IBM WebSphere Business Integration Connect Enterprise
and
Advanced Editions

Programmer Guide for Connect

Version 4.2.2

Note!

Before using this information and the product it supports, read the information in “Notices” on page 153.

First Edition (June 2004)

This edition of the document applies to IBM WebSphere Business Integration Connect Enterprise Edition (5724-E87) and Advanced Edition (5724-E75), version 4.2.2.

To send us your comments about IBM WebSphere Business Integration documentation, email doc-comments@us.ibm.com. We look forward to hearing from you.

Contents

About this document

IBM(R) WebSphere(R) Business Integration Connect Enterprise and Advanced Editions provide a robust, scalable platform for managing business-to-business (B2B) communication.

This document describes a new set of tools available for the programmatic customization of the system as well as for the automation of certain aspects of day to day system administration.

Audience

This document is for consultants, developers, and system administrators for WebSphere Business Integration Connect Enterprise and Advanced Editions at customer sites.

Related documents

The complete set of documentation available with this product includes comprehensive information about installing, configuring, administering, and using WebSphere Business Integration Connect Enterprise and Advanced Editions.

You can download, install, and view the documentation at the following site: http://www.ibm.com/software/integration/wbiconnect/library/infocenter

Note: Important information about this product may be available in Technical Support Technotes and Flashes issued after this document was published. These can be found on the WebSphere Business Integration Support Web site, http://www.ibm.com/software/integration/websphere/support/

Typographic conventions

This document uses the following conventions:

`courier font`	Indicates a literal value, such as a command name, file name, information that you type, or information that the system prints on the screen.
italic	Indicates a new term the first time that it appears, a variable name, or a cross-reference.
blue outline	A blue outline, which is visible only when you view the manual online, indicates a cross-reference hyperlink. Click inside the outline to jump to the object of the reference.
`{ }`	In a syntax line, curly braces surround a set of options from which you must choose one and only one.
`|`	In a syntax line, a pipe separates a set of options from which you must choose one and only one.
`[ ]`	In a syntax line, square brackets surround an optional parameter.
`. . .`	In a syntax line, ellipses indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options.

`< >`	Angle brackets surround individual elements of a name to distinguish them from each other, as in `<server_name><connector_name>tmp.log`.
`/,\`	In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All product path names are relative to the directory where the connector is installed on your machine.
ProductDir	Indicates the directory where the product is installed.

New in this release

Version 4.2.2 is the first release of the *Programmer Guide for Connect*.

Part 1. Customizing Business Integration Connect: User Exits

Business Integration Connect is a business-to-business (B2B) community management solution. With Business Integration Connect, you exchange data and processes within a trading community, crossing enterprise boundaries and extending business integration beyond the enterprise and into the community. A trading community typically revolves around a hub—an enterprise that acts as the Community Manager. Community participants send documents to the hub, the hub performs certain necessary processing on them and the processed documents are then routed to the appropriate destination.

In versions 4.2.1 and previous of Connect, hub processing was limited to a configurable set of options hard-coded into the product. While these options were fairly wide-ranging, some potential trading scenarios were not supported. Starting with the present version, 4.2.2, users who have needs that fall outside the range of options delivered with the Connect product will be able to customize the process at a number of crucial stages by developing and deploying plugin modules that support additional transports, business protocols and so forth, based on a newly developed set of APIs. The points in the process where these plugin modules can be invoked are called *user exits*.

The following chapters document how to customize WBI-C using these user exits.

Chapter 1. User Exits Overview

To understand how the user exit plugins interact with the main body of standard Connect code, it is useful to divide the hub processing flow into three stages:

- "Document Receiving"
- "Document Processing" on page 4
- "Document Transmission" on page 6

This chapter provides a brief description of the WBI Connect components that perform these tasks and what aspects of the process are available for user customization.

Document Receiving

Documents enter the hub processing system through components known as *receivers*. Receivers are responsible for monitoring transports for inbound documents, retrieving the documents that arrive, performing some basic processing on them, and then placing them in a storage queue from which the main processing engine can retrieve them.

Receivers are transport-specific. WBI Connect ships with receivers designed to handle FTP/S, JMS, File, SMTP, and HTTP/S transports. Version 4.2.2 also includes an API so that users can develop their own receivers, based on their specific needs.

To be of use, receivers are associated with transport configurations called *targets*. A target designates the URI of an entry point for documents coming into Business Integration Connect. It specifies a repository location from which the Document Manager is to retrieve documents. Each target supports documents sent using a single transport type. If multiple document formats are supported, there may be one target for each type per given transport. Targets are configured through the Community Console, a GUI-based control component of WBI Connect. A user developed receiver can have one or more targets in the same manner as a provided receiver does. For more information on using the Console to configure targets and associate them with receivers, see the *Hub Configuration Guide*.

In addition to adding completely new receivers, users of Connect 4.2.2 are able to develop user exit plugin modules to customize how receivers process incoming documents. These modules are called *handlers*. There are three places in the receiver processing sequence where user exit handlers can be called to do additional processing: pre-processing, sync check, and post-processing. At each of these places, also called *configuration points*, the user can use the Console to specify one or more handlers. For this release, user designed receivers and the WBI-C provided HTTP receiver are the only ones that support adding handlers.

Pre-processing handlers are used to perform any necessary pre-processing of documents before they are sent to the Document Manger, where main processing takes place. For example, in some situations multiple records may be sent in one wrapper message. Pre-processing could separate the individual messages from each other before they are sent on for actual processing.

Document transmission can be synchronous or asynchronous. In synchronous, or sync, transmission, the initiating partner expects a document in response to the one it transmits. In asynchronous, or async, transmissions, no response document is expected (although some simple information, such as an HTTP status code, may be returned). A user exit can be used to process the checking of documents for sync status with specialized handlers. WBI-C ships with two default sync check handlers, `DefaultSynchronousSyncCheckHandler` and `DefaultAsynchrounousSyncCheckHandler`, but users can supply their own. This is particularly useful in the case of some document types in which defining separate sync and async targets - using different handlers - can increase through-put.

Post-processing handlers are used to deal with the response documents that WBI-C returns to initiating partners when a sync request has been made.

Document Processing

Documents that are being exchanged among community participants often need to be transformed in a number of ways, so that the needs of all partners can be accommodated. The Business Processing Engine (BPE), the heart of the Document Manager component, is responsible for effecting these transformations. It does so by following a series of steps, grouped into three basic units: fixed inbound workflow, variable workflow, and fixed outbound workflow. User exits allow user defined processes to be plugged into each of these basic units.

Fixed inbound workflow

Fixed inbound workflow covers the standard processing done to all documents coming into the Document Manager from a receiver. It is fixed because the number and type of steps are always the same. Through user exits, however, users are able to provide customized handlers for the processing in the first two of these steps: protocol unpackaging and protocol processing.

All messages that come into WBI-C are packaged according the specification of a specific business protocol. For example, a RosettaNet document is packaged according to the RNIF specification. Protocol unpackaging involves unpackaging the message so that it may be further processed. This process may include decryption, decompression, signature verification, extraction of routing information, user authentication, and/or business document parts extraction. WBI-C provides handlers for RNIF, AS2, MIME, EAI, and NONE packaging. If handlers for other packaging types are necessary, they can be developed as user exits.

Protocol processing involves determining protocol specific information, which may include parsing the message to determine routing information (sender ID, destination ID), protocol information (business protocol and version, such as Rosettanet version V02.02), and Document Flow/Process information (such as 3A4 version V02.02.) WBI-C provides processing for XML, RosettaNet, and EDI. Processing for other protocols (for example, CSV) can be provided using a user exit plugin.

If user exits are used to set up new packaging types or new protocol types, new Packaging or new Protocol information must also be set up in the Console. See the *Hub Configuration Guide* for more information.

Variable workflow

The core transformation process in BPE takes place in Variable Workflow. Variable workflow consists of some number of steps, arranged in a sequence, which taken all together make up an *action*. Actions are specified in the Console as part of the process of creating Participant Connections. WBI-C ships with 11 predefined actions. User exits may be used to create new actions either by developing an entirely new set of steps, placed into a new sequence or by copying an existing action and modifying it either by substituting a user-defined step for a pre-existing one, or by inserting a user-defined stop into an existing sequence.

Note: Not all WBI-C delivered steps can be used in new, user-defined actions, as they may be used for internal WBI-C specific purposes. See "WBI-C supplied actions and their status as templates" on page 43 for more information.

Actions consist of sequences of steps. User exit plugins can be used to create those steps. Typically steps may include the following types:

- **Validation**: checking the form of the business document. For example, an XML document can be validated against an XML schema. WBI-C provides an XML validation step, but others could be developed.
- **Transformation**: changing the form of the business document. An XML document can be transformed into a different XML document using XSLT. Again, WBI-C provides an XML transformation step, but others could be developed.
- **Translation**: changing the entire format of a business document from one type to another.
- **Process logging**: creating a log specific to this document. Some business protocols have specific logging requirements.
- **Sequence checks**: checking the sequence of documents. Some business protocols have specific sequencing requirements.

Note: These steps are typical examples only. Variable workflow is designed to implement business processing logic. The logic will dictate the actual steps required.

Once steps have been defined, the sequence in which they are to be executed must be specified in actions. For example, if the defined steps are Validation and Transformation, they could be sequenced into an action consisting only of validation, another of validation followed by transformation and a third of validation followed by transformation followed by validation of the transformed document. Sequences of steps are linked together as actions in the Console. See the *Hub Configuration Guide* for more information.

Fixed outbound

Once a document has been processed by the appropriate variable workflow, it must be packaged in preparation for transmission to its destination. WBI-C provides handlers for RNIF, Backend Integration, AS, and NONE packaging, and for cXML and SOAP protocols. Should other packaging handlers be required, they can be developed as user exit steps. Typically these steps will take care of one or more of the following processes:

- Assembling or enveloping
- Encrypting
- Signing
- Compressing
- Setting business protocol specific transport headers

Once the document is packaged, it is placed a storage queue where it remains until it is picked up by the Delivery Manager to be handed off to a sender.

Document Transmission

In many ways, the WBI-C document transmission process is simply the reverse of the document receiving process. Processed documents are fetched from the BPE storage queue, subjected to minor processing, and sent out to their destinations. Documents are sent out over a transport, by means of a transport specific component, a *sender*, based on a Console based configuration called a *gateway*.

Starting with this release, users can customize this stage by a) creating entirely new senders (to support new transports) and/or b) creating pre- and post-processing handlers that are dropped into the processing flow of standard or user-defined senders at two specific points. Pre-processing handlers affect the processing of the document before it is sent out; post-processing handlers affect the delivery result after the document is sent out (a SenderResult object holds delivery status - if, for example, a SOAP based transmission failed, a post-processing handler could create a SOAP fault message to set in the SenderResult object).

Chapter 2. Customizing Receivers

The receiver handles the first stage in the WBI Connect data flow. It picks up documents from the transport, performs some basic processing on them, and places them in a storage queue to be picked up by the main processing component, Document Manager. In synchronous requests, it also returns the response document to the initiating participant. As of the 4.2.2 release, users may customize the receiver stage of processing in two ways, by creating new receivers or by creating and configuring new receiver handlers. This chapter covers both ways of customizing receivers.:

- "Overview for creating new receivers"
- "Overview for creating new receiver handlers" on page 10

An additional section covers development and deployment issues.

- "Development and deployment" on page 10

The API listing and code and pseudo code sample outlines follow in the next chapter.

Overview for creating new receivers

Receivers are transport specific. WBI-C ships with receivers for FTP directory, JMS, File directory, SMTP (POP3), and HTTP/S transports. To add a new transport to the WBI-C system, users can write their own receivers, using an API provided with the 4.2.2 release. These new receivers can be configured using the Community Console and integrated into the processing flow in the normal way. This section describes the process of developing a new receiver. It covers:

- "Receiver flow"
- "Receiver types" on page 9
- "Receiver architecture" on page 10

Receiver flow

The nature of processing flow inside a receiver is in part dictated by the needs of the particular transport, but there are basic tasks that all receivers must accomplish. This section describes those tasks in a high level, general way.

1. Detect message arrival on transport
Receivers use one of two methods to detect request message arrival: polling the targets defined for this transport, as the provided JMS receiver does, or receiving callbacks from the transport, as the provided HTTP/S receiver does.

2. Retrieve message from transport
The receiver retrieves the request message and any transport attributes, like headers, from the transport. This may require the creation of temporary files on the file system.

3. Generate WBI-C required headers
There are special WBI-C headers that are used for further processing of the document. Receivers should have a mechanism for creating any of these headers that are appropriate, from transport message attributes or in other ways. The list of headers is as follows:

- `InboundCharSet`: the inbound document character set
- `MsgLengthIncHeaders`: the length of the message, including the transport headers
- `requestURI`: in the case of HTTP, the URI of the received servlet
- `timestamp`: the received document timestamp
- `fromIP`: the IP address of the destination where the document is sent
- `certDN`: the DN name of the SSL client certificate

The *receiver request* document which will be forwarded to Document Manager for further processing consists of the transport message, transport headers, and these WBI-C headers.

Note: The following steps, 4 and 5, may be executed in either order.

4. Do pre-processing

The receiver calls a WBI-C component, the Receiver Framework, to actually do the pre-processing. The Framework executes the handlers, either Connect supplied or user defined, that have been specified for this target via the Console, in the order they are shown in the Console configuration screen. The request document is passed as input to the first handler, and then the returned processed document is fed as input to the next handler, and so on. This step may return 1 or more documents, and all receivers must be designed to handle multiple returns.

5. Check sync status

The receiver calls the Receiver Framework to determine whether the received request is synchronous or not. The Framework moves through its configured list of handlers until an appropriate one is located. The outcome of this step determines the receiver's next step. If the request is asynchronous, path A will be followed. If the request is synchronous, path B will be followed.

6A. Process asynchronous request

If the request is asynchronous, (does not require a response document to be returned to the originating trading partner) the receiver calls the Framework to process the request document. The Framework takes care of storing the information in a place from which Document Manager will retrieve it.

6B. Process synchronous request

If the request is synchronous (requires a response document to be returned to the originating trading partner) the receiver calls the Framework to process the request document. There are two possible types of synchronous requests: blocking and non-blocking. In blocking mode, the receiver's calling thread will be blocked until the Framework returns the response document to it from Document Manger. In non-blocking mode, the receiver's calling thread will return immediately. When the Framework receives the response document at a later time, it will call the `processResponse` method on the receiver to pass the response document back. A correlation object is used to sync up the originating request with this response.

7. Do post-processing

In the case of a synchronous request, the receiver calls the Framework to execute post-processing on the response document before it is returned to the originating partner. The Framework executes the handlers, either Connect supplied or user defined, that have been specified for this target via the Console, in the order they are shown in the Console configuration screen. The response document is passed as input to the first handler, and then the returned processed document is fed as input to the next handler, and so on.

8. Complete processing
In case of a synchronous request, the response document is returned to the originating trading partner over the transport. The receiver calls the `setResponseStatus` method on the Framework to report on the success or failure of the response delivery. The receiver then removes the request message from the transport.

Exceptions

Error conditions can occur in the following circumstances:

- Retrieval of message from transport fails
- Call to pre-process fails
- Sync check fails
- Call to carry out async or sync processing fails
- Call to post-process response document fails
- Attempt to return response document on transport fails

If any of these failures occurs, the receiver performs the following actions:

Rejects the message from transport
The message must be removed from the transport. In the case of a JMS receiver, for example, the message is removed from the queue. In the case of an HTTP receiver, a 500 status code is returned to the trading partner.

Archives the rejected message
This is an optional step. The message is archived, either in a queue to be resubmitted later or in a rejected folder on the local file system. In the latter case, the Framework may be able to generate the appropriate storage files for the rejected request document.

Generates event
This is an optional step. Receiver developers may choose to have receivers produce events and/or alerts in the case of error conditions. There are three levels of events: informational, warning, and error. Informational events simply provide more information to the execution flow. Warning events log problems that nonetheless do not stop the execution flow. Error events are logged when processing of document results in an error and further document processing is stopped. For example, if in a synchronous request the receiver is unable to return a response document it has received from the Framework to the originating trading partner, an error event should be logged. A listing of events available for logging problems in the receiver stage is presented in the following API chapter.

Receiver types

There are two general types of receivers, based on how they detect incoming messages on the transport. Some receivers are polling based. They poll their transports at regular intervals to determine if new messages have arrived. Connect-supplied examples of this type of receiver include JMS and FTP. Other receivers are callback based. They receive notification from the transport when messages arrive. The HTTP receiver is an example of a callback based receiver.

Note: Receivers may be deployed in a multi-box mode. In this case, multiple receivers and their configured targets may be picking up messages from the same transport location. In such a deployment model, there needs to be concurrent management access coordination built into the receiver.

Receiver architecture

Receiver development is based on two major parts: the receiver itself, represented in the API by the `ReceiverInterface` interface and the Receiver Framework, represented in the API by the `ReceiverFrameworkInterface` interface. The Framework is responsible for providing an interface between the receiver and the main body of WBI-C code. The receiver calls methods on the Framework to accomplish the various processing steps.

Additional components specified in the API include `ReceiverConfig`, a way of querying and setting the various configuration attributes created in the Community Console; `ReceiverDocumentInterface`, the receiver request document which the receiver creates from the transport message; `ResponseCorrelation`, the provided mechanism for maintaining sync in a non-blocking sync scenario; a general exception class, `BCGReceiverException`; and a utility class, `BCGReceiverUtil`, which provides static methods for getting a Framework object, creating a request document, and finding appropriate file storage information. All of these are covered in detail in the Receiver API chapter.

Overview for creating new receiver handlers

The receiver can ask the Receiver Framework to do special processing at three places during the receiver processing flow: pre-processing, sync check, and post-processing. These places in the document processing flow are called `configuration points`. Pre-processing includes things that should be done before documents are injected into the main processing system and may include things like separating out multiple messages that have been sent by the originating trading partner in a single wrapper. Sync check checks whether the document is to be processed as a synchronous or asynchronous request. Post-processing provides necessary processing for response documents that are returned from Document Manager as a result of a synchronous request.

The Framework relies on `handlers` to execute these processing requests. Users may develop handlers to satisfy their specific needs, using APIs that ship with version 4.2.2. For this release, the only WBI-C supplied receiver that can support handlers is the HTTP receiver. Once the handlers are written and deployed, users will need to configure them using the Console. For further information on this process, see the *Hub Configuration Guide*.

Development and deployment

The following sections describe development and deployment for both user created receivers and handlers.

Development environment

The receiver and receiver handler development API relies on classes and interfaces from two packages:

- `com.ibm.bcg.bcgdk.receiver`
- `com.ibm.bcg.bcgdk.common`

These packages are part of the bcgsdk.jar which is found in the WBI-C installable in the following directories:

- `<install dir>\router\was\wbic`
- `<install dir>\receiver\was\wbic`
- `<install dir>\console\was\wbic`

In all deployed instances, this .jar file is installed in the application server classpath.

For development, the bcgsdk.jar file should be included in the build path of the project that contains the user exit classes, i.e., in the classpath.

Deployment and packaging

All user created code, including custom receivers and handlers, should be packaged and deployed in one of the following ways:

- Placed in a .jar file in `\receiver\was\wbic\userexits`
- Added as classes in `\receiver\was\wbic\userexits\classes`

In addition, all handler classes must be accompanied by an XML descriptor file, which allows key data about the handlers to be imported into the Console GUI. In the brief outline that follows, `HandlerClassName` is the name of the handler class; `Description` is a brief description of the class; `HandlerTypeValues` is the component name (RECEIVER - for receiver handlers, GATEWAY - for sender handlers, FIXEDWORKFLOW - for fixed inbound or outbound handlers, or ACTION - for variable workflow steps) followed by the type of handler (e.g., PREPROCESS.<transport>, PROTOCOL.UNPACKAGING, etc.); and `ComponentAttribute` is any configurable member variable in the handler class which changes the behavior of the handler at runtime.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:HandlerDefinition
     xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import
                                                     /external/types"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1
                             /import/external bcghandler.xsd
                        http://www.ibm.com/websphere/bcg/2004/v0.1
                            /import/external/types bcgimport.xsd ">
 <tns:HandlerClassName>com.ibm.SampleHandler </tns:HandlerClassName>
 <tns:Description>A Sample Handler</tns:Description>
 <tns:HandlerTypes>
     <tns:HandlerTypeValue>COMPONENT.TYPE</tns:HandlerTypeValue>
 </tns:HandlerTypes>
 <tns:HandlerAttributes>
       <tns2:ComponentAttribute>
               <tns2:AttributeName>Attribute 1</tns2:AttributeName>
       </tns2:ComponentAttribute>
       <tns2:ComponentAttribute>
           <tns2:AttributeName>Attribute 2</tns2:AttributeName>
            <tns2:AttributeDefaultValue>Attribute2DefaultValue
           </tns2:AttributeDefaultValue>
            </tns2:ComponentAttribute>
 </tns:HandlerAttributes>
</tns:HandlerDefinition>
```

Chapter 3. APIs and Example Code for Receivers and Receiver Handlers

The following chapter provides an annotated listing of the APIs provided for developing custom receivers and receiver handlers. The following classes and interfaces are documented:

- "ReceiverInterface" on page 14
- "ReceiverDocumentInterface" on page 16
- "ReceiverFrameworkInterface" on page 20
- "ReceiverConfig" on page 24
- "ResponseCorrelation" on page 26
- "BCGReceiverException" on page 27
- "ReceiverPreProcessHandlerInterface" on page 28
- "ReceiverSyncCheckHandlerInterface" on page 30
- "ReceiverPostProcessHandlerInterface" on page 32
- "BCGReceiverUtil" on page 34
- "Events" on page 35
- See also the listings in the Workflow API chapter for more utility, security, and other classes shared across components.

A brief example of code and pseudo-code outlining custom receiver implementation is also included here.

ReceiverInterface

Each receiver must implement this interface. It has the following methods:
- `init`
- `refreshConfig`
- `startReceiving`
- `processResponse`
- `stopReceiving`

Method

`init`

Method description

Initializes the receiver, based on the contents of the `ReceiverConfig` object

Syntax

```
public void init (Context context, ReceiverConfig config)
      throws BCGReceiverException
```

Parameters

`context`
Runtime information such as temp directory path

`config` Configuration details as specified in the Console

Method

`refreshConfig`

Method description

Called by the Receiver Framework if and when it detects changes in the configuration of this receiver

Syntax

```
public void refreshConfig(ReceiverConfig config)
            throws BCGReceiverException
```

Parameters

`config` An object that carries configuration details as specified in the Console

Method

`startReceiving`

Method description

Called by the Receiver Framework in its thread. Once this method is called, the receiver can begin receiving documents. If receiver is of the callback type, it will start processing callbacks in its own thread only after this point. This method should return quickly.

Note: The receiver is responsible for its own thread management.

Syntax

```
public void startReceiving()
             throws BCGReceiverException
```

Parameters

None.

Method

`processResponse`

Method description

In the case of non-blocking synchronous requests, called by the Receiver Framework when the response document has returned from Document Manager. The call comes on a Framework (or internal class) thread. The receiver should return this call quickly.

Syntax

```
public void processResponse(ResponseCorrelation respCorr,
                       ReceiverDocumentInterface response)
             throws BCGReceiverException
```

Parameters

`respCorr`
: An object which contains the information needed to sync the response document to the original request document

`response`
: The response document

Method

`stopReceiving`

Method description

Called by a WBI-C internal component. This method should return quickly. Once the method has been called, receiver should stop receiving documents and perform cleanup. Occurs:

- If an internal component receives a request to terminate
- If receiver is disabled or removed from Console
- If receiver has requested that the Receiver Framework remove it

Syntax

```
public void stopReceiving()
              throws BCGReceiverException
```

Parameters

None

ReceiverDocumentInterface

Represents the document. This object will be created by receiver before it invokes the Framework. It has the following methods:

- getTempObject
- setTempObject
- getAttribute
- setAttribute
- getAttributes
- getTransportHeaders
- setTransportHeaders
- getDocument
- setDocument
- getDocumentUUID

Method

`getTempObject`

Method description

Retrieves temporary information for passing among handlers

Syntax

```
public Object getTempObject(String objectName)
```

Parameters

`objectName`
The name of the object holding the temporary information

Method

`setTempObject`

Method description

Sets temporary information for passing among handlers

Syntax

```
public void setTempObject(String objectName, Object objectValue)
```

Parameters

`objectName`
The name of the object holding the temporary information

objectValue
The temporary information

Method

`getAttribute`

Method description

Retrieves Console defined attribute

Syntax

```
public Object getAttribute(String name)
```

Parameters

name The attribute

Method

`setAttribute`

Method description

Sets Console defined attribute

Syntax

```
public void setAttribute(String name, Object value)
```

Parameters

name The attribute

value The value to be set on the attribute

Method

`getAttributes`

Method description

Retrieves the entire attribute Map

Syntax

```
public Map getAttributes()
```

Parameters

None

Method

`getTransportHeaders`

Method description

Retrieves transport headers

Syntax

```
public HashMap getTransportHeaders()
```

Parameters

None

Method

`setTransportHeaders`

Method description

Sets transport headers

Syntax

```
public setTransportHeaders(Hashmap transportHeaders)
```

Parameters

`transportHeader`
The transport headers

Method

`getDocument`

Method description

Retrieves the document content as a file

Syntax

```
public File getDocument()
```

Parameters

None

Method

`setDocument`

Method description

Sets the document content in the file object

Syntax

```
public void setDocument(File document)
```

Parameters

`document`
The message content

Method

`getDocumentUUID`

Method description

Gets the unique reference id for this document. Every document is assigned a unique id.

Syntax

```
getDocumentUUID()
```

Parameters

None

ReceiverFrameworkInterface

This interface specifies the methods available in the Receiver Framework, which are as follows:

- `remove`
- `preProcess`
- `syncCheck`
- `postProcess`
- `process`
- `setResponseStatus`

Method

`remove`

Method description

Called by the receiver when it detects a fatal condition. The receiver should only call this method if it cannot continue receiving. The Framework marks this receiver for removal and returns immediately. Later an internal WBI-C component calls the `stopReceiving` method on the receiver object.

Syntax

```
public void remove(String transportType)
                   throws BCGReceiverException
```

Parameters

transportType
A string identifying the receiver by the transport it supports

Method

`preProcess`

Method description

Called by the receiver to do the preprocessing of the document based on what pre-processing handlers, either Connect-supplied or user-defined, have been specified for this target via the Console. The Framework executes these handlers by passing the request document as input. The processed document returned by one handler is fed as an input to the second handler and so on. Handlers are called in the order specified in the Console target configuration screen. The resultant documents are returned as an array.

Syntax

```
public ReceiverDocumentInterface[] preProcess(
                                   String transportType,
                                   String target,
                                   ReceiverDocumentInterface request)
               throws BCGReceiverException
```

Parameters

transportType
A String identifying the receiver by the transport it supports

target A String identifying the target, or receiver configuration

request
The request document to be processed

Method

```
syncCheck
```

Method description

Called by receiver to access the list of Console configured syncCheck handlers, including user-provided handlers. The Framework will iterate through the list until it finds an appropriate handler. *True* indicates that the request is synchronous. *False* indicates that the request is configured to be asynchronous or that there are no syncCheck handlers configured for this receiver, which indicates that the request should be handled asynchronously.

Syntax

```
public boolean syncCheck(String transportType, String target,
                              ReceiverDocumentInterface request)
                   throws BCGReceiverException
```

Parameters

transportType
A String identifying the receiver by the transport it supports

target A String identifying the target, or receiver configuration

request
The request document to be processed

Method

```
postProcess
```

Method description

In the case of a synchronous request, the receiver calls the Framework to do the post processing of the response document based on which post -processing handlers, either Connect-supplied or user-defined, have been specified for this target via the Console. The Framework executes these handlers by passing the response document as the input. The processed document returned by one handler is fed as an input to the second handler and so on. Handlers are called in the order specified in the Console target configuration screen The resultant documents are returned as an array.

Syntax

```
public ReceiverDocumentInterface[] postProcess(
                                   String receiverType,
                                   String target,
                                   ReceiverDocumentInterface request)
               throws BCGReceiverException
```

Parameters

receiverType
A string identifying the receiver

target A string identifying the target, or receiver configuration

request
The response document to be processed

Method

`process`

Method description

The main processing method. When it is called, the Framework generates a unique id (UUID) for the request and writes the data into the internally necessary file set in the appropriate input directory. It also sets the UUID on the request document. The method has three distinct signatures, depending on the type of processing (async, blocking sync, or non-blocking sync) required.

Note: The method takes only one request document at a time. If there are multiple documents as a result of pre-processing, it is the receiver's responsibility to iterate through the array and call process for each document.

Syntax

Async request

```
public void process(String transportType, ReceiverDocumentInterface request)
                throws BCGReceiverException
```

Blocking sync request

```
public void process(String transportType,
                    ReceiverDocumentInterface request,
                    ReceiverDocumentInterface response)
                throws BCGReceiverException
```

Non-blocking sync request

```
public void process(String transportType,
                    ReceiverDocumentInterface request,
                    ResponseCorrelation responseCorr)
                throws BCGReceiverException
```

Parameters

`transportType`
A string identifying the receiver

`request`
The input document

`response`
The blank document to hold the response from Document Manager

`responseCorr`
The Response Correlation object that holds information allowing the receiver to sync the original request document with the response document to be returned from Document Manager.

Method

`setResponseStatus`

Method description

Notifies the Framework of the status of the synchronous response document after it has been returned to the originating host

Syntax

```
public void setResponseStatus(String documentUUID,
                    boolean status, String statusMessage)
            throws BCGReceiverException
```

Parameters

documentUUID
The document's unique id

status A boolean representing the state of the response document

statusMessage
Information related to the status of the response document

ReceiverConfig

This object stores receiver configuration information. It provides the following methods:

- `getTransportType`
- `getConfigs`
- `getAttribute`
- `setAttribute`
- `getTargetConfig`
- `getTargetConfigs`

Method

`getTransportType`

Method description

Retrieves the type of receiver

Syntax

```
public String getTransportType()
```

Parameters

None

Method

`getConfigs`

Method description

Retrieves the configuration properties of a receiver

Syntax

```
public Map getConfigs()
```

Parameters

None

Method

`getAttribute`

Method description

Retrieves the value of a configuration property

Syntax

```
public Object getAttribute(String configName)
```

Parameters

`configName`
The name of the property

Method

`setAttribute`

Method description

Sets the value of a configuration property

Syntax

```
public void setAttribute(String configName, Object value)
```

Parameters

`configName`
The name of the property

`value` Definition

Method

`getTargetConfig`

Method description

Retrieves target configuration

Syntax

```
public Config getTargetConfig(String targetName)
```

Parameters

`targetName`
The name of the target

Method

`getTargetConfigs`

Method description

Retrieves configuration of all targets

Syntax

```
public List getTargetConfigs()
```

Parameters

None

ResponseCorrelation

This interface provides a generic way to persist information needed to sync up a request with a response when non-blocking synchronous processing has been invoked. It provides the following methods:

- `set`
- `get`

Method

`set`

Method description

Sets the sync-enabling data

Syntax

```
public Object set(Serializable key, Serializable value)
            throws NullPointerException
```

Parameters

key The key for the sync-enabling data. For example, a JMS receiver would store the JMS correlation ID, so the call would look like:

```
ResponseCorrelation respCorrel = new ResponseCorrelation()
respCorrel.set (CORREL_ID_STRING, correlID);
```

There might be multiple types of information that need to be stored, depending on the transport type.

value The value to be set

Method

`get`

Method description

Retrieves the stored sync-enabling data

Syntax

```
public Object get(Serializable key)
```

Parameters

key The key in which the sync-enabling data is stored. There might be multiple types of information that need to be retrieved, depending on the transport type.

BCGReceiverException

This is a utility class which extends BCGException, which extends Exception. It has four distinct constructors:

```
public class BCGReceiverException extends BCGException {
                public BCGReceiverException()
                {}

                public BCGReceiverException(String s)
                    {}

                public BCGReceiverException(String s,
                                       String args,
                                               String eventCode,
                                               String resourceId)
                    {}

                 public BCGReceiverException(String s,
                                                String args,
                                                String eventCode,
                                                String resourceId,
                                                Throwable th)
                    {}
}
```

Parameters

s A generic description of the exception

args Arguments used to raise the exception

eventCode
The event code associated with the exception

resourceID
The resource ID on which the event was raised

th Throwable object used to raise the event

ReceiverPreProcessHandlerInterface

This interface describes the three methods which all pre-process handlers must implement. They include:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
              throws BCGReceiverException
```

Parameters

`context`
Runtime information such as temp directory path

`handlerConfig`
The object storing configuration information

Method

`applies`

Method description

Determines if handler can handle request document

Syntax

```
public boolean applies(ReceiverDocumentInterface request)
             throws BCGReceiverException
```

Parameters

`request`
The request document

Method

`process`

Method description

Performs pre-processing. Returns an array of documents

Syntax

```
public ReceiverDocumentInterface[] process(
                                       ReceiverDocumentInterface request)
              throws BCGReceiverException
```

Parameters

`request`
The request document

ReceiverSyncCheckHandlerInterface

This interface describes the three methods which all syncCheck handlers must implement. They include:

- `init`
- `applies`
- `syncCheck`

Sync check handlers are configured through the Community Console just as other handlers are.

Method

`init`

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
              throws BCGReceiverException
```

Parameters

`context`
Runtime information such as temp directory path

`handlerConfig`
The object storing configuration information

Method

`applies`

Method description

Determines if handler can handle request document

Syntax

```
public boolean applies(ReceiverDocumentInterface request)
             throws BCGReceiverException
```

Parameters

`request`
The request document

Method

`syncCheck`

Method description

Performs check to see if document is to be processed synchronously. The method returns *true* if the request is synchronous, *false* if asynchronous.

Syntax

```
public boolean syncCheck(ReceiverDocumentInterface request)
              throws BCGReceiverException
```

Parameters

`request`
The request document

Sync check handlers are configured through the Community Console just as other handlers are.

ReceiverPostProcessHandlerInterface

This interface describes the three methods which all post-processing handlers must implement. They include:

- `init`
- `applies`
- `process`

Method

```
init
```

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
             throws BCGReceiverException
```

Parameters

`context`
Runtime information such as temp directory path

`handlerConfig`
The object storing configuration information

Method

```
applies
```

Method description

Determines if handler can handle response document

Syntax

```
public boolean applies(ReceiverDocumentInterface response)
            throws BCGReceiverException
```

Parameters

`response`
The response document

Method

```
process
```

Method description

Performs post-processing on response document. Returns document array.

Syntax

```
public ReceiverDocumentInterface[] process(
                            ReceiverDocumentInterface response)
             throws BCGReceiverException
```

Parameters

`response`
The response document

BCGReceiverUtil

Various static utility methods which include:

- `public static ReceiverDocumentInterface createReceiverDocument()`

  Creates the response document
- `public static ReceiverFrameworkInterface getReceiverFramework()`

  Gets a Receiver Framework object
- `public static File getTempDir()`

  Gets a location for temporary storage of incoming data
- `public static File getRejectDir()`

  Gets a location for archiving rejected messages
- `public static File getArchiveDir(String receiverType, String targetName)`

  Gets a location for the receiver's archive directory. Parameters are the type of receiver (http, JMS, etc.) and the name of this receiver's target.

Events

The following is a list of events available for receiver execution flow.

Informational events

BCG_103207 - Receiver Entrance
Event Text: Receiver ({0}) entrance.
Expected values for arguments: {0} - Receiver class name

BCG_103208 - Receiver Exit
Event Text: Receiver ({0}) entrance.
Expected values for arguments: {0} - Receiver class name

Warning events

BCG_103204 - Target Processing Warning
(This is a General Operational warning and the receiver is expected to run. An example: an error in closing the queue connection)
Event Text: Target '{0},{1}' processing document warning, reason: {2}.
Expected values for arguments: {0} - Target name
{1} - Target (Receiver) type
{2} - Warning reason specific to Target

Error events

BCG_103203 - Target Processing Error
Event Text: Target '{0},{1}' failed to processing document, error: {2}.
Expected values for arguments: {0} - Target name
{1} - Target (Receiver) type
{2} - Exception message

BCG_103205 - Target Error
(An example: a failure in opening a queue connection)
Event Text: Target '{0},{1}' failed to process target: {2}.
Expected values for arguments: {0} - Target name
{1} - Target (Receiver) type
{2} - Exception message

Example Receiver Implementation Outline

The following code and pseudo code outlines an example implementation for a JMS Receiver.

```
public class CustomJMSReceiver implements ReceiverInterface {
         private Context m_context = null;
         private ReceiverConfig m_rcvConfig = null;
         public void init(Context context, ReceiverConfig receiverConfig) {
           this.m_context = context;
           this.m_rcvConfig = receiverConfig;
return;
}
public void refreshConfig(ReceiverConfig rcvconfig)
                 throws BCGReceiverException {
          this.m_rcvConfig = rcvconfig;

// Check which receiver targets are updated, added newly or deleted
// If new target is added, create a new thread and start polling the target
// If current target is updated, stop the thread which is polling the
// target, and using the updated configuration information, start polling
// If current target is deleted, stop the thread which is polling the
// target and delete the thread which is responsible for polling the
//target.
...
return;
}

 public void startReceiving() throws BCGReceiverException {
// Read the list of targets in the ReceiverConfig object
// For each target create a UserJMSThread and start the thread
return;
}
public void processResponse(ResponseCorrelation respCorr,
                             ReceiverDocumentInterface response)
                    throws BCGReceiverException {
// get the correlation information like reply-to-queue, correlation id
// and send the response to that queue
return;
}
public void stopReceiving() throws BCGReceiverException {
// get the list of UserJMSReceiverThreads associated with each target
// call stop method.
...
return;
}
private class UserJMSReceiverThread extends Thread {
 public UserJMSReceiverThread(Config targetConfig) {
// create the queue session, connection, queue receiver
...
}
public void run() {
 while (true) {
       try{
          // call receive method on the queue
          // if message is available read the message and process the
          // document

             processDocument(data);

          // else continue to poll the queue.

        ...

       } catch(Exception e) {

          ...
```

```
      }
  }
}
//Upon receiving the document from the queue, start processing the
//documenting using Receiver FrameWork APIs

public void processDocument(byte[] data) throws BCGReceiverException{
             //Get the temporary location where data can be written
         File tempDir = BCGReceiverUtil.getTempDir();
            // Now create the temp file and write the data into it
         File fileLocation = new File(tempDir, fileStr);
         FileOutputStream fos = new FileOutStream(fileLocation);
         fow.write(data);
         fos.close();
// Create the ReceiverDocument object
        ReceiverDocumentInterface request =
                             BCGReceiverUtil.createReceiverDocument();
       // set document, transport headers and BCG headers in the
       // request
        request.setDocument(fileLocation);
     ...
      //Now start processing the document using ReceiverFrameWork APIs
          ReceiverFrameWorkInterface rcvFramework =
                                BCGReceiverUtil.getReceiverFramework();

          ReceiverDocumentInterface requestDocs[] =
                rcvFramework.preprocess(transportType,target,request);
      //Check if the requestDocs length is 1, if yes document is not
      //split into multiple documents
         boolean sync =
                      rcvFramework.syncCheck(transportType,target,request);
     ...
        if(!sync) {
            //request is not synchronous message
              rcvFramework.process(transportType,target,request);
      }
 }
```

Chapter 4. Customizing Fixed and Variable Workflow

The Business Processing Engine (BPE), the heart of the Document Manager component, is responsible for performing the main transformations at the core of the WBI Connect process. It does so by following a series of distinct processing steps, grouped into three basic units: fixed inbound workflow, variable workflow, and fixed outbound workflow. User exits allow user defined processes to be plugged into each of these basic units.

Fixed inbound and fixed outbound workflow cover standard processing that all documents undergo as they flow into and out of the main processing stage. They are called fixed because the number and type of processing steps are always the same. Main processing takes place in variable workflow, where the number and type of processing steps are completely dependent on the handling requirements of the specific situation. As of the 4.2.2 release, users may customize the workflow stage of WBI-Connect processing in two ways, by creating custom handlers for certain steps in fixed workflows, and by defining new actions (named sequences of steps) in the variable workflow stage. This chapter covers both ways of customizing workflow:

- "Overview for creating handlers in fixed inbound workflow"
- "Overview for creating actions in variable workflow" on page 42
- "Overview for creating handlers in fixed outbound workflow" on page 45

An additional section covers development and deployment issues.

- "Development and deployment" on page 46

An API listing and example code follows in the next chapter. Also included in this listing is information on a number of utility, security, and classes common to all WBI-C components.

Overview for creating handlers in fixed inbound workflow

Before incoming messages can be dealt with, all packaging and protocol specific information with which they are bundled must be extracted and processed. The first two steps in the fixed inbound workflow are designed to accomplish these tasks. WBI-C ships with code to handle RNIF, AS2, MIME, EAI, and NONE packaging and XML, RosettaNet, and EDI protocols. To add new packaging types or to support new protocols, users can write their own handlers, using an API, the `BusinessProcessHandlerInterface`, provided with this release. These new handlers must be configured using the Community Console and integrated into the processing flow in the normal way. For more on the configuration process, see the *Hub Configuration Guide*. This section provides a functional overview of the two user customizable steps in the fixed inbound workflow process. It covers:

- "Protocol unpackaging handlers"
- "Protocol processing handlers" on page 41

Protocol unpackaging handlers

Received messages in the BPE are wrapped in an object that implements the `BusinessDocumentInterface`. This wrapper object, often simply called the business document, contains both the received message, including the transport level and WBI-C defined headers added during the Receiver process, and a variety of metadata associated with the message. BPE uses this metadata in its processing. The first step in fixed processing consists of un-packaging or de-enveloping the business document. This may include any or all of the following steps:

Decryption
If the message is encrypted

Decompression
If the message is compressed

Signature verification
If the message is signed

Routing information extraction
- From, to, and initiating business IDs, if the packaging provides them
- From packaging and versions, such RNIF, v02.00

User authentication
If the packaging provides user information

Business document parts extraction
If the packaging specifies location of various message parts such as payload, attachments, and so forth.

Note: To support decryption and signature verification in user-defined handlers, there are security service utility classes available through the API.

When it is complete, this step produces the un-packaged message. In addition, the handler must update the meta information in the business document in the following ways:

- **Update package level information attributes**:

Packaging attribute name	Description
`FRPACKAGINGNAME`	From: packaging name. This should be the packaging name defined in the Console.
`FRPACKAGINGVER`	From: packaging version. This should be the version defined in the Console.
`PKG_FRBUSINESSID`	If the packaging provides sender's business ID, it should be specified here.
`PKG_TOBUSINESSID`	If the packaging provides destination's business ID, it should be specified here.

- **Update the status of the document**: if the handler cannot un-package the business document, or there are any other fatal errors, the handler should
  - add an event to the business document
  - set the state of the document to fail
  - throw an exception derived from `BCGUserException` or `BCGException`.
- **Optionally, generate events**: handlers may generate document related events in the following three categories:
  - Informational: these events provide additional information in the execution flow. They do not affect the processing of the document.
  - Warning: problems that arise but do not stop the processing flow. An example would be an exception thrown while closing a queue connection after the document has been received and processed.
  - Errors: problems that arise and stop the further processing of the document.

When a user exit returns control to the BPE, the BPE inspects the business document for new events. It will log the events, and WBI-C will update activity tables and generate any necessary alerts. A listing of events available for logging issues in the Workflow stage is presented in the following API chapter.

Protocol processing handlers

After business documents are processed by any un-packaging handlers, they are handed off to the second fixed step, the protocol processing handler. This handler is responsible for parsing the business document to determine:

Routing information
Sender ID and Destination ID

Protocol information
The business protocol and version associated with this business document. Example: `Rosettanet version V02.02`

Document flow/process information
The document flow and version associated with this business document. Example: `3A4 version V02.02`.

Once this information is determined, the handler must update the meta information in the business document in the following ways:

- **Update protocol level information attributes**:

Protocol attribute name	Description
`FRBUSINESSID`	From: business ID from the message.
`TOBUISNESSID`	To: business ID from the message.
`INITBUSINESSID`	This could be the same as `FRBUSINESSID` or it could be a different ID, depending on the protocol.
`FRPROTOCOLNAME`	The protocol name associated with the incoming business document. This should be a valid process name as defined in the Console. An example would be `Rosettanet`
`FRPROTOCOLVER`	The protocol version associated with the incoming business document. This should be a valid process version as defined in the Console. An example would be `V02.00`.
`FRPROCESSCD`	The process code associated with the incoming business document. This should be a valid code as defined in the Console. An example would be `3A4`.
`FRPROCESSVER`	The process version associated with the incoming business document. This should be a valid process version as defined in the Console. An example would be `V02.00`.

- **Update the status of the document**: if the handler cannot parse the business document, or there are any other fatal errors, the handler should
  - add an event to the business document
  - set the state of the document to fail
  - throw an exception derived from `BCGUserException` or `BCGException`.
- **Optionally, generate events**: handlers may generate document related events in the following three categories:

– Informational: these events provide additional information in the execution flow. They do not affect the processing of the document.
– Warning: problems that arise but do not stop the processing flow. An example would be an exception thrown while closing a queue connection after the document has been received and processed.
– Errors: problems that arise and stop the further processing of the document.

When a user exit returns control to the BPE, the BPE inspects the business document for new events. It will log the events, and WBI-C will update activity tables and generate any necessary alerts. A listing of events available for logging issues in the Workflow stage is presented in the following API chapter.

Overview for creating actions in variable workflow

When the inbound workflow path is completed, the appropriate variable workflow path is determined by referencing the Participant Connection This workflow path, or Action, is specified in the Console configuration process. For more information on using the Console to configure Participant Connections and variable workflows, see the *Hub Configuration Guide*.

An action is a number of steps, arranged in a sequence. WBI-C ships with 11 predefined actions. Should other options be required, users can customize variable workflow by defining new actions, either by developing an entirely new set of steps, placed into a new sequence or by copying an existing action and modifying it by adding a step, deleting or replacing a pre-existing step, or modifying the order of the steps.

Note: Not all WBI-C delivered steps can be used in new, user-defined actions, as they may be used for internal WBI-C specific purposes. See "WBI-C supplied actions and their status as templates" on page 43 for more detailed information.

Actions consist of sequences of steps. Typically steps may include the following types:

- **Validation**: checking the form of the business document. For example, an XML document can be validated against an XML schema. WBI-C provides an XML validation step, but others could be developed.
- **Transformation**: changing the form of the business document. An XML document can be transformed into a different XML document using XSLT. Again, WBI-C provides an XML transformation step, but others could be developed.
- **Translation**: changing the entire format of a business document from one type to another.
- **Process logging**: creating a log specific to this document. Some business protocols have specific logging requirements.
- **Sequence checks**: checking the sequence of documents. Some business protocols have specific sequencing requirements.

Note: These steps are typical examples only. Variable workflow is designed to implement business processing logic. The logic will dictate the actual steps required.

Creating steps

Creating a step is a two part process. The actual processing logic resides in a class that implements the `BusinessProcessInterface` interface. But the BPE gets one of these objects by calling the `getBusinessProcess` method on another user-supplied class, a Factory class which implements the `BusinessProcessFactoryInterface` interface and takes care of constructing the processing logic object. A step is defined by a single Factory class.

Once steps have been defined, they are ordered into named sequences using the Community Console. For more information, see the *Hub Configuration Guide*.

As in the case of the fixed inbound handlers, fatal variable workflow processing errors should cause an update in the business document, with its state set to fail and an event set. A `BCGUserException` or a `BCGException` should also be thrown. Other events may be set as well, as needed. A listing of events available for logging issues in the Workflow stage is presented in the following API chapter.

WBI-C supplied actions and their status as templates

WBI-C ships with 11 pre-defined actions. Some, but not all of these actions, and the steps that make them up, are available for user customization. Below is a list of the supplied actions and the degree they may be utilized for user customized actions.

1. Pass through

This action can be copied and modified. The two individual existing steps may not be modified, but an additional step (for example, an XML validation step) can be prepended to the action sequence.

2. HubOwner cancellation of RN process

This action cannot be copied and modified. It is specific to the RosettaNet protocol. The `ValidationFactory` step in it, however can be reused in user actions.

3. RN pass through with process logging

This action can be copied and modified. As delivered, this action provides for the logging of RosettaNet document flow. It can be modified for the logging of user defined protocol document flow by replacing the initial step (`ProcessLoggingFactory`) with a user-defined step.

4. Bi-directional translation of RN and RNSC

This action cannot be copied and modified. It is used for RNIF messages. The `ValidationFactory` step can be reused in user actions.

5. Bi-directional translation of RN and XML

This action cannot be copied and modified. It is used for RNIF messages.However the `ValidationFactory` and the `OutboundValidationFactory` steps can be reused in user actions.

6. Bi-directional translation of Custom XML with validation

This action can be copied and modified. User defined steps can be substituted for the following three supplied steps: `ValidationFactory`, `XSLTTranslationFactory`, and `OutboundValidationFactory`.

The `ValidationFactory` step validates the received Custom XML document. A replacement step needs to set X_PAYLOAD_ROOT_TAG (x-aux-payload-root-tag) to the XML document name on exiting, and, if successful, place a validation successful event.

The `XSLTTranslationFactory` transforms the received XML document into its outbound XML format. A replacement step needs to set TRANSFORMED_DOC attribute to 'true' and return the transformed document to the business document upon exiting.

The `OutboundValidationFactory` validates the transformed document.A replacement step needs to set X_PAYLOAD_ROOT_TAG (x-aux-payload-root-tag) to the XML document name on exiting, and, if successful, place a validation successful event.

All of the substitutable steps above can also be used in user actions.

7. Bi-directional translation of Custom XML with duplicate check and validation

This action can be copied and modified. User defined steps can be substituted for the following three supplied steps: `ValidationFactory`, `XSLTTranslationFactory`, and `OutboundValidationFactory`.

The `ValidationFactory` step validates the received Custom XML document. A replacement step needs to set X_PAYLOAD_ROOT_TAG (x-aux-payload-root-tag) to the XML document name on exiting, and, if successful, place a validation successful event.

The `XSLTTranslationFactory` transforms the received XML document into its outbound XML format. A replacement step needs to set TRANSFORMED_DOC attribute to 'true' and return the transformed document to the business document upon exiting.

The `OutboundValidationFactory` validates the transformed document.A replacement step needs to set X_PAYLOAD_ROOT_TAG (x-aux-payload-root-tag) to the XML document name on exiting, and, if successful, place a validation successful event.

All of the substitutable steps above can also be used in user actions.In addition, the `ContentDuplicateProcessFactory` can also be used in user actions.

8. Bi-directional translation of Owner Custom XML to RN with duplicate check and validation

This action cannot be copied or modified. It is specific to RNIF messages. However the `ValidationFactory` and the `ContentDuplicateProcessFactory` steps can be used in user actions.

9. Custom XML pass through with duplicate check and validation

This action can be copied and modified. A user-defined step may be substituted for the `ValidationFactory` step.

The `ValidationFactory` step validates the received Custom XML document. A replacement step needs to set X_PAYLOAD_ROOT_TAG (x-aux-payload-root-tag) to the XML document name on exiting, and, if successful, place a validation successful event.

The contained `ValidationFactory` step and the `ContentDuplicateProcessFactory` step can be used in user actions.

10. Custom XML pass through with duplicate check

This action cannot be copied and modified. However the `ContentDuplicationProcessFactory` step can be used in user actions.

11. Custom XML pass through with validation

This action can be copied and modified. A user-defined step may be substituted for the `ValidationFactory` step.

The `ValidationFactory` step validates the received Custom XML document. A replacement step needs to set X_PAYLOAD_ROOT_TAG (x-aux-payload-root-tag) to the XML document name on exiting, and, if successful, place a validation successful event.

The contained `ValidationFactory` step can also be used in user actions.

Overview for creating handlers in fixed outbound workflow

The last stage of processing in the BPE consists of packaging or enveloping the outbound message as specified by destination packaging settings. WBI-C provides handlers for RNIF, EAI, AS, and NONE packaging and cXML and SOAP protocols. To add new packaging, for example, SOAP with attachments packaging, users can write their own handler, using an API, the `BusinessProcessHandlerInterface`, provided with this release. This packaging handler must be configured using the Community Console and integrated into the processing flow in the normal way. For more on the configuration process, see the *Hub Configuration Guide*. This section provides a functional overview of this user customizable step in the fixed outbound workflow process. It covers:

- "Protocol packaging handlers"

Protocol packaging handlers

Protocol packaging may include one or more of the following steps:

Assembling or enveloping
If the business protocol requires the message to be packaged as different parts, such as payload, attachments, and so forth

Encrypting
If the packaging type requires encryption

Signing
If the packaging type requires signatures

Compressing
If the packaging type requires compression

Specifying transport headers
As appropriate

The output of this process is the packaged message. The message is written to the in_process directory where it will be picked up and passed on to the Sender component to be processed and transmitted to the destination. Its location is updated in the business document object. The handler must also update the meta information in the business document in the following ways:

- **Update protocol packaging attributes**:

Protocol attribute name	Description
`OUTBOUNDTRANSPORTHEADERS`	The value for this attribute is a HashMap object that contains the list of transport headers that have been set in the outbound message. For example, when an RNIF 2.0 message is sent to a trading partner, the following RN headers are set: • `x-rn-response-type=async` • `x-rn-version=RosettaNet/V02.00`

- **Update the status of the document**: if the handler cannot package the message, or there are any other fatal errors, the handler should
  - add an event to the business document
  - set the state of the document to fail
  - throw an exception derived from `BCGWorkflowException`.
- **Optionally, generate events**: A list of events available for variable workflow use is in the Workflow API chapter. When a user exit returns control to the BPE, the BPE inspects the business document for new events. It will log the events, and WBI-C will update activity tables and generate any necessary alerts.

Development and deployment

The following sections describe development and deployment for both user created handlers in fixed workflows and user created steps in variable workflow.

Development environment

The workflow development API relies on classes and interfaces from three packages:

- `com.ibm.bdg.bcgdk.workflow`
- `com.ibm.bcg.bcgdk.common`
- `com.ibm.bcg.bcgdk.services`

These packages are part of the bcgsdk.jar which is found in the WBI-C installable in the following directories:

- `<`*`install dir`*`>\router\was\wbic`
- `<`*`install dir`*`>\receiver\was\wbic`
- `<`*`install dir`*`>\console\was\wbic`

In all deployed instances, this .jar file should be available in the application server classpath and not in the module classpath.

For development, the bcgsdk.jar file should be included in the build path of the project that contains the user exit classes, i.e., in the classpath.

Deployment and packaging

All user created code, including custom handlers and steps, should be packaged and deployed in one of the following ways:

- Placed in a .jar file in `\router\was\wbic\userexits`
- Added as classes in `\router\was\wbic\userexits\classes`

In addition, all handler classes must be accompanied by an XML descriptor file, which allows key data about the handlers to be imported into the Console GUI. In the brief outline that follows, `HandlerClassName` is the name of the handler class; `Description` is a brief description of the class; `HandlerTypeValues` is the component name (RECEIVER - for Receiver handlers, GATEWAY - for sender handlers, FIXEDWORKFLOW - for fixed inbound or outbound handlers, or ACTION - for variable workflow steps) followed by the type of handler (eg, PREPROCESS.<transport>, PROTOCOL.UNPACKAGING, etc); and `ComponentAttribute` is any configurable member variable in the handler class which changes the behavior of the handler in runtime.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:HandlerDefinition
     xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import
                                                          /external/types"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1
                              /import/external bcghandler.xsd
                         http://www.ibm.com/websphere/bcg/2004/v0.1
                              /import/external/types bcgimport.xsd ">
 <tns:HandlerClassName>com.ibm.SampleHandler </tns:HandlerClassName>
 <tns:Description>A Sample Handler</tns:Description>
 <tns:HandlerTypes>
     <tns:HandlerTypeValue>COMPONENT.TYPE</tns:HandlerTypeValue>
 </tns:HandlerTypes>
 <tns:HandlerAttributes>
       <tns2:ComponentAttribute>
               <tns2:AttributeName>Attribute 1</tns2:AttributeName>
       </tns2:ComponentAttribute>
       <tns2:ComponentAttribute>
           <tns2:AttributeName>Attribute 2</tns2:AttributeName>
            <tns2:AttributeDefaultValue>Attribute2DefaultValue
          </tns2:AttributeDefaultValue>
           </tns2:ComponentAttribute>
 </tns:HandlerAttributes>
</tns:HandlerDefinition>
```

Chapter 5. APIs and Example Code for Workflow Handlers and Steps

The following chapter provides an annotated listing of the APIs provided for developing custom handlers for fixed inbound and outbound workflow and for steps that can be assembled into actions for variable workflow. It also includes listings for utility, security, and other common classes shared across components.

The following classes and interfaces are documented:

From com.ibm.bcg.bcgdk.workflow
- "BusinessProcessFactoryInterface" on page 51
- "BusinessProcessInterface" on page 52
- "BusinessProcessHandlerInterface" on page 53
- "AttachmentInterface" on page 54
- "BusinessProcessUtil" on page 57

From com.ibm.bcg.bcgdk.services
- "SecurityServiceInterface" on page 60
- "MapServiceInterface" on page 64
- "BCGSecurityException" on page 66

From com.ibm.bcg.bcgdk.common
- "Context" on page 68
- "Config" on page 69
- "BusinessDocumentInterface" on page 71
- "exception.BCGException" on page 77
- "BCGUtil" on page 78
- "EventInfo" on page 80
- "BCGDocumentConstants" on page 84

And for workflow events
- "Events" on page 85

Brief examples of code and pseudo-code outlining the implementation of sample protocol processing and protocol unpackaging handlers, and validation and transformation steps are also included. More complete code samples of validation and transformation steps are available in the delivery image in the `DevelopmentKits/UserExits/samples/` directory. More information on these samples can be found in the product readme file.

From com.ibm.bcg.bcgdk.workflow

These are classes and interfaces directly associated with the workflow stage of processing. They include:

- "BusinessProcessFactoryInterface" on page 51
- "BusinessProcessInterface" on page 52
- "BusinessProcessHandlerInterface" on page 53
- "AttachmentInterface" on page 54
- "BusinessProcessUtil" on page 57

BusinessProcessFactoryInterface

Each variable workflow step must implement this factory interface. It has the following methods:

- `getBusinessProcess`
- `returnBusinessProcess`

Method

`getBusinessProcess`

Method description

Gets an instance of `BusinessProcessInterface`. The factory class constructs the `BusinessProcess` instance by calling the appropriate constructor, based on the configuration information that is passed in.

Syntax

```
public BusinessProcessInterface getBusinessProcess(
                                    Context context,
                                    Config workflowConfig,
                                    BusinessDocumentInterface bDoc)
```

Parameters

context
Runtime information such as temp directory path

workflowConfig
Configuration details as specified in the Console

bDoc The business document being processed

Method

`returnBusinessProcess`

Method description

Returns `BusinessProcess` to factory. Called by BPE.

Syntax

```
public void returnBusinessProcess(BusinessProcessInterface bp)
```

Parameters

bp The business process to be returned

BusinessProcessInterface

Each variable workflow step must implement this interface. An instance of the implemented class is what the factory produces. This class executes the actual business logic on the document. It has the following methods:

- `process`
- `reset`

Method

`process`

Method description

Executes the business logic on the business document that is passed in.

Syntax

```
public BusinessDocumentInterface process(
                                     Context context,
                                     BusinessDocumentInterface bDoc)
```

Parameters

context
Runtime information such as temp directory path

bDoc The business document being processed

Method

`reset`

Method description

Resets the `BusinessProcess`. Called by `BusinessProcessFactory`.

Syntax

```
public boolean reset()
```

Parameters

None

BusinessProcessHandlerInterface

Handlers for fixed inbound and fixed outbound workflow must implement this interface. It has the following three methods:

- `init`
- `applies`
- `process`

Method

```
init
```

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context,
                            Config config)
```

Parameters

context
Runtime information such as temp directory path

config Configuration information set by the Console

Method

```
applies
```

Method description

Determines if handler can process business document.

Syntax

```
public boolean applies(BusinessDocumentInterface bDoc)
```

Parameters

bDoc The business document being processed

Method

```
process
```

Method description

Executes the necessary business logic on the document

Syntax

```
public BusinessDocumentInterface process(BusinessDocumentInterface bDoc)
```

Parameters

bDocTerm
The business document being processed

AttachmentInterface

This is a utility interface for handling attachments. It has the following ten methods:

- setContentType
- getContentType
- setDescription
- getDescription
- setURI
- getURI
- setEncoding
- getEncoding
- setFile
- getFile

Method

`setContentType`

Method description

Sets the content type of the attachment.

Syntax

```
public void setContentType(String contentType)
```

Parameters

contentType
The content type

Method

`getContentType`

Method description

Retrieves the content type of the attachment.

Syntax

```
public String getContentType()
```

Parameters

None

Method

`setDescription`

Method description

Sets a string describing the attachment

Syntax

```
public void setDescription(String desc)
```

Parameters

desc	The description

Method

```
getDescription
```

Method description

Retrieves the description

Syntax

```
public String getDescription()
```

Parameters

None

Method

```
setURI
```

Method description

Sets a URI for the attachment

Syntax

```
public void setURI(String URI)
```

Parameters

URI	The URI

Method

```
getURI
```

Method description

Retrieves the URI

Syntax

```
public String getURI()
```

Parameters

None

Method

```
setEncoding
```

Method description

Sets the attachment's character encoding

Syntax

```
public void setEncoding(String encoding)
```

Parameters

encoding
The encoding

Method

```
getEncoding
```

Method description

Retrieves the attachment's character encoding

Syntax

```
public String getEncoding()
```

Parameters

None

Method

```
setFile
```

Method description

Sets a file for the attachment

Syntax

```
public void setFile(File file)
```

Parameters

file The file

Method

```
getFile
```

Method description

Retrieves the file

Syntax

```
public File getFile()
```

Parameters

None

BusinessProcessUtil

This is a WBI-C provided utility class. It has the following methods:

- getSecurityService
- getPartnerService
- getMapService

It also specifies the following constants:

- BCG_DOCSTATE_FAILED = "Failed"
- BCG_DOCSTATE_IN_PROCESS = "In Process"

Method

```
getSecurityService
```

Method description

Retrieves a Security Service object

Syntax

```
public SecurityServiceInterface getSecurityService()
```

Parameters

None

Method

```
getPartnerService
```

Method description

Retrieves a Partner Service object

Syntax

```
public PartnerServiceInterface getPartnerService()
```

Parameters

None

Method

```
getMapService
```

Method description

Retrieves a MapService object

Syntax

```
public MapServiceInterface getMapService()
```

Parameters

None

Constants

These are used for updating the status of the business document.

```
public static final String BCG_DOCSTATE_FAILED = "Failed"
public static final String BCG_DOCSTATE_IN_PROCESS = "In Process"
```

From com.ibm.bcg.bcgdk.services

The following interfaces and classes allow general access to security and mapping services:

- "SecurityServiceInterface" on page 60
- "MapServiceInterface" on page 64
- "BCGSecurityException" on page 66

SecurityServiceInterface

This interface provides access to a range of security related methods and constants. There are five methods, each with two signatures:

- encryptBytes
- verifySignature
- decryptBytes
- signMessage
- generateDigest

There are also five constants:

- BCG_ENC_ALG_DES ="3des"
- BCG_ENC_ALG_RC5 = "rc5"
- BCG_ENC_ALG_RC = "rc2-40"
- BCG_SIGN_ALG_SHA1 = "sha1"
- BCG_SIGN_ALG_MD5 = "md5"

Method

`encryptBytes`

Method description

Encrypts the input

Syntax

Byte array input

```
public byte[] encryptBytes(BusinessDocumentInterface doc,
                                  byte[] buf, String alg)
          throws BCGSecurityException
```

InputStream input

```
public InputStream encryptBytes(BusinessDocumentInterface doc,
                                  InputStream in, String alg)
          throws BCGSecurityException
```

Parameters

doc	The business document
buf	The data (as a byte array) to be encrypted
in	The data (as an InputStream) to be encrypted
alg	The encryption type

Method

`verifySignature`

Method description

Verifies the signature

Syntax

Byte array input

```
   public SignInfo verifySignature (BusinessDocumentInterface doc,
                                    byte[] signature, byte[] buff,
                                    String businessId,
                                    String signatureAlgo)
           throws BCGSecurityException
```

InputStream input

```
public SignInfo verifySignature (BusinessDocumentInterface doc,
                                    byte[] signature,
                                    InputStream in,
                                    String businessId,
                                    String signatureAlgo)
           throws BCGSecurityException
```

Parameters

doc The business document

signature
The signature to be verified

buff The data (as a byte array) whose signature is being verified

in The data (as an InputStream) whose signature is being verified

businessID
The Business ID of the partner against which the signature is to be verified

signatureAlgo
The algorithm used in the signature

Method

```
decryptBytes
```

Method description

Decrypts the input

Syntax

Byte array input

```
public byte[] decryptBytes(BusinessDocumentInterface doc,
                              byte[] buff, String alg)
           throws BCGSecurityException
```

InputStream input

```
public InputStream decryptBytes(BusinessDocumentInterface doc,
                                   InputStream is, String alg)
            throws BCGSecurityException
```

Parameters

doc The business document

buf The data (as a byte array) to be decrypted

is The data (as an InputStream) to be decrypted

alg The encryption type

Method

```
signMessage
```

Method description

Signs the data

Syntax

Byte array input

```
public SignInfo signMessage(BusinessDocumentInterface doc,
                               byte[] data, String alg)
        throws BCGSecurityException
```

InputStream input

```
public SignInfo signMessage(BusinessDocumentInterface document,
                                InputStream is, String alg)
        throws BCGSecurityException
```

Parameters

doc The business document

data The data (as a byte array) to be signed

is The data (as an InputStream) to be signed

alg The encryption type

Method

`generateDigest`

Method description

Generates a digest

Syntax

Byte array input

```
public byte[] generateDigest(byte[] data, String alg)
               throws BCGSecurityException
```

InputStream input

```
public byte[] generateDigest(InputStream dataStream, String alg)
              throws BCGSecurityException
```

Parameters

data The data as a byte array

dataStream
The data as an InputStream

alg The encryption type

SecurityService methods update the business document with the following information whenever applicable:

Attribute name	Description
SIGNING_PUBLIC_CERT_ID	Identifier of the certificate used for signing the document

Attribute name	Description
SIGNING_PRIVATE_KEY_ID	Identifier of the key used for signing the document. This is the hub owner's private key
ENCRYPTION_PUBLIC_CERT_ID	Identifier of the participant's public key for encrypting outbound documents
ENCRYPTION_PRIVATE_KEY_ID	Identifier of the hub owner's private key for decrypting inbound documents
DIGSIGALGORITHM	Algorithm used to sign a message of type 'sha1' or 'md5'
DOC_ENCRYPT_ALGO	Encryption algorithm used for message of type '3des','rc5' etc.

For reference, the `SignInfo` class (returned in some of the above methods):

```
package com.ibm.bcg.bcgdk.services;
public class SignInfo {
 private byte[] data;// signature data
 private byte[] digest;// message digest
 public byte[] getData();
 public void setData(byte[] data);
 public byte[] getDigest();
 public void setDigest(byte[] digest);
}
```

Constants

These define encryption and signature types:

```
public final String BCG_ENC_ALG_DES="3des"
public final String BCG_ENC_ALG_RC5 = "rc5"
public final String BCG_ENC_ALG_RC2 = "rc2-40"
public final String BCG_SIGN_ALG_SHA1="sha1"
public final String BCG_SIGN_ALG_MD5 = "md5"
```

MapServiceInterface

This interface provides access to validation and transformation maps. There are three methods, as follows:

- getFromValidationMap
- getToValidationMap
- getTransformationMap

Method

```
getFromValidationMap
```

Method description

Retrieves the appropriate From validation map

Syntax

```
public byte[] getFromValidationMap(Context context,
                           BusinessDocumentInterface document)
```

Parameters

context
Runtime information such as temp directory path

document
The business document

Method

```
getToValidationMap
```

Method description

Retrieves the appropriate To validation map

Syntax

```
public byte[] getToValidationMap(Context context,
                           BusinessDocumentInterface document)
```

Parameters

context
Runtime information such as temp directory path

document
The business document

Method

```
getTransformationMap
```

Method description

Retrieves the appropriate transformation map

Syntax

```
public byte[] getTransformationMap(Context context,
                          BusinessDocumentInterface document)
```

Parameters

context
Runtime information such as temp directory path

document
The business document

BCGSecurityException

This is a utility class which extends BCGException, which extends Exception. It has two constructors:

```
BCGSecurityException()
```

```
BCGSecurityException(String s)
```

From com.ibm.bcg.bcgdk.common

These are general utility classes and interfaces common to all stages of WBI-C processing. They include:

- “Context” on page 68
- “Config” on page 69
- “BusinessDocumentInterface” on page 71
- “exception.BCGException” on page 77
- “BCGUtil” on page 78
- “EventInfo” on page 80
- “BCGDocumentConstants” on page 84

Context

This class can be used to store various sorts of runtime information, such as temp directory paths. The BPE, for example, uses it to store a reference to the database connection object. It contains two methods:
- getContext
- setContext

Method

```
getContext
```

Method description

Retrieves the stored information

Syntax

```
public Object getContext(String contextName)
```

Parameters

contextName
The name associated with the information

Method

```
setContext
```

Method description

Sets the information

Syntax

```
public void setContext(String contextName, Object context)
```

Parameters

contextName
The name associated with the information

context
The information

Config

This class holds target configuration information. It has four methods.

Note: This class is not thread safe.
- getName
- getAttribute
- setAttribute
- getAttributes

Method

```
getName
```

Method description

Retrieves the name of the target

Syntax

```
public String getName()
```

Parameters

None

Method

```
getAttribute
```

Method description

Retrieves the value of a configuration property

Syntax

```
public Object getAttribute(String name)
```

Parameters

name	The name of the property

Method

```
setAttribute
```

Method description

Sets the value of a configuration property

Syntax

```
public void setAttribute(String name, Object value)
```

Parameters

name	The name of the property
value	The value to be set

Method

```
getAttributes
```

Method description

Retrieves a Map object holding all of the properties of this target

Syntax

```
public Map getAttributes()
```

Parameters

None

BusinessDocumentInterface

This interface represents the business document that is being processed. It has 18 methods.

- getDocumentUUID
- getDocumentParentUUID
- createFile
- getDocument
- setDocument
- getOriginalFile
- getDocumentState
- setDocumentState
- addEvents
- getEvents
- getAttribute
- setAttribute
- getTempObject
- setTempObject
- getAttachments
- addAttachments
- getTransportHeaders

Method

```
getDocumentUUID
```

Method description

Retrieves the Unique ID associated with this document

Syntax

```
public String getDocumentUUID()
```

Parameters

None

Method

```
getDocumentParentUUID
```

Method description

Retrieves the Unique ID associated with this document's parent

Syntax

```
public String getDocumentParentUUID()
```

Parameters

None

Method

`createFile`

Method description

Creates a File object for use in writing files *other* than the Business Document to the file system. Creates a name based on the Business Document name. This method is used, for example, in handling attachments. It is also used to create the File object in the case of a sync response document which must then be set in the Sender Result object.

Syntax

```
public File createFile()
```

Parameters

None

Method

`getDocument`

Method description

Retrieves a File reference for the Business Document in the WBI-C working directory

Syntax

```
public File getDocument()
```

Parameters

None

Method

`setDocument`

Method description

Writes the document to the WBI-C working directory

Syntax

```
public void setDocument(File document)
```

Parameters

document
The business document

Method

`getOriginalFile`

Method description

Gets the location of the original file, that is, the one that came into the in_process directory (the WBI-C working directory)

Syntax

```
public File getOriginalFile()
```

Parameters

None

Method

`getDocumentState`

Method description

Retrieves the status of the document

Syntax

```
public String getDocumentState()
```

Parameters

None

Method

`setDocumentState`

Method description

Sets the status of the document

Syntax

```
public String setDocumentState(String state)
```

Parameters

state The status to be set

Method

`addEvents`

Method description

Adds events to be associated with this document. These events will be displayed in the event viewer and the document viewer.

Syntax

```
public void addEvents(EventInfo[] events)
```

Parameters

events The array of EventInfo objects to be added

Method

`getEvents`

Method description

Retrieves the array of EventInfo objects associated with this document.

Syntax

```
public EventInfo[] getEvents()
```

Parameters

None

Method

`clearEvents`

Method description

Clears events set till now. When BPE has finished processing all events for the business process, it calls this method to clear all events so that it does not process the same events twice.

Syntax

```
public void clearEvents()
```

Parameters

None

Method

`getAttribute`

Method description

Gets the named attribute. Used to retrieve information such as packaging name and version, etc. For list of available attributes refer to the `DocumentConstant` class below.

Syntax

```
public Object getAttribute(String attrName)
```

Parameters

attrName
The name of the attribute requested

Method

`setAttribute`

Method description

Sets the named attribute on this document. For list of available attributes refer to the `DocumentConstant` class below.

Syntax

```
public void setAttribute(String attrName, Object attrValue)
```

Parameters

attrName
The name of the attribute to be set

attrValue
The value to be set

Method

```
getTempObject
```

Method description

Retrieves a temporary object associated with this flow.

Syntax

```
public Object getTempObject(String objectName)
```

Parameters

objectName
The name of the object requested

Method

```
setTempObject
```

Method description

Sets a temporary object associated with this flow.

Syntax

```
public void setTempObject(String objectName, Object objectValue)
```

Parameters

objectName
The name of the object to be set

objectValue
The value to be set

Method

```
getAttachments
```

Method description

Retrieves the list of attachments for this document.

Syntax

```
public ListIterator getAttachments()
```

Parameters

None

Method

```
addAttachment
```

Method description

Adds an attachment to this document.

Syntax

```
public void addAttachment(AttachmentInterface attachment)
```

Parameters

attachment
The attachment to be added.

Method

```
getTransportHeaders
```

Method description

Retrieves the transport headers that were set by the receiver.

Syntax

```
public ListIterator getTransportHeaders()
```

Parameters

None

exception.BCGException

This is a utility class which extends Exception

```
public class BCGException extends Exception {}
```

BCGUtil

This class provides three utility methods and defines some common constants. The methods include:

- generateUUID()
- logEvent
- trace [two signatures]

The constants include:

- BCG_TRACE_SEVERITY_DEBUG = ″Debug″
- BCG_TRACE_SEVERITY_INFO = ″Info″
- BCG_TRACE_SEVERITY_WARNING = ″Warning″
- BCG_TRACE_SEVERITY_ERROR = ″Error″
- BCG_TRACE_SEVERITY_CRITICAL = ″Critical″

Method

```
generateUUID()
```

Method description

Generates a Unique ID

Syntax

```
public String generateUUID()
```

Parameters

None

Method

```
logEvent
```

Method description

Determines whether this event should be logged

Syntax

```
public boolean logEvent(EventInfo eventInfo)
```

Parameters

eventInfo
The event information

Method

```
trace
```

Method description

Sets tracing

Syntax

Without exception object

```
public void trace(String severity, String category, String msg)
```

With exception object

```
public void trace(String severity, String category, String msg, Throwable t)
```

Parameters

severity
A constant indicating severity level. See below.

category
The affected module name

msg The trace message

t The exception

Constants

These indicate trace severity levels:

```
public static final String BCG_TRACE_SEVERITY_DEBUG = "Debug"
public static final String BCG_TRACE_SEVERITY_INFO = "Info"
public static final String BCG_TRACE_SEVERITY_WARNING = "Warning"
public static final String BCG_TRACE_SEVERITY_ERROR = "Error"
public static final String BCG_TRACE_SEVERITY_CRITICAL = "Critical"
```

EventInfo

This class stores event information. It can be initialized in four distinct ways. It includes nine methods:

- getEventCode
- getBusinessDocument
- getDocumentUUID
- getParams
- getStackTrace
- getSourceClass
- setSourceClass
- setFaultType
- getFaultType

It also defines four constants:

- FAULTTYPE_UNKNOWN = ″0″
- FAULTTYPE_SOURCE = ″1″
- FAULTTYPE_TARGET = ″2″
- FAULTTYPE_SYSTEM = ″3″

Constructors

The object can be initialized in four distinct ways:

With a business document

```
   public EventInfo(int eventCode,
                    BusinessDocumentInterface document,
                    String[] params)
```

With a business document and an exception or error

```
   public EventInfo(int eventCode,
                    BusinessDocumentInterface document,
                    String[] params,
                    Throwable t)
```

With a document UUID

```
public EventInfo(int eventCode,
                 String documentUUID,
                 String[] params)
```

With a document UUID and an error or exception

```
public EventInfo(int eventCode,
                 String documentUUID,
                 String[] params,
                 Throwable t)
```

Method

```
getEventCode
```

Method description

Retrieves the event code

Syntax

```
public int getEventCode()
```

Parameters

None

Method

`getBusinessDocument`

Method description

Retrieves the business document

Syntax

```
public BusinessDocument getBusinessDocument()
```

Parameters

None

Method

`getDocumentUUID`

Method description

Retrieves the document UUID

Syntax

```
public String getDocumentUUID
```

Parameters

None

Method

`getParams`

Method description

Retrieves the parameter array

Syntax

```
public String[] getParams()
```

Parameters

None

Method

`getStackTrace`

Method description

Retrieves the stack trace

Syntax

```
public Throwable getStackTrace()
```

Parameters

None

Method

`getSourceClass`

Method description

Retrieves the source class

Syntax

```
public String getSourceClass()
```

Parameters

None

Method

`setSourceClass`

Method description

Sets the source class

Syntax

```
public void setSourceClass(String sourceClass)
```

Parameters

sourceClass
The source class

Method

`setFaultType`

Method description

Sets the fault type. See Constants below.

Syntax

```
public void setFaultType(String faultType)
```

Parameters

faultType
The fault type

Method

`getFaultType`

Method description

Retrieves the fault type. See Constants below.

Syntax

```
public String getFaultType()
```

Parameters

None

Constants

These constants are used to define fault types:

```
public static final String FAULTTYPE_UNKNOWN = "0"
public static final String FAULTTYPE_SOURCE = "1"
public static final String FAULTTYPE_TARGET = "2"
public static final String FAULTTYPE_SYSTEM = "3"
```

BCGDocumentConstants

This class sets constants used in retrieving attribute values from business documents.

Constants

```
public static final String BCG_FRBUSINESSID = "FROMBUSINESSID"
public static final String BCG_TOBUSINESSID = "TOBUSINESSID"
public static final String BCG_INITBUSINESSID = "INITIATINGBUSINESSID"
public static final String BCG_FRPROTOCOLNAME = "FROMPROTOCOLNAME"
public static final String BCG_FRPROTOCOLVER = "FROMPROTOCOLVERSION"
public static final String BCG_FRPROCESSCD = "FROMPROCESSCODE"
public static final String BCG_FRPROCESSVER = "FROMPROCESSVERSION"
public static final String BCG_INPROCESSDIR = "InProcessDirectory"
public static final String BCG_FROM_IPADDRESS = "fromIP"
public static final String BCG_INBOUND_CHARSET = "InboundCharset"
public static final String BCG_REQUEST_URI = "requestURI"
public static final String BCG_CERT_DN = "CertDN"
```

Events

The following is a list of events available for workflow execution flow.

Informational events

BCG_240603 - Packaging Business Process Entrance
Event Text: Packaging business process ({0}) entrance
Expected values for arguments:
{0} - Packaging BusinessProcess class name

BCG_240604- Packaging Business Process Exit
Event Text: Packaging business process ({0}) exit
Expected values for arguments:
{0} - BusinessProcess class name

BCG_240607 - UnPackaging Business Process Entrance
Event Text: Packaging business process ({0}) entrance
Expected values for arguments:
{0} - Unpackaging BusinessProcess class name

BCG_240608- UnPackaging Business Process Exit
Event Text: Packaging business process ({0}) exit
Expected values for arguments:
{0} - BusinessProcess class name

BCG_240612 - Protocol Parse Business Process Entrance
Event Text: Protocol Parse business process ({0}) entrance
Expected values for arguments:
{0} - Protocol Parse BusinessProcess class name

BCG_240613- Protocol Parse Business Process Exit
Event Text: Protocol parse business process ({0}) exit
Expected values for arguments:
{0} - BusinessProcess class name

Warning events

BCG_240605- Packaging warning
Event Text: Packaging warning -{0}
Expected values for arguments:
{0} - Packaging warning information

BCG_240609- UnPackaging warning
Event Text: UnPackaging warning -{0}
Expected values for arguments:
{0} - Unpackaging warning information.

BCG_240614 - Protocol parse warning
Event Text: Protocol parse warning -{0}
Expected values for arguments:
{0} - Protocol parsing warning information.

Error events

BCG_240418 - Digest Generation Failure
Event Text: {0}
Expected values for arguments:
{0} - Digest failure message

BCG_240419 - Unsupported Signature format (i.e signed receipt protocol is not pkcs7-signature)
Event Text: {0}.
Expected values for arguments:
{0} - Exception message containing the signature format

BCG_240420 - Unsupported Signature algorithm (i.e Signature algorithm is not MD5 or SHA1)
Event Text: {0}.
Expected values for arguments:
{0} - Exception message including signature algorithm

BCG_240606 - Packaging Error
Event Text: Packaging Error -{0}
Expected values for arguments:
{0} - Packaging error information

BCG_240611- Encryption failure
Event Text: {0}
Expected values for arguments:
{0} - Encryption failure message

BCG_240610 - UnPackaging Error
Event Text: UnPackaging Error -{0}
Expected values for arguments:
{0} - Unpackaging error information.

BCG_210014 - Error Unpackaging Mime Message
Event Text: Failed to unpackage a MIME multipart document: {0}
Expected values for arguments:
{0} - Exception message

BCG_240417 - Decryption failure
Event Text: {0}
Expected values for arguments:
{0} - Decryption failure message

BCG_240424 - Insufficient message security error
Event Text: {0}.
Expected values for arguments:
{0} - Details of what is missing. For example: a received document is encrypted but the partner agreement requires it to be encrypted and signed.

BCG_240615 - Protocol parse error
Event Text: Protocol parse error: -{0}
Expected values for arguments:
{0} - Protocol parse error message

Example Handlers and Steps Implementation Outline

The following code and pseudo code outline example implementations for fixed workflow handlers and variable workflow steps.

Protocol Processing Handler

An outline of a fixed inbound protocol processing handler implementation, in this case a handler to support CSV processing. Protocol specific code should be added by the developer.

```
public class MyCSVProtocolProcess implements
                                       BusinessProcessHandlerInterface {
   public boolean applies(BusinessDocumentInterface document) {
      //obtain from_protocol from BusinessDocument
       if (from_protocol.equals("CSV_PROTOCOL"))
          return true;
       return false;
      }
  public BusinessDocumentInterface process(
                                    BusinessDocumentInterface document) {
   try {
        //obtain the file contents in a String
        StringTokenizer tokenizer = new StringTokenizer(fileContents, ",");
        String fromBusinessId = tokenizer.nextToken();
        if (fromBusinessId == null) {
                                           EventInfo event = new EventInfo();
           BusinessProcessUtil.logEvent(eventInfo);
        }
        String toBusinessId = tokenizer.nextToken();
        String fromPackaging = tokenizer.nextToken();
        String customerId = tokenizer.nextToken();
        String customerName = tokenizer.nextToken();
        String documentType = tokenizer.nextToken();
        String documentVersion = tokenizer.nextToken();
         ...
        //log obtained parameters
         ...
         document.setValue(DocumentConstant.FRBUSINESSID, fromBusinessId);
         document.setValue(DocumentConstant.TOBUSINESSID, toBusinessId);
         document.setValue(DocumentConstant.INITBUSINESSID, customerId);
         document.setValue(DocumentConstant.FRPROTOCOLNAME,
                                               "CSV_PROTOCOL ");
         document.setValue(DocumentConstant.FRPROCESS, documentType);
         document.setValue(DocumentConstant.FRPROCESSVER, documentVersion);
          ...
          document.setDocumentState(BusinessProcessUtil.BCG_DOC_IN_PROCESS);
      } catch (Exception e) {
                                 EventInfo event = new EventInfo();
        document.addEvent(event);
        document.setDocumentState(BusinessProcessUtil.BCG_DOC_FAILED);
         }
          return document;
      }
  }
```

Protocol Unpackaging Handler

An outline of a fixed inbound protocol unpackaging handler implementation, in this case a handler to support custom XML packaging from Websphere Commerce Business Edition. Protocol specific code should be added by the developer.

```
public class WCBEXMLProtocolUnpackagingHandler implements
                                     BusinessProcessHandlerInterface {

   private Context m_context = null;
   private rConfig m_config = null;

   public void init(Context context,Config config) {
              this.m_context = context;
      this.m_config = config;
      return;
    }
    public boolean applies() {
          //obtain schema for this XML
          if (schemaLocation.startsWith("http://www.ibm.com/WCBE/schemas/"))
                 return true;
           return false;
    }

    public BusinessDocumentInterface process(
                                   BusinessDocumentInterface document) {
          try {
                 document.setValue(DocumentConst.FRPACKAGING, "WCBEPackaging");
                 /*
                  * Parse the file and obtain the following information:
                  * 1. Recipient Name
                  * 2. Type of document - PurchaseOrder, RFQ
                  */

            //obtain receiver_id
            PartnerService partnerService = BusinessProcessUtil.getPartnerService();
            String receiver_id = partnerService.getBusinessId(recipientName,
                                   PartnerService.COMMUNITY_PARTICIPANT);
                 if (receiver_id == null) {
             EventInfo event = new EventInfo();
                     BusinessProcessUtil.logEvent(msg);
             document.setDocumentState(
                       BusinessProcessUtil.BCG_DOCSTATE_FAILED);
                 }
                document.setValue(DocumentConstant.TOBUSINESSID, receiver_id);
                document.setValue(DocumentConst.FRPROCESSCD, documentType);

    document.setDocumentState(BusinessProcessUtil.BCG_DOCSTATE_IN_PROCESS);
       }catch (Exception e) {
            EventInfo event = new EventInfo();
            document.addEvents(event);
document.setDocumentState(BusinessProcessUtil.BCG_DOCSTATE_FAILED);
 }
         return document;
     }
}
```

Validation Step

An outline of a variable workflow step implementation, in this case a step to validate a WCBE purchase order document. It is in two parts. The first part implements the Factory interface, `BusinessProcessFactoryInterface` and the second part implements the Process interface, `BusinessProcessInterface`. Protocol specific code should be added by the developer.

The Factory class:

```
public class WCBEXMLValidationFactory implements
                                   BusinessProcessFactoryInterface {

    public BusinessDocumentInterface getBusinessProcess(Context context,
                                                    Config config,
                                           BusinessDocumentInterface bDoc)
{

// Can use any configuration values from config as necessary.  These
//are set via the Console.

         WCBEValidationBusinessProcess bp = new WCBEValidationBusinessProcess();
            // Set any items in this class as specific to the implementation
// between the factory and the business process class.


            return bp;
     }

public static void returnBusinessProcess(BusinessProcessInterface bp)
                                       throws BCGWorkflowException {
      // if not reusing Business Processes then do nothing.
 }
}
```

The Process class:

```
public class WCBEValidationBusinessProcess implements
                                        BusinessProcessInterface {

    public BusinessDocumentInterface process(BusinessDocumentInterface bDoc,
                                     Context context) {
           /*
            * Obtain document's contents.
            */
         File document = bDoc.getDocument();
   // Read in file.


           //obtain validation map
         MapService mapService = BusinessProcessUtil.getMapService();
         byte[] fromValidationMap = mapService.getFromValidationMap(bDoc, context);

   /* Obtain a validating XML parser instance.
    * Set the validation map location in the parser.
            * Validate the XML by parsing it.
            */


/*
         * Validate the PurchaseOrder:
 *
         * if document type is PurchaseOrder
         *     check if there is at least one orderitem
         *             if there is atleast one orderitem
         *                    check if the quantity ordered is atleast one
         *                    if the quantity ordered is less than 1
         *                           set document status to DOC_FAILED
         *                           throw BCGInvalidDocumentException
         *             else
         *                     set document status to DOC_FAILED
         *                     throw BCGInvalidDocumentException
         *
         */

 return bDoc
    }

    public boolean reset() {
        /*
         * reset internal variables.
         */
     }
}
```

Transformation Step

An outline of a variable workflow step implementation, in this case a a step to transform a document from one format to another. This sample only includes code and pseudo code for the Process implementation, but a related Factory is also necessary. Protocol specific code should be added by the developer.

The Process class:

```
public class WCBETransformationBusinessProcess implements
                                                BusinessProcessInterface {

    public BusinessDocumentInterface process(BusinessDocumentInterface bDoc,
                                           Context context) {
            //obtain transformation map
             MapService mapService = BusinessProcessUtil.getMapService();
              byte[] transformationMap = mapService.getTransformationMap(
                                                       bDoc, context);
    // Transformer is for example only and is part of customer
    // implementation.
     Transformer transformer = new Transformer(transformationMap);

  // Get the Business document file.
    File message = bDoc.getDocument();
  //get contents of File
    byte[] transformOutput = transformer.transform(fileContents);
    File transformedFile = bDoc.createFile();
  //write transformedOutput to transformedFile
   bDoc.setDocument(transformedFile);
         return bDoc;
    }

    public boolean reset() {
         /*
          * reset internal variables.
          */
      }
}
```

Chapter 6. Customizing Senders

The sender handles the final stage in the WBI Connect data flow. It picks up documents from the BPE, packages them, and sends them on to their destination, based on information in the Console configured gateway. In the case of a synchronous request, it may also process the response document. As of the 4.2.2 release, users may customize the sender stage of processing in two ways: by creating new senders or by creating new sender handlers. This chapter covers both ways of customizing senders.

- "Overview for creating new senders"
- "Overview for creating new sender handlers" on page 94

An additional section covers development and deployment issues.

- "Development and deployment" on page 94

The API listing and example code follows in the next chapter.

Overview for creating new senders

Senders are transport specific. WBI-C ships with senders for FTP/S, JMS, File, SMTP, and HTTP/S transports. To add a new capability to the WBI-C system, such as adding a WAP transport, users can write their own senders, using an API provided with the 4.2.2 release. These new senders can be associated with transports using the Community Console and integrated into the processing flow in the normal way. This section describes the process of developing a new sender. It covers:

- "The Sender/Sender Framework flow"
- "Sender architecture" on page 94

The Sender/Sender Framework flow

The nature of processing flow on the sender side of WBI-C is in part dictated by the needs of the particular situation and transport, but there are basic tasks that must always be done. This section describes those tasks in a high level, general way.

1. Receive document
The business document to be processed is placed in the gateway input directory

2. Do pre-processing
The Sender Framework, a WBI-C internal component, is invoked, and its `send` method called. The Framework moves in order through its Console configured list of handlers until one, either Connect-delivered or user-defined, that can process the document is located. The document is processed.

3. Initialize the sender
The Framework calls the sender's `init` method.

4. Send the document
The Framework calls the sender's `send` method.
The sender creates a SenderResult object to store transmission and status

information and sends the message, using the destination specified in the gateway, or, by preference, in the message itself, should it contain that information.

4A. Store response document
In the case of a synchronous request, the sender writes the response document to a file and sets the File object in the SenderResult object.

5. Do post-processing
The Framework moves through its Console configured list of handlers until an appropriate one, either Connect-supplied or user-defined, is located. The SenderResult object is processed.

8. Complete processing
The request document is removed from the gateway queue. In the case of a synchronous request, the response document is placed in a directory to be picked up for response processing.

Sender architecture

Sender development is based on two major parts: the sender itself, represented in the API by the `SenderInterface` interface and the Sender Framework, a Connect-supplied class that is responsible for managing the sender. The sender is responsible for actually sending the message to the destination, and for creating and initially populating the SenderResult object. In the case of a synchronous request, the sender also writes the response document to a file, and places a reference to the File object in the SenderResult object. The Framework is responsible for taking care of pre- and post-processing of documents and for instantiating and utilizing the sender.

Overview for creating new sender handlers

The SenderFramework can invoke handlers at two stages during the sender processing flow: pre-processing and post-processing. These stages are also referred to as `configuration points`. Pre-processing covers what occurs before the request document is given to the sender to be sent to its destination and post-processing occurs after the request document has been sent to its destination and the SenderResult object has been created to document the request's status.

WBI-C ships with a number of pre-defined handlers, but users may also develop their own, if they have specific needs not satisfied by the delivered handlers. If a request document comes from a preferred trading partner, for example, a pre-processing handler could determine the partner's status and set the transport headers accordingly. Once the handlers are written and deployed, users will need to configure them using the Console, just as they would with Connect supplied handlers. For further information on this process, see the *Hub Configuration Guide*.

Development and deployment

The following sections describe development and deployment for both user created senders and handlers.

Development environment

The sender and sender handler development API relies on classes and interfaces from this package:

- `com.ibm.bcg.bcgdk.gateway`

This package is part of the bcgsdk.jar which is found in the WBI-C installable in the following directories:

- `<install dir>\router\was\wbic`
- `<install dir>\receiver\was\wbic`
- `<install dir>\console\was\wbic`

In all deployed instances, this .jar file should be available in the application server classpath and not in the module classpath.

For development, the bcgsdk.jar file should be included in the build path of the project that contains the user exit classes, i.e., in the classpath.

Deployment and packaging

All user created code, including custom senders and handlers, should be packaged and deployed in one of the following ways:

- Placed in a .jar file in `\router\was\wbic\userexits`
- Added as classes in `\router\was\wbic\userexits\classes`

In addition, all handler classes must be accompanied by an XML descriptor file, which allows key data about the handlers to be imported into the Console GUI. In the brief outline that follows, `HandlerClassName` is the name of the handler class; `Description` is a brief description of the class; `HandlerTypeValues` is the component name (RECEIVER - for receiver handlers, GATEWAY - for sender handlers, FIXEDWORKFLOW - for fixed inbound or outbound handlers, or ACTION - for variable workflow steps) followed by the type of handler (e.g., PREPROCESS.<transport>, PROTOCOL.UNPACKAGING, etc.); and `ComponentAttribute` is any configurable member variable in the handler class which changes the behavior of the handler in runtime.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:HandlerDefinition
     xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import
                                                    /external/types"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1
                              /import/external bcghandler.xsd
                         http://www.ibm.com/websphere/bcg/2004/v0.1
                             /import/external/types bcgimport.xsd ">
 <tns:HandlerClassName>com.ibm.SampleHandler </tns:HandlerClassName>
 <tns:Description>A Sample Handler</tns:Description>
 <tns:HandlerTypes>
     <tns:HandlerTypeValue>COMPONENT.TYPE</tns:HandlerTypeValue>
 </tns:HandlerTypes>
 <tns:HandlerAttributes>
       <tns2:ComponentAttribute>
               <tns2:AttributeName>Attribute 1</tns2:AttributeName>
       </tns2:ComponentAttribute>
       <tns2:ComponentAttribute>
           <tns2:AttributeName>Attribute 2</tns2:AttributeName>
            <tns2:AttributeDefaultValue>Attribute2DefaultValue
           </tns2:AttributeDefaultValue>
        </tns2:ComponentAttribute>
 </tns:HandlerAttributes>
</tns:HandlerDefinition>
```

Chapter 7. APIs and Example Code for Senders and Sender Handlers

The following chapter provides an annotated listing of the APIs provided for developing custom senders and sender handlers. The following classes and interfaces are documented:

- "SenderInterface" on page 98
- "SenderResult" on page 100
- "SenderPreProcessHandlerInterface" on page 104
- "SenderPostProcessHandlerInterface" on page 106
- "BCGSenderException" on page 108
- "Events" on page 109
- See also the listings in the Workflow API chapter for more utility, security, and other classes shared across components.

Brief examples of code and pseudo-code outlining the implementation of an example sender and pre-processing and post-processing handlers are also included.

SenderInterface

Each sender must implement this interface. It has the following methods:

- `init`
- `send`
- `cancel`

Method

`init`

Method description

Initializes the sender, based on the contents of the `deliveryConfig` object, which contains gateway configuration information

Syntax

```
public void init (Context context, Config deliveryConfig)
       throws BCGSenderException
```

Parameters

`context`
Runtime information such as temp directory path

`deliveryConfig`
Gateway configuration details as specified in the Console

Method

`send`

Method description

Called by the SenderFramework. It sends the document to the destination using the information specified in the deliveryConfig object. It creates and updates the SenderResult object with delivery status, WBI-C and transport headers and, in the case of synchronous flow, the response document. If delivery fails, sender may try transport retries.

Syntax

```
public SenderResult send(BusinessDocumentInterface document)
```

Parameters

`document`
The business document being sent

Method

`cancel`

Method description

Called by the SenderFramework. Stops message delivery and any transport retries.

Syntax

```
public SenderResult cancel()
```

Parameters

None.

SenderResult

The SenderResult object is created by the sender based on this provided class. It holds meta information on the status of the request business document, and, in the case of synchronous flow, a reference to the File object containing the response document. It contains the following methods:

- `addEvent`
- `getEvents`
- `setSendStatus`
- `getSendStatus`
- `setResponseDocument`
- `getResponseDocument`
- `setTransportStatusCode`
- `getTransportStatusCode`
- `setTransportHeaders`
- `getTransportHeaders`
- `setAttribute`
- `getAttribute`
- `get Attributes`

Method

`addEvent`

Method description

Adds an event to the SenderResult object

Syntax

```
public void addEvent(EventInfo)
```

Parameters

`EventInfo`
EventInfo is a specialized class from the `com.ibm.bcg.bcgdk.common` package, used to hold event information throughout the WBI-C system. It is documented in the Workflow API chapter.

Method

`getEvents`

Method description

Retrieves the events set in this object

Syntax

```
public EventInfo[] getEvents()
```

Parameters

None

Method

`setSendStatus`

Method description

Sets delivery status. Can be success or failure based on transmission status.

Syntax

```
public void setSendStatus(String status)
```

Parameters

`status` The appropriate status

Method

`getSendStatus`

Method description

Retrieves the delivery status

Syntax

```
public String getSendStatus()
```

Parameters

None

Method

`setResponseDocument`

Method description

Sets the File that holds the response document.

Syntax

```
public void setResponseDocument(File responseFile)
```

Parameters

`responseFile`
The File object where the response document is stored

Method

`getResponseDocument`

Method description

Retrieves the File object which holds the response document

Syntax

```
public File getResponseDocument()
```

Parameters

None

Method

`setTransportStatusCode`

Method description

Sets the transport return status code (like HTTP 200 OK)

Syntax

```
public void setTransportStatusCode(Object transportStatusCode)
```

Parameters

`transportStatusCode`
The code

Method

`getTransportStatusCode`

Method description

Retrieves the transport return status code

Syntax

```
public Object getTransportStatusCode()
```

Parameters

None

Method

`setTransportHeaders`

Method description

Sets these headers upon receiving synchronous response

Syntax

```
public void setTransportHeaders(HashMap transportHeaders)
```

Parameters

`transportHeaders`
The HashMap containing the transport headers

Method

`getTransportHeaders`

Method description

Retrieves the transport headers set by the sender

Syntax

```
public HashMap getTransportHeaders()
```

Parameters

None

Method

```
setAttribute
```

Method description

Sets WBI-C specific attributes. These attributes contain headers specific to senders. They are used by the Framework as input into the metadata file: delivery duration, transport status description, etc.

Syntax

```
public void setAttribute(String name, Object obj)
```

Parameters

`name`	The name of the object that stores the attributes
`obj`	The object

Method

```
getAttribute
```

Method description

Retrieves the WBI-C specific attributes

Syntax

```
public Object getAttribute()
```

Parameters

None

Method

```
getAttributes
```

Method description

Retrieves HashMap of all attributes set

Syntax

```
public Map getAttributes set()
```

Parameters

None

SenderPreProcessHandlerInterface

This interface describes the three methods which all pre-process handlers must implement. They include:

- `init`
- `applies`
- `process`

Method

```
init
```

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
              throws BCGSenderException
```

Parameters

`context`
Runtime information such as temp directory path

`handlerConfig`
The object storing configuration information

Method

```
applies
```

Method description

Determines if handler can handle business document

Syntax

```
public boolean applies(BusinessDocumentInterface doc)
              throws BCGSenderException
```

Parameters

`doc` The BusinessDocument that is being processed

Method

```
process
```

Method description

Performs pre-processing by updating the BusinessDocument

Syntax

```
public BusinessDocumentInterface process(BusinessDocumentInterface doc)
               throws BCGSenderException
```

Parameters

`doc`	The BusinessDocument that is being processed

SenderPostProcessHandlerInterface

This interface describes the three methods which all pre-process handlers must implement. They include:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
            throws BCGSenderException
```

Parameters

`context`
Runtime information such as temp directory path

`handlerConfig`
The object storing configuration information

Method

`applies`

Method description

Determines if handler can handle the applicable message

Syntax

```
public boolean applies(BusinessDocumentInterface doc)
           throws BCGRSenderException
```

Parameters

`doc` The BusinessDocument that is being processed

Method

`process`

Method description

Performs post-processing on delivery response by updating the SenderResult object.

Syntax

```
public SenderResult process(SenderResult response,
                            BusinessDocumentInterface doc)
            throws BCGReceiverException
```

Parameters

`response`
: The Sender Result object to be updated

`doc`
: The BusinessDocument that is being processed

BCGSenderException

This is a utility class which extends BCGException, which extends Exception

Events

The following is a list of events available for the sender execution flow.

Informational events

BCG_240616 - Sender Entrance
Event Text: Sender ({0}) entrance
Expected values for arguments:
{0} - Sender class name

BCG_240617 - Sender Exit
Event Text: Sender({0}) exit
Expected values for arguments:
{0} - Sender class name

BCG_250007 - Document Delivered
Event Text: Document was delivered successfully, response: {0}
Expected values for arguments:
{0} - << target response status>>

Warning events

BCG_240618 - Sender warning
Event Text: Sender warning -{0}
Expected values for arguments:
{0} - Sender warning information.

Error Events

BCG_250008 - Document Delivery Failed
Event Text: Document delivery to participant gateway failed: {0}
Expected values for arguments:
{0} - << response status and error message >>

Example Sender and Sender Handlers Implementation Outlines

The following code and pseudo code outline example implementations for senders and sender handlers.

Example Sender

An outline of a sender handler implementation. Protocol specific code should be added by the developer.

```
public class SampleSenderUserExit implements SenderInterface
{
        SenderResult result = new SenderResult ();
        Connection connection = null;
        public SampleSenderUserExit()
        {
             ...
    }
        public void init(Context context, Config deliveryConfig)
                 throws BCGSenderException
        {
               //initialization code
           ...
    }
        public SenderResult send(BusinessDocumentInterface document)
                  throws BCGSenderException
        {
               //destination url set in the BusinessDocumentInterface
               //is preferred over url set in the Gateway
               URL url = (String)document.getAttribute(DocumentConstant.URI);
                if (url == null)
                  url = (String)config.getConfig(DocumentConstant.URI);
                //obtain other configuration information from the
                //DeliveryConfig which holds the configuration information
                //set in the Gateway
                File documentFile = document.getDocument();
                try
                {
                        //read contents from File
                    ...
            }
                catch(FileIOException exception)
                {
                        BCGSenderException e = new BCGSenderException(...);
                        throw e;
            }
                try
                {
                        //establish connection with the destination
                    ...
                        //send contents to destination
                    ...
                        //if DocumentConst.GET_SYNC_RESPONSE is true,
                        //read the response
                        if (String.valueOf(document.getAttribute(
                         DocumentConst.GET_SYNC_RESPONSE).
                                 equalsIgnoreCase(DocumentConstants.TRUE))
                        {
                           //read the response
                          ...
                String uuid = BCGUtil.getDocumentUUID();
                String tmpDir = BCGUtil.TMP_DIR_PATH;
                File responseFile = document.createFile(uuid+".resp");
                //write response to file
                         ...

                result.setResponse(responseFile);
```

```
                         ...
                      }
                        else
                        {
                              //read the transport response and
                              //set in the SenderResult object.
                              ...
                      }
              }
                catch(Exception exception)
                {
                        //create an event and add to the SenderResult
                        Object[] params = {exception};
                        EventInfo eventInfo = EventInfo(eventCode,
                                                          document, params);
                        result.addEvent(eventInfo);
                result.setStatus(DocumentConstant.BCG_DOC_FAILED);
                        return result;
              }
                //close the connection
              ...
           result.setStatus(DocumentConstant.BCG_DOC_SENT);
                return result;
      }

        public SenderResult cancel() throws BCGSenderException
        {
                    connection.close();
                    EventInfo eventInfo = EventInfo(eventCode,
                                                      document, params);
                    result.setEvent(eventInfo);
                    return result;
      }
}
```

Example Pre-processing handler

The following code and pseudo code outlines an example implementation for a sender pre-processing handler. If the `toPartner` is a preferred customer, it sets some transport attributes in the BusinessDocument accordingly.

```
public class SampleSenderPreProcessUserExit implements
SenderPreProcessHandlerInterface
{
    public SampleSenderPreProcessUserExit ()
    {
        ...
    }
    public void init(Config handlerConfig) throws BCGSenderException
    {
        //initialization code
        ...
    }
    public boolean applies(Config deliveryConfig,
                             BusinessDocumentInterface doc)
                throws BCGSenderException
    {
        String toProtocol =
              document.getAttribute(DocumentConstants.TOPROTOCOLNAME);
        boolean isAS2 = (toProtocol!= null &&
                     toProtocol.equalsIgnoreCase(DocumentConstants.AS2));
        return isAS2;
    }
    public BusinessDocumentInterface process (Config deliveryConfig,
                                 BusinessDocumentInterface doc)
                 throws BCGSenderException
    {
        String toPartnerId = (String)doc.getAttribute
                                    (DocumentConstants.TOBUSINESSID);
        //check if partner is a PreferredCustomer
        ...
        if (preferredCustomer)
        {
            //set transport attributes for PreferredCustomer
            doc.setAttribute(PRIORITY, "HIGH");
doc.setAttribute(ACKNOWLEDGEMENT_REQUIRED, "true");
            ...
        }
    }
}
```

Example post-processing handler

The following code and pseudo code outlines an example implementation for a sender post-processing handler. This example assumes a SOAP message. If the response is an exception or other type of error, but not a SOAP fault, the handler creates a SOAP fault and sets the response in the BusinessDocument.

```
public class SampleSOAPPostProcessUserExit implements
                                       SenderPostProcessHandlerInterface
{
    public void init(Config handlerConfig)
             throws BCGSenderException
    {
        //initialization code
        ...
    }
    public SampleSOAPPostProcessUserExit ()
    {
        ...
    }
    public boolean applies(Config deliveryConfig,
                                   BusinessDocumentInterface doc)
               throws BCGSenderException
    {
        String fromProtocol =
                 (String)document.getAttribute(
                                     DocumentConst.FRPROTOCOLNAME);
        boolean isSoap = (fromProtocol != null &&
        fromProtocol.equalsIgnoreCase(
                                     DocumentConst.WEB_SERVICE_PROTOCOL));
        return isSoap;
    }

    public SenderResult process (SenderResult response,
                                 BusinessDocumentInterface doc)
                    throws BCGSenderException
    {
        SoapFault sf = null;
        String deliveryStatus= response. getDeliveryStatus();
        If (deliveryStatus.equalsIgnoreCase("FAILED")){
            //Get transport header, read the http status code,
//    and create SOAP fault message
            String faultMessage="Server error";
            sf = new SoapFault(faultString);
            response.setResponse(sf.getBytes());
        }
        return response;
    }
}
```

Chapter 8. End to End Synchronous Flow: an Overview for Using User Exits

This chapter presents a brief overview of the steps involved as a synchronous request moves through the WBI-C system, highlighting the places where user provided code may function.

Document Receiving

The receiver, which may be WBI-C supplied or user provided, receives a request from an initiating partner. It:

- b) writes the message into a temporary location, which it gets by using a method on the `BCGReceiverUtil` class, `getTempDir`
- c) creates a request document object (based on a class that implements `ReceiverDocumentInterface`) and sets the UUID it has received from the Receiver Framework and the temporary location on it
- d) writes the received transport headers into a HashMap object and sets it on the request document.

Next it creates the WBI-C specific headers (the `BCGHeaders`) and sets them on the request document by calling the `setAttribute` method for each header.

The receiver calls `preProcess` on the Receiver Framework. The Framework executes the handlers, either Connect supplied or user defined, that have been specified for this target via the Console, in the order they are shown in the Console configuration screen. The request document is passed as input to the first handler, and then the returned processed document is fed as input to the next handler, and so on. The document is processed.

The receiver calls `syncCheck` on the Receiver Framework. The Framework iterates through the list of handlers (WBI-C and/or user supplied) defined for this transport and target. When it finds a handler that can manage the document, it executes sync check. In the case of a blocking synchronous request, it returns ’true’.

The receiver calls `process` on the Framework by passing it the target type, the request document, and an empty response document that it has created.

The Framework adds two attributes to the BCG headers on the request document: `RESPONSE_URL` and `REPLY_TO_MSG_ID`. `RESPONSE_URL` contains a WBI-C internal Sync response servlet URL and `REPLY_TO_MSG_ID` contains the request document’s UUID.

The Framework creates the necessary BPE input files in the sync_in directory. It then calls `waitForSyncResponse` on SyncEngine, which will get notified when the Sync servlet receives the response from DeliveryManger (the component that includes Senders.)

A WBI-C internal process reads the files from the sync_in directory, creates the business document object, and non-repudiates the data. It logs the document and related information into the activity table, and writes the business document into the BPE synchronous inbound queue.

Document Processing

The BPE retrieves the document from its synchronous inbound queue.

Fixed inbound

The BPE executes the fixed inbound flow. The first step of fixed inbound flow is protocol un-packaging. The BPE iterates through configured handlers, including any user defined handlers, until it finds a handler that can manage the document. It executes un-packaging.

The second step of fixed inbound flow is protocol processing. The BPE iterates through configured handlers, including any user defined handlers, until it finds a handler that can manage the document. It executes protocol processing. As this is a sync request, the handler sets the GET_SYNC_RESPONSE value in the business document to ’true’. The rest of the fixed inbound steps are processed.

Variable

The BPE executes the configured action, which may be WBI-C supplied, a user-modified version of a WBI-C supplied action, or an entirely new action defined by the user. Steps in user-modified or new actions can be user defined. No constants concerning sync status need to be updated during variable workflow processing.

Fixed outbound

The BPE executes the fixed outbound flow. The first step of fixed outbound flow is protocol packaging. The BPE iterates through configured handlers, including any user defined handlers, until it finds a handler that can manage the document.hi It executes protocol packaging. No constants concerning sync status need to be updated during fixed outbound workflow processing.

The BPE writes the business document to the Delivery Manager’s inbound queue.

Note: Both sync and async requests are written into the same inbound queue.

Document Transmission

The Delivery Manager transfers the business document from the inbound queue to the appropriate gateway folder. DeliveryWorker polls this folder, finds and picks up the business document and calls the `processSend` method on the Sender Framework.

The Framework executes any gateway configured pre-process handlers.

The Framework creates a sender object from a class which implements `SenderInterface`. This class may be user supplied. The Framework calls the `send` method on that sender object. The document is sent. The `send` method also checks if the GET_SYNC_RESPONSE constant on the business document is set to ’true’. If it is, the sender then reads the synchronous response from the connection and uses the data to populate the SenderResult object the method returns. The Framework executes any gateway configured post-process handlers.

The Framework reads the SenderResult object, creates a UUID for the response, and creates the set of input files for the BPE. Included in the metadata file are two constants: BCG_RESPONSE_STATUS, which carries the response transport status code, and BCG_SYNC_RESP, which is set to ’true’. The files are written into the BPE’s sync_in directory.

Response Processing

The BPE retrieves the response business document from its synchronous inbound queue.

Fixed inbound

The BPE executes the fixed inbound flow. The first step of fixed inbound flow is protocol un-packaging. The BPE iterates through configured handlers, including any user defined handlers, until it finds a handler that can manage the document. It executes un-packaging as needed.

The second step of fixed inbound flow is protocol processing. The BPE iterates through configured handlers, including any user defined handlers, until it finds a handler that can manage the document. It executes protocol processing. The handler checks if the BCG_RESPONSE_URL is null. If the value is null, but the handler has received a response document, then the handler sets GET_SYNC_RESPONSE to ’false’ and BCG_SYNC_RESP to null. Otherwise it continues through the rest of the fixed inbound processing steps.

Variable

The BPE executes the configured action, which may be WBI-C supplied, a user-modified version of a WBI-C supplied action, or an entirely new action defined by the user. Steps in user-modified or new actions can be user defined. No constants concerning sync status need to be updated during variable workflow processing.

Fixed outbound

The BPE executes the fixed outbound flow. The first step of fixed outbound flow is protocol packaging. The BPE iterates through configured handlers, including any user defined handlers, until it finds a handler that can manage the document. It executes protocol packaging. No constants concerning sync status need to be updated during fixed outbound workflow processing.

After the successful execution of all workflow steps, the BPE checks the SYNC_RESP attribute on the response document. The response business document is written into the Delivery Manager’s synchronous inbound queue.

Response Transmission

The Sync Delivery Manager sends the response to the RESPONSE_URL set in the response business document. The Sync Engine retrieves the persisted original connection. The Receiver Framework populates a response object (from a class that implements ReceiverDocumentInterface) with the synchronous response information.

The receiver reads the RESPONSE_STATUS attribute and sends the status code and the sync response back to the initiating partner on the original connection.

Chapter 9. Troubleshooting User Exits

This chapter highlights some common troubleshooting situations in setting up and using user exits.

Setting up logging

The trace method of the `BCGUtil` class in the `com.ibm.bcg.bcgdk.common` package is used to set up logging of internal activity for the entire document flow. Full documentation of the method is found in "BCGUtil" on page 78. The following is an example code snippet setting up logging on an XML Translation step in variable workflow:

```
BCGUtil bcgUtil = new BCGUtil ();
:
:
:
    bcgUtil.trace(BCGUtil.BCG_TRACE_SEVERITY_DEBUG,
                      "CustomXMLTranslation",
                       "The Schema is present",
                       null)
```

Receiver logs can be found at:
`<Receiver_installed_dir>/was/logs/server1/wbic_receiver.log`

Fixed and variable workflow and sender logs (grouped together as part of the Router component) can be found at:
`<Router_installed_Dir>/was/logs/server1/wbic_router.log`

By default, debug logging is not enabled. To turn it on, a property in the log4j properties file must be set. The properties file for receivers can be found at:
`<Receiver_installed_dir>/was/wbic/Config/receiver-was.logging.properties`

The properties files for router components can be found at:
`<Router_installed_dir>/was/wbic/Config/ router-was.logging.properties`.

In both cases, the property that needs to be set is the `log4j.rootCategory` property. By default this is set at `error, RollingFile`. This value needs to changed to `debug, RollingFile`. The server must be restarted for the change to be effective.

Common sources of error

The following are five general types of errors commonly encountered in setting up user exits and the steps to take to correct them.

File location errors

It is crucial that the WBI-C system be able to find the user exit classes. A `Class Not Found` exception in either log may occur if:

- The user exit class files are not loaded in the classpath

Or

- The user exit class files are not present as specified in the package hierarchy designated in the XML file that the user uploads through the Community Console.

Additional file location problems can arise if, in a multibox, split topology setup, the appropriate user exits are not deployed with all instances of receivers or routers, as necessary.

Resolution: Make sure the class files are properly loaded in the classpath and that the exact name and location of the user exit class files match the details specified during the upload of the XML descriptor files to through the Console. Make sure all appropriate files exist in all appropriate places.

Handler failure errors

Failure of a pre-processing handler in the receiver component or of either type of handler in the sender component, or failure in an unpackaging, protocol processing, or packaging handler in the router component will produce an error in the appropriate logs and in the Console. Turning on Debug mode will produce a more detailed error report. The error will result in the message or business document not being processed further, and, in the case of an HTTP receiver pre-processing failure, a `500` response code being sent back to the initiating host.

Resolution: Fix the problem in the user exit code, reload the class files, and restart the component.

Processing mode errors

When a document protocol supports synchronous processing, the defined target *must* have a SyncCheck handler specified. If the protocol does not support synchronous processing, a post-process handler must *not* be specified.

Resolution: Make sure that the user exits you specify are appropriate for the defined processing mode.

File update errors

There are two ways to update user exit information in the system:

- Updating the class files (or .jars) themselves
- Updating the XML descriptor files

If you update the class files, you should restart the appropriate components to make sure that the changes are effective. Uploading new XML descriptor files for existing user exits (assuming the files have the same name, and designate the same class) will change whatever attributes and attribute values that are set immediately. In this case, any documents that are processed after the new descriptor files are uploaded will be processed as described in those new files.

Resolution: Updating class files requires a component restart to be effective; updating XML descriptor files takes effect immediately.

File deletion errors

A user defined receiver (transport) cannot be deleted once it has been used in a target definition, even if the target itself has been deleted. A user defined sender (transport) cannot be deleted once a gateway using that transport has been defined, even if the gateway is set to an `offline` or `disabled` state. On the other hand, a fixed workflow handler *can* be deleted, even if it is part of a specified workflow. As soon as it is deleted, the handler is immediately removed from the `Configured List.` Any document flow dependent on that handler that is processed after the deletion will fail.

Resolution: Do not try to delete transports once they have been used to define a target or a gateway. Do not delete handlers for fixed workflow unless you are absolutely sure they are not used anywhere.

Part 2. Customizing Business Integration Connect: Administrative APIs and External Event Delivery

Until now, dealing with the various tasks associated with day to day community hub management required using the Community Console. With the release of WebSphere Business Integration Connect 4.2.2, however, a hub administrator may now use a newly established API to accomplish certain common administrative tasks programmatically, using a simple XML based HTTP POST mechanism. In addition, WBI-C has been modified to allow events, both document related and general system based, to be delivered to an external JMS queue as well as being sent to the internal WBI-C event store.

The following chapters document these new features of WBI-C.

Chapter 10. Using the Administrative API

This chapter describes the Administrative APIs. The Administrative APIs allow certain Hub administrative functions to be executed programmatically. It is divided into two sections:

- "Understanding the Administrative API"
- "The Administrative API" on page 124

Understanding the Administrative API

The WBI-C Administrative API allows certain common administrative functions to be carried out without using the Community Console GUI.

Note: The Console itself must be running for API calls to be processed and the API functionality must have been turned on through the GUI before the calls are made. For more information using the GUI to turn the APIs on, see the *WBI-C Hub Configuration Guide*

A method is called by sending an HTTP POST request with an appropriate XML document as the body. This request is directed to a servlet running on the Console instance, at the relative URL of `/console/bcgpublicapi`.

In general, the XML request document includes the following data:

- User information (this is the same information used in logging in through the Console and must be provided with every request, as there is no notion of session management)
  - User name
  - Password
  - Partner login name
- API information
  - Method name, usually an action given as a concatenated Noun and Verb: e.g., ParticipantCreate
  - Parameters, usually an item: e.g., Partner.

At present, the following 11 methods are supported:

- "ParticipantCreate" on page 125
- "ParticipantUpdate" on page 127
- "ParticipantSearchByName" on page 129
- "ParticipantAddBusinessId" on page 130
- "ParticipantRemoveBusinessId" on page 131
- "ContactCreate" on page 132
- "ListTargets" on page 139
- "ListParticipantCapabilities" on page 136
- "ListParticipantConnections" on page 138
- "ListTargets" on page 139
- "ListEventDefinitions" on page 141

The system processes the request and returns the response (or exception) XML synchronously, i.e., on the same HTTP connection. Each method has a corresponding response. Using an API produces the same internal process that using the console does. If a particular operation executed via the console generates events, that operation executed via the API generates the same events.

"The Administrative API" describes these APIs in detail. More detail can be gathered by looking in the `$(WBICINSTALLROOT)/publicapi` directory at the two provided schemas:

- `bcgpublicapi_v0.1.xsd`: The API signatures
- `bcgpublicapi_vocabulary_v0.1.xsd`: The vocabulary from which the schema is constructed.

In addition to the actual response, the servlet itself also provides standard HTTP status codes, as specified in Table 1

Table 1. Servlet Status Codes

HTTP status code	Situation in which this code is returned
500	• Request XML cannot be parsed • There is any error in processing the request • There is an internal error
405	• An HTTP request other than POST has been received The servlet supports only the POST method
200	• The API has been successfully executed
501	• A non-implemented request has been received • The Administrative API has not been turned on

Security is provided by the use of SSL and, optionally, Client Authorization. Data, but not the elements of the API itself, can be localized, based on the user's locale, as long as character encoding is set to UTF-8, which is the standard expected encoding.

The Administrative API

The following section outlines the structure of the 11 XML method calls and their responses, and the exception XML that is used to report any errors. The general structure of the XML is as follows:

- The root element is always a `BCGPublicAPI` element
- The first child of the root in a request document is the `<MethodName>` element.
  - The first child of the `<MethodName>` element is the `UserInfo` element. This element contains the Console login information for the user making the request. This user must have permissions adequate for the task being attempted.
  - The second child of the `<MethodName>` element represents any input parameters.
- The first child of the root in a response document is the `<MethodName>Response` element. This element represents any results of the execution of the request API.
- The first child of the root in an exception document is a `BCGPublicAPIException` element.

ParticipantCreate

Adds a participant to the hub-community. Participants are the companies that do business with the Community Manager via the hub-community. Once connected, participants can exchange electronic business documents with the Community Manager.

Root element
  `BCGPublicAPI`

First child element
  `ParticipantCreate`

First child of `ParticipantCreate`
  `UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantCreate`
  `ParticipantCreateInfo` element. Contains 7 elements:

- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are: Community Operator, Community Manager, or Community Participant.
- `ParticipantStatus`: Enabled or Disabled. If disabled, participant is not visible in search criteria and drop-down lists. The default value is Enabled.
- `CompanyURL`: The URL of the participant's web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are: Supplier, Contract Manufacturer, Distributor, Logistic Provider, or Other. This is an optional element.
- `Password`: The password this participant will use to access the system.

ParticipantCreateResponse

Response document for the `ParticipantCreate` method.

Root element
  `BCGPublicAPI`

First child element
  `ParticipantCreateResponse`

First child of `ParticipantCreateResponse`
  `ParticipantCreateResponseInfo` element. Contains 7 elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.

- `ParticipantType`: Defines the participant's function in the community. Available values are: Community Operator, Community Manager, or Community Participant.
- `ParticipantStatus`: Enabled or Disabled. If disabled, participant is not visible in search criteria and drop-down lists.
- `CompanyURL`: The URL of the participant's web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are: Supplier, Contract Manufacturer, Distributor, Logistic Provider, or Other. This is an optional element.

ParticipantUpdate

Updates the participant's profile in the system.

Root element
`BCGPublicAPI`

First child element
`ParticipantUpdate`

First child of `ParticipantUpdate`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantUpdate`
`ParticipantUpdateInfo` element. Contains 6 elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are: Community Operator, Community Manager, or Community Participant.
- `ParticipantStatus`: Enabled or Disabled. If disabled, participant is not visible in search criteria and drop-down lists.
- `CompanyURL`: The URL of the participant's web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are: Supplier, Contract Manufacturer, Distributor, Logistic Provider, or Other. This is an optional element.

ParticipantUpdateResponse

Response document for the `ParticipantUpdate` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantUpdateResponse`

First child of `ParticipantUpdateResponse`
`ParticipantUpdateResponseInfo` element. Contains 7 elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are: Community Operator, Community Manager, or Community Participant.
- `ParticipantStatus`: Enabled or Disabled. If disabled, participant is not visible in search criteria and drop-down lists.

- `CompanyURL`: The URL of the participant's web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are: Supplier, Contract Manufacturer, Distributor, Logistic Provider, or Other. This is an optional element.

ParticipantSearchByName

Searches for participant profiles by display name.

Root element
`BCGPublicAPI`

First child element
`ParticipantSearchByName`

First child of `ParticipantSearchByName`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantSearchByName`
`ParticipantName` element: The name the participant wants displayed to the hub community.

ParticipantSearchByNameResponse

Response document for the `ParticipantSearchByName` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantSearchByNameResponse`

First child element of `ParticipantSearchByNameResponse`
`Participants` element

Zero or more children of `Participants`
`ParticipantInfo`. Contains 5 elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are: Community Operator, Community Manager, or Community Participant.
- `ParticipantStatus`: Enabled or Disabled. If disabled, participant is not visible in search criteria and drop-down lists.

ParticipantAddBusinessId

Adds a Business ID to the Participant's Profile.

Root element
: `BCGPublicAPI`

First child element
: `ParticipantAddBusinessId`

First child of `ParticipantAddBusinessId`
: `UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:
  - `UserName`: The Console login username.
  - `Password`: The Console login password.
  - `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantAddBusinessId`
: `ParticipantAddBusinessIdInfo` element. Contains 3 elements:
  - `ParticipantId`: An internal numeric ID that identifies the participant to the system.
  - `BusinessId`: The DUNS, DUNS+4, or Freeform number that the system uses for routing. DUNS numbers must equal nine digits and DUNS+4 thirteen digits. Freeform ID numbers accept up to 60 alpha, numeric, and special characters.
  - `BusinessIdType`: The type of ID being used. Available values are: DUNS, DUNS+4, or Freeform.

ParticipantAddBusinessIdResponse

Response document for the `ParticipantAddBusinessId` method.

Root element
: `BCGPublicAPI`

First child element
: `ParticipantAddBuinessIdResponse`

First child element of `ParticipantAddBuinessIdResponse`
: `ParticipantAddBuinessIdResponseInfo` element: Contains 4 elements.
  - `BusinessIdentifierId`: An internal numeric ID that identifies the BusinessId to the system.
  - `ParticipantId`: An internal numeric ID that identifies the participant to the system.
  - `BusinessId`: The DUNS, DUNS+4, or Freeform number that the system uses for routing. DUNS numbers must equal nine digits and DUNS+4 thirteen digits. Freeform ID numbers accept up to 60 alpha, numeric, and special characters.
  - `BusinessIdType`: The type of ID being used. Available values are: DUNS, DUNS+4, or Freeform.

ParticipantRemoveBusinessId

Removes a Business ID from the Participant's Profile.

Root element
`BCGPublicAPI`

First child element
`ParticipantRemoveBusinessId`

First child of `ParticipantRemoveBusinessId`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantRemoveBusinessId`
`BusinessIdentifierId` element: An internal numeric ID that identifies the BusinessId to the system.

ParticipantRemoveBusinessIdResponse

The response document for the `ParticipantRemoveBusinessId` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantRemoveBuinessIdResponse`

ContactCreate

Creates a contact. Contacts are key personnel who should receive notification when the system generates alerts as a result of specified events in the system.

Root element
`BCGPublicAPI`

First child element
`ContactCreate`

First child of `ContactCreate`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ContactCreate`
`ContactCreateInfo` element. Contains 13 elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `GivenName`: The contact's given name.
- `FamilyName`: The contact's family name.
- `Address`: The contact's address. This is an optional element.
- `ContactType`: The contact's role. This is an optional element. Available values are:
  - Project Manager
  - Business Lead
  - Technical Lead
  - B2B Lead
  - Data Content Lead
  - Backend Application Lead
  - Network Firewall Lead
- `Email`: The contact's email address. This is an optional element.
- `Telephone`: The contact's telephone number. This is an optional element.
- `FaxNumber`: The contact's fax number. This is an optional element.
- `LanguageLocale`: The contact's language locale. This is an optional element.
- `FormatLocale`: Additional locale information for the contact. This is an optional element.
- `TimeZone`: The contact's time zone. This is an optional element.
- `AlertStatus`: Indicates whether the contact should receive alerts. Available values are Enabled or Disabled. The default is Disabled.
- `Visibility`: Indicates the visibility. Available values are Local (restricted to the organization) and Global (the organization and the Community Manager). The default is Local.

ContactCreateResponse

The response document for the `ContactCreate` method.

Root element

`BCGPublicAPI`

First child element

`ContactCreateResponse`

First child of `ContactCreateResponse`

`ContactCreateResponseInfo` element. Contains 14 elements:

- `ContactId`: An internal numeric ID that identifies the contact to the system.
- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `GivenName`: The contact's given name.
- `FamilyName`: The contact's family name.
- `Address`: The contact's address. This is an optional element.
- `ContactType`: The contact's role. This is an optional element. Available values are:
  - Project Manager
  - Business Lead
  - Technical Lead
  - B2B Lead
  - Data Content Lead
  - Backend Application Lead
  - Network Firewall Lead
- `Email`: The contact's email address. This is an optional element.
- `Telephone`: The contact's telephone number. This is an optional element.
- `FaxNumber`: The contact's fax number. This is an optional element.
- `LanguageLocale`: The contact's language locale. This is an optional element.
- `FormatLocale`: Additional locale information for the contact. This is an optional element.
- `TimeZone`: The contact's time zone. This is an optional element.
- `AlertStatus`: Indicates whether the contact should receive alerts. Available values are Enabled or Disabled. The default is Disabled.
- `Visibility`: Indicates the visibility. Available values are Local (restricted to the organization) and Global (the organization and the Community Manager). The default is Local.

ContactUpdate

Updates contact information.

Root element
`BCGPublicAPI`

First child element
`ContactUpdate`

First child of `ContactUpdate`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ContactUpdate`
`ContactUpdateInfo` element. Contains 13 elements:

- `ContactId`: An internal numeric ID that identifies the contact to the system.
- `GivenName`: The contact's given name.
- `FamilyName`: The contact's family name.
- `Address`: The contact's address. This is an optional element.
- `ContactType`: The contact's role. This is an optional element. Available values are:
  - Project Manager
  - Business Lead
  - Technical Lead
  - B2B Lead
  - Data Content Lead
  - Backend Application Lead
  - Network Firewall Lead
- `Email`: The contact's email address. This is an optional element.
- `Telephone`: The contact's telephone number. This is an optional element.
- `FaxNumber`: The contact's fax number. This is an optional element.
- `LanguageLocale`: The contact's language locale. This is an optional element.
- `FormatLocale`: Additional locale information for the contact. This is an optional element.
- `TimeZone`: The contact's time zone. This is an optional element.
- `AlertStatus`: Indicates whether the contact should receive alerts. Available values are Enabled or Disabled. The default is Disabled.
- `Visibility`: Indicates the visibility. Available values are Local (restricted to the organization) and Global (the organization and the Community Manager). The default is Local.

ContactUpdateResponse

The response document for the `ContactUpdate` method.

Root element
`BCGPublicAPI`

First child element

`ContactUpdateResponse`

First child of `ContactUpdateResponse`

`ContactUpdateResponseInfo` element. Contains 14 elements:

- `ContactId`: An internal numeric ID that identifies the contact to the system.
- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `GivenName`: The contact's given name.
- `FamilyName`: The contact's family name.
- `Address`: The contact's address. This is an optional element.
- `ContactType`: The contact's role. This is an optional element. Available values are:
  - Project Manager
  - Business Lead
  - Technical Lead
  - B2B Lead
  - Data Content Lead
  - Backend Application Lead
  - Network Firewall Lead
- `Email`: The contact's email address. This is an optional element.
- `Telephone`: The contact's telephone number. This is an optional element.
- `FaxNumber`: The contact's fax number. This is an optional element.
- `LanguageLocale`: The contact's language locale. This is an optional element.
- `FormatLocale`: Additional locale information for the contact. This is an optional element.
- `TimeZone`: The contact's time zone. This is an optional element.
- `AlertStatus`: Indicates whether the contact should receive alerts. Available values are Enabled or Disabled. The default is Disabled.
- `Visibility`: Indicates the visibility. Available values are Local (restricted to the organization) and Global (the organization and the Community Manager). The default is Local.

ListParticipantCapabilities

Queries participant's functional capabilities

Root element
: `BCGPublicAPI`

First child element
: `ListParticipantCapabilities`

First child of `ListParticipantCapabilities`
: `UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:
  - `UserName`: The Console login username.
  - `Password`: The Console login password.
  - `ParticipantLogin`: The participant (company) login name

Second child of `ListParticipantCapabilities`
: `ParticipantId`: An internal numeric ID that identifies the participant to the system.

ListParticipantCapabilitiesResponse

The response document for the `ListParticipantCapabilities` method

Root element
: `BCGPublicAPI`

First child element
: `ListParticipantCapabilitiesResponse`

First child of `ListParticipantCapabilitiesResponse`
: `ParticipantCapabilities` element.

Zero or more children of ParticipantCapabilities
: `ParticipantCapability` element. Contains 8 elements.
  - `CapabilityId`: An internal numeric ID that identifies this capability to the system.
  - `ParticipantId`: An internal numeric ID that identifies this participant to the system.
  - `ParticipantName`: The name the participant wants displayed to the hub community.
  - `CapabililtyRole`: The functional role the participant has in the system. Available values are:
    - Source
    - Target
    - SourceAndTarget
  - `CapabilityEnabled`: A Boolean.
  - `RoutingObjectRefId`: An internal numeric ID that identifies the routing object reference associated with this capability to the system.
  - `RoutingObjectRefInfo`: Routing objects in WBI-C are hierarchical. They are defined once, but can be referenced at multiple places. The routing object reference uniquely identifies where the routing objects are referenced. This is a complex type holding the following elements:
    - `RoutingObjectRefId`: An internal numeric ID of the routing object reference.

  - `RoutingObjectId`: An internal numeric id of the routing object referenced.
  - `RoutingObjectName`: The name of the routing object.
  - `RoutingObjectVersion`: The routing object version.
  - `RoutingObjectType`: The type of this routing object localized into the user's locale.
  - `RoutingObjectTypeKey`: The key to the type of this routing object. For example: Package, Protocol etc.
  - `RoutingObjectEnabled`: A Boolean.
  - `RoutingObjectParentRefId`: The internal numeric ID of the parent routing object reference. This is an optional element.
- `CapabilityChildren` element. This is an optional element. Contains zero or more `CapabilityChild` elements. Each `CapabilityChild` element holds the same eight elements as the `ParticipantCapability` element.

ListParticipantConnections

Queries the participant's connections.

Root element
`BCGPublicAPI`

First child element
`ListParticipantConnections`

First child of `ListParticipantConnections`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

Second child of `ListParticipantConnections`
`SourceParticipantId`: An internal numeric ID that identifies the participant as source to the system.

Third child of `ListParticipantConnections`
`TargetParticipantId`: An internal numeric ID that identifies the participant as target to the system.

ListParticipantConnectionsResponse

The response document for the `ListParticipantCapabilities` method

Root element
`BCGPublicAPI`

First child element
`ListParticipantConnectionsResponse`

First child of `ListParticipantConnectionResponse`
`ParticipantConnections` element.

Zero or more children of ParticipantConnections
`ParticipantConnection` element. Contains 9 elements.

- `ConnectionId`: An internal numeric ID that identifies this connection to the system.
- `SourceParticipantId`: An internal numeric ID that identifies the participant as source to the system.
- `SourceCapabilityId`: An internal numeric ID identifies the source capability to the system.
- `TargetParticipantId`: An internal numeric ID that identifies the participant as target to the system.
- `TargetCapabilityId`: An internal numeric ID that identifies the target capability to the system.
- `ActionId`: An internal numeric ID that identifies the action to the system.
- `ActionName`: The display name of the action.
- `TransformMapId`: An internal numeric ID that identifies the transform map associated with this action. This is an optional element.
- `ConnectionEnabled`: A Boolean.

ListTargets

Queries for the targets configured on the system.

Root element
`BCGPublicAPI`

First child element
`ListTargets`

First child of `ListTargets`
`UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:

- `UserName`: The Console login username.
- `Password`: The Console login password.
- `ParticipantLogin`: The participant (company) login name

ListTargetsResponse

The response document for the `ListTargets` method.

Root element
`BCGPublicAPI`

First child element
`ListTargetsResponse`

First child of `ListTargetsResponse`
`Targets`

Zero or more children of `Targets`
`Target` element. Contains 5 elements.

- `TargetId`: An internal numeric ID that identifies the target to the system.
- `Description`: A string describing the target.
- `ClassName`: The name of the target class. This is an optional class.
- `TransportType Name`: The transport type
- `TargetAttributes`: A complex type holding zero or more of the following:
  - `TargetAttribute`: A complex type holding the following elements. This is an optional element.
    - `AttributeName`: The target attribute's name
    - `AttributeValue`: The target attribute's value. This is an optional value.
- `TargetConfigPoints`: There are three target configuration points: PreProcess, PostProcess, and SyncCheck. Each of them is represented by a complex type holding the following elements:
  - `<ConfigPointName>`: A complex type holding the following element:
    - `Handlers`: A complex type holding zero or more of the following:
      - `Handler`: A complex type holding the following 3 elements. This is an optional element:
        - `ClassName`: The name of the handler class
        - `HandlerType`: The handler type
        - `HandlerAttributes`: A complex type holding zero or more of the following:

- HandlerAttribute: A complex type holding the following two elements. This is an optional element:
  - AttributeName: The attribute's name
  - AttributeValue: The attribute's value. This is an optional element.

ListEventDefinitions

Queries for the events configured on the system.

Root element
: `BCGPublicAPI`

First child element
: `ListEventDefinitions`

First child of `ListEventDefinitions`
: `UserInfo` element. This is the same information used in logging in through the Console. It contains 3 elements:
  - `UserName`: The Console login username.
  - `Password`: The Console login password.
  - `ParticipantLogin`: The participant (company) login name

ListEventDefinitionsResponse

The response document for the `ListEventsDefinitions` method.

Root element
: `BCGPublicAPI`

First child element
: `ListEventsDefinitionsResponse`

First child of `ListEventsDefinitionsResponse`
: `EventDefinitions`

Zero or more children of `EventDefinitions`
: `EventDefinition` element. A complex type holding the following 6 elements. This is an optional element.
  - `EventCode`: The code for this event
  - `Event name`: The event's name
  - `InternalDescription`: A string with the event's internal specific description
  - `Visibility`: The event's visibility in the system. A complex type holding 3 elements.
    - `CommunityManager`: A Boolean
    - `CommunityOperator`: A Boolean
    - `CommunityParticipant`: A Boolean
  - `Severity`: The event's severity. Available values are:
    - Info
    - Debug
    - Warning
    - Error
    - Critical
  - `Alertable`: A Boolean

BCGPublicAPIException

The response document in the case of an exception.

Root element
`BCGPublicAPI`

First child element
`BCGPublicAPIException`

First child of `BCGPublicAPIException`
`ErrorMsg`: A string containing the error message

Chapter 11. Using External Event Delivery

WebSphere Business Integration Connect generates and stores events as a way of monitoring the activity inside the Connect system. Events are published to an internal queue from which the WBI-C event server fetches them. The event server then sends them on to the internal event store. Until this release the only way to access the record of these events was through the Community Console Event Viewer. As of the 4.2.2 release of Connect, however, events can also be delivered to an external JMS queue, where they can be fetched by other concerned processes, such as monitoring applications. This chapter provides an overview of this process. It consists of two sections:

- "The external event delivery process"
- "The structure of delivered events" on page 145

The external event delivery process

There are two different types of events in the Connect system: document events and message events. Document events are events directly associated with a business document. The Business Processing Engine is responsible for publishing these events to a WBI-C internal queue. In the case of either a Sent or a Failed document state, the Delivery Manager also publishes business document events to this queue. Message events, on the other hand, are published by all components of WBI-C. Message events are not necessarily related to a business document, although one or more message events *may* be associated with a business document.

Events that are published to the internal queue are sent by the event server to Connect's event store. As of the 4.2.2 release, however, users may also decide to have events delivered to an external JMS queue. Turning external delivery on and off and configuring the external queue are done through the Community Console. See the *WBI-C Hub Configuration Guide* for help in setting up this up.

Events are delivered to the JMS queue in CBE XML format. The CBE, or Common Base Event, format is a part of a larger evolving IBM initiative, the Common Event Infrastructure, or CEI, designed to standardize the handling of events across applications. CBE structure covers three basic types of information:

- CBE standard properties, consisting of details such as creation time, event type, source, severity, and so forth
- CBE context data, including information about the environment in which the event was generated
- CBE extended data, holding generic data that is specific to the event type

The specifics of the CBE format as it is used in external event delivery are detailed in "The structure of delivered events" on page 145.

If external delivery is turned on, all events will be delivered to the external queue. The visibility flag, which limits which type of user may see which type of event in the Console Event Viewer, is not used in external delivery. Event names and descriptions in external delivery are localized in the same manner as they are in the Event Viewer.

Incorrect JMS configuration and/or JMS provider problems can cause errors in external event delivery. If they are not detected on startup, and an external delivery error occurs, the following happens:

- Future event delivery is turned off.
- Events will be re-delivered upon system re-start *only* if the user re-initializes the system in the following ways:
  - By correcting and updating the JMS properties on the *Event Publishing Properties* screen in the Console (see the *Hub Configuration Guide* and Console online Help for more information)

  Or

  - By correcting the JMS provider issues and clicking **Save** on the *Event Publishing Properties* screen in the Console
- An alert-able event is logged, so that the WBI-C alert engine can produce an alert. If for some reason the alert event cannot be logged, however, the event is ignored. No retries for logging this event are done.
- Normal internal event processing continues normally.

The structure of delivered events

This section covers the structure of events delivered to the external JMS queue.

The CBE event document structure

The CBE event document structure is complex, so the description of it is broken down into two parts:

- "The CBE event document structure"
- "CBE event structure for WBI-C message events and business document events"

For the full canonic description of the CBE structure, see the schema file, located at `\B2BIntegrate\events\schemas\commonbaseevent1_0_1.xsd`. In the same directory there is an additional schema file, `eventdelivery.xsd`. This file defines a WBI-C extension to the main schema which serves to define the `OtherSituation` type of the `SituationType` type used in the `situation` element in the main schema. Further information on CBE and the schema can be found at the Eclipse.org website, in the context of the Hyades project: www.eclipse.org/hyades/

The basic CBE document structure

The root element of a CommonBaseEvent document is a CommonBaseEvent element. The children of the CommonBaseEvent are as follows:

- `contextDataElements`: Provides context for the event. It is an optional event. WBI-C does not provide it.
- `extendedDataElements`: Captures information not captured directly by the basic CBE structure. It is an optional element. WBI-C provides it.
- `associatedEvents`: Captures associated events. It is an optional element. WBI-C does not provide it.
- `reporterComponentId`: Specifies the component that reports the event. It is an optional element. WBI-C does not provide it.
- `sourceComponentId`: Specifies the component that generated the event. It is a required element. WBI-C provides it.
- `msgDataElement`: Represents the data that is used to specify all of the related information that is associated with the message that this event holds. It is an optional element. It is generated for CBE events created for message events. For business document events, this element is *not* generated. WBI-C always generates this element as follows:
  ```
  <msgDataElement msgLocale="en-US"></msgDataElement>
  ```
- `situation`: Describes the type of situation that caused the event. It is a required element. WBI-C provides it.

CBE event structure for WBI-C message events and business document events

This section provides an element by element description of the CBE elements supplied in the event documents generated by the WBI-C external event delivery system. It includes a detailed listing of the main elements' attributes. Some descriptions includes a brief example of that element as it would appear in CBE XML for message and business document events, as appropriate.

The CommonBaseEvent element: This is the root element of all CBE event documents. The following table describes this element and its attributes.

Table 2. The CommonBaseEvent element

Property Name	Description
`Version`	1.0.1 WBI-C 4.2.2 supports this version of the schema
`localInstanceId`	Unique identifier in WBI-C store: • Message events: the event ID of the source event • Business document events: the UUID of the business document
`creationTime`	Creation time of this CBE event: • Message events: the creation time of the event • Business document events: since logging time is not stored in business document, set to current time
`severity`	• Message events: – Debug: 8 – Information: 10 – Warn: 30 – Error: 50 • Business document events: business documents have no severity, so this is set at 10 (Information)
`priority`	WBI-C has no notion of priority. Always set at 50
`msg`	• Message event: description of this event Localized. • Business document event: not specified
`repeatCount`	Not specified by WBI-C
`elapsedTime`	Not specified by WBI-C
`extensionName`	Used to distinguish message events from business document events • Message event: `BCG_EVENT` • Business document event: `BCG_BUSINESSDOCUMENT`
`sequenceNumber`	Not specified by WBI-C

This is a sample of the `CommonBaseEvent` element for a message event:

```
<cbe:CommonBaseEvent
         creationTime="2004-06-20T06:26:01"
         extensionName="BCG_EVENT"
         localInstanceId="1087712761674000C766F006F0178601DF89630A39DF6CA"
         msg="ASValidation"
         priority="50"
         severity="10"
         version="1.0.1"
         xsi:schemaLocation=
                        "http://www.ibm.com/AC
                        /commonbaseevent1_0_1commonbaseevent1_0_1.xsd">
:
:
<cbe:CommonBaseEvent />
```

This is a sample of the CommonBaseEvent element for a business document event:

```
<cbe:CommonBaseEvent
          creationTime="2004-06-20T06:26:02"
          extensionName="BCG_BUSINESSDOCUMENT"
          localInstanceId="1087712759944000C766F006F017860B7583EB51E26A336"
          priority="50"
          severity="10"
          version="1.0.1"
          xsi:schemaLocation="http://www.ibm.com/AC
             /commonbaseevent1_0_1 commonbaseevent1_0_1.xsd">
:
:
<cbe:CommonBaseEvent />
```

The sourceComponentId element: This element specifies the component that generated the event. WBI-C fills this in in the normal CBE way. Please see the schema for more information.

The situation element: This element describes the type of situation that generated the event. The following table describes this element and its attributes.

Table 3. The situation element

Property Name	Description
categoryName	OtherSituation
reasoningScope	INTERNAL
faultType	WBI-C defines this attribute for OtherSituation in eventdelivery.xsd. Message events: • SOURCE • TARGET • SYSTEM • UNKNOWN Business document events: • UNKNOWN

This is an example of the situation element for a message event:

```
<cbe:situation categoryName="OtherSituation">
               <cbe:situationType
               reasoningScope="INTERNAL"
                xsi:type="cbe:OtherSituation">
               <bcg:faultType/>
           </cbe:situationType>
</cbe:situation>
```

The extendedDataElements element: This element captures information not captured directly by the basic CBE structure. The following three tables describes this element, its attributes, and its specialized child elements, covering message event extended elements and business document event extended elements.

Table 4. The extendedDataElements element

Property Name	Description
name	Used to distinguish message events from business document events • Message event: BCG_EVENT • Business document event: BCG_BUSINESSDOCUMENT
type	WBI-C sets this to noValue
children	One or more elements are created, depending on the type (message or business document) of event. Descriptions in the tables below

The message event extended data elements:

Table 5. Message event extended data elements

Name	Value
BCG_EVENTCD	Event code from the message event
BCG_HOSTIPADDRESS	Host IP address. Specified if available
BCG_PARTNERID1	Internal ID for participant. Specified if available
BCG_PARTNERID2	Internal ID for participant. Specified if available
BCG_STACKTRACE	Stack trace. Specified if available
BCG_FRIPADDRESS	From: IP address. Specified if available
BCG_PARENTBCGDOCID	Unique ID for parent business document. Specified if available
BCG_BCGDOCID	The ID of the business document with which this message event is associated **Note:** Monitoring applications can use this element for correlating this event with any associated business document
BCG_USERID	User ID. Specified if available
BCG_BUSINESSID1	From: business ID. Specified if available
BCG_INITBUSINESSID	Initiating business ID. Specified if available
BCG_INITASMESSAGEID	Initiating AS message ID. Specified if available
BCG_BUSINESSID2	To: business ID. Specified if available

A partial example of an `extendedDataElements` element in a message event.

```
<cbe:extendedDataElements name="BCG_EVENT" type="noValue">
        <cbe:values/>
               <cbe:children name="BCG_EVENTTIMESTAMP" type="string">
               <cbe:values>1087712761674</cbe:values>
           </cbe:children>
              <cbe:children name="BCG_PARENTBCGDOCID" type="string">
              <cbe:values>1087712759944000C766F006F017860B7583EB51E26A336
              </cbe:values>
          </cbe:children>
              <cbe:children name="BCG_ARGUMENTSTRING" type="string">
              <cbe:values>ASValidation</cbe:values>
          </cbe:children>
              <cbe:children name="BCG_HOSTIPADDRESS" type="string">
             <cbe:values>127.0.0.1</cbe:values>
          </cbe:children>
```

```
:
:
:
</cbe:extendedDataElements>
```

The business document event extended data elements:

Table 6. The business document event extended data elements

Attribute	**Value**
`BCG_BCGDOCID`	Business document's unique document ID
`BCG_PARENTBCGDOCID`	Document ID for parent business document. Specified if available
`BCG_DOCLOCATION`	Location of business document with complete path. Specified if available
`BCG_DOCSTATE`	Current state of the business document. • DOC_IN_PROCESS="In Process" • DOC_SENT = "Sent" • DOC_RECEIVED = "Received" • DOC_FAILED = "Failed"
`BCG_DOCSIZE`	Obtained from business document. Specified if available
Business document related data	In addition, business document events may contain other information concerning: • routing related data • flow related data • business protocol related data The name attribute of the child elements will be set to one of the constants specified in the BCGDocumentConstants class. See "BCGDocumentConstants" on page 84 for further information. Specified only if available

A partial example of an `extendedDataElements` element in a business document event.

```
<cbe:extendedDataElements name="BCG_BUSINESSDOCUMENT" type="noValue">
   <cbe:values/>
          <cbe:children name="BCG_BCGDOCID" type="string">
         <cbe:values>1087712755684000C766F006F01786046684D6EAC6FAC22
         </cbe:values>
      </cbe:children>
          <cbe:children name="BCG_DOCLOCATION" type="string">
            <cbe:values>
              /opt/IBM/WBIConnect/common/data/Inbound/process
              /520/D9
              /1087712753565000C766F006F003149F07FF1FC6C41D8D9.ascontent
        </cbe:values>
      </cbe:children>
          <cbe:children name="BCG_PARENTBCGDOCID" type="string">
        <cbe:values>1087712753565000C766F006F003149F07FF1FC6C41D8D9
        </cbe:values>
      </cbe:children>
          <cbe:children name="BCG_DOCSTATE" type="string">
```

```
    <cbe:values>In Process</cbe:values>
  </cbe:children>
      <cbe:children name="BCG_DOCRESTARTED" type="string">
    <cbe:values>false</cbe:values>
  </cbe:children>
      <cbe:children name="BCG_FRPARTNERTYPE" type="string">
    <cbe:values>0</cbe:values>
  </cbe:children>

  :
  :
  :

</cbe:extendedDataElements>
```

Index

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800

Burlingame, CA 94010
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not necessarily tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program’s tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

IBM
the IBM logo
AIX
CrossWorlds
DB2
DB2 Universal Database
Domino
Lotus
Lotus Notes
MQIntegrator

IBM WebSphere Business IntegrationConnect Enterprise and Advanced Editions Version 4.2.2.