data in the specified field (for example, the SHIPTO field or the BILLTO field).	SO_INVALID_DATA_<field_name> For example: SO_INVALID_DATA_SHIPTO SO_INVALID_DATA_BILLTO SO_INVALID_DATA_BILLTO_ZIP	Error
Invalid Account Code	STD_ACCOUNT_INVALID	Error
Errors occurred while processing one or more line items	SO_ERROR_ORDERITEM	Error
General error such as back-end failure	STD_ERROR_GENERIC	Error

Other APIs

More sophisticated component development may require the use of other Commerce One supplied APIs, including:

- The XDK package, which allows for the direct manipulation of the contents of documents and envelopes
- The Util package, which contains utility classes that may be used to create unique identifiers

For information about using these APIs, see the Javadoc contained in the *\doc\api\xdk* and *doc\api\util* directories of the XPC installation.

Deploying a Component

After modifying a component, you compile it to a class file such as myPriceCheckIntegrator.class. This section describes how to package the component as a JAR file and how to deploy without generating a JAR file.

Deploying with JAR Files

After modifying the component, compile it to a class file (for example, myPriceCheckIntegrator.class), put the compiled files into a JAR file, and add the JAR file to the classpath.

The following sample batch file uses Sun Microsystems JDK compiler to compile a sample integrator. The batch code could be copied into a batch file (for example, compile.bat) then executed from a command line (for example, compile \sample\com\commerceone\xpc\ my_integrators\myPriceCheckIntegrator30.java).

```
   "javac" -d classes -classpath
".;%XPCROOT%\lib\activation.jar;%XPCROOT%\lib\bussdocs.jar;%XPCROOT%\lib\XPC.jar;%X
PCROOT%\lib\ccs_server.jar;%XPCROOT%\lib\ccs_dir.jar;%XPCROOT%\lib\ccs_event.jar;%X
PCROOT%\lib\ccs_util.jar;%XPCROOT%\lib\ccs_xdk.jar;%XPCROOT%\lib\iaik.jar;%XPCROOT%
\lib\iaik_jce_applet.jar;%XPCROOT%\lib\iaik_ssl_applet.jar;%XPCROOT%\lib\jigsawlite
.jar;%XPCROOT%\lib\jmail.jar;%XPCROOT%\lib\jndi.jar;%XPCROOT%\lib\jsafe.jar;%XPCROO
T%\lib\jsdk.jar;%XPCROOT%\lib\mail.jar;%XPCROOT%\lib\swingall.jar;%XPCROOT%\lib\lda
p.jar;%XPCROOT%\lib\providerutil.jar;%XPCROOT%\lib\sax.jar;%XPCROOT%\lib\ccs_xdkdir
.jar"  %*
```

To add a new JAR to XPC's classpath:

1. Open the file \etc\classpath\default.
2. Add a line containing the full path to the new JAR file.
3. Ensure there is an empty line after the last line in the file.
4. Stop and restart XPC.

To verify that the JAR file was correctly added:

1. Start the server.
2. Open the file \bin\CCSNTService.log.
3. Confirm the new JAR is present in this log file.

Deploying Without Generating a JAR File

You may also use the following option to deploy without generating a JAR file:

1. Compile the custom component's .java file to a .class file.
2. Copy the .class file to the following directory:
   \lib\com\commerceone\xpc\my_integrators\
3. Stop and restart XPC.

4 Trading Partner Preconfiguration

In This Chapter

This chapter provides information about the XPC Trading Partner Preconfiguration, a set of preconfigured business services designed to support the xCBL 3.0 transactions used by buyers and suppliers.

Loading the Trading Partner Preconfiguration

Loading the Trading Partner Preconfiguration loads a set of service definitions to support xCBL 3.0. Once XPC has been installed on your computer, you can use the following steps to load the Trading Partner Preconfiguration:

1. From the **Start** menu, select **Programs** | **XMLPortal Connector 4.0** | **Configure**. The **Configure XPC** window appears.
2. Click the **Preconfigure Trading Partner** button.

Note ..... The Trading Partner Preconfiguration can only be loaded one time. Once it has been loaded, the **Preconfigure Trading Partner** button is disabled.

After loading the Trading Partner Preconfiguration, you must enable each of the services you will be running. For information about which services to enable, see the remaining sections of this chapter. For information about how to enable services, see the **XPC Manager**chapter.

Before using XPC services, it is likely that you will want to reconfigure them. XPC Manager allows you to make various types of modifications to a service's Action Director. For information about reconfiguring services, see the **XPC Manager** chapter.

Supported Services

Each of the following sections provides information about the services associated with a particular type of transaction:

Advance Shipment Notice Services

This section describes how to configure the XPC services used for Advance Shipment Notice transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Advance shipment notice transactions are one-way asynchronous document exchanges. When suppliers send an AdvanceShipmentNotice to buyers, they expect to receive a MessageAcknowledgement but do not expect a response document.

Following is an overview of advance shipment notice transactions:

1. A supplier sends AdvanceShipmentNotice to a buyer.
2. The buyer sends a MessageAcknowledgement document indicating that the AdvanceShipmentNotice was received.

The following diagram illustrates the flow of xCBL documents:

Note ..... The diagrams in this chapter do not include arrows depicting MessageAcknowledgement documents. These are included in all peer-to-peer and one-way asynchronous document exchanges for every transaction.

Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable to process these transactions:

XPC Service	Used By Buyers	Used By Supplier
XPCAdvanceShipmentNotice30Outbound Service on page 5		●
XPCAdvanceShipmentNotice30Inbound Service on page 5	●	

Document Exchange Details

This section provides details about the services that manage Advance Shipment Notice transactions:

1. The supplier's back office system creates the AdvanceShipmentNotice and leaves it in a designated location on its local file system.
2. The supplier's XPCAdvanceShipmentNotice30Outbound service wakes up periodically and searches the specified directory for the AdvanceShipmentNotice. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the AdvanceShipmentNotice to the buyer.
4. The buyer's XPCAdvanceShipmentNotice30Inbound service extracts the AdvanceShipmentNotice from its envelope, sends a MessageAcknowledgement document, and stores and stores the AdvanceShipmentNotice envelope, document, and attachment in designated locations on its local file system.

XPCAdvanceShipmentNotice30Outbound Service

Supplier's XPCAdvanceShipmentNotice30Outbound service wakes up periodically and searches the specified directory for the AdvanceShipmentNotice. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the AdvanceShipmentNotice document from the file system, archives it, and uses it to build the document object.
GetStringFromDocument	Extracts a string from the document object by applying the xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCAdvanceShipmentNotice30Inbound Service

The Buyer's XPCAdvanceShipmentNotice30Inbound service extracts the AdvanceShipmentNotice from its envelope, sends a MessageAcknowledgement document, and stores the AdvanceShipmentNotice envelope, document, and attachment in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives AdvanceShipmentNotice document and send MessageAcknowledgement

Action List Component	Description
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the AdvanceShipmentNotice envelope, document, and attachment.
FileStore.storeEnvelope	Stores the AdvanceShipmentNotice Envelope, document and attachment on the file system.

Auction Management Services

This section describes how to configure the XPC services used for auction management transactions. It includes the following information:

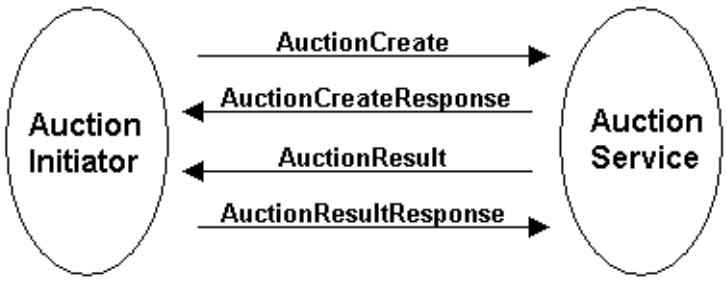
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Following is an overview of auction management transactions:

1. An auction is initiated by sending an AuctionCreate document. Forward auctions are initiated by suppliers; reverse auctions by buyers.
2. Auction Services receives the AuctionCreate, organizes an auction, and notifies the initiator and other interested parties by sending an AuctionCreateResponse document.
3. Auction Services selects the winning bid and notifies the auction initiator by sending an AuctionResult document.
4. The auction initiator replies with an AuctionResultResponse document.

The following diagram illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that auction initiators and auction services must enable and configure to process auction management transactions:

XPC Service	Used By Auction Initiator	Used By Auction Service
XPCAuctionCreate30Outbound Service on page 10	●	
XPCAuctionCreate30Inbound Service on page 11		●
XPCAuctionCreateResponse30Outbound Service on page 12		●
XPCAuctionCreateResponse30Inbound Service on page 12	●	
XPCAuctionResult30Outbound Service on page 13		●
XPCAuctionResult30Inbound Service on page 13	●	
XPCAuctionResultResponse30Outbound Service on page 14	●	
XPCAuctionResultResponse30Inbound Service on page 15		●

Document Exchange Details

Details of the document exchange are provided below:

1. An auction is initiated by sending an AuctionCreate document. Forward auctions are initiated by suppliers; reverse auctions by buyers.
   - **a)** The auction initiator's back office system creates the AuctionCreate document and leaves it in a designated location on its local file system.
   - **b)** The auction initiator's XPCAuctionCreate30Outbound service wakes up periodically and searches the specified directory for the AuctionCreatedocument. It places the document in an xCBL envelope and transmits it to MarketSite.
   - **c)** MarketSite routes the AuctionCreate document to Auction Services.
2. Auction Services receives the AuctionCreate, organizes an auction, and notifies the initiator and other interested parties by sending an AuctionCreateResponse document.

a) Auction Services's XPCAuctionCreate30Inbound service extracts the AuctionCreate document from its envelope, builds a default AuctionCreateResponse document, and stores both the AuctionCreate document and the AuctionCreateResponse document in designated locations on its local file system.

b) Auction Services's XPCAuctionCreateResponse30Outbound service wakes up periodically and searches the specified directory for the AuctionCreateResponse document. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the AuctionCreateResponse document to the initiator of the auction.

d) The initiator's XPCAuctionCreateResponse30Inbound service extracts the AuctionCreateResponse document from its envelope, sends a MessageAcknowledgement, and stores the AuctionCreateResponse document and its envelope in a designated location on the local file system.

3. Auction Services selects the winning bid and notifies the auction initiator by sending an AuctionResult document.

a) Auction Services selects the winning bid.

b) Auction Services creates an AuctionResult document and stores it in a specified location on its local file system.

c) Auction Services' XPCAuctionResult30Outbound service wakes up periodically and searches the specified directory for the AuctionResult document. It places the document in an xCBL envelope and transmits it to MarketSite.

d) MarketSite routes the AuctionResult document to the initiator of the auction.

e) The auction initiator's XPCAuctionResult30Inbound service extracts the AuctionResult document from its envelope, sends a MessageAcknowledgement, and stores the AuctionResult document and its envelope in a designated location on the local file system.

4. The auction initiator replies with an AuctionResultResponse document.

a) The auction initiator creates an AuctionResultResponse document and stores it in a designated location on the local file system.

b) The auction initiator's XPCAuctionResultResponse30Outbound service wakes up periodically and searches the specified directory for the AuctionResultResponse document. it places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the AuctionResultResponse document to Auction Services.

d) Auction Services' XPCAuctionResultResponse30Inbound service extracts the AuctionResultResponse document from its envelope, sends a MessageAcknowledgement, and stores the AuctionResultResponse document and its envelope in a designated location on the local file system.

XPCAuctionCreate30Outbound Service

The auction initiator's XPCAuctionCreate30Outbound service wakes up periodically and searches the specified directory for the AuctionCreatedocument. (Forward auctions are initiated by suppliers; reverse auctions by buyers.) The service places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the AuctionCreate document from the file system, archives it, and uses it to build the document object.
GetStringFromDocument	Gets the key string specified in the configuration.
StringMapper	Gets the TPID matching the key string from the map file.
CreateEnvelope	Creates Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCAuctionCreate30Inbound Service

Auction Services's XPCAuctionCreate30Inbound service extracts the AuctionCreate document from its envelope, builds a default AuctionCreateResponse document, and stores both the AuctionCreate document and the AuctionCreateResponse document in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the AuctionCreate document and sends a MessageAcknowledgement.
GetCorrelationKey	Builds a CorrelationKey from the MessageId of the envelope is used
DefaultAuctionCreateResponse30Builder	Builds a default AuctionCreateResponse document.
FileStore.storeEnvelope	Stores the AuctionCreate Envelope, document and attachment on the file system. File names are formed by concatenating the appropriate prefix with the CorrelationKey.
FileStore.storeDocument	Stores the default AuctionCreateResponse document on the file system.

XPCAuctionCreateResponse30Outbound Service

Auction Services's XPCAuctionCreateResponse30Outbound service wakes up periodically and searches the specified directory for the AuctionCreateResponse document. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the AuctionCreateResponse document from the file system and archives it. Builds the document object from this document and builds filename key string from the name of the file.
Filestore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the AuctionCreateResponse document to MarketSite.

XPCAuctionCreateResponse30Inbound Service

The initiator's XPCAuctionCreateResponse30Inbound service extracts the AuctionCreateResponse document from its envelope, sends a MessageAcknowledgement, and stores the AuctionCreateResponse document and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the AuctionCreateResponse document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the AuctionCreateResponse envelope, document.
FileStore.storeEnvelope	Stores the AuctionCreateResponse Envelope, document and attachment on the file system. File names are formed by concatenating the appropriate prefix with the CorrelationKey.

XPCAuctionResult30Outbound Service

Auction Services' XPCAuctionResult30Outbound service wakes up periodically and searches the specified directory for the AuctionCreateResponse document. It places the document in an xCBL envelope correlated with the AuctionCreate envelope, and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the AuctionCreate document from file system and archive it. Builds the document object from the document read. Gets a key string from the filename. The key string must match the filename key used to persist the AuctionCreate envelope.
FileStore.lookupEnvelope	Looks up AuctionCreate envelope that matches the key string.
CreateCorrelatingEnvelope	Creates the AuctionResult envelope using the header information of the AuctionCreate envelope looked up. Keep the correlationId same. Swap the senderId and recipientId.
Transmitter	Transmits the envelope to MarketSite.

XPCAuctionResult30Inbound Service

The auction initiator's XPCAuctionResult30Inbound service extracts the AuctionResult document from its envelope, sends a MessageAcknowledgement, and stores the AuctionResult document and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the AuctionResult document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the AuctionResult envelope, document, and attachment and the default AuctionResultResponse document.

Action List Component	Description
DefaultAuctionResultResponse30Builder	Builds a default AuctionResultResponse document.
FileStore.storeEnvelope	Stores the AuctionResult Envelope, document and attachment on the file system. File names are formed by concatenating the appropriate prefix with the CorrelationKey.
FileStore.storeDocument	Stores the default AuctionResultResponse document on the file system. The file name is formed by concatenating the appropriate prefix with the CorrelationKey.

XPCAuctionResultResponse30Outbound Service

The auction initiator's XPCAuctionResultResponse30Outbound service wakes up periodically and searches the specified directory for the AuctionResultResponse document. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads AuctionResultResponse document from file system and archive it. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Transmits the response document to MarketSite.

XPCAuctionResultResponse30Inbound Service

Auction Services' XPCAuctionResultResponse30Inbound service extracts the AuctionResultResponse document from its envelope, sends a MessageAcknowledgement, and stores the AuctionResultResponse document and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the AuctionResultResponse document and sends a MessageAcknowledgement
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the AuctionResultResponse envelope, document, and attachment.
FileStore.storeEnvelope	Stores the AuctionResultResponse Envelope, document and attachment on the file system. File names are formed by concatenating the appropriate prefix with the CorrelationKey.

Availability Check Request Services

This section describes how to configure the XPC services used for availability check request transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Availability check transactions are synchronous document exchanges. The XPC connection used to receive the request document remains open until the response document is returned.

Following is an overview of the document exchange:

1. A buyer sends an AvailabilityCheckRequestDocument to determine the quantity of a product the supplier has available to sell.
2. The supplier returns an AvailabilityCheckResult document indicating the available quantity of the product.

The following diagram illustrates the flow of xCBL documents:

Associated XPC Services

The following table lists the XPC services that suppliers must enable to process availability check request transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCAvailabilityCheckRequest30Inbound Service on page 17		●

Document Exchange Details

Details of the document exchange are provided below:

1. The buyer's back office system creates an AvailabilityCheckRequest document and transmits it to MarketSite.
2. MarketSite forwards the AvailabilityCheckRequest to the supplier.
3. The supplier's XPCAvailabilityCheckRequest30Inbound service receives the AvailabilityCheckRequest document, builds a default AvailabilityCheckResult document based upon the request, updates the default AvailabilityCheckResult document with business data from the back office system, and sends the customized AvailabilityCheckResult document to MarketSite.
4. MarketSite forwards the AvailabilityCheckResult document to the buyer.

XPCAvailabilityCheckRequest30Inbound Service

Action List Component	Description
DefaultAvailabilityCheckResponse30Builder	Receives the AvailabilityCheckRequest document and uses it to build a default AvailabilityCheckResult document
Customizer	Customizes and updates the default AvailabilityCheckResult document
Responder	Sends the AvailabilityCheckResult document to MarketSite.

Availability To Promise Services

This section describes how to configure the XPC services used for availability to promise transactions. It includes the following information:

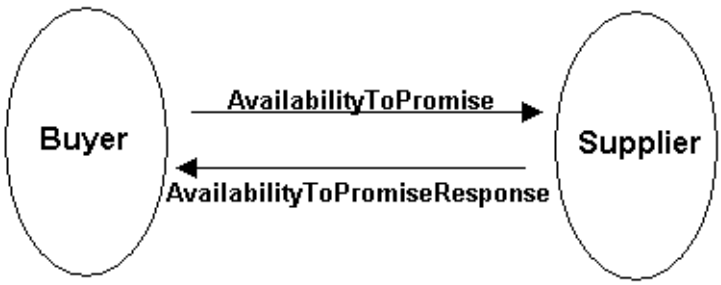
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transaction

A typical document exchange is as follows:

- A buyer sends an AvailabilityToPromise document to a supplier to determine when, where, and how many of the requested goods the supplier can provide.
- In response, the supplier sends the buyer an AvailabilityToPromiseResponse indicating the quantity of goods available for the buyer.

The following diagram illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable to process availability to promise transactions:

XPC Service	Used By Buyer	Used By Supplier
Associated XPC Services on page 19	●	
XPCAvailabilityToPromise30Inbound Service. on page 20		●
XPCAvailabilityToPromiseResponse30Outbound Service. on page 21		●
XPCAvailabilityToPromiseResponse30Inbound Service. on page 21	●	

Document Exchange Details

Following is a detailed description of the exchange of documents:

1. The buyer's back office system prepares an AvailabilityToPromise document and leaves it in a designated location on its local file system.
2. Buyer's XPCAvailabilityToPromise30Outbound service wakes up periodically and searches the specified directory for the AvailabilityToPromise document. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the AvailabilityToPromise document to the supplier.
4. The supplier's XPCAvailabilityToPromise30Inbound service extracts the AvailabilityToPromise document from its envelope, builds a default AvailabilityToPromiseResponse document, and stores both the AvailabilityToPromise and the AvailabilityToPromiseResponse documents in designated locations on its local file system.
5. The supplier's XPCAvailabilityToPromiseResponse30Outbound service wakes up periodically and searches the specified directory for the AvailabilityToPromiseResponse document. It places the document in an xCBL envelope and transmits it to MarketSite.
6. MarketSite routes the AvailabilityToPromiseResponse document to the buyer.
7. The buyer's XPCAvailabilityToPromise30ResponseInbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and

stores the AvailabilityToPromiseResponse document and its envelope in a designated location on the local file system.

XPCAvailabilityToPromise30Outbound Service.

Action List Component	Description
FileStore.readDocument	Reads the AvailabilityToPromise document from the file system and archives it. Builds the document object from the document.
GetStringFromDocument	Extracts a string from the document object using xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an envelope for the document. Uses the input string as the recipientId and the sender TPID configuration as senderId.
Transmitter	Transmits the envelope to MarketSite.

XPCAvailabilityToPromise30Inbound Service.

Action List Component	Description
MessageAcknowledgementSender	Builds envelope containing MessageAcknowledgement document and transmits it.
GetCorrelationKey	Gets the correlation key from the envelope.
DefaultAvailabilityToPromiseResponseBuilder	Builds a default AvailabilityToPromiseResponse document.
FileStore.storeEnvelope	Stores the envelope, document, and attachments on the file system.
FileStore.storeDocument	Stores the response document on the file system.

XPCAvailabilityToPromiseResponse30Outbound Service.

Action List Component	Description
FileStore.readDocument	Reads the AvailabilityToPromiseResponse document from file system and archive it. Builds the document object from the document.
FileStore.lookupEnvelope	Finds the original request envelope using the filename key.
Responder	Transmits the response document to MarketSite.

XPCAvailabilityToPromiseResponse30Inbound Service.

Action List Component	Description
MessageAcknowledgementSender	Builds and envelope for the MessageAcknowledgement and transmits it.
GetCorrelationKey	Gets the correlation key from the envelope.
FileStore.storeEnvelope	Stores the envelope, document, and attachments on the file system.

Invoice Services

This section describes how to configure the XPC services used for invoice transactions. It includes the following information:

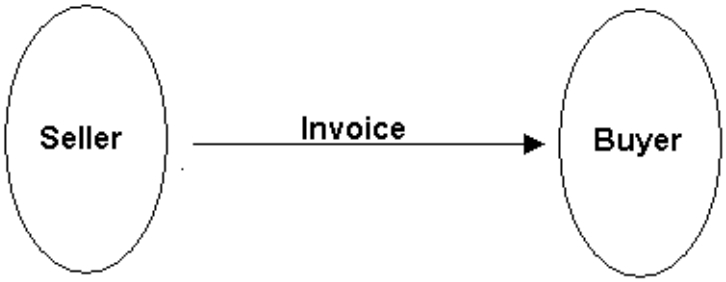
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transaction

Invoice transactions are one-way document exchanges. Following is a an overview of an invoice transaction:

1. A supplier sends an Invoice document to a buyer.
2. The buyer sends a MessageAcknowledgement document indicating that the Invoice was received.

The following diagram illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable to process invoice transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCInvoice30Outbound Service on page 23		●
XPCInvoice30Inbound Service on page 24	●	

Document Exchange Details

Following are the detailed steps involved in the exchange:

1. Supplier's back office system creates the Invoice and leaves it in a designated location on its local file system.
2. Supplier's XPCInvoice30Outbound service wakes up periodically and searches the specified directory for the Invoice. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the Invoice to the Buyer.
4. The Buyer's XPCInvoice30Inbound service extracts the Invoice from its envelope, sends a MessageAcknowledgement document, and stores both documents in designated locations on its local file system.

XPCInvoice30Outbound Service

Supplier's XPCInvoice30Outbound service wakes up periodically and searches the specified directory for the Invoice. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the Invoice document from file system, archives it and uses it to builds the document object.
GetStringFromDocument	Extracts a string from document object using xPath configuration.

Action List Component	Description
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCInvoice30Inbound Service

The Buyer's XPCInvoice30Inbound service extracts the Invoice from its envelope, sends a MessageAcknowledgement document, and stores both documents in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the Invoice document and sends a MessageAcknowledgement
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the Invoice envelope, document, and attachment.
FileStore.storeEnvelope	Stores the Invoice Envelope, document and attachment on the file system. File names are formed by concatenating the appropriate prefix with the CorrelationKey.

Message Acknowledgement and Error Services

When an inbound peer-to-peer or one-way service receives an xCBL document, it responds by sending a MessageAcknowledgement document. If the inbound service cannot read the incoming document, it sends an Error document.

XPCMessageAcknowledgement30Inbound Service

When an asynchronous xCBL document is transmitted, the receiving trading partner is responsible for automatically replying with a MessageAcknowledgement document. This service receives and stores this receipt confirmation.

Action List Component	Description
CorrelationKeyBuilder	Receives the document. Builds a CorrelationKey from the MessageId of the envelope.
FileStore.storeDocument	Stores the MessageAcknowledgement document on the file system.

Order Management Services

This section describes the XPC services used to process order management transactions. It includes the following information:

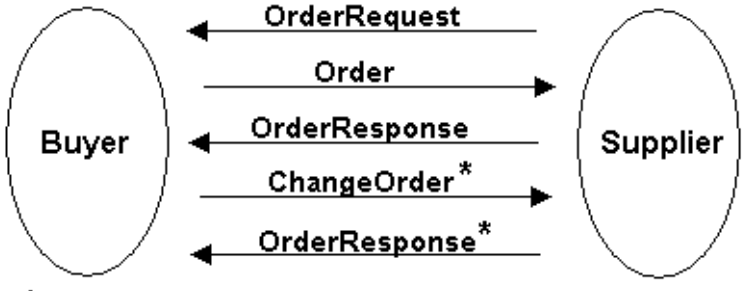
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Following is a description of the flow of documents between buyer and supplier. This flow may be initiated by either a supplier or a buyer. Each time a trading partner receives a document from the other, it sends a MessageAcknowledgement document.

1. A supplier may solicit an order by sending an OrderRequest.
2. Buyer sends an Order, either in response to the supplier's OrderRequest or independently of any such request.
3. Seller sends an OrderResponse.
4. After sending the Order, buyer may send one or more ChangeOrder documents, changing the details of the order.
5. Supplier sends an OrderResponse for each ChangeOrder received.

The diagram below illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable and configure to process order management transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCOrderRequest30Outbound Service on page 29		●
XPCOrderRequest30Inbound Service on page 30	●	
XPCCorrelatedOrder30Outbound Service on page 31	●	
XPCOrder30Outbound Service on page 32	●	
XPCOrder30Inbound Service on page 32		●
XPCOrderResponseFromOrder30Outbound Service on page 33		●
XPCOrderResponse30Inbound Service on page 34	●	
XPCChangeOrder30Outbound Service on page 34	●	
XPCChangeOrder30Inbound Service on page 35		●
XPCOrderResponseFromChangeOrder30Outbound Service on page 36		●

Document Exchange Details

Following is a description of the flow of documents between buyer and supplier:

1. Supplier may solicit an order by sending an OrderRequest.
   - **e)** The supplier's back office system creates an OrderRequest and leaves it in a designated location on its local file system.
   - **f)** The supplier's **XPCOrderRequest30Outbound Service** service wakes up periodically and searches the specified directory for the OrderRequest document. It places the document in an xCBL envelope and transmits it to MarketSite.
   - **g)** MarketSite routes the OrderRequest document to the buyer.
2. Buyer sends an Order, either in response to the supplier's OrderRequest or

independently of any such request.

a) The buyer's **XPCOrderRequest30Inbound Service** service extracts the OrderRequest document from its envelope, builds a default Order document, and stores both the OrderRequest and Order documents in designated locations on its local file system.

b) The buyer's **XPCCorrelatedOrder30Outbound Service** service and **XPCOrder30Outbound Service** wake up periodically and search the appropriate directories for the Order. XPCCorrelatedOrder30Outbound searches for Orders created in response to an OrderRequest. XPCOrder30Outbound searches for Orders that were not initiated by an OrderRequest. If either service finds an unprocessed order, it creates an xCBL envelope for the document, archives the envelope to the local file system, and transmits it to MarketSite.

c) MarketSite routes the Order document to the supplier.

3. The supplier sends an OrderResponse.

a) The supplier's **XPCOrder30Inbound Service** service extracts the Order document from its envelope, sends a MessageAcknowledgement document to MarketSite, builds a default OrderResponse document. It stores both the Order envelope and document in a designated location on the local file system, using a prefix of OrderResponseFromOrder_. It stores the OrderResponse document in a designated locations on its local file system.

b) The supplier's **XPCOrderResponseFromOrder30Outbound Service** service wakes up periodically and searches the designated directory for the OrderResponse document with a prefix of OrderResponseFromOrder_. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the OrderResponse to the buyer.

d) The buyer's **XPCOrderResponse30Inbound Service** service extracts the OrderResponse document from its envelope, sends a MessageAcknowledgement, and stores the OrderResponse document and any attachments in a designated location on its local file system.

4. The buyer may send one or more ChangeOrder documents, changing the details of an Order.

a) The buyer's back office system prepares a ChangeOrder and leaves it in a designated location on its local file system.

b) The buyer's **XPCChangeOrder30Outbound Service** service wakes up periodically and searches the specified directory for the ChangeOrder. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the ChangeOrder to the supplier.

5. The supplier receives the ChangeOrder.
   a) The supplier's **XPCChangeOrder30Inbound Service** service extracts the ChangeOrder document from its envelope, sends a MessageAcknowledgement, and builds a default OrderResponse document. It stores the ChangeOrder envelope, document, and any attachments in a designated location on its local file system with a prefix of OrderResponseFromChangeOrder_. It stores the OrderResponse in another designated location on its local file system.
6. The supplier sends a new OrderResponse incorporating the changes.
   a) The supplier's back office system creates a customized OrderResponse and stores it in a designated location on its local file system.
   b) The supplier's **XPCOrderResponseFromChangeOrder30Outbound Service** service wakes up periodically and searches the specified location for the OrderResponse. It places the document in an xCBL envelope and transmits it to MarketSite.
   c) MarketSite routes the OrderResponse to the buyer.
   d) The buyer's**XPCOrderRequest30Inbound Service** service extracts the OrderResponse document from its envelope, sends a MessageAcknowledgement, and stores the OrderResponse document and any attachments in a designated location on its local file system.

XPCOrderRequest30Outbound Service

The supplier's XPCOrderRequest30Outbound service wakes up periodically and searches the specified directory for the OrderRequest document. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the OrderRequest document from the file system and archive it. Builds the document object from the document read.
GetStringFromDocument	Extracts a string from document object using xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.

Action List Component	Description
CreateEnvelope	Creates Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCOrderRequest30Inbound Service

The Buyer's OrderRequest30Inbound service extracts the OrderRequest document from its envelope, builds a default Order document, and stores both the OrderRequest and Order documents in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the OrderRequest document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope.
DefaultOrder30Builder	Builds a default Order document.
FileStore.storeEnvelope	Stores the OrderRequest Envelope, document and attachment on the file system. File names are formed by concatenating the appropriate prefix with the CorrelationKey.
FileStore.storeDocument	Stores the default Order document on the file system.

XPCCorrelatedOrder30Outbound Service

The buyer's XPCCorrelatedOrder30Outbound service wakes up periodically and searches the appropriate directory for Orders created in response to an OrderRequest. If it finds an unprocessed order, it creates an xCBL envelope for the document, archives the envelope to the local file system, and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the Order document from the file system, archives it and uses it to build the document object. Builds filename key string from the name of the file read. The filename key string must be same as the filename key of the OrderRequest envelope persisted by XPCOrderRequest30Inbound service
FileStore.lookupEnvelope	Looks up the OrderRequest envelope using the filename key.
CreateCorrelatingEnvelope	Creates the Order envelope using the header information from the OrderRequest envelope. Keeps the correlationId same. Swap the senderId and recipientId.
GetCorrelationKey	Builds filename key string to be used to store the envelope just created. Use the xPath configuration to extract the string from the Envelope's document (document that's read in from the file).
OrderStore.storeDocument	Stores the envelope on the file system for use by XPCChangeOrderOutbound service in creating ChangeOrder envelope.
Transmitter	Transmits the Order envelope to MarketSite.

XPCOrder30Outbound Service

The buyer's XPCOrderOutbound wakes up periodically and searches the appropriate directory for Orders that were not initiated by an OrderRequest. If it finds an unprocessed order, it creates an xCBL envelope for the document, archives the envelope to the local file system, and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the Order document from file system and archive it. Builds the document object from the document read.
GetStringFromDocument	Extracts a string from document object by applying xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
GetCorrelationKey	Builds filename key string to be used to store the envelope just created. Use the xPath configuration to extract the string from the Envelope's document (document that's read in from the file).
OrderStore.storeDocument	Stores the envelope to the file system. This envelope is used when creating ChangeOrder envelope in XPCChangeOrder30Outbound services.
Transmitter	Transmits the Order envelope to MarketSite.

XPCOrder30Inbound Service

The supplier's Order30Inbound service extracts the Order document from its envelope, sends a MessageAcknowledgement document to MarketSite, builds a default OrderResponse document. It stores both the Order envelope and document in

a designated location on the local file system, using a prefix of OrderResponseFromOrder_. It stores the OrderResponse document in a designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the Order document and sends a MessageAcknowledgement
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the Order envelope, document, and attachment and the default OrderResponse document.
DefaultOrderResponse30Builder	Builds a default OrderResponse document.
FileStore.storeEnvelope	Stores the Order Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default OrderResponse document on the file system.

XPCOrderResponseFromOrder30Outbound Service

The supplier's OrderResponseFromOrder30Outbound service wakes up periodically and searches the designated directory for the OrderResponse document with a prefix of OrderResponseFromOrder_. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads OrderResponse document from file system and archive it. This document is supposed to be an OrderResponse to an Order document. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Transmits the response document to MarketSite.

XPCOrderResponse30Inbound Service

The buyer's XPCOrderResponse30Inbound service extracts the OrderResponse document from its envelope, sends a MessageAcknowledgement, and stores the OrderResponse document and any attachments in a designated location on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the OrderResponse document and send MessageAcknowledgement
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the OrderResponse envelope, document, and attachment.
FileStore.storeEnvelope	Stores the OrderResponse Envelope, document and attachment on the file system.

XPCChangeOrder30Outbound Service

The buyer's XPCChangeOrder30Outbound service wakes up periodically and searches the specified directory for the ChangeOrder. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the ChangeOrder document from file system and archive it. Builds the document object from the document read.
GetCorrelationKeyFromDocument	Extracts a string from document object using xPath configuration. This string must match the filename key used when the outgoing Order envelope is saved in XPCOrder30Outbound service.
FileStore.lookupEnvelope	Looks up the envelope that matches the extracted string. This is the Order envelope saved.

Action List Component	Description
CreateCorrelatingEnvelope	Creates the ChangeOrder envelope using the header information of the Order envelope retrieved. Keep the correlationId, senderId and recipientId same.
Transmitter	Transmits the envelope to MarketSite.

XPCChangeOrder30Inbound Service

The supplier's XPCChangeOrder30Inbound service extracts the ChangeOrder document from its envelope, sends a MessageAcknowledgement, and builds a default OrderResponse document. It stores the ChangeOrder envelope, document, and any attachments in a designated location on its local file system with a prefix of OrderResponseFromChangeOrder_. It stores the OrderResponse in another designated location on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the ChangeOrder document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the ChangeOrder envelope, document, and attachment and the default OrderResponse document.
DefaultOrderResponse30Builder	Builds a default OrderResponse document
FileStore.storeEnvelope	Stores the ChangeOrder Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default OrderResponse document on the file system.

XPCOrderResponseFromChangeOrder30Outbound Service

The supplier's XPCOrderResponseFromChangeOrder30Outbound service wakes up periodically and searches the specified location for the OrderResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the OrderResponse document from the file system and archives it. This document is supposed to be an OrderResponse to an ChangeOrder document. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the OrderResponse document to MarketSite.

Order Status Request Services

This section describes the XPC services used to process order status request transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Price check transactions are synchronous document exchanges. The XPC connection used to receive the request document remains open until the response document is returned.

Following is an overview of the document exchange:

1. A buyer sends a PriceCheckRequestDocument to determine the price at which a supplier will sell a particular quantity of goods.
2. The supplier returns a PriceCheckResult document indicating the available quantity of the product.

The following diagram illustrates the flow of xCBL documents:

Associated XPC Services

The following table lists the XPC services that buyers must enable and configure to process order management transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCOrderStatusRequest30Inbound Service on page 38	●	

Document Exchange Details

Details of the document exchange are provided below:

1. The buyer's back office system creates an OrderStatusRequest document and transmits it to MarketSite.
2. MarketSite forwards the OrderStatusRequest to the supplier.
3. The supplier's OrderStatusRequest30Inbound service receives the OrderStatusRequest document, builds a default OrderStatusResult document based upon the request, updates the default OrderStatusResult document with business data from the back office system, and sends the customized OrderStatusResult document to MarketSite.
4. MarketSite forwards the OrderStatusResult document to the buyer.

XPCOrderStatusRequest30Inbound Service

Action List Component	Description
DefaultOrderStatusResponse30Builder	Receives the OrderStatusRequest document and builds default OrderStatusResult document
Customizer	Customizes and updates the default OrderStatusResult document
Responder	Sends the customized OrderStatusResult document to MarketSite.

Payment Request Services

This section describes the XPC services used to process payment request transactions. It includes the following information:

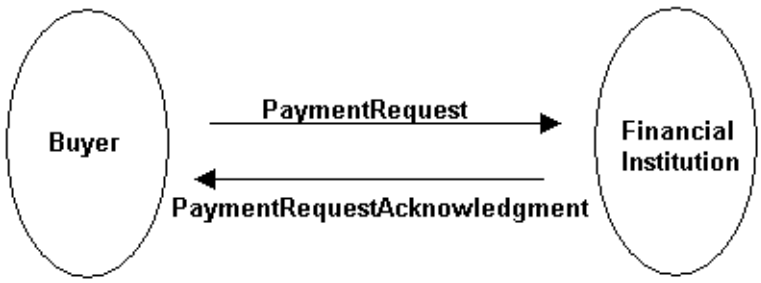
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Following is a description of the flow of financial management documents between buyer and supplier:

- The buyer sends his financial institution a PaymentRequest document to initiate payment to the supplier. The request can reference multiple orders, advance shipment notices, or invoices.
- The financial institution sends the buyer a PaymentRequestAcknowledgment.

The following diagram illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable to process payment request transactions:

XPC Service	Used By Buyer	Used By Financial Institution
XPCPaymentRequest30Outbound Service on page 41		●
XPCPaymentRequest30Inbound Service on page 41	●	
XPCPaymentRequestAcknowledgement30Outbound Service on page 42	●	
XPCPaymentRequestAcknowledgment30Inbound Service on page 42		●

Document Exchange Details

1. The buyer's back office system prepares a PaymentRequest document and leaves it in a designated place on the local file system.
2. Buyer sends his financial institution a PaymentRequest document to initiate payment to the supplier. The request can reference multiple orders, advance shipment notices, or invoices.
3. The financial institution sends the buyer a PaymentRequestAcknowledgment.

XPCPaymentRequest30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the PaymentRequest document from the file system and archives it. Builds the document object from this document.
GetStringFromDocument	Gets the key string specified in the configuration
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header
Transmitter	Transmits the Envelope to MarketSite.

XPCPaymentRequest30Inbound Service

Action List Component	Description
MessageAcknowledgementSender	Receives the PaymentRequest document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the PaymentRequest envelope, document, and attachment and the default PaymentRequestAcknowledgement document.
DefaultPaymentRequestAck30Builder	Builds the default PaymentRequestAcknowledgement document.
FileStore.storeEnvelope	Stores the PaymentRequest Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default PaymentRequestAcknowledgement document on the file system.

XPCPaymentRequestAcknowledgement30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the PaymentRequestAcknowledgement document from the file system and archive it. Builds the document object from the document read. Builds the filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the PaymentRequestAcknowledgement document to MarketSite.

XPCPaymentRequestAcknowledgment30Inbound Service

Action List Component	Description
MessageAcknowledgementSender	Receives the PaymentRequestAcknowledgment document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the PaymentRequestAcknowledgement envelope, document, and attachment.
FileStore.storeEnvelope	Stores the PaymentRequestAcknowledgment Envelope, document and attachment on the file system.

Planning and Shipping Schedule Services

This section describes the XPC services used to process planning and shipping schedule transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

The typical scenario is as follows:

- At fixed intervals, a buyer sends a supplier a PlanningSchedule that forecasts product requirements over a long period of time.
- The supplier sends a PlanningScheduleResponse.
- At more frequent intervals, the buyer sends the supplier a ShippingSchedule, associated with a particular PlanningSchedule, to communicate precise short-term product delivery requirements.
- The supplier sends a ShippingScheduleResponse.

The diagram below illustrates the exchange:

Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable and configure to process planning and shipping schedule transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCPlanningSchedule30Outbound Service on page 46	●	
XPCPlanningSchedule30Inbound Service on page 47		●
XPCPlanningScheduleResponse30Outbound Service on page 48		●
XPCPlanningScheduleResponse30Inbound Service on page 48	●	
XPCShippingSchedule30Outbound Service on page 49	●	
XPCShippingSchedule30Inbound Service on page 50		●
XPCShippingScheduleResponse30Outbound Service on page 51		●
XPCShippingScheduleResponse30Inbound Service on page 51	●	

Document Exchange Details

Following are the detailed steps involved in the exchange:

1. Buyer's back office system creates the PlanningSchedule and leaves it in a designated location on its local file system.
2. Buyer's XPCPlanningSchedule30Outbound service wakes up periodically and searches the specified directory for the PlanningSchedule. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the PlanningSchedule to the supplier.
4. The supplier's XPCPlanningSchedule30Inbound service extracts the PlanningSchedule from its envelope, builds a default PlanningScheduleResponse, and stores both the PlanningSchedule and the PlanningScheduleResponse in designated locations on its local file system.
5. The supplier's XPCPlanningScheduleResponse30Outbound service wakes up periodically and searches the specified directory for the

PlanningScheduleResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

6. MarketSite routes the PlanningScheduleResponse to the buyer.
7. The buyer's XPCPlanningSchedule30ResponseInbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the PlanningScheduleResponse and its envelope in a designated location on the local file system.
8. The buyer's back office system prepares a ShippingSchedule associated with a particular PlanningSchedule and leaves it in a designated location on the local file system.
9. Buyer's XPCShippingSchedule30Outbound service wakes up periodically and searches the specified directory for the ShippingSchedule. It places the ShippingSchedule in an xCBL envelope, correlated with the envelope of the associated PlanningSchedule, and transmits it to MarketSite.
10. MarketSite routes the ShippingSchedule to the supplier.
11. The supplier's XPCShippingSchedule30Inbound service extracts the ShippingSchedule from its envelope, builds a default ShippingScheduleResponse, and stores both the ShippingSchedule and the ShippingScheduleResponse in designated locations on its local file system.
12. The supplier's XPCShippingScheduleResponse30Outbound service wakes up periodically and searches the specified directory for the ShippingScheduleResponse. It places the document in an xCBL envelope and transmits it to MarketSite.
13. MarketSite routes the ShippingScheduleResponse to the buyer.
14. The buyer's XPCShippingSchedule30ResponseInbound service extracts the response document from its envelope, sends a message acknowledgement, and stores the ShippingScheduleResponse and its envelope in a designated location on the local file system.

XPCPlanningSchedule30Outbound Service

Buyer's XPCPlanningSchedule30Outbound service wakes up periodically and searches the specified directory for the PlanningSchedule. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the PlanningSchedule document from the file system and archives it. Builds the document object from the document read.
GetStringFromDocument	Extracts a string from document object using xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
GetCorrelationKey	Uses xPath configuration to extract the correlation key from the PlanningSchedule document.
PlanningScheduleStore.storeDocument	Stores the Envelope on the file system for later use in creating Envelopes for ShippingSchedule documents associated with this PlanningSchedule. The name of the file is specified by the correlation key.
Transmitter	Transmits the Planning Schedule Envelope to MarketSite.

XPCPlanningSchedule30Inbound Service

The supplier's XPCPlanningSchedule30Inbound service extracts the PlanningSchedule from its envelope, builds a default PlanningScheduleResponse, and stores both the PlanningSchedule and the PlanningScheduleResponse in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the PlanningSchedule document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the PlanningSchedule envelope, document, and attachment and the default PlanningScheduleResponse document.
DefaultPlanningScheduleResponse30 Builder	Builds a default PlanningScheduleResponse document.
FileStore.storeEnvelope	Stores the PlanningSchedule Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default PlanningScheduleResponse document on the file system.

XPCPlanningScheduleResponse30Outbound Service

The supplier's XPCPlanningScheduleResponse30Outbound service wakes up periodically and searches the specified directory for the PlanningScheduleResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the PlanningScheduleResponse document from the file system and archive it. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the PlanningScheduleResponse document to MarketSite.

XPCPlanningScheduleResponse30Inbound Service

The buyer's XPCPlanningSchedule30ResponseInbound service extracts the response document from its envelope, sends a message acknowledgement, and stores the PlanningScheduleResponse and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the PlanningScheduleResponse document and send MessageAcknowledgement
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the PlanningScheduleResponse envelope, document, and attachment.
FileStore.storeEnvelope	Stores the PlanningScheduleResponse Envelope, document and attachment on the file system.

XPCShippingSchedule30Outbound Service

Buyer's XPCShippingSchedule30Outbound service wakes up periodically and searches the specified directory for the ShippingSchedule. It places the ShippingSchedule in an xCBL envelope, correlated with the envelope of the associated PlanningSchedule, and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the ShippingSchedule document from the file system and archives it. Builds the document object from the document read.
GetCorrelationKeyFromDocument	Uses xPath configuration to extract a key string from the file. The key string must match the filename key that was used by the XPCPlanningSchedule30Outbound service to persist the associated PlanningSchedule envelope.
FileStore.lookupEnvelope	Looks up the PlanningSchedule envelope that matches the extracted key string.
CreateCorrelatingEnvelope	Creates the ShippingSchedule envelope using the correlationId, senderId, and recipientId information from the PlanningSchedule envelope.
Transmitter	Transmits the ShippingSchedule envelope to MarketSite.

XPCShippingSchedule30Inbound Service

The supplier's XPCShippingSchedule30Inbound service extracts the ShippingSchedule from its envelope, builds a default ShippingScheduleResponse, and stores both the ShippingSchedule and the ShippingScheduleResponse in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the ShippingSchedule document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the ShippingSchedule envelope, document, and attachment and the default ShippingScheduleResponse document.
DefaultShippingScheduleResponse30 Builder	Builds a default ShippingScheduleResponse document.
FileStore.storeEnvelope	Stores the ShippingSchedule Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default ShippingScheduleResponse document

XPCShippingScheduleResponse30Outbound Service

The supplier's XPCShippingScheduleResponse30Outbound service wakes up periodically and searches the specified directory for the ShippingScheduleResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the ShippingScheduleResponse document from the file system and archive it. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the ShippingScheduleResponse document to MarketSite.

XPCShippingScheduleResponse30Inbound Service

The buyer's XPCShippingSchedule30ResponseInbound service extracts the response document from its envelope, sends a message acknowledgement, and stores the ShippingScheduleResponse and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the ShippingScheduleResponse document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the ShippingScheduleResponse envelope, document, and attachment.
FileStore.storeEnvelope	Stores the ShippingScheduleResponse Envelope, document and attachment on the file system.

Price Check Services

This section describes the XPC services used to process price check transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

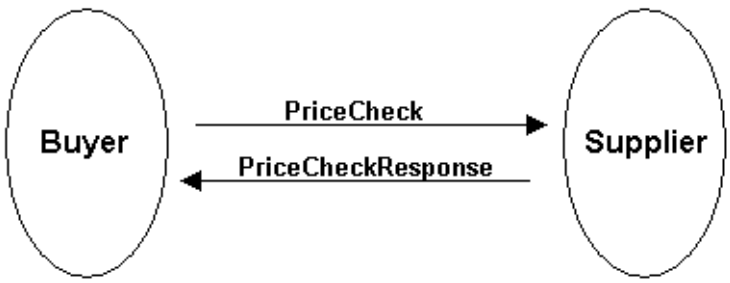
Overview of Transactions

Price check transactions are synchronous document exchanges. The XPC connection used to receive the request document remains open until the response document is returned.

Following is an overview of the document exchange:

1. A buyer sends a PriceCheckRequestDocument to determine the price at which a supplier will sell a particular quantity of goods.
2. The supplier returns a PriceCheckResult document indicating the available quantity of the product.

The following diagram illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that suppliers must enable to process price check transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCPriceCheckRequest30Inbound Service on page 53		●

Document Exchange Details

Details of the document exchange are provided below:

1. The buyer's back office system creates a PriceCheckRequest document and transmits it to MarketSite.
2. MarketSite forwards the PriceCheckRequest to the supplier.
3. The supplier's XPCPriceCheckRequest30Inbound service receives the PriceCheckRequest document, builds a default PriceCheckResult document based upon the request, updates the default PriceCheckResult document with business data from the back office system, and sends the customized PriceCheckResult document to MarketSite.
4. MarketSite forwards the PriceCheckResult document to the buyer.

XPCPriceCheckRequest30Inbound Service

Action List Component	Description
DefaultPriceCheckResponse30Builder	Receives the PriceCheckRequest document and builds a default PriceCheckResult document.
Customizer	Customizes and updates the default PriceCheckResult document.
Responder	Sends the customized PriceCheckResult document to MarketSite.

Product Catalog Services

This section describes the XPC services used to process product catalog transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

ProductCatalog transactions are one-way document exchanges. Suppliers send a ProductCatalog to a buyer to communicate product offerings and their prices. The supplier expects to receive a MessageAcknowledgment indicating that the ProductCatalog was received but do not expect a response document.

The following diagram illustrates the flow of xCBL documents:

Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable and configure to process product catalog transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCProductCatalog30Outbound Service on page 55		●
XPCProductCatalog30Inbound Service on page 56	●	

Document Exchange Details

Following are the detailed steps involved in the exchange:

1. Supplier's back office system creates the ProductCatalog and leaves it in a designated location on its local file system.
2. Supplier's XPCProductCatalog30Outbound service wakes up periodically and searches the specified directory for the ProductCatalog. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the ProductCatalog to the Buyer.
4. The Buyer's XPCProductCatalog30Inbound service extracts the ProductCatalog from its envelope, sends a MessageAcknowledgement document, and stores both documents in designated locations on its local file system.

XPCProductCatalog30Outbound Service

Supplier's XPCProductCatalog30Outbound service wakes up periodically and searches the specified directory for the ProductCatalog. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the ProductCatalog document from the file system and archives it. Builds the document object from the document read.
GetStringFromDocument	Gets the key string specified in the configuration.

Action List Component	Description
StringMapper	Gets the TPID matching the key string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCProductCatalog30Inbound Service

The Buyer's XPCProductCatalog30Inbound service extracts the ProductCatalog from its envelope, sends a MessageAcknowledgement document, and stores both documents in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the ProductCatalog document and sends a MessageAcknowledgement.
GetCorrelationKey	Builds a CorrelationKey to be used for file name when persisting. MessageId of the envelope is used
FileStore.storeEnvelope	Stores the ProductCatalog Envelope, document and attachment on the file system.

Quote Services

This section describes the XPC services used to process quote transactions. It includes the following information:

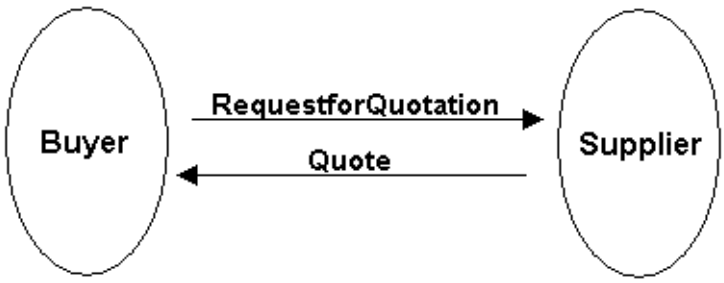
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Quote transactions are peer-to-peer transactions. Following is an overview of quote transactions:

1. A buyer sends a supplier a RequestForQuotation document requesting the price of goods or services.
2. The supplier responds with a Quote document indicating the price of goods or services.

The following diagram illustrates the flow of xCBL documents:



Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable and configure to process quote transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCRequestForQuotation30Outbound Service on page 59	●	
XPCRequestForQuotation30Inbound Service on page 59		●
XPCQuote30Outbound Service on page 60		●
XPCQuote30Inbound Service on page 61	●	

Document Exchange Details

Following are the detailed steps involved in the exchange:

1. Buyer's back office system creates the RequestForQuotation document and leaves it in a designated location on its local file system.
2. Buyer's XPCRequestForQuotation30Outbound service wakes up periodically and searches the specified directory for the RequestForQuotation document. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the RequestForQuotation document to the Supplier.
4. Supplier's XPCRequestForQuotation30Inbound service extracts the RequestForQuotation document from its envelope, sends a MessageAcknowledgement document to the buyer, builds a default Quote document, and stores both the RequestForQuotation document and the Quote document in designated locations on its local file system.
5. Supplier's XPCQuote30Outbound service wakes up periodically and searches the specified directory for the Quote document. It places the document in an xCBL envelope and transmits it to MarketSite.
6. MarketSite routes the Quote document to Buyer.
7. Buyer's XPCQuote30Inbound service extracts the Quote document from its envelope, sends a MessageAcknowledgement document to the supplier, and stores the Quote document and its envelope in a designated location on the local file system.

XPCRequestForQuotation30Outbound Service

Buyer's XPCRequestForQuotation30Outbound service wakes up periodically and searches the specified directory for the RequestForQuotation document. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the RequestForQuotation document from the file system and archive it. Builds the document object from the document read.
GetStringFromDocument	Extracts a string from document object using xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCRequestForQuotation30Inbound Service

Supplier's XPCRequestForQuotation30Inbound service extracts the RequestForQuotation document from its envelope, builds a default ResponseDocument, and stores both the RequestDocument and the ResponseDocument in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the RequestForQuotation document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the RequestForQuotation envelope, document, and attachment and the default Quote document.

Action List Component	Description
DefaultRequestForQuotationResponse30Builder	Builds a default Quote document.
FileStore.storeEnvelope	Stores the RequestForQuotation Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default Quote document on the file system.

XPCQuote30Outbound Service

Supplier's XPCQuote30Outbound service wakes up periodically and searches the specified directory for the Quote document. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the Quote document from file system and archive it. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the Quote document to MarketSite.

XPCQuote30Inbound Service

Buyer's XPCQuote30Inbound service extracts the Quote document from its envelope, sends a message acknowledgement, and stores the Quote document and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgmentSender	Receives the Quote document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the Quote envelope, document, and attachment.
FileStore.storeEnvelope	Stores the Quote Envelope, document and attachment on the file system.

Remittance Advice Services

This section describes the XPC services used to process remittance advice transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Remittance advice transactions are one-way document exchanges.

Following is an overview of remittance advice transactions:

1. A buyer sends a RemittanceAdvice to a supplier to indicate that payment was initiated.
2. The supplier sends a MessageAcknowledgement to the buyer indicating that the RemittanceAdvice was received.

Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable and configure to process remittance advice transaction:

XPC Service	Used By Buyer	Used By Supplier
XPCRemittanceAdvice30Outbound Service on page 64	●	
XPCRemittanceAdvice30Inbound Service on page 64		●

Document Exchange Details

Following are the detailed steps involved in the exchange:

1. Buyer's back office system creates the RemittanceAdvice and leaves it in a designated location on its local file system.
2. Buyer's XPCRemittanceAdvice30Outbound service wakes up periodically and searches the specified directory for the RemittanceAdvice. It places the document in an xCBL envelope and transmits it to MarketSite.
3. MarketSite routes the RemittanceAdvice to the Supplier.
4. The Supplier's XPCRemittanceAdvice30Inbound service extracts the RemittanceAdvice from its envelope, sends a MessageAcknowledgement document, and stores both documents in designated locations on its local file system.

XPCRemittanceAdvice30Outbound Service

Buyer's XPCRemittanceAdvice30Outbound service wakes up periodically and searches the specified directory for the RemittanceAdvice. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the RemittanceAdvice document from the file system and archives it. Builds the document object from the document read.
GetStringFromDocument	Extracts a string from document object using xPath configuration.
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the envelope to MarketSite.

XPCRemittanceAdvice30Inbound Service

The Supplier's XPCRemittanceAdvice30Inbound service extracts the RemittanceAdvice from its envelope, sends a MessageAcknowledgement document, and stores both documents in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the RemittanceAdvice document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the RemittanceAdvice envelope, document, and attachment.
FileStore.storeEnvelope	Stores the RemittanceAdvice Envelope, document and attachment on the file system.

Time Series Services

This section describes the XPC services used to process time series transactions. It includes the following information:

- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

Following is an overview of a time series transaction:

1. The supplier may request time series data by sending the buyer a TimeSeriesRequest.
2. The buyer sends a TimeSeries document either in response to the supplier's TimeSeriesRequest or independently of any such request.
3. The supplier sends the buyer a TimeSeriesResponse document identifying any errors in the TradingPartner document.

The following diagram illustrates the flow of xCBL documents:

Associated XPC Services

The following table lists the XPC services that buyers and suppliers must enable to process time series transactions:

XPC Service	Used By Buyer	Used By Supplier
XPCTimeSeriesRequest30Outbound Service on page 67		●
XPCTimeSeriesRequest30Inbound Service on page 68	●	
XPCCorrelatedTimeSeries30Outbound Service on page 69	●	
XPCTimeSeries30Outbound Service on page 69	●	
XPCTimeSeries30Inbound Service on page 70		●
XPCTimeSeriesResponse30Outbound Service on page 71		●
XPCTimeSeriesResponse30Inbound Service on page 71	●	

Document Exchange Details

1. The supplier may request time series data by sending the buyer a TimeSeriesRequest.
   - **a)** Supplier's back office system creates the TimeSeriesRequest and leaves it in a designated location on its local file system.
   - **b)** Supplier's XPCTimeSeriesRequest30Outbound service wakes up periodically and searches the specified directory for the TimeSeriesRequest. It places the document in an xCBL envelope and transmits it to MarketSite.
   - **c)** MarketSite routes the TimeSeriesRequest to the buyer.
2. The buyer sends a TimeSeries document either in response to the supplier's TimeSeriesRequest or independently of any such request.
   - **a)** The buyer's XPCTimeSeriesRequest30Inbound service extracts the TimeSeriesRequest from its envelope, builds a default TimeSeries document, and stores both the TimeSeriesRequest and the TimeSeries in designated locations on its local file system.
   - **b)** The buyer's XPCTimeSeries30Outbound and XPCCorrelatedTimeSeries30Outbound services wake up periodically.

XPCCorrelatedTimeSeries30Outbound searches the appropriate directory for the TimeSeries document created in response to a TimeSeriesRequest.

XPCTimeSeries30Outbound searches the appropriate directory for the TimeSeriesRequest.

If either service finds a TimeSeries document, it creates and xCBL envelope, archives the envelope to the local file system, and transmits the TimeSeries document to MarketSite.

c) MarketSite routes the TimeSeries document to the supplier.

3. The supplier sends the buyer a TimeSeriesResponse document identifying any errors in the TradingPartner document.

a) The supplier's XPCTimeSeries30Inbound service extracts the TradingPartnerResponse document from its envelope, sends a MessageAcknowledgement document to MarketSite, and builds a default TimeSeriesResponse document. It stores both the TimeSeries envelope and document in a designated location on the local file system. It stores the TimeSeriesResponse document in a designated location on its local file system.

b) The supplier's XPCTimeSeriesResponse30Outbound service wakes up periodically and searches the specified directory for the TimeSeriesResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the TimeSeriesResponse to the buyer.

d) The buyer's XPCTimeSeriesResponse30Inbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the TimeSeriesResponse and its envelope in a designated location on the local file system.

XPCTimeSeriesRequest30Outbound Service

The supplier's XPCTimeSeriesRequest30Outbound service wakes up periodically and searches the specified directory for the TimeSeriesRequest. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the TimeSeriesRequest document from the file system and archives it. Builds the document object from the document read.
getStringFromDocument	Extracts a string from document object using xPath configuration.

Action List Component	Description
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCTimeSeriesRequest30Inbound Service

The buyer's XPCTimeSeriesRequest30Inbound service extracts the TimeSeriesRequest from its envelope, builds a default TimeSeries document, and stores both the TimeSeriesRequest and the TimeSeries in designated locations on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the TimeSeriesRequest document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the TimeSeriesRequest envelope, document, and attachment and the default TimeSeries document.
DefaultTimeSeriesBuilder	Builds a default TimeSeries document.
FileStore.storeEnvelope	Stores the TimeSeriesRequest Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default TimeSeries document on the file system.

XPCCorrelatedTimeSeries30Outbound Service

The buyer's XPCCorrelatedTimeSeries30Outbound service wakes up periodically and searches the appropriate directory for the TimeSeries document created in response to a TimeSeriesRequest. If it finds a TimeSeries document, it creates an xCBL envelope, archives the envelope to the local file system, and transmits the TimeSeries document to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the TimeSeries document from file system and archive it. Builds the document object from the document read. Builds filename key string from the name of the file read. The filename key string must be same as the filename key of the TimeSeriesRequest envelope persisted by XPCTimeSeriesRequest30Inbound service.
FileStore.lookupEnvelope	Looks up the TimeSeriesRequest envelope using the filename key.
CreateCorrelatingEnvelope	Creates the TimeSeries envelope based on the header information of the TimeSeriesRequest envelope. Keep the correlationId same. Swap the senderId and recipientId.
Transmitter	Transmits the TimeSeries envelope to MarketSite.

XPCTimeSeries30Outbound Service

The buyer's XPCTimeSeries30Outbound service wakes up periodically and searches the appropriate directory for the TimeSeries document created independently of a TimeSeriesRequest. It creates an xCBL envelope, archives the envelope to the local file system, and transmits the TimeSeries document to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the TimeSeries document from file system and archive it. Builds the document object from the document read.
GetStringFromDocument	Extracts a string from document object using xPath configuration.

Action List Component	Description
StringMapper	Gets the TPID matching the extracted string from the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Order envelope to MarketSite.

XPCTimeSeries30Inbound Service

The supplier's XPCTimeSeries30Inbound service extracts the TradingPartnerResponse document from its envelope, sends a MessageAcknowledgement document to MarketSite, and builds a default TimeSeriesResponse document. It stores both the TimeSeries envelope and document in a designated location on the local file system. It stores the TimeSeriesResponse document in a designated location on its local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the TimeSeries document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the TimeSeries envelope, document, and attachment and the default TimeSeriesResponse document.
DefaultTimeSeriesResponse30Builder	Builds a default TimeSeriesResponse document.
FileStore.storeEnvelope	Stores the TimeSeries Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default TimeSeriesResponse document on the file system.

XPCTimeSeriesResponse30Outbound Service

The supplier's TradingPartner1's XPCTimeSeriesResponse30Outbound service wakes up periodically and searches the specified directory for the TimeSeriesResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

Action List Component	Description
FileStore.readDocument	Reads the TimeSeriesResponse document from the file system and archives it. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the response document to MarketSite.

XPCTimeSeriesResponse30Inbound Service

The supplier's XPCTimeSeriesResponse30Inbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the TimeSeriesResponse and its envelope in a designated location on the local file system.

Action List Component	Description
MessageAcknowledgementSender	Receives the TimeSeriesResponse document and sends a MessageAcknowledgement.
GetCorrelationKey	Builds a CorrelationKey to be used for file name when persisting. MessageId of the envelope is used
FileStore.storeEnvelope	Stores the TimeSeriesResponse Envelope, document and attachment on the file system.

Trading Partner Management Services

This section describes the XPC services used to process trading partner management transactions. It includes the following information:

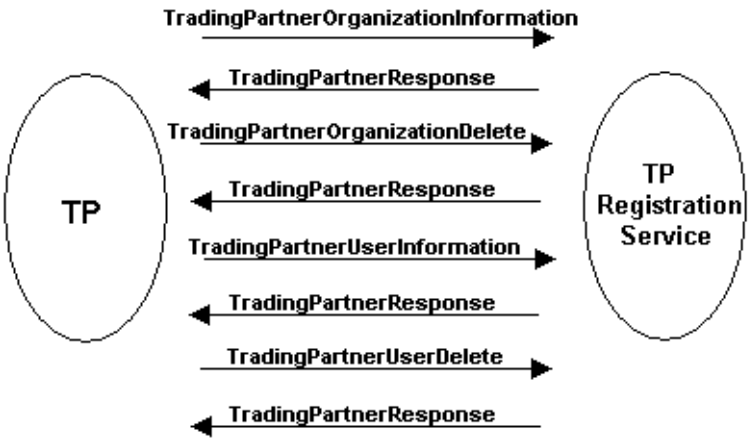
- An overview of the transaction
- A list of XPC services to enable and configure
- A detailed description of the document exchange
- A description of each service's default Action Director

Overview of Transactions

This section describes the document exchanges involved in the following transactions:

- Registering a trading partner
- Deleting a trading partner
- Registering a trading partner user
- Deleting a trading partner user

The following diagram illustrates the exchange of xCBL documents:



Associated XPC Services

The following table lists the XPC services that trading partners and registration services must enable to process trading partner management transaction:

XPC Service	Used By Trading Partner	Used By Registration Service
XPCTradingPartnerOrganizationInformation30Outbound Service on page 81	●	
XPCTradingPartnerOrganizationInformation30Inbound Service on page 78		●
XPCTradingPartnerResponseFromTPOrganizationInfo30Outbound Service on page 82		●
XPCTradingPartnerResponse30Inbound Service on page 80	●	
XPCTradingPartnerOrganizationDelete30Outbound Service on page 82	●	
XPCTradingPartnerOrganizationDelete30Inbound Service on page 79		●
XPCTradingPartnerResponseFromTPOrganizationDelete30Outbound Service on page 83		●
XPCTradingPartnerUserInformation30Outbound Service on page 80	●	
XPCTradingPartnerUserInformation30Inbound Service on page 77		●
XPCTradingPartnerResponseFromTPUserInfo30Outbound Service on page 83		●
XPCTradingPartnerUserDelete30Outbound Service on page 81	●	
XPCTradingPartnerUserDelete30Inbound Service on page 78		●
XPCTradingPartnerResponseFromTPUserDelete30Outbound Service on page 84		●

Document Exchange Details: Registering a Trading Partner

1. A trading partner registers with the registration service.
   a) Organization1's back office system creates the TradingPartnerOrganizationInformation document and leaves it in a designated location on its local file system.
   b) Organization1's XPCTradingPartnerOrganizationInformation30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerOrganizationInformation document. It places the document in an xCBL envelope and transmits it to MarketSite.
   c) MarketSite routes the RequestDocument to the registration service.
2. The registration service sends a TradingPartnerResponse document.
   a) The registration service's XPCTradingPartnerOrganizationInformation30Inbound service extracts the TradingPartnerOrganizationInformation document from its envelope, builds a default TradingPartnerResponse, and stores both the TradingPartnerOrganizationInformation document and the TradingPartnerResponse document in designated locations on its local file system. The TradingPartnerResponse is stored in a file with the prefix TradingPartnerResponseFromTPOrganizationInfo_.
   a) The registration service's XPCTradingPartnerResponseFromTPOrganizationInfo30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerResponse. It places the document in an xCBL envelope and transmits it to MarketSite.
   b) MarketSite routes the TradingPartnerResponse to TradingPartner1.
   c) TradingPartner1's XPCTradingPartnerResponse30Inbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the TradingPartnerResponse and its envelope in a designated location on the local file system.

Document Exchange Details: Deleting a Trading Partner

1. A trading partner notifies the registration service to remove it from the trading partner registry.
   a) The trading partner's back office system creates the TradingPartnerOrganizationDelete document and leaves it in a designated location on its local file system.
   b) The trading partner's XPCTradingPartnerOrganizationDelete30Outbound

service wakes up periodically and searches the specified directory for the TradingPartnerOrganizationDelete document. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the TradingPartnerOrganizationDelete to the registration service.

2. The registration service sends a TradingPartnerResponse document.

a) The registration service's XPCTradingPartnerOrganizationDelete30Inbound service extracts the TradingPartnerOrganizationDelete document from its envelope, builds a default TradingPartnerResponse, and stores both the TradingPartnerOrganizationDelete document and the TradingPartnerResponse document in designated locations on its local file system. The TradingPartnerResponse is stored in a file with the prefix TradingPartnerResponseFromTPOrganizationDelete_.

a) The registration service's XPCTradingPartnerResponseFromTPOrganizationDelete30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

b) MarketSite routes the TradingPartnerResponse to the trading partner.

c) The trading partner's XPCTradingPartnerResponse30Inbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the TradingPartnerResponse and its envelope in a designated location on the local file system.

Document Exchange Details: Registering a Trading Partner User

1. A trading partner registers one of its users with the registration service.

a) The trading partner's back office system creates the TradingPartnerUserInformation document and leaves it in a designated location on its local file system.

b) The trading partner's XPCTradingPartnerUserInformation30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerUserInformation document. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the TradingPartnerUserInformation document to the registration service.

2. The registration service sends a TradingPartnerResponse document.

a) The registration service's XPCTradingPartnerUserInformation30Inbound

service extracts the TradingPartnerUserInformation document from its envelope, builds a default TradingPartnerResponse, and stores both the TradingPartnerUserInformation document and the TradingPartnerResponse document in designated locations on its local file system. The TradingPartnerResponse is stored in a file with the prefix TradingPartnerResponseFromTPUserInfo_.

a) The registration service's XPCTradingPartnerResponseFromTPUserInfo30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

b) MarketSite routes the TradingPartnerResponse to the trading partner.

c) The trading partner's XPCTradingPartnerResponse30Inbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the TradingPartnerResponse and its envelope in a designated location on the local file system

Document Exchange Details: Deleting a Trading Partner User

1. A trading partner notifies the registration service that a user has been deleted.

a) The trading partner's back office system creates the TradingPartnerUserDelete document and leaves it in a designated location on its local file system.

b) Organization1's XPCTradingPartnerUserDelete30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerUserDelete document. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the TradingPartnerUserDelete to the registration service.

2. The registration service sends a TradingPartnerResponse document.

a) The registration service's XPCTradingPartnerUserDelete30Inbound service extracts the TradingPartnerUserDelete document from its envelope, builds a default TradingPartnerResponse, and stores both the TradingPartnerUserDelete document and the TradingPartnerResponse document in designated locations on its local file system. The TradingPartnerResponse is stored in a file with the prefix TradingPartnerResponseFromTUserDelete_.

b) The registration service's XPCTradingPartnerResponseFromTPUserDelete30Outbound service wakes up periodically and searches the specified directory for the TradingPartnerResponse. It places the document in an xCBL envelope and transmits it to MarketSite.

c) MarketSite routes the TradingPartnerResponse to the trading partner.

d) The trading partner's XPCTradingPartnerResponse30Inbound service extracts the response document from its envelope, sends a MessageAcknowledgement, and stores the TradingPartnerResponse and its envelope in a designated location on the local file system.

XPCTradingPartnerUserInformation30Inbound Service

Action List Component	Description
MessageAcknowledgementSender	Receives the TradingPartnerUserInformation document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the TradingPartnerUserInformation envelope, document, and attachment and the default TradingPartnerResponse document.
DefaultTPResponseFromUserInfo30Builder	Builds a default TradingPartnerResponse document
FileStore.storeEnvelope	Stores the TradingPartnerUserInformation Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default TradingPartnerResponse document in a file whose name begins with the prefix TradingPartnerResponseFromTPUserInfo_.

XPCTradingPartnerUserDelete30Inbound Service

Action List Component	Description
MessageAcknowledgmentSender	Receives the TradingPartnerUserDelete document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the TradingPartnerUserDelete envelope, document, and attachment and the default TradingPartnerResponse document.
DefaultTPResponseFromUserDelete30 Builder	Builds a default TradingPartnerResponse document.
FileStore.storeEnvelope	Stores the TradingPartnerUserDelete Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default TradingPartnerResponse document on the file system.

XPCTradingPartnerOrganizationInformation30Inbound Service

Action List Component	Description
MessageAcknowledgementSender	Receives the TradingPartnerOrganizationInformation document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the TradingPartnerOrganizationInformation envelope, document, and attachment and the default TradingPartnerResponse document.
DefaultTPResponseFromOrganization Info30Builder	Builds a default TradingPartnerResponse document.

Action List Component	Description
FileStore.storeEnvelope	Stores the TradingPartnerOrganizationInformation Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default TradingPartnerResponse document on the file system.

XPCTradingPartnerOrganizationDelete30Inbound Service

Action List Component	Description
MessageAcknowledgementSender	Receives the TradingPartnerOrganizationDelete document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey to be used for file name when persisting. MessageId of the envelope is used
DefaultTPResponseFromOrganization Delete30Builder	Builds a default TradingPartnerResponse document.
FileStore.storeEnvelope	Stores the TradingPartnerOrganizationDelete Envelope, document and attachment on the file system.
FileStore.storeDocument	Stores the default TradingPartnerResponse document on the file system.

XPCTradingPartnerResponse30Inbound Service

Action List Component	Description
MessageAcknowledgementSender	Receives the TradingPartnerResponse document and sends a MessageAcknowledgement.
CorrelationKeyBuilder	Builds a CorrelationKey from the MessageId of the envelope. The CorrelationKey, concatenated with the appropriate prefix, forms the names of the files used to store the TradingPartnerResponse envelope, document, and attachment.
FileStore.storeEnvelope	Stores theTradingPartnerResponse Envelope, document and attachment on the file system.

XPCTradingPartnerUserInformation30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerUserInformation document from the file system and archive it. Builds the document object from the document read.
GetStringFromDocument	Gets the key string specified in the configuration.
StringMapper	Gets the TPID matching the key string the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCTradingPartnerUserDelete30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerUserDelete document from the file system and archives it. Builds the document object from the document read.
GetStringFromDocument	Gets the key string specified in the configuration
StringMapper	Gets the TPID matching the key string the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCTradingPartnerOrganizationInformation30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerOrganizationInformation document from file system and archives it. Builds the document object from the document read.
GetStringFromDocument	Gets the key string specified in the configuration.
StringMapper	Gets the TPID matching the key string the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCTradingPartnerOrganizationDelete30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerOrganizationDelete document from the file system and archives it. Builds the document object from the document read.
GetStringFromDocument	Gets the key string specified in the configuration.
StringMapper	Gets the TPID matching the key string the map file.
CreateEnvelope	Creates an Envelope using the TPID as RecepientId and senderTPID configuration as senderId of the Envelope header.
Transmitter	Transmits the Envelope to MarketSite.

XPCTradingPartnerResponseFromTPOrganizationInfo30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerResponse document from the file system and archives it. This document is supposed to be a TradingPartnerResponse to a TradingPartnerOrganizationInfomation document. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope
Responder	Sends the response document to MarketSite.

XPCTradingPartnerResponseFromTPOrganizationDelete30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerResponse document from the file system and archive sit. This document is supposed to be a TradingPartnerResponse to a TradingPartnerOrganizationDelete document. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the response document to MarketSite.

XPCTradingPartnerResponseFromTPUserInfo30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerResponse document from the file system and archives it. This document is supposed to be a TradingPartnerResponse to a TradingPartnerUserInformation document. Builds the document object from the document read. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the response document to MarketSite.

XPCTradingPartnerResponseFromTPUserDelete30Outbound Service

Action List Component	Description
FileStore.readDocument	Reads the TradingPartnerResponse document from the file system and archives it. Builds the document object from the document. Builds filename key string from the name of the file read.
FileStore.lookupEnvelope	Finds the original requesting envelope using the filename key passed. The original envelope is needed to send correlated response envelope.
Responder	Sends the TradingPartnerResponse document to MarketSite.

5 XPC Component Library

In This Chapter

This chapter describes the XPC Component Library. It includes a description of the component, required inputs, expected outputs, and values that can be externally configured.

Components fall into the following categories:

- Default response builders, which create default responses to incoming request documents
- MarketSite Messaging Layer (MML) components, which create envelopes for response documents, and XPath-based document querying components.
- File system components, which pass information to and from the file system
- Sample integrators, which can be customized to update the default response documents with data from the back office system
- Other system components, which performs such tasks as handling exceptions and transmitting documents to MarketSite.

Components marked as deprecated should be used only with xCBL 2.x documents.

For more information about the components and methods in this table, see the **API Reference** chapter.

Component Location

When configuring a component from the Component Library using XPC Manager, you must specify the fully qualified class name. For all standard XPC components, the component code is found at:

```
com.commerceone.xpc.components.<component class>.
```

Default Response Builders

The following components are used to build default responses to incoming request documents:

Name and Description	Inputs, Outputs, and Configurations
DefaultAuctionCreateResponse30Builder Builds a default AuctionCreateResponse document in response to an incoming AuctionCreate document.	**Inputs:** DefaultAuctionCreateDoc - DocumentObject - the incoming AuctionCreate request document. **Outputs:** DefaultAuctionCreateResponseDoc - DocumentObject - the default AuctionCreateResponse document. **Configurations:** None
DefaultAuctionResultResponse30Builder Builds a default AuctionResultResponse document in response to an incoming AuctionResult document.	**Inputs:** DefaultAuctionResultDoc - DocumentObject - the incoming AuctionResult document. **Outputs:** DefaultAuctionResultResponseDoc - DocumentObject - the default AuctionResultResponse document. **Configurations:** None
DefaultAvailabilityCheckResponse30Builder Returns a default AvailabilityCheckResult document in response to an incoming AvailabilityCheckRequest document.	**Inputs:** DefaultAvailabilityCheckRequestDoc - DocumentObject - the AvailabilityCheckRequest document. **Outputs:** DefaultAvailabilityCheckResponse - DocumentObject - the default AvailabilityCheckResult document **Configurations:** Quantity - the available quantity of the item UOMCode - the unit of measurement code in which the quantity is expressed

Name and Description	Inputs, Outputs, and Configurations
DefaultAvailabilityCheckResponseBuilder (deprecated) Returns a default xCBL 2.x AvailabilityCheckResponse document in response to an incoming xCBL 2.x AvailabilityCheckRequest document.	**Inputs:** DefaultAvailabilityCheckRequestDoc - DocumentObject - the AvailabilityCheckRequest document. **Outputs:** DefaultAvailabilityCheckResponse - DocumentObject - the default AvailabilityCheckResult document **Configurations:** Quantity - the available quantity of the item UOMCode - the unit of measurement code in which the quantity is expressed
DefaultAvailabilityToPromiseResponse30Builder Builds a default AvailabilityToPromiseResponse document based on an incoming AvailabilityToPromise document.	**Inputs:** DefaultAvailabilityToPromiseDoc - DocumentObject - the incoming AvailabilityToPromise document **Outputs:** DefaultAvailabilityToPromiseResponseDoc - DocumentObject - the default AvailabilityToPromise response document. **Configurations:** None
DefaultOrder30Builder Builds a default Order document in response to an incoming OrderRequest document.	**Inputs:** DefaultOrderRequest30Doc - DocumentObject - the incoming OrderRequest document **Outputs:** DefaultOrder30Doc - DocumentObject - the default Order document. **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
DefaultOrderResponse30Builder Builds a default OrderResponse document in response to an incoming Order document.	**Inputs:** DefaultOrder30Doc - DocumentObject - the incoming Order document. **Outputs:** DefaultOrderresponse30Doc - DocumentObject - the default OrderResponse document. **Configurations:** None
DefaultOrderResponseFromChangeOrder30Builder Builds a default OrderResponse document in response to an incoming ChangeOrder document.	**Inputs:** DefaultChangeOrder30Doc - DocumentObject - the incoming ChangeOrder document. **Outputs:** DefaultOrderResponse30Doc - DocumentObject - the default OrderResponse document. **Configurations:** None
OrderStatusResponse30Builder Builds a default OrderStatusResponse document in response to an incoming OrderStatusRequest document.	**Inputs:** DefaultOrderStatusRequestDoc - DocumentObject - the incoming OrderStatusRequest document. **Outputs:** DefaultOrderStatusResponse - DocumentObject - the default OrderStatusResponse document. **Configurations:** Status - the status of the order. “OrderStatusNotAccepted” by default.

Name and Description	Inputs, Outputs, and Configurations
DefaultOrderStatusResponseBuilder (deprecated) Returns a default xCBL 2.x OrderStatusResponse document in response to an incoming OrderStatusRequest document.	**Inputs:** DefaultOrderStatusRequestDoc - DocumentObject - the OrderStatusRequest document **Outputs:** DefaultOrderStatusResponse - DocumentObject - the default OrderStatusResult document **Configurations:** Status - the status of the order
DefaultPaymentRequestAck30Builder Builds a default PaymentRequestAcknowledgment document in response to an incoming PaymentRequest document.	**Inputs:** DefaultPaymentRequest30Doc - DocumentObject - the incoming PaymentRequest document. **Outputs:** DefaultPaymentrequestAck30Doc - DocumentObject - the default PaymentRequestAcknowledgment document. **Configurations:** None
DefaultPlanningScheduleResponse30Builder Builds a default PlanningScheduleResponse document in response to an incoming PlanningSchedule request document.	**Inputs:** DefaultPlanningScheduleDoc - DocumentObject - the incoming **Outputs:** DefaultPlanningScheduleResponseDoc - DocumentObject - the default PlanningScheduleResponse document. **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
DefaultPriceCheckResponse30Builder Returns a default PriceCheckResponse in response to an incoming PriceCheckRequest document.	**Inputs:** DefaultPriceCheckRequestDoc - DocumentObject - the input PriceCheckRequest document **Outputs:** DefaultPriceCheckResponse - DocumentObject - the default PriceCheckResult document **Configurations:** UnitPrice - the unit price for the item UOMCode - the unit of measurement code in which the unit price is expressed
DefaultPriceCheckResponseBuilder (deprecated) Returns a default xCBL 2.x PriceCheckResponse in response to an incoming PriceCheckRequest document.	**Inputs:** DefaultPriceCheckRequestDoc - DocumentObject - the input PriceCheckRequest document **Outputs:** DefaultPriceCheckResponse - DocumentObject - the default PriceCheckResult document **Configurations:** UnitPrice - the unit price for the item UOMCode - the unit of measurement code in which the unit price is expressed
DefaultPurchaseOrderResponseBuilder (deprecated) Returns a default xCBL 2.x PurchaseOrderResponse in response to an incoming PurchaseOrderRequest document.	**Inputs:** DefaultPurchaseOrderRequestDoc - DocumentObject - the input PurchaseOrderRequest document **Outputs:** DefaultPurchaseOrderResponse - DocumentObject - the default PurchaseOrderResponse document **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
DefaultQuote30Builder Builds a default Quote document in response to a RequestForQuotation document.	**Inputs:** DefaultRequestForQuoteDoc - DocumentObject - the incoming RequestForQuotation document. **Outputs:** DefaultQuoteDoc - DocumentObject - the default Quote document **Configurations:** None
DefaultShippingScheduleResponse30Builder Builds a default ShippingScheduleResponse document in response to a ShippingSchedule request document.	**Inputs:** DefaultShippingScheduleDoc - DocumentObject - the incoming ShippingSchedule document. **Outputs:** DefaultShippingScheduleResponseDoc - DocumentObject - the default ShippingScheduleResponse document. **Configurations:** None
DefaultTimeSeries30Builder Builds a default TimeSeriesResponse document in response to a TimeSeries document.	**Inputs:** DefaultTimeSeriesRequest30Doc - DocumentObject - the incoming TimeSeriesRequest document. **Outputs:** DefaultTimeSeries30Doc - DocumentObject - the default TimeSeries **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
DefaultTimeSeriesResponse30Builder Builds a default TimeSeriesResponse document in response to an incoming TimeSeriesRequest document.	**Inputs:** DefaultTimeSeries30Doc - DocumentObject - the incoming TimeSeries document. **Outputs:** DefaultTimeSeriesResponse30Doc - DocumentObject - the default TimeSeriesResponse document. **Configurations:** None
DefaultTPRResponseFromOrganizationDelete30Builder Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerOrganizationDelete document.	**Inputs:** DefaultTPOrganizationDelete30Doc - DocumentObject - the incoming TradingPartnerOrganizationDelete document. **Outputs:** DefaultTradingPartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document. **Configurations:** None
DefaultTPRResponseFromOrganizationInfo30Builder Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerOrganizationInformation document.	**Inputs:** DefaultTPOrganizationInformation30Doc - DocumentObject - the incoming TradingPartnerOrganizationInformation document. **Outputs:** DefaultTradingPartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document. **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
DefaultTPResponseFromUserDelete30Builder Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerUserDelete document.	**Inputs:** DefaultTPUserDelete30Doc - DocumentObject - the incoming TradingPartnerUserDelete document. **Outputs:** DefaultTradingPartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document. **Configurations:** None
DefaultTPRResponseFromUserInfo30Builder Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerUserInformation document.	**Inputs:** DefaultTPUserInformation30Doc - DocumentObject - the incoming TradingPartnerUserInformation document. **Outputs:** DefaultTradingpartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document. **Configurations:** None

MarketSite Messaging Layer (MML) and Document Querying Components

The following components are used to create envelopes for response documents.

Name and Description	Inputs, Outputs, and Configurations
CreateCorrelatingEnvelope Creates an Envelope for a response document. The RecipientTPID of the new envelope matches the SenderTPID of the request envelope. The CorrelationId of the new envelope matches that of the request envelope.	**Inputs:** Document - the incoming request document Envelope - the envelope that transmitted the request document **Outputs:** Envelope - the output envelope created for the response document **Configurations:** Addressing - String - the addressing mechanism for the new envelope. If “swap” (the default), the sender of the original document becomes the recipient of the new document and the recipient of the original document becomes the sender of the new document. If “keep” the new document’s sender and recipient are the same as those of the request document. TransmissionMode - String - The transmission mode for the document exchange. One of the following: one_way for one-way transmission, peer_peer for peer-to-peer transmission, or sync for synchronous transmission

Name and Description	Inputs, Outputs, and Configurations
CreateEnvelope Creates an envelope for a document.	**Inputs:** document - Document - the document for which the envelope will be created. **Outputs:** recipientTPID - String - the document recipient's Trading Partner ID (TPID). **Configurations:** senderTPID - String - the document sender's Trading Partner ID key string to be looked up from the mapping file specified by the mapFile configuration. This is required. TransmissionMode - String - the transmission mode for the document exchange. One of the following: one_way for one-way transmission, peer_peer for peer-to-peer transmission, or sync for synchronous transmission. mapFile - Full path to the file which contains the key string and TPID value pairs.
GetCorrelationKey Returns the correlation key of an envelope.	**Inputs** envelope - Envelope - the request envelope **Outputs** correlationKey - String - the correlation ID of the envelope **Configurations** KeyProperty - determines whether the correlation key is generated from the incoming envelope's CorrelationId (the default) or its MessageId. Xpath - a correlation key specification that reflects the values of one or more xCBL elements or attributes. Takes precedence over the KeyProperty setting.
GetStringFromDocument Queries a document using an XPath string and outputs the results of the query as a string. This component uses the XPCDocHandle helper class for XPath-based querying.	**Inputs:** document - DocumentObject - the xCBL document **Outputs:** correlationKey - String - the value queried from the document **Configurations:** DefaultString - an alternative correlation key to be output xPath - query string. For more informtion, see the **Building Custom Integrations** chapter.

File System Components

The components below pass information to and from the file system:

Name and Description	Inputs, Outputs, and Configurations
All components in the FileStore class	**The following configurations are common to all methods of the FileStore class:** ■ Overwrite - determines whether new files overwrite existing files with the same name. A “no” setting throws an error if an existing file with the specified name is found. A “yes” setting archives the most recent version of the file by appending a dollar sign ($) to its file name. A “history” setting archives **all** versions of the file by appending a dollar sign and version number to the most recent version’s file name is appended by a dollar sign ($) and version number to an existing file when it is overwritten. ■ RootDirectory - the root archive directory used for all configurations.
FileStore.copyFile Copies the file named by the input string from the File.Source.Directory to the File.Target.Directory defined in the configuration file. File name format will be [Prefix][filename key][extension].	**Inputs:** ■ filenameKey - String - the correlation key of the file to be copied **Outputs:** ■ None **Configurations:** ■ File.Source.Directory - the directory in which the original file is located ■ File.Target.Directory - the directory into which the file will be copied ■ File.Prefix - a set of characters to be added at the beginning of the file name specified by filenameKey ■ File.Extension - a set of characters to be added at the end of the file name specified by filenameKey

Name and Description	Inputs, Outputs, and Configurations
FileStore.lookupEnvelope Reads the envelope with the specified correlation key from the file system. Archives the envelope to a separate Envelope.Archive.Directory if one is defined.	**Inputs:** ■ correlationKey - String - the correlation key of the envelope **Outputs:** ■ lookupEnvelope - Envelope - the archived envelope **Configurations:** ■ Envelope.Directory - the directory in which the envelope is currently stored ■ Envelope.Archive.Directory - the directory in which the envelope archive will be stored ■ Envelope.Prefix - string (by default, E_) with which each envelope file name begins ■ Envelope.Extension - string (by default, .env) with which each envelope file name ends

Name and Description	Inputs, Outputs, and Configurations
FileStore.readDocument Reads a document from a directory. Outputs the document along with the correlation key. Archives the document to a separate Document.Archive.Directory if one is defined. This method follows the conventions required for an initiating action in a Timed Service. It continues to read and issue documents until no more are found.	**Inputs:** ■ None **Outputs:** ■ documentRead - Document - the document to be read ■ correlationKey - String - the correlation key of the document **Configurations:** ■ Document.Directory - directory in which the document is stored. The default directory name varies by XPC service and reflects the type of document being archived. For services that archive request documents, a default directory called “default_request” is used. For services that archive response documents, a default directory called “default_response” is used. ■ Document.Archive.Directory - the directory in which the document will be archived ■ Document.Prefix - string (by default, D_) with which the document file name begins ■ Document.Extension - string (by default, .xml) with which the document file name ends

Name and Description	Inputs, Outputs, and Configurations
FileStore.readStream Reads a file from a directory. Outputs a streamed input file and a correlation key to the Stream.Directory. Archives the file to a separate Stream.Archive.Directory if one is defined. This method follows the conventions required for an initiating action in a Timed Service. It will keep reading files and issuing streams until no more are found.	**Inputs:** ■ None **Outputs:** ■ streamRead - InputStream - the streamed input file ■ correlationKey - String - the correlation key **Configurations:** ■ Stream.Directory - the directory in which the streamed input file will be stored. ■ Stream.Archive.Directory - the directory in which the file will be archived once it has been streamed. ■ Stream.prefix - string (by default, D_) with which the stream file name begins. ■ Stream.Extension - string (by default, .xml) with which the stream file name ends.
FileStore.storeDocument Stores the input document.	**Inputs:** ■ documentToStore - Document - the input document ■ correlationKey - String - the correlation key **Outputs:** ■ None **Configurations:** ■ Document.Directory - directory in which the document will be stored ■ Document.Prefix - string (by default, D_) with which the document file name begins ■ Document.Extension - string (by default, .xml) with which the document file name ends

Name and Description	Inputs, Outputs, and Configurations
FileStore.storeEnvelope Stores an envelope, request document and attachments.	**Inputs:** ■ envelopeToStore - Envelope - the envelope to be stored ■ correlationKey - String - the correlation key **Outputs:** ■ None **Configurations:** ■ Document.Directory - directory in which the document is archived. ■ Envelope.Attachment.Description.Directory - directory in which descriptions of attachments are stored ■ Envelope.Attachment.Description.Extension - string (by default, .adf) with which each attachment description file name ends ■ Envelope.Attachment.Description.Prefix - string with which each attachment description file name begins. ■ Envelope.Attachment.Directory - directory in which the attachments are stored. ■ Envelope.Attachment.Extension - string (by default, .att) with which each attachment file name ends ■ Envelope.Attachment.NameURI.[Name].Directory - the attachment directory naming convention, which is based upon attachment name ■ Envelope.Attachment.Prefix - string with which each attachment file name begins. (continued on following page)

Name and Description	Inputs, Outputs, and Configurations
FileStore.storeEnvelope (continued from previous page)	**Configurations:** ■ Envelope.Directory - directory in which the envelope is stored. ■ Envelope.Document.Directory - directory in which the document is stored. ■ Envelope.Document.Extension - string with which each document file name ends ■ Envelope.Document.Prefix - string with which each document file name begins. ■ Envelope.Extension - string (by default, .env) with which each envelope file name ends ■ Envelope.Prefix - string with which each envelope file name begins.
FileStore.storeStream Stores the input stream.	**Inputs:** ■ streamToStore - InputStream - streamed data ■ correlationKey - String - correlation key **Outputs:** ■ None **Configurations:** ■ Stream.Directory - the directory in which the streamed input is stored. ■ Stream.Prefix - string (by default, D_) with which the stream file name begins. ■ Stream.Extension - string (by default, .xml) with which the stream file name ends.
StreamToDocument Converts the input stream to an xCBL document.	**Inputs:** streamRead - InputStream - streamed input **Outputs:** document - DocumentObject - an xCBL document **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
StringMapper.process Uses a flat file to match the recipient ID to the corresponding recipient TPID used by MarketSite. If no matching recipient TPID is found an exception is thrown.	**Inputs:** receiverID - String - the receiver ID from the EDI file **Outputs:** receiverTPID - String - the Trading Partner ID for the receiver **Configurations:** mapFile - the *map.txt* file used to map Receiver IDs to Trading Partner IDs

Sample Integrators

The following components are used to customize the default response documents with actual business data:

Name and Description	Inputs, Outputs, and Configurations
myAvailabilityCheckIntegrator30.process Customizes the default response document by calling the doAvailabilityCheck helper method once for each item whose availability is being checked.	**Inputs:** RequestDoc -XCBL30_sox.AvailabilityRequest - the AvailabilityCheckRequest document ResultDoc - XCBL30_sox.AvailabilityResult - the default AvailabilityCheckResult document **Outputs:** ResultDoc - XCBL30_sox.AvailabilityResult - the AvailabilityCheckResult document **Configurations:** None
myAvailabilityCheckIntegrator.process (deprecated) Customizes the default response document by calling the doAvailabilityCheck helper method once for each item whose availability is being checked.	**Inputs:** RequestDoc - CBL_sox.AvailabilityRequest - the AvailabilityCheckRequest document ResultDoc - CBL_sox.AvailabilityResult - the default AvailabilityCheckResult document **Outputs:** ResultDoc - CBL_sox.AvailabilityResult - the AvailabilityCheckResult document **Configurations:** None
myOrderStatusIntegrator30.process Customizes the default response document by calling the doOrderStatus() method once for each item whose status is being checked.	**Inputs:** RequestDoc -XCBL30_sox.OrderStatusRequest - the OrderStatusRequest document ResultDoc - XCBL30_sox.OrderStatusResult - the default OrderStatusResult document **Outputs:** ResultDoc - XCBL30_sox.OrderStatusResult - the OrderStatusResult document **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
myOrderStatusIntegrator.process (deprecated) Customizes the default response document by calling the doOrderStatus() method once for each item whose status is being checked.	**Inputs:** RequestDoc - CBL_sox.OrderStatusRequest - the OrderStatusRequest document ResultDoc - CBL_sox.OrderStatusResult - the default OrderStatusResult document **Outputs:** ResultDoc - CBL_sox.OrderStatusResult - the OrderStatusResult document **Configurations:** None
myPriceCheckIntegrator30.process Customizes the default response document by calling the doPriceCheck() method once for each item whose price is being checked.	**Inputs:** RequestDoc - XCBL30_sox.PriceCheckRequest - the PriceCheckRequest document ResultDoc - XCBL30_sox.PriceCheckResult- the default PriceCheckResult document **Outputs:** ResultDoc - XCBL30_sox.PriceCheckResult - the PriceCheckResult document **Configurations:** None
myPriceCheckIntegrator.process (deprecated) Customizes the default response document by calling the doPriceCheck() method once for each item whose price is being checked.	**Inputs:** RequestDoc - CBL_sox.PriceCheckRequest - the PriceCheckRequest document ResultDoc - CBL_sox.PriceCheckResult- the default PriceCheckResult document **Outputs:** ResultDoc - CBL_sox.PriceCheckResult - the PriceCheckResult document **Configurations:** None

Other System Components

The following components performs such tasks as handling exceptions and transmitting documents to MarketSite

Name and Description	Inputs, Outputs, and Configurations
ExceptionHandler Logs an event and builds an Error document based on a previously thrown exception.	**Inputs:** exception - Exception - a previously thrown exception componentName - String - the component that threw the exception **Outputs:** replyDocument - Document - an Error document based on the exception **Configurations:** None
LookupXCCArchive Retrieves the archived envelope whose referenceId or correlationId matches that of the input envelope. The standard XPC archive mechanism is checked for the envelope. This archive is described in the **Administration** chapter of the *XPC Installation and Administration Guide.*	**Inputs:** inputEnvelope - Envelope - the envelope containing the error document **Outputs:** outputEnvelope - Envelope - the archived envelope whose correlation key matches that of the input envelope. correlationKey - String - the correlation key **Configurations:** None
MessageAcknowledgmentSender Creates an envelope containing a MessageAcknowledgement document and transmits it to the sender of the original document.	**Inputs:** inputEnvelope - Envelope - the envelope containing the original document **Outputs:** None **Configurations:** None

Name and Description	Inputs, Outputs, and Configurations
Responder Sends a response document to the initiator of a previously received request envelope.	**Inputs:** replyDocument - DocumentObject - the response document **Outputs:** None **Configurations:** None
Transmitter Transmits an envelope to the destination specified in the envelope's properties.	**Inputs:** requestEnv - Envelope - the request envelope **Outputs:** None **Configurations:** None

6 Testing Your Integrations

In This Chapter

Once you have customized the way XPC handles a particular request document you can use the XPC Invoker in test mode to verify that the customization is working as it should. The Invoker allows you to send sample xCBL documents to XPC and to examine XPC's reply documents.

This chapter describes how to use the Invoker test your integrations. It includes the following information:

Overview of the Invoker

The Invoker is a graphical user interface that runs on the same machine as the XPC server. To facilitate the creation of new Envelopes, the Invoker comes packaged with pre-configured xCBL Documents for each supported transaction. You can modify these documents so that they more closely represent your own business data.

You can use the Invoker in Test mode to send the modified xCBL documents to the running XPC server. XPC replies either synchronously or asynchronously with the appropriate response document, which you can view with the Invoker. As you send and receive envelopes, you can examine the behavior of each component to verify that your customizations are working correctly.

For more information about the Invoker, please see **Using the XPC Invoker** in the **Administration** chapter of the *XPC Installation and Administration Guide*.

Modifying the Sample Request Documents

To facilitate the creation of new Envelopes, the Invoker comes packaged with pre-configured xCBL Documents for the supported transaction types. These samples are stored in the \sample\xpc\instances folder of your XPC installation, which contains a separate subfolder named after each transaction type. Each folder contains one or more samples of the named xCBL document.

These templates are provided as examples of each request type. To provide a better test of your integration, you need to modify the templates so that they include the values you want to test. Following are some examples of changes you could make:

- Change PartIDs to part numbers your company actually uses.
- Make sure the AvailabilityCheckRequest document includes parts for which there is no available supply as well as parts that are currently available.

Testing Your Customizations

Before testing the XPC installation, verify that XPC is started and that you have configured XPC correctly.

1. Start XPC Invoker.
   To start the Invoker on NT, go to **Start | Programs | XML Portal Connector 4.0 | Invoker**.

Note .....The recommended screen resolution for the Invoker is 1024x768.

2. From the Transaction Filter pull-down menu, select the first transaction you want to test. For example, to check your price check integration, select the **PriceCheck** transaction.
3. Click **New** to create a new envelope for the transaction.
4. Go to the \sample\xpc\instances folder, then to the folder for the transaction you want to test. Select the xCBL document you modified for the transaction.
5. Click **Send**.
   After a few seconds, the Invoker displays the response Envelope.
6. Open the response Envelope and verify that the response document contains the expected information.

Debugging Your Components

When testing your components within the debugging environment of your integrated development environment, you execute the XPC Server as a standalone Java program rather than as an NT service. This involves executing the main() method of the com.commerceone.ccs.server. CCSServer class, supplying the location of your XPC server root as the '-root' command line argument. Additionally, your IDE will need to know the location of each JAR file used by the server. The required JAR files are listed in the file \etc\classpath\default in your XPC installation.

Following is a sample command line using Sun Microsystems jdk:

```
java -classic -Xms4m -Xmx512m
com.commerceone.ccs.server.CCSServer –root
c:\commerceone\xpc\runtime\servers\defaultserver
```

Note ..... This command line assumes that the required JAR files have already been configured within the IDE.

7 API Reference

This chapter contains reference information about the XML Portal Connector application programming interface. It describes the major XPC packages, classes, interfaces, and methods.

More sophisticated component development may require the use of the following Commerce One supplied APIs:

- The XDK package, which allows for the direct manipulation of the contents of documents and envelopes
- The Util package, which contains utility classes that may be used to create unique identifiers

For information about using these APIs, see the Javadoc contained in the *\doc\api\xdk* and *doc\api\util* directories of the XPC installation.

Packages

This reference provides information about the following packages

com.commerceone.xpc.abs	This package contains XPCAbstractComponent, the superclass of XPC components. All components—those provided by XPC for out-of-the-box use, user-developed extensions of these components, and new components developed by users—are subclasses of this class. This provides a link between components and the XCC Server's service framework, which is used for publishing document replies, logging events, and so forth.
com.commerceone.xpc.common	This package contains classes used by all components. They include classes used to pass configuration parameters, input and output arguments, and the result of executing a component.

com.commerceone.xpc.components	Contains component classes used to build error documents, send default responses to incoming CBL documents, store envelopes, request documents, and attachments on the file system, and convert streamed input to xCBL documents.
com.commerceone.xpc.helpers	This package contains classes whose methods can be used to access the information in request documents and to build both the data portion and the error portion of response documents
com.commerceone.xpc.my_integrators	Contains working samples of component classes used to build and send default responses to requests for item availability, item price, or order status. You can use these components as models when extending XPCAbstractComponent to build your own integrations. Unlike the sample components in this class, your own components should issue response documents with actual availability, price, or order status information retrieved from your backend system.
com.commerceone.xpc.swi.common	This package contains an interface implemented by all XPC components. Its methods are used to define a component's input and output arguments.
com.commerceone.xpc.swi.framework	This package contains a number of interfaces that are implemented by each XPC component. These interfaces provide methods for naming the component, setting its configuration parameters, invoking its execution method, and transmitting request envelopes and response documents.
com.commerceone.xpc.gedi	This package contains component classes used by the XPC Generic EDI Wrapper service to receive and transport ANSI X12 and EDIFACT formatted EDI files through MarketSite. These classes are used to parse data from EDI files, map RecipientIDs to RecipientTPIDs in a flat file, compress and decompress files, and create and populate envelopes.

package com.commerceone.xpc.abs

Description

Provides the superclass of XPC components.

Classes

XPCAbstractComponent	Is the superclass of XPC components. All components, including components provided by XPC for out-of-the-box use, user-developed extensions of these components, and new components developed by users, are subclasses of this component.

package com.commerceone.xpc.abs

class XPCAbstractComponent

Description

Is the superclass of XPC components. All components, including components provided by XPC for out-of-the-box use, user-developed extensions of these components, and new components developed by users, are subclasses of this component.

Implements

- interface **XPCAdmin**
- interface **XPCConfig**
- interface **XPCProcess**
- interface **XPCTransmit**
- interface **XPCContract**

Extended By

All XPC components

Methods

getEntityManager()
getEnvelopeFactory()
public final String getName() Returns the component name.
public void setName(String name) Sets the name for the component.
public final boolean config(XPCTransmit container, XPCConfigParams configParams) Configures the component using the configParams. Returns true if the configuration was successful; false, otherwise. Note: this method is called by the infrastructure only; it is not for use by XPC users. container - XPCTransmit - the container within which the component is running configParams - XPCConfigParams - parameters used to configure the component

public final String getConfigParam(String key) Returns the value for the given key from the configuration parameters.
protected final void logMessage(String logMessage) Creates a standard event with the specified message and logs the event with an event ID of "XPC_COMP_ERRORGENERIC."
protected final void logError(String errMessage) Creates a standard event with the specified message and invokes the EventGenerator to log the event with an event ID of "XPC_COMP_ERRORGENERIC."
protected final void reply(Envelope reqEnv, Document replyDoc) Sends the reply document to the initiator of the original request. reqEnv - Request envelope replyDoc - Reply document
protected final void transmit(Envelope env) Transmits the envelope to the destination specified in the envelope's properties.
protected final EnvelopeFactory getEnvelopeFactory() Returns the Envelope factory. Components call this method to get the EnvelopeFactory, which they use for creating envelopes.
public abstract XPCContractDescriptor[] getInputList(String methodName) Gets the list of input argument descriptors for the named method. Implemented by subclasses to define the input arguments of their methods.
public abstract XPCContractDescriptor[] getOutputList(String methodName) Gets the list of output argument descriptors for the named method. Implemented by subclasses to define the output arguments of their methods.
public XPCResult process(XPCDataMgr dataMgr) Invocation method for standard components. Implemented by subclasses to define the invocation method of standard components.

package com.commerceone.xpc.common

Description

This package contains classes used by all components. They include classes used to pass configuration parameters, input and output arguments, and the results of executing a component.

Classes

XPCConfigParams	This utility class gets configuration parameters from the default.prop file. These parameters may be specified either as Java.util.Properties objects or as strings in the form *parameter_name=parameter_value.* Consecutive parameters are separated by commas.
XPCDataMgr	This class is used to pass data from one component to another.
XPCResult	This class is used to pass the integer code and descriptive string that result from the execution of a component. Successful executions have a ResultCode of 0 and a ResultString of "SUCCESS." Unsuccessful executions have a non-zero ResultCode.
XPCContractDescriptor	This class is used to describe the name and type of an input or output argument of a component's method.

package com.commerceone.xpc.common

class XPCConfigParams

Description

This utility class gets configuration parameters from the default.prop file of the service in which the component is running. These parameters may be specified either as Java.util.properties objects or as strings in the form *parameter_name=parameter_value,.*

Constructors

public XPCConfigParams(String nameValueString)
This class is only constructed by the XPC Framework. Creates a comma-separated list of configuration parameters in the form *`parameter_name1=parameter_value1, parameter_name2=parameter_value2...`*
For example, the following string specifies the values of three parameters, url, userid, and password:
url=http://www.commerceone.com,userid=admin,password=mypassword

public XPCConfigParams(Properties prop)
Creates a configuration parameter from a java.util.Properties object.

Methods

public boolean getConfigParam(String key)
Returns the value of the configuration parameters specified by the key.

public void listContents(java.io.PrintStream out)
Prints the values of all configuration parameters.

package com.commerceone.xpc.common

class XPCDataMgr

Description

This class is used to pass data from one component to another. A component accesses its input arguments through calls to the get() method and writes its output arguments through calls to the set() method.

Constructors

public XPCDataMgr(XPCPropertiesPrototype prototype)
This class is constructed only by the XPC insfrastructure. Uses the XPCPropertiesPrototype object passed to it to control the reading of input arguments and the writing of output arguments.
prototype - The prototype that controls the getting and setting of arguments

Methods

public Object get(String key)
Gets the value of the named argument. If the argument is not get-enabled, throws an IllegalArgumentException.
key - String - the name of the argument to get

public void set(String key, Object value)
Sets the argument to the specified value. If the argument is not set-enabled, throws an IllegalArgumentException.
key - the name of the argument to set.

package com.commerceone.xpc.common

class XPCResult

Description

This class is used to pass the integer code and descriptive string that result from the execution of a component. Successful executions have a ResultCode of 0 and a ResultString of “SUCCESS.” Unsuccessful executions have a non-zero ResultCode.

Constructors

public XPCResult(int resultCode)

Variables

public final static int SUCCESS=0

Methods

public int getResultCode() Returns the integer result code.
public void setResultCode(int resultCode) Sets the integer result code.
public String getResultString() Returns a text string that describes the result of executing the action.
public void setResultString(String resultStr) Sets the text string that describes the result of executing the action.

package com.commerceone.xpc.common

class XPCContractDescriptor

Description

This class is used to describe the name and type of an input or output argument.

Constructors

public XPCContractDescriptor(final String aName, final Class aClass)

Methods

public String getName() Returns the name of an input or output argument.
public Class getContract() Returns the type of an input or output argument.

package com.commerceone.xpc.components

Description

Contains component classes used to handle exceptions, build and send default responses to incoming xCBL documents, store envelopes, request documents, and attachments on the file system, and convert streamed input to xCBL documents.

Classes

CreateCorrelatingEnvelope	Creates an Envelope for a response document. The RecipientTPID of the new envelope matches the SenderTPID of the request envelope. The CorrelationId of the new envelope matches that of the request envelope.
CreateEnvelope	Creates an envelope for a document.
DefaultAuctionCreateResponse30Builder	Builds a default AuctionCreateResponse document in response to an incoming AuctionCreate document.
DefaultAuctionResultResponse30Builder	Builds a default AuctionResultResponse document in response to an incoming AuctionResult document.
DefaultAvailabilityCheckResponse30Builder	Returns a default AvailabilityCheckResult document in response to an incoming AvailabilityCheckRequest document. Do not use this component with release 2.x xCBL documents; use the DefaultAvailabilityCheckResponseBuilder instead.
DefaultAvailabilityCheckResponseBuilder (deprecated)	Returns a default xCBL 2.x AvailabilityCheckResponse document in response to an incoming xCBL 2.x AvailabilityCheckRequest document. Do not use this component with release 3.x xCBL documents.
DefaultAvailabilityToPromiseResponse30Builder	Builds a default AvailabilityToPromiseResponse document based on an incoming AvailabilityToPromise document.

DefaultOrder30Builder	Builds a default Order document in response to an incoming OrderRequest document.
DefaultOrderResponse30Builder	Builds a default OrderResponse document in response to an incoming Order document.
DefaultOrderResponseFromChangeOrder30Builder	Builds a default OrderResponse document in response to an incoming ChangeOrder document.
OrderStatusResponse30Builder	Builds a default OrderStatusResponse document in response to an incoming OrderStatusRequest document.
DefaultOrderStatusResponseBuilder (deprecated)	Returns a default OrderStatusResponse document in response to an incoming OrderStatusRequest document. This component should be used only with release 2.x xCBL documents
DefaultPaymentRequestAck30Builder	Builds a default PaymentRequestAcknowledgment document in response to an incoming PaymentRequest document.
DefaultPlanningScheduleResponse30Builder	Builds a default PlanningScheduleResponse document in response to an incoming PlanningSchedule request document.
DefaultPriceCheckResponse30Builder	Returns a default PriceCheckResponse in response to an incoming PriceCheckRequest document. Do not use this component with release 2.x xCBL documents; Use the DefaultPriceCheckResponseBuilder component instead.
DefaultPriceCheckResponseBuilder (deprecated)	Returns a default PriceCheckResponse in response to an incoming PriceCheckRequest document. This component should be used only with release 2.x xCBL documents.
DefaultPurchaseOrderResponseBuilder (deprecated)	Returns a default PurchaseOrderResponse in response to an incoming PurchaseOrderRequest document. This component should be used only with release 2.x xCBL documents.

DefaultQuote30Builder	Builds a default Quote document in response to a RequestForQuotation document.
DefaultShippingScheduleResponse30Builder	Builds a default ShippingScheduleResponse document in response to a ShippingSchedule request document.
DefaultTimeSeries30Builder	Builds a default TimeSeriesResponse document in response to a TimeSeries document.
DefaultTimeSeriesResponse30Builder	Builds a default TimeSeriesResponse document in response to an incoming TimeSeriesRequest document.
DefaultTPRResponseFromOrganizationDelete 30Builder	Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerOrganizationDelete document.
DefaultTPRResponseFromOrganizationInfo30 Builder	Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerOrganizationInformation document.
DefaultTPResponseFromUserDelete30Builder	Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerUserDelete document.
DefaultTPRResponseFromUserInfo30Builder	Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerUserInformation document.
ExceptionHandler	This standard component logs an event and builds an Error document based on a previously thrown exception.
FileStore	This extended component provides methods for storing envelopes, request documents, and attachments in directories on the file system and for reading the archived files.
GetCorrelationKey	This standard component returns the correlation key of an envelope.
GetStringFromDocument	Reads the correlation key in the xCBL document and returns it to the Data Manager.

LookupXCCArchive	Accepts an Error document as input and returns the archived envelope with the corresponding correlation key.
MessageAcknowledgmentSender	Creates an envelope containing a MessageAcknowledgment document and transmits it to the sender of the original document.
Responder	Sends a response document to the initiator of a previously received request envelope.
StreamToDocument	This standard component converts streamed input to an xCBL document.
Transmitter	Transmits an envelope to the destination specified in the envelope's properties.

package com.commerceone.xpc.components

class CreateCorrelatingEnvelope

Description

Creates a new envelope based on the properties of the incoming request envelope.The TransmissionMode configuration detemines whether the document exchange is one-way, peer-to-peer, or synchronous.

The new envelope has the same CorrelationId as the original envelope. If the Addressing configuration is “keep” the new envelope has the same senderTPID and RecipientTPID as the original envelope. If the Addressing configuration is “swap” the SenderTPID of the original envelope is used as the RecipientTPID of the new envelope and the RecipientTPID of the new envelope is used as the SenderTPID of the new envelope.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
Document - the incoming request document
Envelope - the envelope that transmitted the request document
Outputs:
Envelope - the output envelope created for the response document
Configurations:
Addressing - String - the addressing mechanism for the new envelope. If “swap” (the default), the sender of the original document becomes the recipient of the new document and the recipient of the original document becomes the sender of the new document. If “keep” the new document’s sender and recipient are the same as those of the request document.
TransmissionMode - String - The transmission mode for the document exchange. One of the following: one_way for one-way transmission, peer_peer for peer-to-peer transmission, or sync for synchronous transmission

package com.commerceone.xpc.components

class CreateEnvelope

Description

Creates an envelope for a document.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
document - Document - the document for which the envelope will be created.
Outputs:
recipientTPID - String - the document recipient's Trading Partner ID (TPID).
Configurations:
senderTPID - String - the document sender's Trading Partner ID key string to be looked up from the mapping file specified by the mapFile configuration. This is required.
TransmissionMode - String - the transmission mode for the document exchange. One of the following: one_way for one-way transmission, peer_peer for peer-to-peer transmission, or sync for synchronous transmission.
mapFile - full path to the file that contains the key string and TPID value pairs.

package com.commerceone.xpc.components

class DefaultAuctionCreateResponse30Builder

Description

Builds a default AuctionCreateResponse document in response to an AuctionCreate document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as “Other” or set to the current ID or date and time. The following table lists the field settings:

Field	Setting
AuctionCreateResponsePurpose	“Other”
AuctionCreateResponseIssueDate	current date and time
AuctionCreateResponseID	new ID
RefNum	copied from request document
RefDate	current date and time
BasicResponseCode	“Other”
AuctionResponseCodedOther	null
AuctionCreateHeader	null
Language	copied from request document
AuctionCreateResponseNote	null
AuctionItemID	copied from request document
AuctionItemName	copied from request document
AuctionItemDescription	copied from request document
AuctionItemHierarchyLevel	copied from request document
AuctionLineItemNum	copied from request document
AuctionItemResponseCode	“Other”
AuctionItemResponseCodedOther	null

Field	Setting
ListOfAuctionItemComponentResponse (multiple fields)	copied from request document
TotalNumberOfAuctionItems	copied from request document
TotalNumberOfParticipants	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultAuctionCreateDoc - DocumentObject - the incoming AuctionCreate request document.
Outputs:
DefaultAuctionCreateResponseDoc - DocumentObject - the default AuctionCreateResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultAuctionResultResponse30Builder

Description

Builds a default AuctionResultResponse document in response to an AuctionResult document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as “Other” or set to the current ID or date and time. The following table lists the field settings:

Field	Setting
Purpose	“Other”
AuctionResultResponseIssueDate	current date and time
AuctionResultResponseID	new UUID
AuctionCreateReference	copied from request document
AuctionResultReference	copied from request document
AuctionresultResponseCoded	“Other”
AuctionResultResponseCodedOther	null
Language	copied from request document
GeneralNote	null

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultAuctionResultDoc - DocumentObject - the incoming Auctionresult document.
Outputs:
DefaultAuctionResultResponseDoc - DocumentObject - the default AuctionResultResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultAvailabilityCheckResponse30Builder

Description

Builds a default AvailabilityCheckResult document in response to an incoming AvailabilityCheckRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.Both the QuantityValue and the UnitOfMeasurement can be reset using the XPC Manager.

Fields in the default response document are set as follows:

Field	Setting
AvailabilityCheckResultID	copied from request document
AvailabilityCheckResultIssueDate	copied from request document
SupplierParty	copied from request document
SupplierIDReferenceDate	copied from request document
BuyerParty	copied from request document
BuyerIDReferenceDate	copied from request document
AvailabilityShipToParty	copied from request document
AvailabilityCheckResultLanguage	copied from request document
AvailabilityCheckResultNote	null
ResultListOfAttachment	copied from request document
LineItemNum	copied from request document
LineItemType	copied from request document
ParentItemNumber	copied from request document
ItemIdentifiers	copied from request document

Field	Setting
ListOfDimension	copied from request document
TotalQuantity	copied from request document
MaxBackOrderQuantity	copied from request document
OffCatalogFlag	copied from request document
ListOfItemReferences	copied from request document
CountryOfOrigin	copied from request document
CountryOfDestination	copied from request document
FinalRecipient	copied from request document
ConditionsOfSale	copied from request document
HazardousMaterials	copied from request document
CheckResultTransport	copied from request document
QuantityValue	copied from request document, if supplied, or set to -1. Can be reconfigured in XPC Manager.
UnitOfMeasurement	copied from request document, if supplied, or set to 1. Can be reconfigured in XPC Manager.
AvailabilityErrorInfo	null
GeneralLineItemNote	copied from request document
LineItemAttachment	copied from request document
AvailabilityItemErrors	0
SummaryErrorInfo	null
TotalNumberOfLineItem	copied from request document

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the input document from the Data Manager, calls the buildResponse method, and sets the default AvailabilityCheckResult document in the Data Manager.

Inputs:

DefaultAvailabilityCheckRequestDoc - DocumentObject - the AvailabilityCheckRequest document.

Outputs:

DefaultAvailabilityCheckResponse - DocumentObject - the default AvailabilityCheckResult document

Configurations:

Quantity - the available quantity of the item

UOMCode - the unit of measurement code in which the quantity is expressed

package com.commerceone.xpc.components

class DefaultAvailabilityCheckResponseBuilder (deprecated)

Builds a default AvailabilityCheckResult document in response to an incoming AvailabilityCheckRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
AvailabilityCheckHeader (multiple fields)	copied from request document
Quantity	copied from request document, if supplied; otherwise, -1. May be reset in XPC Manager.
UnitOfMeasurement	copied from request document, if supplier; otherwise UOMCode.10. May be reset in XPC Manager.
ErrorInfo	null
AvailabilityItemErrors	0

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the input document from the Data Manager, calls the buildResponse method, and sets the default AvailabilityCheckResponse document in the Data Manager.

Inputs:

DefaultAvailabilityCheckRequestDoc - DocumentObject - the AvailabilityCheckRequest document.

Outputs:

DefaultAvailabilityCheckResponse - DocumentObject - the default AvailabilityCheckResult document

Configurations:

Quantity - the available quantity of the item

UOMCode - the unit of measurement code in which the quantity is expressed

ackage com.commerceone.xpc.components

class DefaultAvailabilityToPromiseResponse30Builder

Description

Builds a default AvailabilityToPromiseResponse document based on an incoming AvailabilityToPromise document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
RefNum	new ID
RefDate	current date and time
AvailabilityResponseIssueDate	current date and time
AvailabilityToPromiseID	copied from request document
AvailabilityToPromiseResponseCode	“Other”
AvailabilityToPromiseResponseCodedOther	null
ListOfReferenceCoded	copied from request document
AvailabilityDeliveryOption	copied from request document
InitiatingParty	copied from request document
AvailabilityShipToParty	copied from request document
AvailabilityResponseHeaderTransport	copied from request document
GeneralNote	copied from request document
ListOfAttachment	copied from request document
AvailabilityToPromisePurposeCode	“Other”
AvailabilityToPromisePurposeCodedOther	null
AvailabilityToPromiseBaseItemDetail	copied from request document

Field	Setting
AvailabilityToPromiseResponseDeliveryDetail	copied from request document
AvailabilityToPromiseResponseTransportDetail	copied from request document
AvailabilityToPromiseResponseItemListOfAttachment	copied from request document
AvailabilityToPromiseGeneralNote	null
TotalNumberOfLineItems	

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultAvailabilityToPromiseDoc - DocumentObject - the incoming AvailabilityToPromise document.
Outputs:
DefaultAvailabilityToPromiseResponseDoc - DocumentObject - the default AvailabilityToPromiseresponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultOrder30Builder

Description

Builds a default Order document in response to an incoming OrderRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings:

Field	Setting
BuyerOrderNumber	copied from request document
SellerOrderNumber	copied from request document
ListOfMessageID	copied from request document
OrderIssueDate	current date and time
OrderReferences	copied from request document
ReleaseNumber	null
Purpose	copied from request document
RequestedResponse	copied from request document
OrderType	NewOrder
OrderCurrency	copied from request document
TaxAccountingCurrency	copied from request document
OrderLanguage	copied from request document
OrderTaxReference	copied from request document
OrderInvoiceMediumTypeCoded	copied from request document
OrderInvoiceMediumTypeCodedOther	copied from request document
OrderDates	copied from request document

Field	Setting
BuyerParty	copied from request document if non-null; otherwise, UN.
AgencyCodedOther	null
AgencyDescription	null
CodeListIdentifierCoded	null
CodeListIdentifierCodedOther	null
Ident	“ “
ListOfIdentifier	null
MDFBusiness	false
NameAddress	null
OrderContact	null
ReceivingContact	null
ShippingContact	null
Correspondencelanguage	null
BuyerTaxInformation	copied from request document
SellerParty	copied from request document
SellerTaxInformation	copied from request document
ShipToParty	copied from request document
BillToParty	copied from request document
RemitToParty	copied from request document
ShipFromParty	copied from request document
WherehouseParty	null
SoldToParty	null
ManufacturingToParty	null
MaterialIssuer	null

Field	Setting
ListOfPartyCoded	copied from request document
PartLocation	null
ListOfTransport	copied from request document
OrderTermsOfDelivery	copied from request document
OrderHeaderPrice	copied from request document
OrderPaymentInstructions	copied from request document
OrderAllowancesOrCharges	copied from request document
OrderHeaderNote	copied from request document
ListOfStructuredNote	copied from request document
OrderHeaderAttachments	copied from request document
OrderDetail	copied from request document
OrderSummary	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultOrderRequest30Doc - DocumentObject - the incoming Orderrequest document.
Outputs:
DefaultOrder30Doc - DocumentObject - the default Order document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultOrderResponse30Builder

Description

Builds a default OrderResponse document in response to an incoming Order document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
BuyerOrderNumber	copied from request document
SellerOrderNumber	copied from request document
ListOfMessageID	copied from request document
OrderResponseIssueDate	current date and time
OrderResponseDocTypeCoded	OrderResponse
OrderResponseDocTypeCodedOther	null
RefNum	“ “
RefDate	“ “
ChangeOrderReference	null
SellerParty	copied from request document
BuyerParty	copied from request document
ListOfReferenceCoded	copied from request document
Purpose	copied from request document
ResponseTypeCoded	NotAccepted

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultOrder30Doc - DocumentObject - the incoming Order document.
Outputs:
DefaultOrderresponse30Doc - DocumentObject - the default OrderResponse document.
Configurations:

package com.commerceone.xpc.components

class DefaultOrderResponseFromChangeOrder30Builder

Description

Builds a default OrderResponse document in response to an incoming ChangeOrder document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
BuyerOrderNumber	copied from request document
SellerOrderNumber	copied from request document
ListOfMessageID	copied from request document
OrderResponseIssueDate	current date and time
OrderresponseDocTypeCoded	ChangeOrderResponse
OrderResponseDocTypeCodedOther	null
OrderReference	copied from request document
RefNum	“ “
RefDate	null
SellerParty	copied from request document
BuyerParty	copied from request document
ListOfReferenceCoded	copied from request document
Purpose	copied from request document
ResponseTypeCoded	NotAccepted
ResponseTypeCodedOther	null
ChangeOrderHeader	copied from request document

Field	Setting
OrderHeaderChanges	null
OrderResponseHeaderNote	copied from request document
ListOfStructuredNote	copied from request document
itemDetailResponseCoded	New
ItemDetailResponseCodedOther	null
PriceErrorInfo	null
AvailabilityErrorInfo	null
ListOfErrorInfo	null
ListOfReferenceCoded	copied from request document
ChangeOrderItemDetail	copied from request document
ItemDetailChanges	null
LineItemNote	copied from request document
ListOfStructuredNote	copied from request document
PackageDetailResponseCoded	New
PackageDetailResponseCodedOther	null
ChangeOrderPackageDetail	copied from request document
PackageDetailChanges	null
PackageDetailNote	copied from request document
ErrorInfo	null
OriginalOrderSummary	copied from request document
RevisedOrderSummary	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultChangeOrder30Doc - DocumentObject - the incoming ChangeOrder document.
Outputs:
DefaultOrderResponse30Doc - DocumentObject - the default OrderResponse document.
Configurations:

package com.commerceone.xpc.components

class OrderStatusResponse30Builder

Description

Builds a default OrderStatusResponse document in response to an incoming OrderStatusRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
OrderStatusID	copied from request document
OrderStatusIssueDate	current date and time
OrderStatusResultParty	copied from request document
OrderStatusResultLanguage	copied from request document
OrderStatusResultNote	copied from request document
ResultListOfAttachment	copied from request document
AccountCode	copied from request document
BuyerReferenceNumber	copied from request document
SellerReferenceNumber	copied from request document
OrderReference	copied from request document
OrderDate	copied from request document
OrderStatusDate	current date and time
StatusEventCoded	configured in XPC Manager
StatusEventCodedother	null
StatusReasonCoded	Other
StatusReasonCodedOther	null

Field	Setting
StatusNote	null
ErrorInfo	null
LineItemNum	copied from request document
LineItemType	copied from request document
ParentItemNumber	copied from request document
ItemIdentifiers	copied from request document
ListOfDimension	copied from request document
TotalQuantity	copied from request document
MaxBackOrderQuantity	copied from request document
OffCatalogFlag	copied from request document
ListOfitemReferences	copied from request document
CountryOfOrigin	copied from request document
CountryOfDestination	copied from request document
FinalRecipient	copied from request document
ConditionsOfSale	copied from request document
HazardousMaterials	copied from request document
OrderStatusItemResultTransport	copied from request document
VarianceQty	copied from request document
ItemStatusQuantity	copied from request document
StatusEventCoded	configured in XPC Manager
StatusEventCodedOther	null
StatusReasonCoded	Other
StatusReasonCodedOther	null
StatusNote	null

Field	Setting
ErrorInfo	null
PaymentStatusEvent	null
ShipmentStatusEvent	null
GeneralLineItemNote	copied from request document
LineItemAttachment	copied from request document
OrderStatusCheckItemError	0
OrderStatusSummaryErrorInfo	null
TotalNumberOfLineItem	OrderStatusCheckItemError

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultOrderStatusRequestDoc - DocumentObject - the incoming OrderStatusRequest document.
Outputs:
DefaultOrderStatusResponse - DocumentObject - the default OrderStatusResponse document.
Configurations:
Status - the status of the order. “OrderStatusNotAccepted” by default.

package com.commerceone.xpc.components

class DefaultOrderStatusResponseBuilder (deprecated)

Description

Returns a default OrderStatusResponse in response to an incoming OrderStatusRequest document. For each line item whose status is being requested, the default response specifies a StatusEventCodeElement of “NoInfo,” which can be reset in the default.prop file, and the following hardcoded values:

- StatusNote of ““
- StatusEventCodeOther of ““
- StatusReasonCodeElement of StatusReasonCode._Other
- StatusReasonCodeOther of ““
- OrderStatusDate of NULL

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Gets the input document from the Data Manager, calls the buildResponse method, and sets the default AvailabilityCheckResponse document in the Data Manager.
Inputs:
DefaultOrderStatusRequestDoc - DocumentObject - the OrderStatusRequest document
Outputs:
DefaultOrderStatusResponse - DocumentObject - the default OrderStatusResult document
Configurations:
Status - the status of the order

package com.commerceone.xpc.components

class DefaultPaymentRequestAck30Builder

Description

Builds a default PaymentRequestAcknowledgment document in response to an incoming PaymentRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
PaymentRequestAcknCoded	Other
PaymentrequestAcknCodedOther	null
PaymentRequestAcknIssueDate	current date and time
PaymentRequestAcknID	“”
PaymentRequestIDReference	copied from request document
CertificateAuthority	null
SuccessfulReceptIndicator	true
GeneralNote	null
Language	copied from request document
ConfirmationID	“”
PaymentDocumentID	copied from request document
PaymentSequenceNo	copied from request document
SettlementAmount	copied from request document
DebitAmount	null
CreditAmount	null
PayerParty	copied from request document

Field	Setting
PayeeParty	copied from request document
BuyerParty	copied from request document
SupplierParty	copied from request document
BillToParty	copied from request document
ListOfPartyCoded	copied from request document
FinancialInstitutionDetail	copied from request document
ListOfRateOfExchangeDetail	copied from request document
ExceptionNote	null
PaymentRequestNote	null
ListOfPaymentException	null
EncryptedInfo	null
PaymentRequestAcknSummary	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultPaymentRequest30Doc - DocumentObject - the incoming PaymentRequest document.
Outputs:
DefaultPaymentrequestAck30Doc - DocumentObject - the default PaymentRequestAcknowledgment document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultPlanningScheduleResponse30Builder

Description

Builds a default PlanningScheduleResponse document in response to an incoming PlanningSchedule request document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
ScheduleResponseID	new UUID
ScheduleResponseIssueDate	current date and time
RefNum	copied from request document
RefDate	current date and time
ListOfReferenceCoded	null
BuyerParty	copied from request document
SellerParty	copied from request document
PurposeCode	“Other”
PurposeCodedOther	null
ResponseTypeCode	“Other”
ResponseTypeCodedOther	null
Language	copied from request document
OriginalPlanningScheduleHeader	copied from request document
ChangedPlanningScheduleHeader	null
PlanningScheduleResponseHeaderNote	null
ListOfStructuredNote	null

Field	Setting
ListOfAttachment	null
DetailResponseCoded	“Other”
DetailResponseCodedOther	null
OriginalLocationGroupedPlanningDetail OR OriginalMaterialGroupedPlanningDetail	copied from request document
ChangedLocationGroupedPlanningDetail OR ChangedMaterialGroupedPlanningDetail	null
LineItemNote	null
ListOfStructuredNote	null
ListOfAttachment	null
TotalNumberOfLineItems	the number of LocationGroupedPlanningResponse elements OR the number of MaterialGroupedPlanningResponse elements

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultPlanningScheduleDoc - DocumentObject - the incoming
Outputs:
DefaultPlanningScheduleResponseDoc - DocumentObject - the default PlanningScheduleResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultPriceCheckResponse30Builder

Description

Returns a default PriceCheckResponse in response to an incoming PriceCheckRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
PriceCheckResultID	copied from request document
PriceCheckResultIssueDate	current date and time
SupplierParty	copied from request document
SupplierPartyReferenceDate	copied from request document
BuyerParty	copied from request document
BuyerIDReferenceDate	copied from request document
PriceCheckShipToParty	copied from request document
PriceCheckCurrency	copied from request document
QuoteDate	copied from request document
PriceCheckResultLanguage	copied from request document
PriceCheckResultNote	copied from request document
ResultListOfAttachment	copied from request document
LineItemNum	copied from request document
LineItemType	copied from request document
ParentItemNumber	copied from request document
ItemIdentifiers	copied from request document

Field	Setting
ListOfDimension	copied from request document
TotalQuantity	copied from request document
MaxBackOrderQuantity	copied from request document
OffCatalogFlag	copied from request document
ListOfItemReferences	copied from request document
CountryOfOrigin	copied from request document
CountryOfDestination	copied from request document
FinalRecipient	copied from request document
ConditionsOfSale	copied from request document
HazardousMaterials	copied from request document
CheckResultTransport	copied from request document
UnitPriceValue	-1. may be reconfigured with XPC Manager
Currency	copied from request document. May be reconfigured with XPC Manager.
UOMCoded	must be configured with XPC manager
UOMCodedOther	null
PriceBasisQuantity	null
ValidityDates	null
PriceQuantityRange	null
PriceMultiplier	null
PriceErrorInfo	null
GeneralLineItemNote	copied from request document
LineItemAttachment	copied from request document

Field	Setting
PriceCheckItemError	0
PriceCheckSummaryErrorInfo	null
TotalNumberOfLineItem	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the input document from the Data Manager, calls the buildResponse method, and sets the default PriceCheckResponse document in the Data Manager.

Inputs:

DefaultPriceCheckRequestDoc - DocumentObject - the input PriceCheckRequest document

Outputs:

DefaultPriceCheckResponse - DocumentObject - the default PriceCheckResult document

Configurations:

UnitPrice - the unit price for the item

UOMCode - the unit of measurement code in which the unit price is expressed

package com.commerceone.xpc.components

class DefaultPriceCheckResponseBuilder

Description

Returns a default PriceCheckResponse in response to an incoming PriceCheckRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings:.

Field	Setting
StartDate	null
EndDate	null
WuantityRange	null
UnitPrice	-1. may be reconfigured with XPC Manager.
CurrencyCode	copied from request document. May be reconfigured with XPC Manager.
UnitOfMeasure	copied from request document. May be reconfigured with XPC Manager.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Gets the input document from the Data Manager, calls the buildResponse method, and sets the default PriceCheckResponse document in the Data Manager.
Inputs:
DefaultPriceCheckRequestDoc - DocumentObject - the input PriceCheckRequest document
Outputs:
DefaultPriceCheckResponse - DocumentObject - the default PriceCheckResult document
Configurations:
UnitPrice - the unit price for the item
UOMCode - the unit of measurement code in which the unit price is expressed

package com.commerceone.xpc.components

class DefaultPriceCheckResponseBuilder (deprecated)

Description

Returns a default PriceCheckResponse in response to an incoming PriceCheckRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings:.

Field	Setting
StartDate	null
EndDate	null
QuantityRange	null
UnitPrice	-1. may be reconfigured with XPC Manager.
CurrencyCode	copied from request document. May be reconfigured with XPC Manager.
UnitOfMeasure	copied from request document. May be reconfigured with XPC Manager.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Gets the input document from the Data Manager, calls the buildResponse method, and sets the default PriceCheckResponse document in the Data Manager.
Inputs:
DefaultPriceCheckRequestDoc - DocumentObject - the input PriceCheckRequest document
Outputs:
DefaultPriceCheckResponse - DocumentObject - the default PriceCheckResult document
Configurations:
UnitPrice - the unit price for the item
UOMCode - the unit of measurement code in which the unit price is expressed

package com.commerceone.xpc.components

class DefaultPurchaseOrderResponseBuilder (deprecated)

Description

Returns a default PurchaseOrderResponse in response to an incoming PurchaseOrderRequest document.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the input document from the Data Manager, calls the buildResponse method, and sets the default PurchaseOrderResponse document in the Data Manager.

Inputs:

DefaultPurchaseOrderRequestDoc - DocumentObject - the input PurchaseOrderRequest document

Outputs:

DefaultPurchaseOrderResponse - DocumentObject - the default PurchaseOrderResponse document

Configurations:

None

package com.commerceone.xpc.components

class DefaultQuote30Builder

Description

Builds a default Quote document in response to a RequestForQuotation document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings

Field	Setting
QuoteIssueDate	current date and time
RefNum	new UUID
RefDate	current date and time
Reference	copied from request document's RequestQuoteID field if supplied; otherwise, null
ReferenceReleaseNumber	"0" if request document's RequestQuoteID field is supplied; otherwise, null
QuoteTypeCode	"Other"
QuoteTypeCodedOther	null
QuoteParty	copied from request document
QuoteTransport	copied from request document
QuoteCurrency	copied from request document
QuoteAllowOrCharge	copied from request document
QuoteTermsOfPayment	copied from request document
QuoteTermsOfDelivery	copied from request document
TaxReference	array with 0 elements
QuoteLanguage	copied from request document

Field	Setting
GeneralNotes	null
ListOfAttachment	null
QuoteTypeCode	"Other"
QuoteTypeCodedOther	null
LineItemNum	copied from request document
LineItemType	copied from request document
ParentItemNumber	copied from request document
ItemIdentifiers	copied from request document
ListOfDimension	copied from request document
TotalQuantity	copied from request document
MaxBackOrderQuantity	copied from request document
OffCatalogFlag	copied from request document
ListOfItemReferences	copied from request document
CountryOfOrigin	copied from request document
CountryOfDestination	copied from request document
FinalRecipient	copied from request document
ConditionsOfSale	copied from request document
HazardousMaterials	copied from request document
RequestQuoteReference	null
OrderParty	null
ListOfQuotePackageDetail	copied from request document
TotalNumberOfLineItems	copied from request document from

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultRequestForQuoteDoc - DocumentObject - the incoming RequestForQuotation document.
Outputs:
DefaultQuoteDoc - DocumentObject - the default Quote document
Configurations:
None

package com.commerceone.xpc.components

class DefaultShippingScheduleResponse30Builder

Description

Builds a default ShippingScheduleResponse document in response to a ShippingSchedule request document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
ScheduleResponseID	new UUID
ScheduleResponseIssueDate	current date and time
RefNum	copied from request document
RefDate	current date and time
ListOfReferenceCoded	null
BuyerParty	copied from request document
SellerParty	copied from request document
PurposeCode	“Other”
PurposeCodedOther	null
ResponseTypeCode	“Other”
ResponseTypeCodedOther	null
Language	copied from request document
OriginalShippingScheduleHeader	copied from request document
ChangedShippingScheduleHeader	null
ShippingScheduleResponseHeaderNote	null
ListOfStructuredNote	null

Field	Setting
ListOfAttachment	null
DetailResponseCode	“Other”
DetailResponseCodedOther	null
OriginalLocationGroupedShippingDetail OR OriginalMaterialGroupedShippingDetail	copied from request document
ChangedLocationGroupedShippingDetail OR ChangedMaterialGroupedShippingDetail	null
LineItemNote	null
ListOfStructuredNote	null
ListOfAttachment	null
TotalNumberOfLineItems	number of LocationGroupedShippingResponse or MaterialGroupedShippingResponse elements

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultShippingScheduleDoc - DocumentObject - the incoming ShippingSchedule document.
Outputs:
DefaultShippingScheduleResponseDoc - DocumentObject - the default ShippingScheduleResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultTimeSeries30Builder

Description

Builds a default TimeSeries document in response to a TimeSeriesRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
TimeSeriesHeader (multiple fields)	all fields copied from request document
ListOfCharacteristicCombinations	copied from request document
TimeSeriesKeyFigurePurposeCoded	copied from request document
TimeSeriesKeyFigurePurposeCodedOther	copied from request document
TimeSeriesKeyFigureResponseCoded	null
TimeSeriesKeyFigureResponseCodedOther	null
CharacteristicCombinationID	copied from request document
KeyFigureInformation	copied from request document
UnitOfMeasurement	copied from request document
StartDate	current date and time
EndDate	current date and time
TimeSeriesValue	-1.0
TimeSeriesDataNote	null
TimeSeriesSummary	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultTimeSeriesRequest30Doc - DocumentObject - the incoming TimeSeriesRequest document.
Outputs:
DefaultTimeSeries30Doc - DocumentObject - the default TimeSeries
Configurations:
None

package com.commerceone.xpc.components

class DefaultTimeSeriesResponse30Builder

Description

Builds a default TimeSeriesResponse document in response to an incoming TimeSeriesRequest document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
TimeSeriesResponseIssueDate	current date and time
RefNum	“ “
RefDate	null
TimeSeriesReference	copied from request document
TimeSeriesPlanningData	copied from request document
TimeSeriesResponseParty	copied from request document
TimeSeriesResponseCodedOther	copied from request document
TimeSeriesHeaderResponseCodedOther	“ “
ChangedTimeSeriesHeader	copied from request document
Language	copied from request document
GeneralNotes	copied from request document
TimeSeriesDetailResponseCoded	copied from request document
TimeSeriesDetailResponseCodedOther	“ “
ListOfChangedCharacteristicCombinations	copied from request document
ListOfChangedTimeSeriesKeyFigureData	copied from request document
TimeSeriesResponseSummary	copied from request document

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultTimeSeries30Doc - DocumentObject - the incoming TimeSeries document.
Outputs:
DefaultTimeSeriesResponse30Doc - DocumentObject - the default TimeSeriesResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultTPRResponseFromOrganizationDelete30Builder

Description

Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerOrganizationDelete document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings

Field	Setting
RefNum	“ “
RefDate	null
PrimaryReturnCoded	PrimaryReturnCode.00
PrimaryReturnCodeDescription	null
ListOfSecondaryMessageInformation	null
TradingPartnerPrimaryID	copied from request document
AlternateID	null
UserID	null
ReturnedIdentificationURN	null
RedirectURL	SURI
ServiceID	null

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultTPOrganizationDelete30Doc - DocumentObject - the incoming TradingPartnerOrganizationDelete document.
Outputs:
DefaultTradingPartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultTPRResponseFromOrganizationInfo30Builder

Description

Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerOrganizationInformation document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
RefNum	" "
RefDate	null
PrimaryReturnCoded	PrimaryReturnCode.00
PrimaryReturnCodeDescription	null
ListOfSecondaryMessageInformation	null
TradingPartnerPrimaryID	copied from request document
AlternateID	null
UserID	null
ReturnedIdentificationURN	null
RedirectURL	SURI
ServiceID	null

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultTPOrganizationInformation30Doc - DocumentObject - the incoming TradingPartnerOrganizationInformation document.
Outputs:
DefaultTradingPartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultTPResponseFromUserDelete30Builder

Description

Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerUserDelete document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
RefNum	“ “
RefDate	null
PrimaryReturnCoded	PrimaryReturnCode.00
PrimaryReturnCodeDescription	null
ListOfSecondaryMessageInformation	null
TradingPartnerPrimaryID	copied from request document
AlternateID	null
UserID	copied from request document
ReturnedIdentificationURN	null
RedirectURL	SURI
ServiceID	null

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultTPUserDelete30Doc -DocumentObject - the incoming TradingPartnerUserDelete document.
Outputs:
DefaultTradingPartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document.
Configurations:
None

package com.commerceone.xpc.components

class DefaultTPRResponseFromUserInfo30Builder

Description

Builds a default TradingPartnerResponse document in response to an incoming TradingPartnerUserInformation document.

Optional fields in the response document are filled with nulls. Required fields are either copied from the request document, set to a constant value such as -1 or set to the current ID or date and time.

The following table lists the field settings.

Field	Setting
RefNum	“ “
RefDate	null
PrimaryReturnCoded	PrimaryReturnCode.00
PrimaryReturnCodeDescription	null
ListOfSecondaryMessageInformation	null
TradingPartnerPrimaryID	copied from request document
AlternateID	null
UserID	copied from request document
ReturnedIdentificationURN	null
RedirectURL	SURI
ServiceID	null

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
DefaultTPUserInformation30Doc - DocumentObject - the incoming TradingPartnerUserInformation document.
Outputs:
DefaultTradingpartnerResponse30Doc - DocumentObject - the default TradingPartnerResponse document.
Configurations:
None

package com.commerceone.xpc.components

class ExceptionHandler

Description

This standard component logs an event and builds an Error document based on a previously thrown exception.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Logs an event and builds an Error document based on a previously thrown exception. Returns the result of the component's execution.

Inputs:

exception - Exception - a previously thrown exception

componentName - String - the component that threw the exception

Outputs:

replyDocument - Document - an Error document based on the exception

Configurations:

None

package com.commerceone.xpc.components

class FileStore

Description

This extended component provides methods for storing envelopes, request documents, and attachments in directories on the file system and for reading the archived files. Both the archive directories and the filename formats can be configured.

Extends

XPCAbstractComponent

Class-wide Configurations

- Overwrite - determines whether new files overwrite existing files with the same name. Three settings--no, yes, or history--are available.
  - A “no” setting throws an error if an existing file with the specified name is found.
  - A “yes” setting archives the most recent version of the file by appending a dollar sign ($) to its file name.
  - A “history” setting archives **all** versions of the file by appending a dollar sign and version number to the most recent version’s file name is appended by a dollar sign ($) and version number to an existing file when it is overwritten. For example, when the file xyz is overwritten, the first version of the file is renamed xyz$1. The next time xyz is overwritten, the second version of the file is renamed xyz$2. The file named xyz always contains the most current version.
- RootDirectory - the root archive directory. This directory is used for all configurations.

Methods

public XPCResult storeEnvelope(XPCDataMgr dataMgr)

Stores an envelope, request document and attachments.

Inputs:

- envelopeToStore - Envelope - the envelope to be stored
- correlationKey - String - the correlation key

Outputs:

- None

Configurations:

- Document.Directory - directory in which the document is archived. The default directory name varies by XPC service and reflects the type of document being archived. For services that archive request documents, a default directory called “default_request” is used. For services that archive response documents, a default directory called “default_response” is used.
- Envelope.Directory - directory (by default, envelope) in which the envelope is stored.
- Envelope.Document.Directory - directory in which the document is stored. The default directory name varies by XPC service and reflects the type of document being archived. For services that archive response documents, a default directory called “response” is used. For services that archive request documents, the default directory name is “request.” For services that archive one-way documents, a default directory called “oneway” is used.
- Envelope.Attachment.Directory - directory (by default, attachment) in which the attachments are stored
- Envelope.Attachment.Description.Directory - directory (by default, meta) in which descriptions of attachments are stored
- Envelope.Prefix - string with which each envelope file name begins. By default, this is the name of the document being archived followed by an underscore (for example, Quote_ or AvailabilityToPromiseResponse_).
- Envelope.Document.Prefix - string with which each document file name begins. By default, this is the name of the document being archived followed by an underscore (for example, Quote_ or AvailabilityToPromiseResponse_).

(Configurations continued on next page)

- Envelope.Attachment.Prefix - string with which each attachment file name begins. By default, this is the name of the document being archived followed by an underscore (for example, Quote_ or AvailabilityToPromiseResponse_).
- Envelope.Attachment.Description.Prefix - string with which each attachment description file name begins. By default, this is the name of the document being archived followed by an underscore (for example, Quote_ or AvailabilityToPromiseResponse_).
- Envelope.Attachment.NameURI.[Name].Directory - the attachment directory naming convention, which is based upon attachment name
- Envelope.Document.Extension - string (by default, .xml) with which each document file name ends
- Envelope.Extension - string (by default, .env) with which each envelope file name ends
- Envelope.Attachment.Extension - string (by default, .att) with which each attachment file name ends
- Envelope.Attachment.Description.Extension - string (by default, .adf) with which each attachment description file name ends

public XPCResult storeStream(XPCDataMgr dataMgr)

Stores the input stream.

Inputs:

- streamToStore - InputStream - streamed data
- correlationKey - String - correlation key

Outputs:

- None

Configurations:

- Stream.Directory - the directory in which the streamed input is stored.
- Stream.Prefix - string (by default, D_) with which the stream file name begins.
- Stream.Extension - string (by default, .xml) with which the stream file name ends.

public XPCResult storeDocument(XPCDataMgr dataMgr)

Stores the input document.

Inputs:

- documentToStore - Document - the input document
- correlationKey - String - the correlation key

Outputs:

- None

Configurations:

- Document.Directory - directory in which the document will be stored
- Document.Prefix - string (by default, D_) with which the document file name begins
- Document.Extension - string (by default, .xml) with which the document file name ends

public XPCResult readStream(XPCDataMgr dataMgr)

Reads a file from a directory. Outputs a streamed input file and a correlation key to the Stream.Directory. Archives the file to a separate Stream.Archive.Directory if one is defined.

Inputs:

- None

Outputs:

- streamRead - InputStream - the streamed input file
- correlationKey - String - the correlation key

Configurations:

- Stream.Directory - the directory in which the streamed input file will be stored.
- Stream.Archive.Directory - the directory in which the file will be archived once it has been streamed.
- Stream.prefix - string (by default, D_) with which the stream file name begins.
- Stream.Extension - string (by default, .xml) with which the stream file name ends.

public XPCResult readDocument(XPCDataMgr dataMgr)

Reads a document from a directory. Outputs the document along with the correlation key. Archives the document to a separate Document.Archive.Directory if one is defined.

Inputs:

- None

Outputs:

- documentRead - Document - the document to be read
- correlationKey - String - the correlation key of the document

Configurations:

- Document.Directory - directory in which the document is stored. The default directory name varies by XPC service and reflects the type of document being archived. For services that archive request documents, a default directory called “default_request” is used. For services that archive response documents, a default directory called “default_response” is used.
- Document.Archive.Directory - the directory in which the document will be archived
- Document.Prefix - string (by default, D_) with which the document file name begins
- Document.Extension - string (by default, .xml) with which the document file name ends

public XPCResult lookupEnvelope(XPCDataMgr, dataMgr)

Reads the envelope with the specified correlation key from the file system. Archives the envelope to a separate Envelope.Archive.Directory if one is defined.

Inputs:

- correlationKey - String - the correlation key of the envelope

Outputs:

- lookupEnvelope - Envelope - the archived envelope

Configurations:

- Envelope.Directory - the directory in which the envelope is currently stored
- Envelope.Archive.Directory - the directory in which the envelope archive will be stored
- Envelope.Prefix - string (by default, E_) with which each envelope file name begins
- Envelope.Extension - string (by default, .env) with which each envelope file name ends

public XPCResult copyFile(XPCDataMgr dataMgr))
This extended method copies a file named by the input string from the File.Source.Directory to the File.Target.Directory defined in the configuration file. File name format will be [Prefix][filename key][extension].

Inputs:

- filenameKey - String - the correlation key of the file to be copied

Outputs:

- None

Configurations:

- File.Source.Directory - the directory in which the original file is located
- File.Target.Directory - the directory into which the file will be copied
- File.Prefix - a set of characters to be added at the beginning of the file name specified by filenameKey
- File.Extension - a set of characters to be added at the end of the file name specified by filenameKey

package com.commerceone.xpc.components

class GetCorrelationKey

Description

Returns the correlation key of an envelope.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Generates a correlation key that can be used to identify a transaction. If xPath is set, key string is the value corresponding to the xPath in the root document of the envelope. If xPath is not set, correlation id of the envelope will be the output.

Inputs

envelope - Envelope - the request envelope

Outputs

correlationKey - String - the correlation ID of the envelope

Configurations

KeyProperty - determines whether the correlation key is generated from the incoming envelope's CorrelationId (the default) or its MessageId. The Xpath configuration may be used to specify a correlation key other than the CorrelationId or MessageId; this setting takes precedence over the KeyProperty setting.

Xpath - a correlation key specification that reflects the values of one or more xCBL elements or attributes. The specification is constructed from one or more xpath-like strings, each of which represents the path to an xCBL element or attribute. Nodes within the path are separated by forward slashes (/), attribute names are preceded by the at symbol (@), and the entire xPath string is surrounded by angle brackets (< and >). The string <PurchaseOrder/OrderHeader/OrderReference/BuyerRefNum>, for example, represents the value of the BuyerRefNum element in the PurchaseOrder document.

If an xPath configuration contains more than one xpath string, individual strings are separated by one or more characters. With the exception of the angle bracket characters, any characters may be used to separate consecutive xpath strings. In the following xPath configuration, for example, the underscore character (_) is used to separate the two xpath strings: <PurchaseOrder/OrderHeader/POIssuedDate>_<PurchaseOrder/OrderHeader/OrderReference/BuyerRefNum>

The resulting xPath configuration consists of the date the purchase order was issued followed by an underscore character and the buyer's purchase order number (for example: 200000805T01:01:01_123456789)

package com.commerceone.xpc.components

class GetStringFromDocument

Description

Reads the correlation key from an xCBL document and returns it to the Data Manager.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Reads the correlation key from an xCBL document and returns it to the Data Manager. The xPath configuration is used to specify which portion of the document is used as the correlation key. The DefaultString configuration is used may to specify the default correlation key returned when the xPath configuration is not set or returns an empty string.

Inputs:

document - DocumentObject - the xCBL document

Outputs:

correlationKey - String - the correlation key of the xCBL document

Configurations:

DefaultString - an alternative correlation key to be output if xPath is not configured or returns an empty string

xPath - a correlation key specification that reflects the values of one or more xCBL elements or attributes. The specification is constructed from one or more xpath-like strings, each of which represents the path to an xCBL element or attribute. Nodes within the path are separated by forward slashes (/), attribute names are preceded by the at symbol (@), and the entire xPath string is surrounded by angle brackets (< and >). The string <PurchaseOrder/OrderHeader/OrderReference/BuyerRefNum>, for example, represents the value of the BuyerRefNum element in the PurchaseOrder document.

If an xPath configuration contains more than one xpath string, individual strings are separated by one or more characters. With the exception of the angle bracket characters, any characters may be used to separate consecutive xpath strings. In the following xPath configuration, for example, the underscore character (_) is used to separate the two xpath strings: <PurchaseOrder/OrderHeader/POIssuedDate>_<PurchaseOrder/OrderHeader/OrderReference/BuyerRefNum>

The resulting xPath configuration consists of the date the purchase order was issued followed by an underscore character and the buyer's purchase order number (for example: 200000805T01:01:01_123456789).

package com.commerceone.xpc.components

class LookupXCCArchive

Description

Retrieves the archived envelope whose referenceId matches that of the input envelope. If no match is found, retrieves the archived envelope whose correlationId matches that of the input envelope. If neither match is found, returns an outputEnvelope of null.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
inputEnvelope - Envelope - the envelope containing the error document
Outputs:
outputEnvelope - Envelope - the archived envelope whose correlation key matches that of the input envelope.
correlationKey - String - the correlation key
Configurations:
None

package com.commerceone.xpc.components

class MessageAcknowledgmentSender

Description

Creates an envelope containing a MessageAcknowledgement document and transmits it to the sender of the original document.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Inputs:
inputEnvelope - Envelope - the envelope containing the original document
Outputs:
None
Configurations:
None

package com.commerceone.xpc.components

class Responder

Description

This standard component sends a response to a previously-received request envelope.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Sends a response to a previously-received request envelope. Gets the request envelope and response document from the Data Manager. Returns the result of the component's execution.
Inputs:
replyDocument - DocumentObject - the response document
Outputs:
None
Configurations:
None

package com.commerceone.xpc.components

class StreamToDocument

Description

This standard component converts streamed input to an xCBL document.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Converts the input stream to an xCBL document.
Inputs:
streamRead - InputStream - streamed input
Outputs:
document - DocumentObject - an xCBL document
Configurations:
None

package com.commerceone.xpc.components

class Transmitter

Description

Gets the request envelope from the data manager and transmits it to its destination.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Gets the request envelope from the data manager and transmits it to its destination.
Inputs:
requestEnv - Envelope - the request envelope
Outputs:
None
Configurations:
None

package com.commerceone.xpc.helpers

Description

This package contains classes whose methods can be used to access the information in request documents and to build both the data portion and the error portion of response documents.

XPCDocHandle	Provides methods for traversing documents, obtaining the values of specified fields, and setting the values of fields. In practice, this is used to obtain the values of key fields from xCBL request documents, such as the PriceCheckRequest document, and to set the values of corresponding fields in xCBL response documents, such as the PriceCheckResponse document.
XPCErrorInfo	Builds the error information portion of a response document.

package com.commerceone.xpc.helpers

class XPCDocHandle

Description

Provides methods for traversing documents, obtaining the values of specified fields, and setting the values of fields. In practice, this is used to obtain the values of key fields from xCBL request documents, such as the PriceCheckRequest document, and to set the values of corresponding fields in xCBL response documents, such as the PriceCheckResponse document.

Methods

public Object get(final String atPath)

Returns a node from the xCBL request document.

The method takes as its input an XPath string to the xCBL element or attribute whose value will be returned. The string has the same structure as the xCBL request document, and includes each node that must be traversed to get to the desired element or attribute. Nodes are separated by forward slashes (/). Attribute names are preceded by the at symbol (@)

For example, get("PriceCheckRequest/PriceCheckRequestDetail/ ListOfPriceCheckRequestItemDetail/PriceCheckRequestItemDetail/ PriceCheckRequestBaseItemDetail/ItemIdentifiers/PartNumbers/SellerPartNumber/ PartNum/PartID") returns an array containing the suppliers part id of each item as specified in the PriceCheckRequest document. The method selects the PartID element:
from the PartNum element
from the SellerPartNumber element
from the PartNumbers element
from the ItemIdentifiers element
from the PriceCheckRequestBaseItemDetail element
from the PriceCheckRequestItemDetail element
from the ListOfPriceCheckRequestItemDetail element
from the PriceCheckRequestDetail element
from the PriceCheckRequest input document

Rather than specifying complete paths to each element or attribute, as above, you can create symbolic names for these paths in your integrators. For example, the following entry in the sample myPriceCheckIntegrator30 creates a variable, XPATH_PARTID_SUPPLIER, whose value is the path to the PartID element in the PriceCheckRequest document:

```
static private final String XPATH_PARTID_SUPPLIER = "PriceCheckRequest/
PriceCheckRequestDetail/ListOfPriceCheckRequestItemDetail/
PriceCheckRequestItemDetail/PriceCheckRequestBaseItemDetail/ItemIdentifiers/
PartNumbers/SellerPartNumber/PartNum/PartID";
```

The variable name, XPATH_PARTID_SUPPLIER, can then be specified as the input string in get(XPATH_PARTID_SUPPLIER).

public Object set(final String atPath, final Object objParams)

Sets the value of a node in the xCBL response document.

The method takes two arguments. The first specifies the path to the xCBL element(s) or attribute(s) whose value will be set. The second specifies the value, or array of values, to be assigned to these element(s) or attribute(s).

The path to the xCBL element or attribute is specified by an Xpath string with the same structure as the xCBL result document. The string includes each node that must be traversed to get to the desired element or attribute. Nodes are separated by forward slashes (/). Attribute names are preceded by the at symbol (@)

For example, set("PriceCheckResult/PriceCheckResultDetail/ListOfPriceCheckResultItemDetail/PriceCheckResultItemDetail/ResultPrice/Price/UnitPrice/UnitPriceValue", resultPrices) sets the unit price for each item whose price was requested. For each item, the method sets the UnitPriceValue element:
of the UnitPrice element
within the Price element
within the ResultPrice element
within the PriceCheckResultItemDetail element
within the ListOfPriceCheckResultItemDetail element
within the PriceCheckResultDetail element
within the PriceCheckResult response document

Rather than specifying complete paths to each element or attribute, as above, you can create symbolic names for these paths in your integrators. For example, the following entry in the sample myPriceCheckIntegrator30 creates a variable, XPATH_RESULT_PRICE, whose value is the path to the UnitPriceValue element in the PriceCheckResult document:

```
static private final String XPATH_RESULT_PRICE = "PriceCheckResult/PriceCheckResultDetail/ListOfPriceCheckResultItemDetail/PriceCheckResultItemDetail/ResultPrice/Price/UnitPrice/UnitPriceValue";
```

The variable name, XPATH_RESULT_PRICE, can then be specified as the first argument in the set method, as below:
set(XPATH_RESULT_PRICE, resultPrices)

package com.commerceone.xpc.helpers

class XPCErrorInfo

Description

Builds the error information portion of a response document.

Constructors

public XPCErrorInfo()
Creates a new string buffer for the error code.
Creates a new string buffer for the error message to be displayed to the user.
Creates a new string buffer for the vendor-specific error message that may be used for troubleshooting.
Sets a default value of 0 for the minimum number of seconds to wait before retrying the request.
Sets withParams to NULL, indicating that there are no other error parameters.

Methods

public boolean isSet()
Returns “false” if the length of the ErrCode is 0, “false” otherwise.

final public StringBuffer getCode()
Returns the error code.

final public StringBuffer getMessage()
Returns an error message suitable for display to application users.

final public StringBuffer getVendorRef()
Returns an error message suitable for troubleshooting.

final public int getSeverity()
Returns the severity of the error.

final public int getMinRetry()
Returns the minimum number of seconds to wait before trying the request again.

final public String[] getParams()
Returns other error parameters.

public ErrorInfo buildErrorInfo()
Builds a new ErrorInfo CBL element for inclusion in the response document.

package com.commerceone.xpc.my_integrators

Description

Contains working samples of components used to send default responses to requests for item availability, item price, or order status. You can use these components as models when extending XPCAbstractComponent to build your own integrations. Unlike the sample components in this class, your own components should issue response documents with actual availability, price, or order status information retrieved from your backend system.

myAvailabilityCheckIntegrator30	This sample component is provided for you to use as a model when developing your own Availability Check integration.
myAvailabilityCheckIntegrator (deprecated)	This sample component is provided for you to use as a model when developing your own Availability Check integration.
myOrderStatusIntegrator30	This sample component is provided for you to use as a model when developing your own Order Status integration.
myOrderStatusIntegrator (deprecated)	This sample component is provided for you to use as a model when developing your own Order Status integration.
myPriceCheckIntegrator30	This sample component is provided for you to use as a model when developing your own Price Check integration.
myPriceCheckIntegrator (deprecated)	This sample component is provided for you to use as a model when developing your own Price Check integration.

package com.commerceone.xpc.my_integrators

class myAvailabilityCheckIntegrator30

Description

This class provides methods that may be used to access the contents of an AvailabilityCheckRequest document, determine the availability of each item, and return an AvailabilityCheckResponse document. You may use these methods as models when building your own Availability Check integration. Unlike these methods, however, your own methods should extract actual data from your backend system and use it to build your AvailabilityCheckResponse document.

Extends

XPCAbstractComponent

Methods

private XPCResult doAvailabilityCheck
(final String acctCode_Buyer,
final String partID_Supplier,
final String partExt_Supplie
final String quantity,
final String uomCode,
StringBuffer resultQuantity,
StringBuffer resultUOM,
StringBuffer errorCode,
StringBuffer errorMessage,
StringBuffer errorVendorMessage)

You may use this sample helper method as a model when building your own Availability Check integration. It retrieves key information from the incoming AvailabilityCheckRequest document, displays information about the part whose availability is being checked, and returns a default AvailabilityCheckResponse document.

When building your own integration, you must add custom code to retrieve the available quantity and unit of measurement from your backend system and set these fields in the AvailabilityCheckResponse document. Your custom code must handle any errors in the request document by setting the optional error code, error message, and other error information in the ErrorInfo portion of the AvailabilityCheckResponse document.

Inputs:

acctCode_Buyer - String - the buyer's account code
partID_Supplier - String - the supplier's part number for the item
partExt_Supplier - String - the supplier's part number extension for the item
quantity - String - the requested quantity
uomCode - String - the unit of measurement in which the quantity is expressed

Outputs:

resultQuantity - StringBuffer - the available quantity of the item
resultUOM - StringBuffer - the unit of measurement in which the available quantity is expressed
errorCode - StringBuffer - the error code, if any
errorMessage - StringBuffer - a descriptive error message suitable for display to users
vendorErrorMessage - StringBuffer - a vendor-specific error message suitable for troubleshooting

public XPCResult process(XPCDataMgr dataMgr)

Calls doAvailabilityCheck helper method once for each item whose availability is being checked.

Inputs:

RequestDoc - XCBL30_sox.AvailabilityRequest - the AvailabilityCheckRequest document

ResultDoc - XCBL30_sox.AvailabilityResult - the default AvailabilityCheckResult document

Outputs:

ResultDoc - XCBL30_sox.AvailabilityResult - the AvailabilityCheckResult document

Configurations:

None

class myAvailabilityCheckIntegrator (deprecated)

Description

This class provides methods that may be used to access the contents of an AvailabilityCheckRequest document, determine the availability of each item, and return an AvailabilityCheckResponse document. You may use these methods as models when building your own Availability Check integration. Unlike these methods, however, your own methods should extract actual data from your backend system and use it to build your AvailabilityCheckResponse document.

Extends

XPCAbstractComponent

Methods

private XPCResult doAvailabilityCheck

You may use this sample helper method as a model when building your own Availability Check integration. It retrieves key information from the incoming AvailabilityCheckRequest document, displays information about the part whose availability is being checked, and returns a default AvailabilityCheckResponse document.

When building your own integration, you must add custom code to retrieve the available quantity and unit of measurement from your backend system and set these fields in the AvailabilityCheckResponse document. Your custom code must handle any errors in the request document by setting the optional error code, error message, and other error information in the ErrorInfo portion of the AvailabilityCheckResponse document.

Inputs:

acctCode_Buyer - String - the buyer's account code
partID_Buyer - String - the buyer's part number for the item
partExt_Buyer - String - the buyer's part number extension for the item
partID_Supplier - String - the supplier's part number for the item
partExt_Supplier - String - the supplier's part number extension for the item
quantity - String - the requested quantity
uomCode - String - the unit of measurement in which the quantity is expressed

Outputs:

resultQuantity - StringBuffer - the available quantity of the item
resultUOM - StringBuffer - the unit of measurement in which the available quantity is expressed
errorCode - StringBuffer - the error code, if any
errorMessage - StringBuffer - a descriptive error message suitable for display to users
vendorErrorMessage - StringBuffer - a vendor-specific error message suitable for troubleshooting

public XPCResult process(XPCDataMgr dataMgr)

Calls doAvailabilityCheck helper method once for each item whose availability is being checked.

Inputs:

RequestDoc - CBL_sox.AvailabilityRequest - the AvailabilityCheckRequest document
ResultDoc - CBL_sox.AvailabilityResult - the default AvailabilityCheckResult document

Outputs:

ResultDoc - CBL_sox.AvailabilityResult - the AvailabilityCheckResult document

Configurations:

None

package com.commerceone.xpc.my_integrators

class myOrderStatusIntegrator30

Description

This class provides methods that may be used to access the contents of an OrderStatusRequest document, determine the status of the order, and return an OrderStatus Response document. You may use these methods as models when building your own Order Status integration. Unlike these methods, however, your own methods should extract actual data from your backend system and use it to build your OrderStatusResponse document.

Extends

XPCAbstractComponent

Methods

private XPCResult doOrderStatus(final String accountCode,
final String refNum_Buyer,
final String orderDate,
StringBuffer resultStatusCode,
StringBuffer resultStatusNote,
StringBuffer resultStatusDate,
StringBuffer errorCode,
StringBuffer errorMessage,
StringBuffer errorVendorMessage)

You may use this sample helper method as a model when building your own Order Status integration. It retrieves key information from the incoming OrderStatusRequest document, displays information about the line item whose status is being checked, and returns a default OrderStatusResponse document.

When building your own integration, you must add custom code to retrieve the status from your backend system and set this information in the OrderStatusResponse document. Your custom code must handle any errors in the request document by setting the optional error code, error message, and other error information in the ErrorInfo portion of the OrderStatusResponse document.

Inputs:

- accountCode - String - the buyer's account code
- refNum_Buyer - String - the buyer's reference number for the order
- refNum_Supplier - String - the supplier's reference number for the order
- orderDate - String - the date the item was ordered

Outputs:

- resultStatusCode - StringBuffer - the status code
- resultStatusNote - StringBuffer - a description of the status
- resultStatusDate - StringBuffer - the date associated with the status
- errorCode - StringBuffer - error code if any
- errorMessage - StringBuffer - an error message suitable for user display
- errorVendorMessage - StringBuffer - an error message suitable for troubleshooting

public XPCResult process(XPCDataMgr dataMgr)
Calls the doOrderStatus helper method once for each item whose status is being checked.
Inputs:
RequestDoc - XCBL30_sox.OrderStatusRequest - the OrderStatusRequest document
ResultDoc - XCBL30_sox.OrderStatusResult - the default OrderStatusResult document
Outputs:
ResultDoc - XCBL30_sox.OrderStatusResult - the OrderStatusResult document
Configurations:
None

package com.commerceone.xpc.my_integrators

class myOrderStatusIntegrator (deprecated)

Description

This class provides methods that may be used to access the contents of an OrderStatusRequest document, determine the status of the order, and return an OrderStatus Response document. You may use these methods as models when building your own Order Status integration. Unlike these methods, however, your own methods should extract actual data from your backend system and use it to build your OrderStatusResponse document.

Extends

XPCAbstractComponent

Methods

```
private XPCResult doOrderStatus(final String account Code, // [IN] Buyer account code
final String refNum_Buyer, // [IN] Buyer's ref num for the order
//final String refNum_Supplier, // [IN] Supplier's ref num for the order
final String orderDate // [IN] Ordered Date
StringBuffer resultStatusCode, // [OUT] Order Status code
StringBuffer resultStatusNote, // [OUT] Descriptive string for Status
StringBuffer resultStatusDate, // [OUT] date of the status
StringBuffer errorCode, // [OUT,OPTIONAL] Error code if any
StringBuffer errorMessage, // [OUT,OPTIONAL] Descriptive error Message
StringBuffer errorVendorMessage  // [OUT,OPTIONAL] vendor specific error message
)
```

You may use this sample helper method as a model when building your own Order Status integration. It retrieves key information from the incoming OrderStatusRequest document, displays information about the line item whose status is being checked, and returns a default OrderStatusResponse document.

When building your own integration, you must add custom code to retrieve the status from your backend system and set this information in the OrderStatusResponse document. Your custom code must handle any errors in the request document by setting the optional error code, error message, and other error information in the ErrorInfo portion of the OrderStatusResponse document.

Inputs:

- accountCode - String - the buyer's account code
- refNum_Buyer - String - the buyer's reference number for the order
- refNum_Supplier - String - the supplier's reference number for the order
- orderDate - String - the date the item was ordered

Outputs:

- resultStatusCode - StringBuffer - the status code
- resultStatusNote - StringBuffer - a description of the status
- resultStatusDate - StringBuffer - the date associated with the status
- errorCode - StringBuffer - error code if any
- errorMessage - StringBuffer - an error message suitable for user display
- errorVendorMessage - StringBuffer - an error message suitable for troubleshooting

public XPCResult process(XPCDataMgr dataMgr)
Calls the doOrderStatus helper method once for each item whose status is being checked.
Inputs:
RequestDoc - CBL_sox.OrderStatusRequest - the OrderStatusRequest document
ResultDoc - CBL_sox.OrderStatusResult - the default OrderStatusResult document
Outputs:
ResultDoc - CBL_sox.OrderStatusResult - the OrderStatusResult document
Configurations:
None

package com.commerceone.xpc.my_integrators

class myPriceCheckIntegrator30

Description

This class provides methods that may be used to access the contents of a PriceCheckRequest document, determine the price of each item, and return a PriceCheckResponse document. You may use these methods as models when building your own PriceCheck integration. Unlike these methods, however, your own methods should extract actual data from your backend system and use it to build your Price CheckResponse document.

Extends

XPCAbstractComponent

Methods

private XPCResult doPriceCheck
(final String acctCode_Buyer,
final String partID_Supplier,
final String partExt_Supplier,
final String quantity,
final String uomCode,
StringBuffer resultPrice,
StringBuffer resultCurrency,
StringBuffer errorCode,
StringBuffer errorMessage,
StringBuffer errorVendorMessage)

You may use this sample helper method as a model when building your own Price Check integration. It retrieves key information from the incoming PriceCheckRequest document, displays information about the part whose availability is being checked, and returns a default PriceCheckResponse document.

When building your own integration, you must add custom code to retrieve the available quantity and unit of measurement from your backend system and set these fields in the PriceCheckResponse document. Your custom code must handle any errors in the request document by setting the optional error code, error message, and other error information in the ErrorInfo portion of the PriceCheckResponse document.

Inputs:

- acctCode_Buyer - String - the buyer's account code
- partID_Supplier - String - the supplier's part number for the item
- partExt_Supplier - String - the supplier's part number extension for the item
- quantity - String - the requested quantity of the item
- uomCode - String - the code for the unit in which the requested quantity is expressed

Outputs:

- resultPrice - StringBuffer - the supplier's unit price for the item
- resultCurrency - StringBuffer - the currency code in which the supplier's price is expressed.
- errorCode - StringBuffer - the error code
- errorMessage - StringBuffer - an error message suitable for display to users
- errorVendorMessage - StringBuffer - an error message suitable for debugging purposes

public XPCResult process(XPCDataMgr dataMgr)
Calls the doPriceCheck helper method once for each item whose price is being checked.
Inputs:
RequestDoc - XCBL30_sox.PriceCheckRequest - the PriceCheckRequest document
ResultDoc - XCBL30_sox.PriceCheckResult- the default PriceCheckResult document
Outputs:
ResultDoc - XCBL30_sox.PriceCheckResult - the PriceCheckResult document
Configurations:
None

package com.commerceone.xpc.my_integrators

class myPriceCheckIntegrator (deprecated)

Description

This class provides methods that may be used to access the contents of a PriceCheckRequest document, determine the price of each item, and return a PriceCheckResponse document. You may use these methods as models when building your own PriceCheck integration. Unlike these methods, however, your own methods should extract actual data from your backend system and use it to build your Price CheckResponse document.

Extends

XPCAbstractComponent

Methods

private XPCResult doPriceCheck
(final String acctCode_Buyer, // [IN] Buyer Account Code
final String partID_Supplier, // [IN] Supplier's PartId
final String partExt_Supplier, // [IN] Supplier's PartId extension
final String quantity, // [IN] Requested Quantity
final String uomCode, // [IN] UOM
StringBuffer resultPrice, // [OUT] Supplier's Price
StringBuffer resultCurrency, // [OUT] Currency of Supplier's Price
StringBuffer errorCode, // [OUT,OPTIONAL] ErrorCode if any
StringBuffer errorMessage, // [OUT,OPTIONAL] Descriptive Error Message
StringBuffer errorVendorMessage // [OUT,OPTIONAL] vendor Error Message
)

Determines the price of a single line item. Returns string buffers containing either the item's price and its currency code or information about errors that occurred while checking the price.

Inputs:

- acctCode_Buyer - String - the buyer's account code
- partID_Supplier - String - the supplier's part number for the item
- partExt_Supplier - String - the supplier's part number extension for the item
- quantity - String - the requested quantity of the item
- uomCode - String - the code for the unit in which the requested quantity is expressed

Outputs:

- resultPrice - StringBuffer - the supplier's unit price for the item
- resultCurrency - StringBuffer - the currency code in which the supplier's price is expressed.
- errorCode - StringBuffer - the error code
- errorMessage - StringBuffer - an error message suitable for display to users
- errorVendorMessage - StringBuffer - an error message suitable for debugging purposes

---

public XPCResult process(XPCDataMgr dataMgr)

Calls the doPriceCheck helper method once for each item whose price is being checked.

Inputs:

RequestDoc - CBL_sox.PriceCheckRequest - the PriceCheckRequest document
ResultDoc - CBL_sox.PriceCheckResult- the default PriceCheckResult document

Outputs:

ResultDoc - CBL_sox.PriceCheckResult - the PriceCheckResult document

Configurations:

None

package com.commerceone.xpc.swi.common

Description

This package contains an interface implemented by all XPC components. Its methods are used to retrieve the input and output argument descriptors.

Interfaces

XPCContract	This interface is implemented by all XPC components. Its methods return descriptions of the component's methods' input and output arguments.

package com.commerceone.xpc.swi.common

interface XPCContract

Description

This interface is implemented by all XPC components.

Implemented By

XPCAbstractComponent

Methods

public XPCContractDescriptor[] getInputList(String methodName) Returns an array of contract descriptors for the input arguments of the specified method. methodName - the name of the component's execution method ("" for standard components that use the process method).
public XPCContractDescriptor[] getOutputList(String methodName) Returns an array of contract descriptors for the output arguments of the specified method. methodName - the name of the component's execution method ("" for standard components that use the process method).

package com.commerceone.xpc.swi.framework

Description

This package contains a number of interfaces that are implemented by each XPC component. These interfaces provide methods for naming the component, setting its configuration parameters, invoking its execution method, and transmitting request envelopes and response documents.

XPCConfig	This interface is implemented by all XPC components. Its methods retrieve configuration parameters and use them to configure the component.
XPCProcess	This interface is implemented by all XPC standard components. Its process() method is invoked by the Action Director.
XPCTransmit	This interface is implemented by the XPCAbstractComponent subclass.

package com.commerceone.xpc.swi.framework

interface XPCConfig

Description

This interface is implemented by all XPC components. Its methods retrieve configuration parameters and use them to configure the component.

Implemented By

XPCAbstractComponent

Methods

public boolean config(XPCTransmit container, XPCConfigParams configParams);
Configures the component using configParams. Returns “true” if the configuration was successful; “false” otherwise. This method is final within its only implementation, XPCAbstractComponent and is invoked only by the XPC Framework. Concrete subclasses can override this method to access the configuration parameters.
container - the Transmit container
configParams - parameters for configuring the component

public boolean config(XPCConfigParams configParams);
Configures the component using configParams. Returns “true” if the configuration was successful; “false” otherwise.
configParams - parameters for configuring the component

public String getConfigParam(String key);
Retrieves the specified configuration parameter from configParams.
Key - the name of the parameter to be retrieved

package com.commerceone.xpc.swi.framework

interface XPCProcess

Description

This interface is implemented by all XPC standard components. Its process() method is invoked by the Action Director.

Implemented By

All subclasses of XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Invokes the component's execution method and returns the results of the execution.
dataMgr - the data with which the component is to be invoked.

package com.commerceone.xpc.swi.framework

interface XPCTransmit

Description

This interface is implemented by the XPCAbstractComponent subclass.

Implemented By

XPCAbstractComponent

Methods

public void transmitReply(Envelope reqEnv, Document replyDoc) Transmits a reply document. Throws the DocumentExchangeException if the envelope is not correctly formed or the data is invalid. reqEnv - the original request envelope replyDoc - the document to be sent as a reply
public void transmitEnvelope(Envelope env) Transmits a request envelope.Throws the DocumentExchangeException if the envelope is not correctly formed or the data is invalid. env - the envelope sent as a request

package com.commerceone.xpc.gedi

Description

This package contains classes used by the XPC Generic EDI Wrapper service to receive and transport ANSI X12 and EDIFACT formatted EDI files through MarketSite. These classes are used to parse data from EDI files, map RecipientIDs to RecipientTPIDs in a flat file, compress and decompress files, and create and populate envelopes.

Descriptor	Takes a key string of an EDI file as input and creates a GEDI document as output. The Receiver and Sender IDs are parsed from the EDI file and are used to populate the GEDI document.
StringMapper	Finds the ReceiverID's matching ReceiverTPID in a flat file.
CompressStream	Takes in an input stream from the data manager, compresses the input stream and converts the zipped output stream to a document. The "zipped" document is returned to the data manager.
CreateGEDIEnvelope	Creates an envelope and populates it with a GEDI document, an attachment document, and a receiverTPID from the data manager as well as a senderTPID and attachmentURI from the default.prop configuration. Note: The URI used by the trading partner sending the transaction must match the URI used by the trading partner receiving the transaction.
GetAttachment	Unwraps the GEDIEnvelope from the env file, converts the attachment in the GEDIEnvelope into an input stream and returns the input stream to the data manager.
DecompressStreamToFileSystem	Decompresses an input stream and saves it to a specified directory on the local file system with a specified file name. Takes in an input stream and file name from the data manager and receives a directory path from the default.prop configuration.

package com.commerceone.xpc.gedi

class Descriptor

Description

Takes a key string of an EDI file as input and creates a GEDI document as output. The Receiver and Sender IDs are parsed from the EDI file and are used to populate the GEDI document.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Reads the EDI file input stream one character at a time until the entire header line has been read, determines what type of parsing is needed based on the first three characters of the header line (ISA or UNB), creates and populates the GEDI document, and returns the GEDI document to the dataMgr.

Inputs:

fileKey - String - the filename key of the EDI file. This string is used to construct the path to the file to be processed.

Outputs:

xmlDocument - GEDI Document - the GEDI document.

Configurations:

The following configurations reflect the EDI standard (X12 or EDIFACT) that is being used. All are set in the **Descriptor.config** configuration in the *XPCGEDIOutBound default.prop* file. The delimiters and segment terminator should be consistent with the EDI standard being used. If a delimiter or terminator is an unprintable symbol, use the ASCII code for the symbol instead of the actual symbol.

subDataElement - the X12 or EDIFACT delimiter

dataElementTerminator - the second EDIFACT delimiter

segmentTerminator - the escape character that indicates the beginning of a new line

ediStandard - the EDI standard that determines which parser to use The default configuration is **EDIFACT**. If you are using the X12 standard, set **ediStandard** to **X12**.

Prefix - the prefix of the file name to be constructed.

Extension - the extension of the file name to be constructed.

Directory - the full path to the directory where the file resides.

package com.commerceone.xpc.gedi

class StringMapper

Description

Finds the Recipient TPID (Trading Partner ID)that corresponds to the Recipient ID in the EDI file. Mappings between Recipient ID and Recipient TPID are stored in the *map.txt* file. This file has the following structure:

```
SAMPLERECEIVERID=SAMPLETPID
SAMPLE2RECEIVERID=SAMPLE2TPID
```

To map a Recipient TPID to a Recipient ID, use a text editor to modify the *map.txt* file in the *\gedi\tpid_map* directory. Be sure that there are unnecessary blank spaces following the TPID information, as such spaces will be included in the TPID that is copied to the XML document and envelope, causing a mismatch with the MarketSite Trading Partner ID.

Note ..... You must stop and restart the server for these changes to take effect.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Uses a flat file to match the recipient ID to the corresponding recipient TPID used by MarketSite. If no matching recipient TPID is found an exception is thrown.
Inputs:
receiverID - String - the receiver ID from the EDI file
Outputs:
receiverTPID - String - the Trading Partner ID for the receiver
Configurations:
mapFile - the *map.txt* file used to map Receiver IDs to Trading Partner IDs

package com.commerceone.xpc.gedi

class CompressStream

Description

This class takes in an input stream from the Data Manager, compresses it, and converts the zipped output stream to a document. The "zipped" document is returned to the Data Manager.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)
Takes in an input stream from the Data Manager, compresses it, and converts the zipped output stream to a document. The "zipped" document is returned to the Data Manager.
Inputs:
inputStream - InputStream - the input stream
Outputs:
zipDocument - DocumentObject - the compressed output stream
Configurations:
None

package com.commerceone.xpc.gedi

class CreateGEDIEnvelope

Description

Creates a Generic EDI Wrapper MIME envelope and populates it with a GEDI document, an attachment document ("zipped" document), and a receiverTPID from the data manager as well as a senderTPID and attachmentURI from the default.prop configuration file.

Note ..... The URI used by the trading partner sending the transaction must match the URI used by the trading partner receiving the transaction.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the receiving Trading partner ID, GEDI document, and attachment document from the Data Manager, creates a URI from the configuration string, creates the envelope, sets information and documents to the envelope, sets the EnvelopePropertyValue to peer to peer transmission, and returns the envelope to the Data Manager.

Inputs:

receiverTPID - String - The receiver's Trading Partner ID
FilenameKey - String - the configuration string
xmlDocument - GEDI - the GEDI document
zipDocument - Document - the attachment

Outputs:

GEDIEnvelope - Envelope - the envelope returned to the Data Manager

Configurations:

senderTPID - the sender's ID key to be looked up from the mapping file specified by mapFile configuration
attachmentURI - the URI of the attachment
mapFile - Full path to the file that contains the key string and TPID value pairs.

package com.commerceone.xpc.gedi

class GetAttachment

Description

Unwraps the GEDIEnvelope from the envelope, converts the attachment in the GEDIEnvelope into an input stream and returns the input stream to the Data Manager.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the GEDIEnvelope from the Data Manager, unwraps the attachment from the envelope with the specified URI, converts it into a document. The "zipped" document is converted into an input stream and returned to the dataMgr. When utilizing the toStream method, the 8859_1 character encoding is required.

Inputs:

GEDIEnvelope - Envelope - the GEDI envelope

Outputs:

attachment - Input Stream - the input stream created from the attachment

Configurations:

URI - the URI of the envelope

package com.commerceone.xpc.gedi

class DecompressStreamToFileSystem

Description

Decompresses an input stream and saves it to a specified directory on the local file system with a specified file name. Takes an input stream and file name from the Data Manager and receives a directory from the default.prop configuration.

Extends

XPCAbstractComponent

Methods

public XPCResult process(XPCDataMgr dataMgr)

Gets the InputStream and the envelope correlation ID from the dataMgr. The component unzips the InputStream. The correlation ID is then used to name the unzipped file that is placed in a configurable directory on the local file system.

Inputs:

attachment - InputStream - the input stream from the Data Manager

filename - String - the path to the file

Outputs:

None

Configurations:

fileDir - the directory in which the decompressed file will be stored

A Using a Transmitter API

In this Appendix

This appendix describes how to use a TransmitterAPI to create stand-alone clients for sending document(s)/envelope and provides some examples of creating entity manager, document(s) and envelope.

This appendix includes the following sections:

For more information on how to create a document(s)/envelope, please refer to *XDK Developer's Guide* under XPC documentation directory, or other XDK documentations from the XDK Pro product.

Stand-Alone Client

You can use the Stand-alone client to interact with a MarketSite XPC.

The stand-alone client can synchronously transmit envelopes and receive responses to the envelope, and asynchronously transmit envelopes. A stand-alone client is external to the XPC Server (which doesn't require a XPC server to run) and cannot receive and process asynchronous replies. In MarketSite 4.0, the TransmitterAPI will do the transformation without any explicit API call, but by setting three versioning related properties.

An XPC client can communicate with a MarketSite/XPC server in one of three ways:

- Synchronous

For each transmitted message, the calling thread within the client blocks until either a response envelope is received, or error document such as timeout is received. The PriceCheck is an example of a synchronous document.

- **Peer-to-peer**

  For each transmitted message, the calling thread will receive either a transmission OK document (for internal use) right away, or an error document, so the client is not blocked. However, it will not receive the response envelope. In order to receive the response envelope, you must run the XPC Server containing the service that subscribes to this response document. The Order is an example of a peer-to- peer document.

- **One-way**

  The message is sent asynchronously. No reply is expected and the client is not blocked.

Setting Up a Client Environment

The simplest way to create a working stand-alone client is to clone the existing DocSender Sample included in the XPC Installation. This sample demonstrates how TransmitterAPI can be used as well as how XDK Document Framework, SOX Bean are used.

You can find the sample DocSender under

`<XPCInstallRoot>/sample/com/commerceone/sample/xpc/docsender`

in the package of `com/commerceone/sample/xpc/docsender.`

For additional information about the sample, such as the contents of each java class in the sample or information on compiling or running, refer to the README.txt file at the above location.

Configuring a Client

XPC looks for the necessary SOX schema path, transmission data, and other necessary properties, in either the `client.prop` file in the current directory where the client is started, or in the Config object passed programmatically. For additional information, refer to **Transmitter API** on page 6 in this chapter.

The Configure XPC tool modifes the XPC Server's inbound and outbound connectivity information. This tool also modifies two client.prop files for either purpose:

- <install:root>/bin/client.prop. This file always points to MarketSite. It is used by the PingMarketSite program as described in the Install/Admin guide.
- <install:root>/etc/config/client.prop. This file always points to the local server. It is used by the Invoker tool.

Your stand-alone client can use either of these client.prop files. If you make a copy of either of these files and move them, you will need to manually synchronize any changes made in Configure XPC.

If client.prop is used and it couldn't be found in the current directory where the client is started, the program will exit. You can find a sample of client.prop under the `<XPCInstallRoot>/bin` directory.

Please note there is a case that client.prop can't locate in the same directory that client is started. To solve this problem, you can read the property file in other location, load it to Properties object, and pass it to Config object as these:

```
String propertyFilePath = …;
ConfigProperties prop = null;
       try {

              FileInputStream fis = new
FileInputStream(propertyFilePath);
              Properties tranProps = new Properties();
              tranProps.load(fis);
              prop = new ConfigProperties(tranProps);

       } Catch(Exception ex){
…
}
```

The properties that need to be set either in `client.prop`, or programmatically in Config object, are listed later in the section and explained in the *XPC Installation and Administration Guide*.

Required Jar Files

In addition to the jar files required for XDK API to create the Document and Envelope and the jar files for the CBL documents (depending on the version), the following jar files are needed to use Transmitter API:

- client.jar
- broker.jar
- jndi.jar
- jms.jar
- ccs_server.jar
- ccs_event.jar
- ccs_util.jar
- vgateway.jar
- iaik.jar

Transmitter Parameters

For each transmission method used in a stand-alone client, the following specific keys are valid. You can obtain the values for the constant values from:

com.commerceone.xdk.swi.metadox.property.Properties Constants

Synchronous

TIMEOUT_PARAM_KEY

Request time-out in milliseconds or `INFINITE_TIMEOUT_VALUE.`

ACK_PARAM_KEY

Must be set to ACK_NO_PARAM_VALUE.

For example,

```
String timeout = ...
Envelope request_env = …;
ParameterList modeParams = new ParameterList();
modeParams.set(PropertiesConstants.ACK_PARAM_KEY,
PropertiesConstants.ACK_NO_PARAM_VALUE);
modeParams.set(PropertiesConstants.TIMEOUT_PARAM_KEY, timeout);
EnvelopePropertyValue propValue = new
EnvelopePropertyValue(PropertiesConstants.SYNC_MODE_VALUE,
modeParams);
request_env.setRequestMode(propValue);
```

Peer-to-peer and One-way

ACK_PARAM_KEY

Must be set to ACK_YES_PARAM_VALUE.

For example,

```
Envelope request_env = …;
ParameterList modeParams = new ParameterList();
modeParams.set(PropertiesConstants.ACK_PARAM_KEY,
PropertiesConstants.ACK_YES_PARAM_VALUE);


EnvelopePropertyValue propValue = new
EnvelopePropertyValue(PropertiesConstants.PEER_PEER_MODE_VALUE,
modeParams);// for PEER-TO-PEER mode
//EnvelopePropertyValue propValue = new
EnvelopePropertyValue(PropertiesConstants.ONEWAY_MODE_VALUE,
modeParams);// for ONEWAY mode
request_env.setRequestMode(propValue);
```

Transmitter API

When Using the client.prop file

Step 1: Instantiate TransmitterFactory object by using the properties in client.prop from the current directory where the client is started.

```
TransmitterFactor transmitterFactory = new TransmitterFactory()
```

Step 2:Create DocumentResponder/DocumentListener from MarketSite/destination MPID and user Properties object, depending on the request mode.

```
DocumentResponder responder = transmitterFactory.getResponder(TradingPart-
nerAddres s internalized_destination, Properties userProps);
```
// for Synchronous

OR

```
      DocumentListener listener =
transmitterFactory.getListener(TradingPartnerAddress
internalized_destination, Properties userProps);// for Peer-t-Peer or
OneWay
```

The following fields are required to be set in the user Properties Object (userProps):

Note**.....** These keys are case sensitive and must be entered exactly as listed below:

- **DOCTYPE** - This field is always required
- **marketparticipantid** - This field is always required
- **authpref** - This field is required when http(s) protocol is used. The value could be uidpswd, or cert, or none (used mostly for testing purposes)
- **sonicmq.authpref** - This field is required when sonic protocol is used. The value could be uidpswd, or none (used from MarketSite to XPC, or testing purposes)
- **userid** - This field is required when `authpref/ sonicmq.authpref` is set to `uidpswd` as a way of authentication
- **password** - This field is required when `authpref/ sonicmq.authpref` is set to `uidpswd` as the way of authentication
- **server_root-** This field is required when authpref is set to cert as the way of authentication. This specifies a directory with the following subdirectories:
  - certs subdirectory that contains a file named serverstore for keystore
  - config/startup subdirectory that contains a file named https- server.prop and in this file, these three properties need to be set:
    - iaik.jigsaw.ssl.keystore=serverstore// this file name should match file name above for keystore
    - iaik.jigsaw.ssl.keystore.password=...//encrypted password
    - iaik.jigsaw.ssl.rsa.keyAndCertificate=...//entry name for keystore

    An example of server_root directory is:

    <RootOfServerInstall>/transmitter/ccs

  You can also include the above user properties in client.prop, load all the properties to Properties object, and selectively pass the above user properties to Properties object userProps used in getResponder() and getListener().

Step3: Send the request envelope and receive the response envelope only if in Synchronous mode.

Envelope reply_env = responder.processDocument(request_env); //for Synchronous

OR

listener.handleDocument(request_env);/for Peer-to-Peer or Oneway

```
Properties userProps = new Properties();
QName qname_doctype = new QName(null,"urn:x-
commerceone:document:com:commerceone:CBL:CBL.sox$1.0",
"PurchaseOrder");//or what ever your document is other than
PurchaseOrder under the urn of CBL

userProps.put("DOCTYPE", qname_doctype);
userProps.put("marketparticipantid",
request_env.getSenderId().toString());

userProps.put("authpref", "uidpswd");
userProps.put("userid", "admin");
userProps.put("password", "mypassword");
userProps.put("Sonicmq.authpref", "none");

String destinationMPID = …
TradingPartnerAddress internalized_destination =
TradingPartnerAddress.internalize(destinationMPID);

TransmitterFactory tf = new TransmitterFactory();
```

For Synchronous Envelope:

```
DocumentResponder responder =
(DocumentResponder)tf.getResponder(internalized_destination,
userProps);
reply_env = responder.processDocument((Envelope)request_env);
```

For PEER-TO-PEER/ONEWAY Envelope:

```
DocumentListener listener =
(DocumentListener)tf.getListener(internalized_destination,
userProps);
listener.handleDocument((Enevelope)request_env);
```

When Not Using the client.prop file

Step 1:Instantiate TransmitterFactory object by using the properties in Config object and EntityManager object

```
TransmitterFactory transmitterFactory = new TransmitterFactory(Config
config, EntityManager entityManager)
```

The following keys must be set in the Config object. The keys are case sensitive and should be entered exactly as listed. The possible values and descriptions of the keys are provided in the *XPC Installation and Administration Guide*.

Key
ccs.comm.em.fs
ccs.comm.em.fs.path
ccs.comm.em.ldap
ccs.comm.dir.schemaroot
ccs.comm.dir.username
ccs.comm.dir.password
ccs.comm.tx.fs
ccs.comm.tx.ldap
ccs.comm.tx.dir.tproot
ccs.comm.tx.dir.username

ccs.comm.tx.dir.password
ccs.comm.transmitter.cache
ccs.comm.transmitter.cache.refresh
ccs.comm.transmitter.destination.name
<destinationMPID>.doctype
<destinationMPID>.<doctype>.docformat
<destinationMPID>.<doctype>.protocols
<destinationMPID>.<doctype>.protocol.https.args
<destinationMPID>.<doctype>.protocol.sonic.args
Versioning Related Key
transformation.registry
transformation.internalversion
transformation.externalversion
Sonic MQ Related Key
sonicmq.broker.url
sonicmq.broker.username
sonicmq.broker.password
sonicmq.syncresponsequeue.name
sonicmq.connection.close
`sonicmq.authpref`
jms.client.ssl.enable
jms.client.ssl.provider.class
jms.client.ssl.cipher.suites
jms.client.ssl.client.requireTrustedRoot

jms.client.ssl.client.trustedRoot.dir

```
Import com.commerceone.ccs.config.ConfigProperties;
ConfigProperties cp  = new ConfigProperties();
cp.put("ccs.comm.em.fs", "true");
…
EntityManager entityManager = …//refer example in XDK API at the
end of the chapter
TransmitterFactory transmitterFactory = new TransmitterFactory(cp,
entityManager);
```

Step 2. Create DocumentResponder/DocumentListener from MarketSite/destination MPID and user Properties object, depending on the request mode.(The same as when using client.prop)

```
DocumentResponder r = transmitterFactory.getResponder(Address destination,
Properties userProps);
```

OR

```
DocumentListener l = transmitterFactory.getListener(Address destination, Properties
userProps);
```

The following fields must be set in the Properties Object (userProps) which is the same as when using client.prop:

Note ..... These keys are case sensitive and must be entered exactly as listed below:

- **DOCTYPE** - This field is always required
- **marketparticipantid** - This field is always required
- **authpref** - This field is required when http(s) protocol is used. The value could be uidpswd, or cert, or none(used mostly for testing purposes)
- **sonicmq.authpref** - This field is required when sonic protocol is used. The value could be uidpswd, or none (used from MarketSite to XPC, or testing purposes)
- **userid** - This field is required when `authpref/ sonicmq.authpref` is set to

uidpswd as a way of authentication

- **password** - This field is required when authpref/sonicmq.authpref is set to uidpswd as the way of authentication
- **server_root-** This field is required when authpref is set to cert as the way of authentication. This specifies a directory that has the structure of:
  - - certs subdirectory that contains a file named serverstore for keystore
  - - config/startup subdirectory that contains a file named https- server.prop and in this file, these three properties need to be set:

    iaik.jigsaw.ssl.keystore=serverstore// this file name should match file name above for keystore

    iaik.jigsaw.ssl.keystore.password=...//encrypted password

    iaik.jigsaw.ssl.rsa.keyAndCertificate=...//entry name for keystore

  An example of server_root directory is:
  <RootOfServerInstall>/transmitter/ccs

  You can also include the above user properties in the place that the other Config object properties are kept, load all the properties to Properties object, and selectively pass the above user properties to Properties object userProps used in getResponder() and getListener().

Step3: Send request envelope and receive the response envelope only if in Synchronos mode. (The same as when using client.prop)

```
Envelope reply_env = responder.processDocument(request_env); //for Synchronous

OR

listener.handleDocument(request_env);/for Peer-to-Peer or Oneway
```

Changing Debug Level

By default, the debug level is set up with the most print out(Level 0). This can be adjusted programmatically between 0-5 with Level 5 the least print out. This will be changed in next release to set in client.prop. Here is how to set:

```
String level = …;

Debug.log.setLevel(Integer.parseInt(level));
```

Exception Handling

Catching Exceptions in a Stand-alone Client

Client-side errors in using a transmitter are always seen as exceptions. The error happens before or during connection time. All Exceptions are derived from DocumentExchangeException.

A client can experience the following types of exceptions:

- XDK Related Error

  Error occurs with bean processing or conversion, marshalling/unmarshalling of envelope, etc.

- Sonic Related Error

  This can be an error to authenticate to local broker, to connect to local broker, queue not found, message size too big, etc.

- Transformation Related Error

  Error occurs during the transformation before the envelope is sent out

- TransmitterProperty Related Error

  Error with the transmitter information set in the envelope header, user Properties object as well as in the client.prop or Config object.

- Connection Related Error

  An error occurs while establishing connection

- Transfer Related Error

  A connection is established, and an error occurs during the transmission of document and evenlope data

- Server Related Error

  The message was transferred, but an error occurred during processing. An error doc is sent back. It could be the error of authentication, recipient not found, etc.

The exceptions that can be thrown for any transmitter include but are not limited to:

Type	Exception	Description
TransmitterProperty	com.commerceone.ccs.excp.comm.sender.TransmitterPropertyException	Error with the transmitter information set in the envelope header, user Properties object as well as in the client.prop or Config object.
	com.commerceone.ccs.excp.comm.communicator.InvalidParameterException	Used only for Sonic. Missing/ incorrect sonic related properties set in client.prop or Config object, or in user Properties object.
	com.commerceone.ccs.excp.comm.communicator.EnvelopeParameterException	Used only for Sonic. Missing/ incorrect properties set in the envelope header.
Connection	com.commerceone.xdk.excp.metadox.send.EstablishException	Error while establishing connection.
	com.commerceone.ccs.excp.comm.sender.ConnectionException	Extends from EstablishException. Could be bad destination address, etc.
	com.commerceone.ccs.excp.comm.communicator.DestinationNotFoundException	Used only for Sonic. Couldn't find Queue specified for destination or sync response.
	com.commerceone.ccs.excp.comm.communicator.TransportException	Used only for Sonic. JMS transport error.
Transfer	com.commerceone.xdk.excp.metadox.send.TransferException	A connection was established, and an error occurs during the transmission of document and envelope data. Could be a problem of I/O, etc.

Type	Exception	Description
Server	com.commerceone.ccs.excp.comm.sender.ServerException	The message was transferred, but something goes wrong in processing the message and error doc is sent back. Could be problem of authentication, recipient not found, etc.
	com.commerceone.ccs.excp.comm.communicator.ServerException	Used only for Sonic. The same purpose as ServerException above.
	com.commerceone.ccs.excp.comm.communicator.TimeoutException	Used only for Sonic. Timeout error for Sync request.
Sonic	com.commerceone.ccs.excp.comm.communicator.SecurityException	Error to authenticate to local broker
	com.commerceone.ccs.excp.comm.communicator.ConnectionNotEstablishedException	Error to connect to local broker
	com.commerceone.ccs.excp.comm.communicator.MessageTooBigException	Message size too big that exceeds the max of 10MB
Transformation	com.commerceone.versiongateway.exception.TransformExecException	Error occurs during the transformation before the envelope is sent out
XDK	com.commerceone.xdk.excp.metadox.send.ProcessingException	Error occurs when doing the bean processing or conversion. Could be a problem related to marshalling and unmarshalling.

Type	Exception	Description
	com.commerceone.xdk.excp.metadox.model.RepresentationConversionException	Extends from ProcessingException. Could be problem to translate from one representation to other.
	com.commerceone.xdk.excp.metadox.model.ResourceConversionException	Extends from RepresentationConversionException. Error accessing a resource when trying to convert the old representation into the new representation.
	com.commerceone.xdk.excp.metadox.factory.EnvelopeInvariantException	Could be problem with envelope header, main document, attachment(s), catalog, or corrupted/incorrect mime data etc.
	com.commerceone.xdk.excp.metadox.factory.CatalogUnmarshallingException	Extends EnvelopeInvariantException, Error unmarshalling the Envelope Catalog.
	com.commerceone.xdk.excp.metadox.meta.NoSuchAttachmentException	Attachment reference does not exist in Envelope.
	com.commerceone.xdk.excp.metadox.meta.AttachmentRetrievalException	Error retrieving the attachment.

Examples to Create EntityManager/ Document/Envelope

The following examples have been tested and some are also included in the DocSender sample.

Example 1

To initiate XDK before using any XDK API

```
import com.commerceone.xdk.initialize.XDK;
import
com.commerceone.xdk.excp.initialize.AlreadyInitializedExcep
tion;

try{
XDK.init();
}catch(AlreadyInitializedException e){
...
}
```

Example 2

To create EntityManager object from a schema path directories with each directory in a String object:

```
import com.commerceone.xdk.base.parser.URIEntityManager;
import com.commerceone.xdk.base.parser.EntityManager;
import
com.commerceone.xdk.base.parser.DirectorySearchURIDomain;

java.util.Vector fileVector = new java.util.Vector();
java.io.File file = null;

String schemaPath1 = ...;
Java.io.File File = new  java.io.File(schemaPath1);
fileVector.addElement(file);

String schemaPath2 = ...;
file = new  java.io.File(schemaPath2);
fileVector.addElement(file);

URIEntityManager em = new URIEntityManager();
DirectorySearchURIDomain domain = new
DirectorySearchURIDomain(fileVector,"urn");
em.addDomain(domain);
```

Example 3

To create EntityManager object from a schema path directories with all directories separated by separator(;) in a String object:

```
import com.commerceone.xdk.base.parser.URIEntityManager;
import com.commerceone.xdk.base.parser.EntityManager;
import
com.commerceone.xdk.base.parser.DirectorySearchURIDomain;

java.util.Vector fileVector = new java.util.Vector();
java.io.File file = null;

String path = schemaPath1 + ";" + schemaPath2 + ... // ";"
is used as separator here

java.util.StringTokenizer st = new
java.util.StringTokenizer( path,";"); // ";" is a separator
used in the String path
while (st.hasMoreTokens()){
fileVector.addElement(new java.io.File(st.nextToken()));
}
URIEntityManager em = new URIEntityManager();
DirectorySearchURIDomain domain = new
DirectorySearchURIDomain(fileVector,"urn");
em.addDomain(domain);
```

Example 4

To create Document object from a String (DocType needs to be passed):

```
import com.commerceone.xdk.swi.metadox.marshall.DataSource;
import
com.commerceone.xdk.metadox.model.stream.StringDataSource;
import com.commerceone.xdk.metadox.type.DocumentType;
import
com.commerceone.xdk.metadox.model.stream.DocumentStream;
import com.commerceone.xdk.swi.metadox.meta.Document;

String xmlString = ...
DataSource dataSource = new StringDataSource(xmlString);
QName qname_doctype = new QName(null,"urn:x-
commerceone:document:com:commerceone:CBL:CBL.sox$1.0",
"PurchaseOrder");//or what ever your document is other than
PurchaseOrder under the urn of CBL
DocumentType docType = new DocumentType(qname_doctype);

Document doc = new DocumentStream(dataSource, docType);
```

Example 5

To create Document object from a file (option 1: DocType needs to be passed):

```
import com.commerceone.xdk.swi.metadox.marshall.DataSource;
import
com.commerceone.xdk.metadox.model.stream.ReaderNowDataSourc
e;
import com.commerceone.xdk.metadox.type.DocumentType;
import
com.commerceone.xdk.metadox.model.stream.DocumentStream;
import com.commerceone.xdk.swi.metadox.meta.Document;

String fileName = ...
DataSource dataSource = new FileReaderNowDataSource(new
Reader(fileName));
QName qname_doctype = new QName(null,"urn:x-
commerceone:document:com:commerceone:CBL:CBL.sox$1.0",
"PurchaseOrder");//or what ever your document is other than
PurchaseOrder under the urn of CBL
DocumentType docType = new DocumentType(qname_doctype);//
or what ever your document is other than PurchaseOrder
Document doc = new DocumentStream(dataSource, docType);
```

Example 6

To create Document object from a file. (option2: gets doc type automatically from the document by parsing part of the document with some performance overhead):

```
import com.commerceone.util.net.URI;
import com.commerceone.xdk.base.parser.FileURIDomain;
import com.commerceone.xdk.base.parser.URIEntityManager;
import com.commerceone.xdk.metadox.factory.DocumentFactory
import com.commerceone.xdk.swi.metadox.meta.Document;

String fileName = ...
URI instanceURI = FileURIDomain.toURI(new File(filename));
URIEntityManager em = ...// use the example to create
EntityManager

ExternalSource source = em.open(instanceURI);
DocumentFactory docFactory = new DocumentFactory();
Document doc = docFactory.fromSource(source);
source.close();
```

Example 7

To create a request envelope:

```
For Example,
import com.commerceone.xdk.metadox.factory.EnvelopeFactory;
import
com.commerceone.xdk.metadox.message.TradingPartnerAddress;
import
com.commerceone.xdk.swi.metadox.property.PropertiesConstants;
import javax.mail.internet.ParameterList;
import com.commerceone.xdk.metadox.meta.EnvelopePropertyValue;
import com.commerceone.xdk.base.parser.EntityManager;
import com.commerceone.xdk.swi.metadox.meta.Envelope

EntityManager entityManager =...;// use example to create
EntityManager
EnvelopeFactory envFactory = new
EnvelopeFactory(entityManager);
Envelope request_env = null;
try{
request_env = envFactory.createEnvelope(doc);
}catch(Exception ex){
...
}

request_env.setCorrelationId(env.getIdentity());
String recipientID = ...;
request_env.setRecipientId(new
TradingPartnerAddress(recipientID()));
request_env.setSenderId(new TradingPartnerAddress("TEST"));
…//refer to Transmitter Parameters session for detail what to
set
request_env.setRequestMode(..);
```

B Security Credential

In This Appendix

This appendix provides information about the security credential. It includes the following sections:

Credential of Document Originator

All document envelopes received by a target business service (either deployed on an XPC Server running in e-marketplace or a XPC Server running on a Trading Partner site) will contain the authenticated credential of the document Originating party, that is, the trading partner that sent the document to the target business service. The Credential is included into the document envelope at the time of authentication of the sending Trading partner (the document sender) by the MarketSite Authentication Service running in the Portal Router in MS 4.0 based e-marketplace (or the MarketSite Authentication Service running in Level-1 Server in a MS 3.x based e-marketplace).

Included here is the definition and function of Credential; the Java Bean API for the SOX/XML Credential document represented as Java bean component; and an example of how Credentials can be used by end-user business services application (to make application-level authorization decisions) while integrating with the MarketSite 4.0/MarketSet 2.0 platform environments.

Function of Credential

Credentials represent the identity of the document sending trading partner as a result of completion of the process of authenticating the documents sender by the MarketSite/MarketSet Portal Router. Credentials are created by the MarketSite Authentication Service independent of the transport protocol used by the document sending party. Hence, a trading partner A involved in a document exchange

transaction with another trading partner B via its e-marketplace's Portal Router will get the same Credential of the trading partner A regardless of the transport protocol (e.g., Sonic/JMS or HTTPS) employed during the document exchange.

The credential of an authenticated sender contains registered, authenticated identity attributes of the principal, i.e. trading partner transacting via MarketSite/MarketSet. Credentials are included in the XML document envelope (as a XML document attachment) along with the original business document(s) that are part of the business transaction. The document envelope, containing the business document and authenticated Credential, is forwarded by the MarketSite/MarketSet Portal Router to the target trading partner business service. The identity and TP registration information contained in the Credential can be used by the target business service to make application-level authorization decisions within the business service logic.

Furthermore, the Credential feature is identical to older versions of MarketSite 3.x based e-marketplaces (which use Level-1 Server).

Access Control Application of Credential in Business Services

Business Service and/or Document Service writers can design their services to process Credentials that do application-level authorizations such that sending Trading partner's document is processed by business service keeping into consideration the following access control criteria:

- Registered e-marketplace identity of the sending Trading partner (i.e., MPID);
- Business Service level Access rights of the trading partner

The Business Service will need to maintain its own access control list (for example, a Customer account table in a SQL database or a mapping of the MPID or role in a Credential to specific transaction authorizations) which can be employed in conjunction with the Credential included in the Document Envelope.

The Credential API below describes the relevant authenticated attributes contained in the Credential.

XML/SOX Credential Public APIs

Included here is the Java Bean API for SOX/XML based Credential.

com.commerceone.ccs.doclet.security.Security_sox

Interface Credential

All Superinterfaces:

com.commerceone.xdk.swi.metadox.meta.Doclet, com.commerceone.xdk.swi.metadox.meta.Document, com.commerceone.xdk.maplibs.jbschema.jbmapping.ElementType, com.commerceone.xdk.swi.metadox.marshall.Marshaller, java.io.Serializable

All Known Implementing Classes:

CredentialImpl

public interface **Credential**

extends com.commerceone.xdk.maplibs.jbschema.jbmapping.ElementType

This is the interface emitted for element type :Credential

Field Summary	
static com.commerceone.xdk.base.event.QName	**DOC_TYPE**
static java.lang.String	**SYSTEM_ID**

Method Summary

/*
* This interface, although accessible in the Credential API, currently does not do anything useful .
*
*/
AccessRights **getAccessRightsList**()

/*
* This API returns the AuthorizingEntityID of the sending Trading Partner.

```
 *
 */
java.lang.String getAuthorizingEntityID()

/*
 * This API returns the registered e-mail, address of the sending Trading Partner.
 *
 */
java.lang.String getEmailAddress()
/*
 * This API returns the registered MarketParticipant ID (aka MPID) of the sending TP.
 *
 */
java.lang.String getMktSitePartpntID()

/*
 * This API returns the registered location (i.e., city/town) of the sending TP.
 *
 */
java.lang.String getSubjectLocation()


 /*
 * This API returns the registered organization name of the sending TP.
 *
 */
java.lang.String getSubjectOrgName()
 /*
 * This API returns the registered organization unit (e.g., company division/department) name of
 * the sending TP.
 *
 */
java.lang.String getSubjectOrgUnitName()

 /*
 * This API returns all the registered TP roles (e.g., “Buyer” or “Supplier”) of the sending TP.
 *
 */
```

```
java.lang.String[] getSubjectTPRole()

 /*
 * This API returns a specific registered TP role (within the roles list) of the sending TP.
 *
 * @param int index of the specific role entry in the roles array
 */
java.lang.String getSubjectTPRole(int index)

 /*
 * This API returns the registered legal name (e.g.,  company name registered with a D&B Registration
 * authority or equivalent company registration agencies elsewhere in the world) of the sending TP.
 *
 */
java.lang.String getTPName()

/*
 * This API returns the registered “well-known” name (e.g.,  an abbreviated company name) of
 * the sending TP.
 *
 */
java.lang.String getTPShortName()
```

Credential Usage Example

An end-user business service can access the Credential packaged in the document envelope that is routed to in the following way. The getCredenitial() is an example of a convenience function that returns the Credential of the sender that is transacting with the receiving business service. This Credential can be used to check the authorization level of the document sending party as part of the application logic implemented in the business service.

```
import com.commerceone.xdk.swi.metadox.meta.Envelope;
import com.commerceone.ccs.doclet.security.Security_sox.Credential;
import com.commerceone.xdk.excp.metadox.send.DocumentExchangeException;

    private Credential getCredential(Envelope envelope)
    throws DocumentExchangeException
    {
```

```
    Credential cred =
   (Credential)envelope.getCredential(DocumentObject.REPRESENTATION);

   if (cred != null)
   {
       String[] roles = cred.getSubjectTPRole();
       if (roles != null)
       {
      for (int i=0; i< roles.length; i++)
      {
          System.out.println("Credential:getSubjectTPRole[" + i +
                 "]: " + roles[i]);
      }
      }

      System.out.println("Credential:getMktSitePartpntID()  " +
cred.getMktSitePartpntID());
      System.out.println("Credential:getTPName()  " + cred.getTPName());
      System.out.println("Credential:getTPShortName() " + cred.getTPShortName());
      System.out.println("Credential:getSubjectLocation(): " +
cred.getSubjectLocation());
   }
           return cred;
    }
```

C Generic EDI

In This Chapter

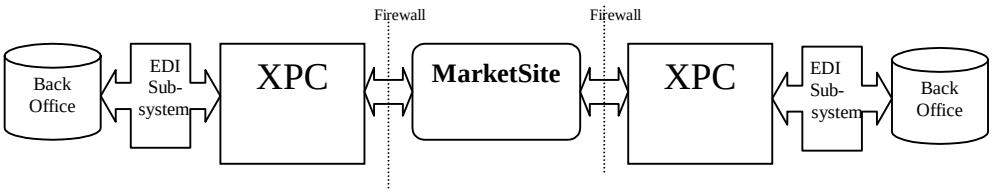
This chapter provides information about Generic EDI. It includes the following information:

Overview of Generic EDI

Generic EDI gives trading partners the ability to exchange *any* ANSI X12 or EDIFACT EDI document over MarketSite. Trading partners place EDI documents into the local file system. From there XPC picks up the EDI documents, places each document into an xCBL envelope, and transmits the envelopes to MarketSite. MarketSite routes the envelopes to the destination trading partner where the EDI document is extracted from the envelope.

The processes at the trading partner's site are performed by XPC. XPC runs inside the trading partner's firewall. It functions as the communication component that sits between the trading partner's EDI subsystem and MarketSite. XPC wraps and unwraps EDI documents.

The following diagram shows how an EDI document is routed from one trading partner's back office system to another.



GEDI Envelope Structure

For transmission between XPC and MarketSite, an EDI document is wrapped in a GEDI envelope. The envelope has three important areas.

Header

The header contains the routing information telling MarketSite where the EDI envelope goes. XPC determines the MarketSite Trading Partner ID (TPID) of the destination. XPC extracts the recipient identifier from the EDI document, and uses an external file to translate it into a TPID.

Body

The body contains the GEDI xCBL document. This document can include the following information:

- Receiver Identifier (key identifying destination in an EDI document)
- EDI Standard (ANSI, EDIFACT, Other)
- Transaction count (or, message count)
- Transaction count breakdown (message count per transaction type)
- Sender Identifier (key in an EDI document)

All entries in this document are optional.

Attachment

The envelope has a single attachment. This is the compressed EDI document.

EDI File Properties

- EDI files sent to MarketSite must contain only one interchange.

- EDI file interchange must contain a Recipient ID and a Sender ID.
- EDI files placed in the outbound directory by trading partners must not be compressed. The Descriptor component will not be able to parse the Receiver ID and Sender ID from compressed EDI files.
- EDI files placed in the outbound directory by trading partners must not be encrypted. The Descriptor component will not be able to parse the Receiver ID and Sender ID from encrypted EDI files.
- Only ANSI X12 and EDIFACT formatted EDI files will be parsed correctly by the Descriptor component.
- ANSI X12 EDI files must have an *ISA* tag denoting the beginning of an interchange.
- EDIFACT EDI files must have a *UNB* tag denoting the beginning of an interchange.

EDI Recipient ID Mapping

The Recipient ID parsed from the EDI file must have a corresponding Recipient TPID in the **map.txt** file. The **map.txt** file must not contain any extra (unnecessary) blank spaces after the Recipient TPID information.

Outbound Processing

XPC periodically wakes up and searches a specified directory for new EDI files. For each EDI document found, XPC:

1. Creates a GEDI xCBL document and fills it with the receiver ID and sender ID extracted from the EDI document
2. Translates this receiver ID to a receiver TPID using an external properties file
3. Compresses the EDI document
4. Creates the GEDI envelope
5. Transmits the GEDI envelope to MarketSite

Inbound Processing

When a GEDI envelope is received from MarketSite, XPC:

1. Detaches the EDI attachment

2. Decompresses the EDI document
3. Saves the EDI document to the file system

Error Processing

If any failures are encountered during outbound processing, the EDI document is moved from the archive directory to the error document directory.

However when an error occurs during inbound processing an error document is sent back to the sending XPC. At the sending XPC the XPCError service will search the XCC archive for the matching envelope and copy it to the error document directory.

Generic EDI Components

The following table provides a summary of component methods, inputs, outputs, and externally defined configurations. For more information about the components and methods in this table, see the **API Reference** chapter.

Component.method	Inputs, Outputs, and Configurations	Description
CompressStream.process	**Input Name and Type** n inputStream - InputStream **Output Name and Type** n zipDocument - DocumentObject **Configurations** n None	Takes an input stream from the data manager, compresses the input stream and converts the zipped output stream to a document. The “zipped” document is returned to the data manager.

Component.method	Inputs, Outputs, and Configurations	Description
CreateGEDIEnvelope.process	**Input Name and Type** n receiverTPID - String n FilenameKey - String n xmlDocument - GEDI n zipDocument - Document **Output Name and Type** n GEDIEnvelope - Envelope **Configurations** n SENDER_TPID n URI	Gets the RECEIVER_TPID, XML_DOC (GEDI document), and ATTACH_DOC ("zipped" document) from the data manager, creates a URI from the configuration string, creates the envelope, sets information and documents to the envelope, sets EnvelopePropertyValue to peer to peer transmission, and returns the envelope to the data manager.
DecompressStreamToFileSystem.process	**Input Name and Type** n attachment - Input Stream n filename - String **Output Name and Type** n None **Configurations** n fileDir	Gets the InputStream and the envelope correlation ID from the dataMgr. The component unzips the InputStream. The correlation ID is then used to name the unzipped file which is placed in a configurable directory on the local file system.
Descriptor.process	**Input Name and Type** n ediInputStream - Input Stream **Output Name and Type** n xmlDocument - GEDI **Configurations** n subDataElement n dataElementTerminator n segmentTerminator n ediStandard	Takes in a keystring of an EDI file as an input and creates a GEDI document as an output. The Receiver and Sender IDs are parsed from the EDI file and are used to populate the GEDI document.

Component.method	Inputs, Outputs, and Configurations	Description
GetAttachment.process	**Input Name and Type** n GEDIEnvelope - Envelope **Output Name and Type** n attachment - Input Stream **Configurations** n URI	Gets the GEDIEnvelope from the data manager, unwraps the attachment from the envelope with specified URI, converts it into a document. The “zipped” document is converted into an input stream and returned to the dataMgr. When utilizing the toStream method, the 8859_1 character encoding is required.

Generic EDI SOX Schema

This appendix contains the SOX schema for a Generic EDI document.

```
<?xml version="1.0"?>

<!DOCTYPE schema SYSTEM "urn:x-
commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri="urn:x-commerceone:document:com:commerceone:xpc:gedi:GEDI.sox$1.0">

<!-- All strings are in GEDI are defined as gedistring so that in future this can be
     extended to add constraints and length. Right now it is set to 255 max -->
<datatype name ="gedistring">
   <varchar maxlength = "255" />
</datatype>

<datatype name="EDIStandardCode">
    <explain>
   <p> Defines the various EDI Standards like X12, EDIFACT</p>
    </explain>
    <enumeration datatype="NMTOKEN">
```

```
    <!-- Other standard -->
    <option>Other</option>

    <!-- ANSI Standard EDI -->
    <option>ANSI</option>

    <!-- EDIFACT Standard -->
    <option>EDIFACT</option>

     </enumeration>
</datatype>
```