Connect Customization Guide

IBM **Connect** *for iSeries*

Date: October, 2002

Contents

This page intentionally left blank.

1.0 Introduction

1.1 Overview

IBM Connect for iSeries was announced October 3, 2000 as a software integration framework for Business to Business. It provides secure integration of your existing core business applications with the business applications of your trading partners. It is meant to be a high function, low-cost, easy to deploy framework. As a framework, it is designed to be customized to provide an end-customer solution. This customization is in two major areas: A) the customization to accept a diverse set of XML protocols and b) the customization to a diverse set of legacy, business applications. The Connect Customization Guide provides the needed information to perform these two areas of customization. It is used as a suppliment to the iSeries Connect API Javadoc and the set of Connect Samples.

The objective of IBM Connect for iSeries is to NOT require any coding in order to integrate your core business applications with the Connect framework. However, we realize that we cannot foresee all the ways you would want to integrate their applications or customize the framework. That is why Connect provides a series of APIs to extend the framework. This Customization Guide is a suppliment to the API documentation. It is meant to provide the needed information to effectively utilize the APIs that the Connect product provides.

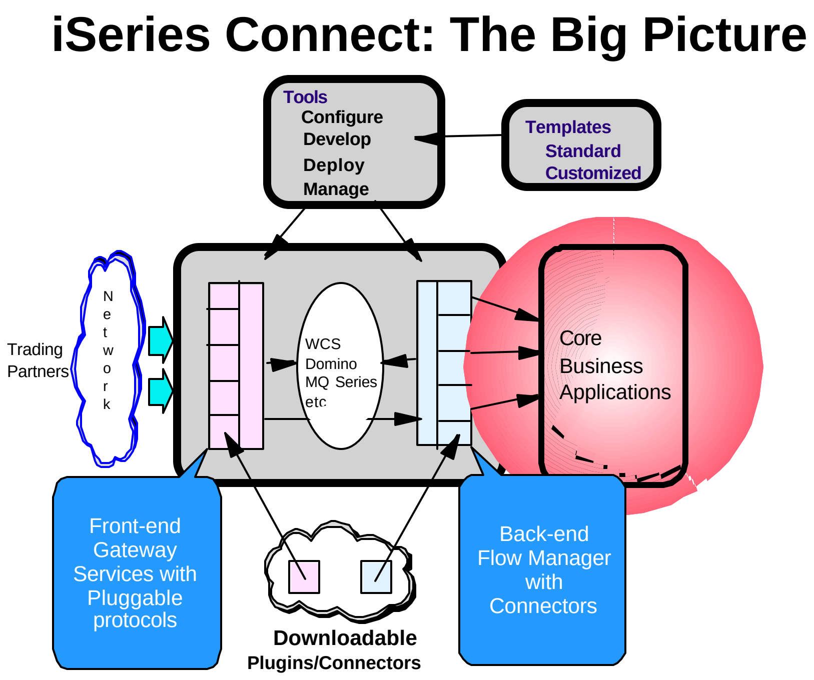
There are several tasks that can be accomplished by using the Connect APIs. Among these tasks are the following:

- Utilize the tools to define a unique XML Protocol to Connect.
- Create the protocol flow to allow gateway services to be rendered for the XML message.
- Create a connector (that can be one of various types) to interface with a backend application and learn how that connector can then be plugged into an application flow.
- Exchange information and provide a mapping between a Connect Provider or Partner entity and a corresponding Backend Application entity (such as a customer, patient, agent, client, merchant, store, etc).
- Enhance or share the information stored in the Provider or Partner Registry with a backend application and learn how you can automatically keep the Connect Provider or Partner registry in sync with the backend entity information through the use of registered exit programs.
- Generate a request to send an asynchronous message from an application

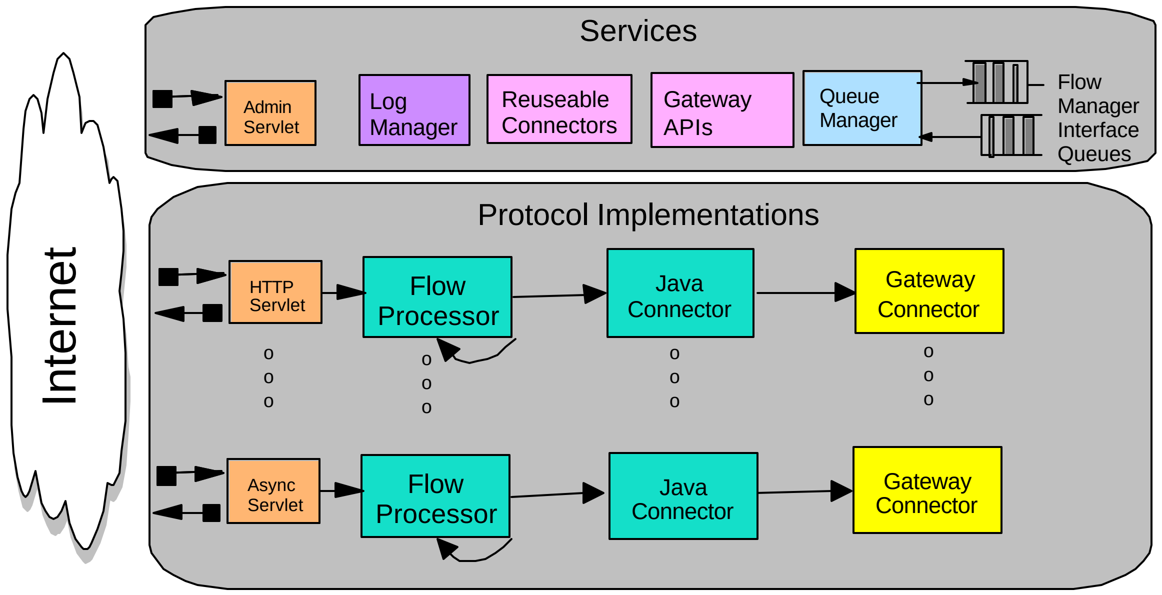
.

2.0 Connect Architecture Overview

Connect for iSeries is logically divided into two halves. The first half is called the Delivery Gateway, or Gateway for short. The Gateway accepts requests from remote trading partners in currently-supported trading partner “language” (XML protocol), like cXML.. (Note: throughout this paper we use the term “trading partner” to describe both customers and marketplaces). The second part is the Flow Manager. The Flow Manager handles connecting (mapping) the protocol “payload” (the information sent within the protocol request) to your existing business processes.


iSeries Connect: The Big Picture
Tools
Configure
Develop
Deploy
Manage
Templates
Standard
Customized
Trading Partners
Network
WCS
Domino
MQ Series
etc
Core Business Applications
Front-end Gateway Services with Pluggable protocols
Downloadable Plugins/Connectors
Back-end Flow Manager with Connectors

B2B Delivery Gateway Framework



- Handles the interfacing with various business partners over a variety of business protocols e.g. cXML
- Does marketplace and protocol authentication
- Forwards request (and response template) to Flow Manager
- Sends the response back to the requester
- Protocol connectors handle authentication and request/response processing

3.0 Connectors

The IBM Connect for iSeries product has two main runtime components when processing a request: Delivery Gateway and Flow Manager. Both runtime components share the same runtime architecture and code. This architecture/code is called the flow engine. The flow engine provides the runtime environment functions of process control flow and data storage flow. From an interface point of view, the flow engine provides a similar and consistent programming environment for both the Gateway and Flow Manager. One of the benefits of sharing the flow engine is that once you understand the environment in one component, that knowledge will apply to the other component.

3.1 Connector Overview

3.1.1 Connectors

The flow engine provides the capability to invoke legacy or newly written applications. The flow engine provides this capability through its connector architecture. Each connector can be thought of as an application access method. The following are the currently supported access methods (connectors):

Program Call Connector - This connector allows invoking applications via a program call. It is used to invoke applications that are designed with a program call interface. These applications can exist on either the local or a remote system. This connector utilizes the Program Call Meta Language (PCML) support from the AS/400 Java Toolbox. You can use this connector to invoke C, C++, RPG and Cobol progams via a call and return mechanism.

Queue Connector - This connector allows communicating with applications by putting a message on a queue. It is used to access applications that are designed to accept/return messages via data queues. These applications can exist on either the local or a remote system. The connector will utilize either AS/400 Data Queues or MQ Series queues.

JDBC Connector - This connector allows accessing data from relational databases. It utilizes JDBC drivers to query, insert, update or delete rows in a relational database.

Java Connector - This connector allows the invocation of a Java Class that implements a specific Java Interface. This is the main source of user written code that uses the APIs provided by Connect. The Java Connector is used to invoke a Java Connector Application. The Java Connector Application can be written to do anything the end user requires. For example: it may take the request and generate a HTTP message and post it to a URL. The Java Connector Application can utilize the APIs that are described below to retrieve data associated with Requests, set data for the Response and retrieve data that is maintained by the flow engine.

3.1.2 Named Data Contexts

Named Data Contexts allow the protocol writer to define a data storage area that they can store information and have that information available to either the Gateway, Flow Manager, or both. Contexts have different attributes and are defined for each protocol. Contexts can be mappable. Connect provides support for several types of named data contexts.

The following provides a description of the context types that are available in the Flow Engine for version 2.0.

- **Header -** The header context type provides a data context that supports name/value pairs of simple string fields. There are two header contexts predefined by Connect. The header type data contexts are the “MessageHeader” and “InternalHeader” message header contexts used by the Gateway and Flow Manager infrastructure. In developing a custom protocol, users have the ability to define more named contexts of type header.

- **DOM -** The DOM context type is a data context type that provides mapping to a w3c DOM document. It only provides access for storing data as strings. The DOM type context allows repeating fields and structures. DOM type contexts are generally used for the request and response documents.

- **Intermediate (IDA) -** The IDA context type provides mapping to a data store that provides a DOM-like structure for repeating fields and structures, but also allows for storage of any of the Connect supported data types (not just string fields). The IDA context type was used in 1.1 to support the Intermediate context that is used to map data between steps in a flow.

Protocol writers must take context requirements into account when developing their protocol. There are some fields they will be required to map into the pre-defined contexts, and some fields that are set in the infrastructure that may be useful for them. They will also have some control over the data contexts that are available for their protocol flows and all process flows deployed for that protocol. When creating new contexts for the protocol, the protocol tools and protocol writers must consider the following factors:

- Context names must be unique. There is a set of reserved names for the Connect pre-defined contexts. The reserved names are:
  - MessageHeader
  - InternalHeader
  - Partner
  - Provider
  - Protocol
- They should consider if the context should be available to the process flows running in the Flow Manager to be used by backend applications. In this case, the context will be sent across on the queue to the Flow Manager by the Gateway’s FMCommunicatorConnector.
- Likewise, they should consider if the context could be updated by the process flow and should be returned to the Gateway when the response step is encountered in the process flow. In this case, the context will be sent back on the queue from the Flow Manager to the Gateway’s FMCommunicatorConnector.
- A context should contain related data when possible. The data in the sendable contexts are serialized to be sent between the Delivery Gateway and Flow Manager. In the Flow Manager, changed data contexts are serialized between each step of a restartable process flow for flow recovery. Extraneous data included in those contexts will affect performance of the Gateway and Flow Manager communications, and in the case of flow recovery, may require a lot of extra data to be serialized and written to the recovery database when other unrelated data in the context changes.

The following sections describe the data context requirements and assumptions from the flow engine’s perspective for each data context.

3.1.2.1 Message Header (required by Flow Engine)

The Flow Engine relies on the following fields to be set in the “MessageHeader” message header context. The Delivery Gateway infrastructure sets these fields prior to the start of any protocol flow. These fields are documented in the XML Protocol Customization section.

- **GATEWAY_INSTANCE**
- **CONTENT_REQUEST** - used in the Flow Manager for flow selection
- **PROTOCOL_GROUP** - used in the Flow Manager for flow selection
- **PROTOCOL_TYPE** - used in the Flow Manager for flow selection

3.1.2.2 Internal Header (required by Flow Engine)

The Flow Engine sets the following fields into the “InternalHeader” message header context. These fields are available for the Flow Engine running either in the Gateway (GW in the access column below) or the Flow Manager (FM in the access column), but are not sent across on the queue, so the values apply to the environment in which the Flow Engine is running. (In the access column, GW means Delivery Gateway, FM means Flow Manager, R means Read access, W means read and write access).

Header Constant Name	Mapping Name	Access	Description
CURRENT_STEP_NAME	`CurrentStep`	GW - R FM - R	This field contains the name of the current step running in the flow. It is set by the Flow Engine.
FAILED_STEP_NAME	`FailedStep`	GW - R FM - R	This field contains the name of the failed step in the flow. It is set by the Flow Engine.
IN_ERROR_PATH	`IsError`	GW - R FM - R	This field indicates whether the flow is running in the error path. The values are “true” or “false”, and it is set by the Flow Engine.
RESTART	`IsRestart`	GW - N/A FM - R	This field indicates if this flow has been restarted during startup based on information stored in the Flow Manager recovery database. The values are “true” or “false”. It is set by the Flow Engine when running in the Flow Manager.
FM_ERROR_INFO_CODE	`FlowErrorCode`	GW - R FM - R	This field contains the error code for the flow.
FM_ERROR_INFO_STRING	`FlowErrorText`	GW - R FM - R	This field contains the error text for the flow error.
STEP_APP_ERROR_INFO_CODE	`StepApplicationErrorCode`	GW - R FM - W	This field is the error code for the application connector. The values are application dependant. An error can be indicated by mapping a non-zero value into this field or by returning a non-zero return code from a Java connector. A non-zero return code from a Java connector will override anything mapped into this field. When the next step begins, this error code is reset back to zero. A copy of the error code is available to subsequent error steps via the ApplicationErrorCode field.
STEP_APP_ERROR_INFO_STRING	`StepApplicationErrorText`	GW - R FM - W	This field is the descriptive error text for the application

			connector. This field can be set by mapping or returned from a Java connector. A non-zero error code from a Java connector will cause the returned error text to override anything mapped into this field. When the next step begins, this error text is reset to an empty string. A copy of the error text is available to subsequent error steps via the ApplicationErrorCode field.
APP_ERROR_INFO_CODE	`ApplicationErrorCode`	GW - RW FM - R	This field is a copy of the StepApplicationErrorCode and represents the error code of the previous step.
APP_ERROR_INFO_STRING	`ApplicationErrorText`	GW - RW FM - R	This field is a copy of the StepApplicationErrorText and represents the error text of the previous step.

3.1.2.3 Partner (required by Flow Engine)

The partner context is used to map partner (buyer) business data from a protocol or process flow. A key must first be mapped out to the partner context that represents the partner token used to access the appropriate partner information. Once the key is mapped to the partner context, the remaining business data defined in the Request Message Format (RMF) will be available for mapping. (see the XML Protocol Customization section for an explanation of RMFs). The keys should be writeable from the protocol flows, but read only in the process flows (controlled by the RMF). Business data set from mapping is written to the partner configuration database and available from outside the flow.

The Flow Manager accesses the partner token from the partner context for use during flow selection. If a partner definition is required by the protocol, the partner token must be mapped out to the partner context in the protocol flow before sending a request to the Flow Manager. If the protocol does not have partner information, the partner will be null when processed as part of the flow selection. In this case, any configured flow with *ALL for the partner would be considered a match for the null partner in the flow. Connect has provided a utility connector to retrieve the partner token.

3.1.2.4 Provider (required by Flow Engine)

The provider context is used to map provider (supplier) business data into a protocol or process flow. A key must first be mapped out to the provider context that represents the provider token used to access the appropriate provider information. Once the key is mapped to the provider context, the remaining business data defined in the RMF will be available for mapping. The keys will be defined in the RMF as writeable from the protocol flows, but read only in the process flows. Business data set from mapping is written to the partner configuration database and available from outside the flow.

The Flow Manager accesses the provider token from the provider context for use during flow selection, so the provider token must be mapped out to the provider context in the protocol flow before sending a request to the Flow Manager. Connect provides a utility connector to retrieve the provider token.

3.1.2.5 Protocol

The protocol context is used to map the protocol information from within a protocol or process flow. The protocol and protocol group will be set by the Gateway infrastructure which will provide an initial set of data available for mapping. When the provider key is set into the provider context, additional data or new values will be available in the protocol context based on the current provider. When the partner key is set into the partner context, additional data or new values will be available in the protocol context based on the current partner. The keys to the Protocol context will be defined in the RMF as read only in the protocol and process flows.

3.1.2.6 DOM Contexts

A protocol can create an arbitrary number of DOM contexts to provide XML documents available for mapping and programmatic access. Two typical contexts would be one to represent the incoming request and one to represent the outgoing response, although any number of contexts with any context names (except for those names reserved by Connect) may be used.

3.1.2.7 IDA Contexts

Several IDA type contexts will be used in protocol and process flows. One or more IDA type contexts will be used to hold the data mapped between steps in the flow. This is the “Intermediate” context from version 1.1, and does not have an RMF associated with its data.

3.1.2.8 Key Attributes

The Flow Engine will now allow for special key fields to be mapped to or from certain contexts to affect the context as a whole. These key fields are associated with the context, but are not stored in the actual data store associated with the context. The following key fields are available for version 2.0:

Context	Key Reference	Description
DOM contexts	#ValidationLevel	This key field is used to override the level of validation that is performed when the DOM is parsed. The valid values are “Full”, “EntityResolver”, and “None”. This will override the default validation level for the context for all future validation in the flow.
DOM contexts	#XMLByteArray	This represents the full XML document for the DOM context. When a mapin (or copy step) occurs to read this key field, the current DOM will be serialized and provided in the byte array. When a mapout (or copy step) occurs to update this key field, the document in the byte array will be parsed and used to populate the DOM context.
DOM contexts	#XMLString	This represents the full XML document for the DOM context. When a mapin (or copy step) occurs to read this key field, the current DOM will be serialized and provided as a string. When a mapout (or copy step) occurs to update this key field, the document in the String will be parsed and used to populate the DOM context.
Provider	#Provider_Refno	This is the provider key that is used to initialize the Provider context. Once it is set by the protocol, the remaining provider fields are available for mapping. It also causes the provider specific fields in the Protocol context to be available.
Partner	#Partner_Refno	This is the partner key that is used to initialize the Partner

		context. Once it is set by the protocol, the remaining partner fields are available for mapping. It also causes the partner specific fields in the Protocol context to be available. .
Protocol	#Protocol_Group	This is the protocol group key that is used to initialize the protocol context. Once this and the #Protocol field are set by the Delivery Gateway framework, the Protocol fields are available for mapping.
Protocol	#Protocol	This is the protocol key that is used to initialize the protocol context. Once this and the #Protocol_Group field are set by the Delivery Gateway framework, the Protocol fields are available for mapping.

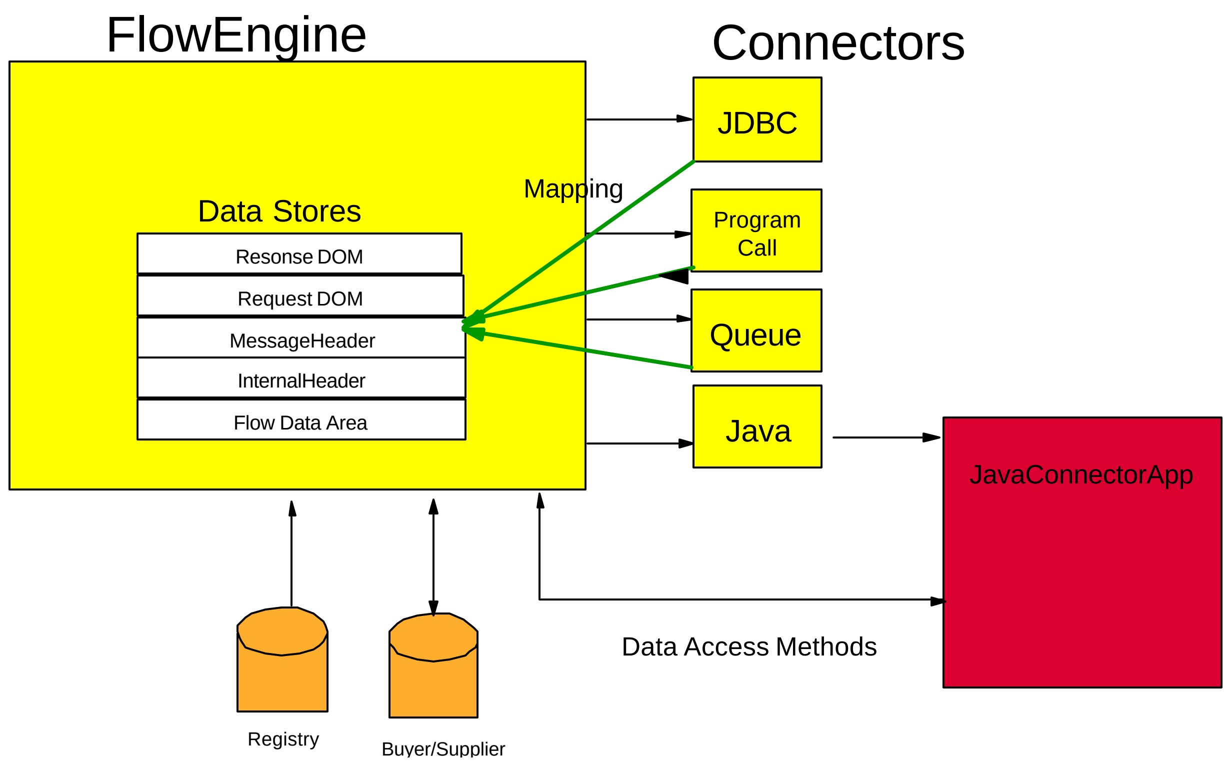
3.1.3 Gateway and Flow Manager Architecture

There is one important difference between the two environments: the Gateway and the Flow Manager. The Flow Manager only allows access to the data stores through mapping. The Gateway provides a mappable access plus direct access to the data stores. Mappings lets you associate fields from the data stores with input or output fields in your application or process step. Mappable access is strictly one field at a time processing. The API methods that provide mappable access have the architecture of retrieving an object associated with a field and then retrieving or setting a value associated with that field. For example, mappable access will only allow the retrieving and setting of a specific field of a DOM context. One of the main benefits of the mappable interface is that it isolates the user from the format of the data. When a program requests/sets data one field at a time, the Connect product will ensure that the data is found no matter what format the data is in. In addition, Connect will format the data correctly when setting response data. The Mappable interfaces provides Connect the ability for applications to be protocol independent.

In contrast to the mappable interface, direct access will allow access to a broader range of data elements of the data store objects. The API methods that provide direct access have the architecture to allow retrieving and setting data element objects. For example, direct access provides a method to retrieve or set an entire DOM in a DOM context. This interface is not protocol independent. If an application wants to retrieve an entire DOM then that application needs to understand the format of the DOM (which makes it protocol dependent).

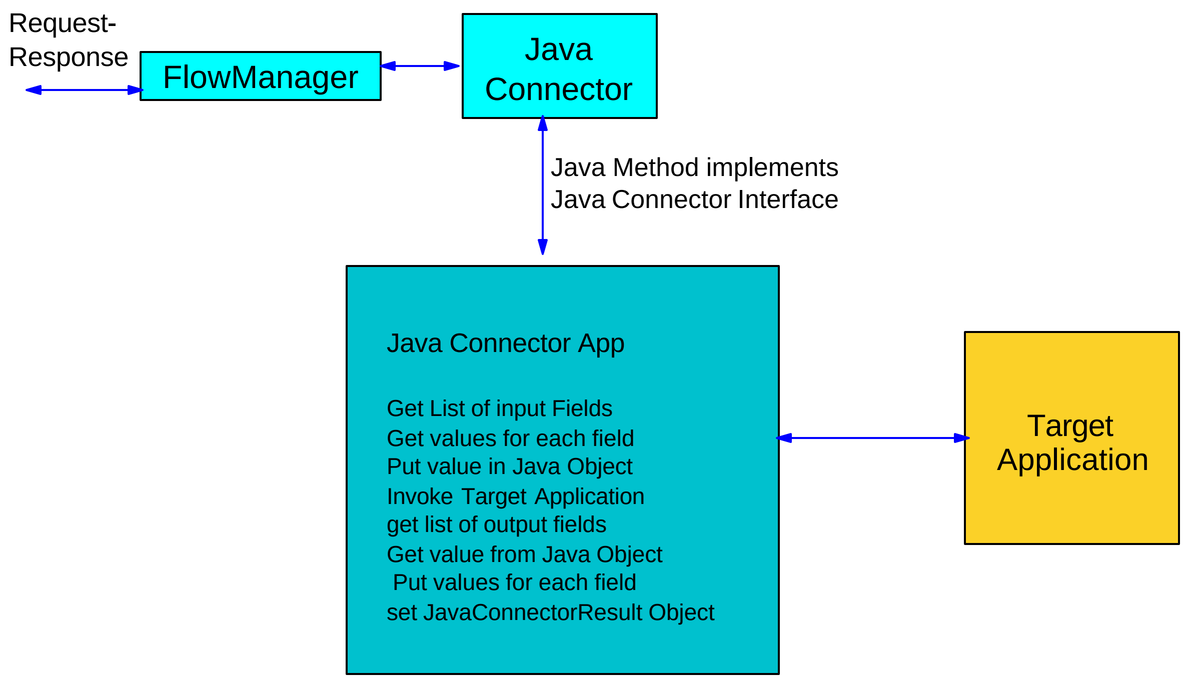
The diagram below depicts the above overview:

Interface Architecture



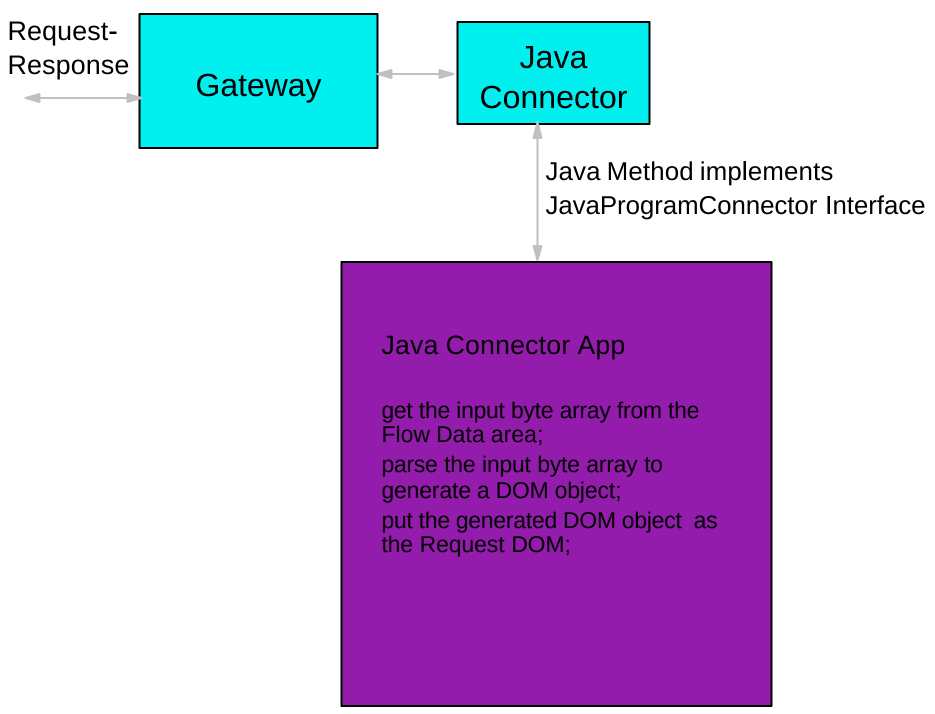
In the section above we described that the Flow Manager and the Gateway runtime components share the same flow engine. However, we noted that the Flow Manager only provides a mapping interface to the data stores that are maintained in the flow engine. The main purpose of the Flow Manager component is to communicate with legacy applications that will process the request message and generate the response message. So the Java Connector Application that is invoked via the Java Connector generally will be involved with transforming the request message into a format that is acceptable to a target legacy application. It will take the results from the legacy application and utilizing the Connect APIs, it will return the response in the correct format. The diagram below depicts a generic algorithm for a Java Connector Application that will utilize mapping to get request data, invoke a legacy application and then set the response data.

Flow Manager Java Connector


Request-Response
FlowManager
Java Connector
Java Method implements Java Connector Interface
Java Connector App
Get List of input Fields
Get values for each field
Put value in Java Object
Invoke Target Application
get list of output fields
Get value from Java Object
Put values for each field
set JavaConnectorResult Object
Target Application

In contrast to the Flow Manager component environment, the Gateway component has the ability to have direct access to the flow engine data stores (in addition to mappable access). The main purpose of the Gateway component is to provide a set of functions that are specific to each protocol. These functions generally involve authentication, authorization, data extraction, encoding, logging, and so forth.   In order for the Gateway Java Connectors Applications to provide their desired functions, it is necessary to provide the direct access to the data stores. One possible algorithm for using the direct access APIs is depicted in the diagram below:

Gateway Java Connector


Request-Response
Gateway
Java Connector
Java Method implements JavaProgramConnector Interface
Java Connector App
get the input byte array from the Flow Data area;
parse the input byte array to generate a DOM object;
put the generated DOM object as the Request DOM;

3.2 Where can code run in Connect?

There are generally two places in the Connect runtime where code can run: A java Connector Application and a Flow Manager Exit.

Java Connector Applications

The Connect for iSeries connectors allow the integration of business applications with Connect and allow the ability to support custom protocols. Generally, the connectors provide a specific access method to applications or data (for example, program call connectors allow the invocation of an application via a call/return access method). The exception to this is the Java Connector which invokes the Java connector application. The Java connector application can provide any function that is required. The Java connector applications can invoke a set of connect APIs to get information about a request and set response information. This is a key point for users and business partners to provide custom code for Connect.

Flow Manager Exits

These exit points will allow user written code to get executed at distinct points in the Flow Manager’s life. There are two exit points, one at initialization time and one at termination time. The initialization one is executed after the Flow Manager goes through it’s initialization phase, prior to waiting for requests on the Gateway queue. The termination point is during the cleanup phase of the Flow Manager prior to shutting down any processes/threads. The Flow Manager instantiates a Java class that implements a specified Java interface. Multiple Java exit classes can be registered with Connect. The Flow Manager will instantiate and run all classes that are registered.

To register the exit programs, the administrator/programmer must use the Connect administration GUI. Select an instance in the Instance tab and then select the Properties action button. In the Properties panel, press the Flow Manager tab and look down for an User Exit Class field. There will be a button to add the class name that contains the exit programs; eg com.user.MyInitializer. There can be multiple exit program properties registered.

The Java interface that the exit classes need to implement is:

```
public interface B2BExit {
    public void initialize() throws Exception;
    public void terminate() throws Exception;
}
```

The Flow Manager startup code will call all the initialize methods. If a user exit method throws an exception during the initialization exit, the Flow Manager will log a message and the Flow Manager will NOT start. The Flow Manager stop code that runs after all the connection threads have finished, will call the terminate methods. If a user exit method throws an exception during the termination exit, the Flow Manager will log a message and continue on. The Flow Manager will call the exit methods in "user threads" subject to the user security manager restrictions.

3.3 Data for tools compared to data at runtime

The tools for Connect are responsible for assisting users in configuring the runtime elements of Connect. They provide the data required by runtime to determine how to process request and generate responses. In dealing with the Connect runtime, the tool set has two main documents that are relevant to the Connect runtime:

Application Connector Document : this document is generated by the Connect tool set and it describes the interface to the applications that can be accessed by connectors in a process flow.

Process Flow: this document is generated by the Connect tool set and it describes the flow of steps that occur for a given request. This document indicates which applications are going to be invoked and in what sequence will they get invoked.

The deployment tool will then take these generated documents and build a runtime representation of the information contained in these documents. The runtime representation differs from the generated documents so that they can be optimized for runtime processing. At the start of the Delivery Gateway or Flow Manager, the runtime reads the runtime representation of the documents and build Java object representations of the information. These objects are then made available via the ConnectorParm and JavaConnectorParm objects.

3.4 Connector Interfaces

In Connect for iSeries, there are two Java Connector Application Interfaces that are supported. One is available only in the Gateway and is used for Custom Protocols. The other is available only in the Flow Manager and is used for integrating business applications with Connect. When the Flow Manager invokes a Java Connector Application, the com.ibm.connect.flowmanager.interfaces.JavaConnectorInterface is used. This interface allow user code to execute and makes available an object that provides the mappable access to data objects. This interface has a single method that takes a ConnectorParm object as input

run(ConnectorParm)
    execute the user java code..

The Java Connector Application must implement this Java Interface and make the class available to the Flow Manager (via it's classpath).

When the Gateway invokes a Java Connector Application, the JavaProgramConnectorInterface is used. This interface supplies the object that provide the mappable access to data objects. In addition, it provides direct access to data object that the flow engine maintains. The JavaProgramConnectorInterface has a single method that takes a ProgramConnectorParm object as input.

run(ProgramConnectorParm)
    Execute the user java code.

The Java Connector Application must implement this Java Interface and make the class available to the Gateway (via it's classpath).

The run methods above both return a JavaConnectorResult object. The JavaConnectorResult object contains two fields, a return code of type int and a return message of type String. It is the responsibility of the Java Connector Application to set the return code. If the return code is set to non-zero, then this will signal to the Flow engine that an error has occurred. The Flow Engine will then perform it's error processing. During error processing, the flow engine will stop the normal step flow and jump into the error step flow. Each step in the flow engine can optionally have an associated error step defined. This step will be the step that control will go to in case of an error. Setting the Java Connector Result object return code value to a non-zero value will signal that control will go to the error step. If an error step is not defined, then the flow will stop processing. In addition, the JavaConnectorResult object return code and return message will be automatically set in a InternalHeader field. The reason we will do this is that the error step can then use the mapping technology to find out what the return code and return message values were that caused the error step to gain control. The fields in the InternalHeader that corresponds to the return code is ApplicationErrorCode and the field that corresponds to the return message is ApplicationErrorText.

3.4.1 Mappable Access

The ConnectorParm object provides all the methods required to access data that is mapped. The ConnectorParm object provides methods to get Field Objects that are associated with the input and output parameters identified for this Java Connector Application. Each Field Object contains information about the field, including the mapping information to get or set the runtime value associated with the parameter. The MapCursor object allows the moving of a pointer to navigate a DOM. It is used for getting/setting values for a collection of related fields within a structure.

3.4.1.1 ConnectorParm Object

The ConnectorParm object provides all the methods required to access data that is mapped. The ConnectorParm object provides methods to get Field objects that are associated with the input and output parameters identified for this Java Connector Application. Each Field object contains information about the field, including the mapping information to get or set the runtime value associated with the parameter. The Field Object contains all the data associated with an input or output parameter defined for the Java Connector Application.

Step 1) Get Field objects. The first required step in every Java Connector Application is to get the set of field objects associated with the Java Connector Application. The Java Connector Application can get this set of objects using methods on the ConnectorParm object. The methods allow getting the input list of fields separate from the output list. This allows the mapping of the input fields, then the processing associated with the Java Connector Application and then the mapping of the output fields. The methods to get the list of input fields are:

getInputField(String)

Get a specific field defined for input. This method is used to get a single specific input field object. The name of the field is the input String and the output is the associated field object. The name of the field can be a path type name to get fields within structures. For example; if the input parameter is a structure such that the order_item structure contains a field called unit_price;to get the field object associated with the field unit_price then the input String must be "/order_item/unit_price".

getInputFieldList()

Get of list of fields defined for input.

getInputFieldList(Field)

Get of list of children fields for a field defined for input.

The getInputFieldList() methods (with no parameters) only get the list of first level parameters. If your input parameters contain structures then the getInputFieldList() will get the field object associated with the structure. In order to get the list of children field, the method getInputFieldList(Field) must be used. The Field parameter in getInputFieldList(Field) is the field object of the structure whose children you want to get field objects for.

For example; if the input parameters to a Java Connector Application were defined in the following PCML xml sample.

```
<pcml version="1.0">
<!-- PCML source for calling PCMLTest -->
  <program name="pcmltest" path="/QSYS.lib/B2BTEST.lib/PCMLTEST.pgm">
    <data name="incount" type="int" length="4" usage="input" />
    <data name="instance" type="char" length="10" usage="input" />
    <data name="mktplc" type="char" length="20" usage="input" />
    <struct name="fromc"  usage="input" >
      <data name="from_domain" type="char" length="64" usage="input" />
      <data name="from_identity" type="char" length="64" usage="input" />
    </struct>
    <struct name="toc"  usage="input" >
      <data name="to_domain" type="char" length="64" usage="input" />
      <data name="to_identity" type="char" length="64" usage="input" />
    </struct>
    <data name="orderID" type="char" length="6" usage="input" />
    <data name="t_type" type="char" length="6" usage="input" />
    <struct name="item"  usage="input" count="pcmlapperr.incount" >
      <data name="lineNumber" type="int" length="4" init="0" usage="input" />
      <data name="identifier" type="char" length="16" usage="input" />
      <data name="quantity" type="int" length="4" usage="input" />
      <data name="price" type="float" length="4" usage="input" />
      <data name="description" type="char" length="64" init="none" usage="input" />
    </struct>
  </program>
</pcml>
```

Then the following Java code snippet demonstrates how to get the Field Objects for the input parameters:

```
import com.ibm.connect.flowmanager.interfaces.*;
import com.ibm.connect.flowmanager.metadata.Field;
import java.util.*;


public JavaConnectorResult run(ConnectorParm parms)
{
        // Get the high level fields.  The Vector of Fields will contain the
       // following input fields
       //        incount
       //        instance
       //        mktplc
       //        fromc
       //        toc
```

```
    //      orderID
    //      t_type
    //      item
    Vector FirstLevelFields = parms.getInputFieldList();

    // The following will get a specific field (item) and then will get all the member fields
    Field itemfld = parms.getInputField("item");

     // The Vector of Fields will contain field objects associated with the following input fields:
     //     lineNumber
     //     identifier
     //     quantity
     //      price
     //      description
    Vector item_children_flds = parms.getInputFieldList(itemfld);

     // The following will get a specific child field in the structure item
     Field quantity = parms.getInputField("/item/quantity");
}
```

Once the Field object is retrieved, you can use it as input into subsequent ConnectorParm methods. In addition, you can get information about the field using the "get" methods on the Field object.

There is a synonymous group of methods for retrieving the output list of Field objects. These methods act the same as the getInputField methods except they retrieve the list of Field objects associated with output parameters defined for the Java Connector Application. These methods are:

getOutputField(String)
  Get a specific field defined for output.
getOutputFieldList()
  Get of list of fields defined for output.
getOutputFieldList(Field)
  Get of list of children fields for a field defined for output.

Step 2) (optional) Associate a Cursor with a Field. After getting the Field objects associated with the input and output parameters defined for the Java Connector Application, then the next step is to associate a MapCursor with a Field Object. This is an optional step. The MapCursor object allows the moving of a pointer to navigate a DOM. It is used for getting/setting values for a collection of related fields within a structure. For example; let's say the input to the Java Connector Application is a repeating structure that represents an order item. The order item structure has 4 child fields (price, quantity, identifier and description). It is important that the values for the child fields are the same relative to each other. This means that if you have 3 order items, that the values for the child fields of price, quantity, identifier and description of the first order are grouped together. If MapCursors were not present it is possible to get the values for price, quantity and identifier of the second order grouped with the description of the third order (this could happen if the description field had no value). MapCursor objects are not constructed, they are returned when getting or setting fields that are of type "struct". Once a MapCursor is returned then it can be "bound" to a field object. When bound to a field object then the getting or setting of that field object will be relative to that MapCursor position. If the field is not bound to any cursor, then it is automatically bound to the "root" or default cursor position. Generally, binding to the "root" or default cursor is sufficient if the field is not in a repeating structure. The methods for binding fields to cursor are as follows:

bindInfieldToCursor(Field, MapCursor)
    Bind an input field to a cursor
    This will create a relationship between a field and a specific cursor.
bindOutfieldToCursor(Field, MapCursor)
    Bind an output field to a cursor
    This will create a relationship between a field and a specific cursor.

Step 3) Retrieve or Set a value. The methods of ConnectorParm that provide the getting and setting of values perform many functions. First, they perform the mappings as defined by the Business Process Workbench tools. The second is that they do automatic data type conversion. They perform any default processing if the value to be retrieved is null. They will also perform any operations defined. The methods for getting values will convert from a String object to a target data type for all data stores except the Intermediate Data Area. The methods for setting values will convert from a source data type to the String object. In addition, the set methods will ensure that valid XML is generated if mapping to the Response DOM.

The methods for retrieving values follow a naming algorithm; getFieldAsXXXX(Field), where XXXX are the valid data types that Connect supports. The current list of data types that Connect supports are:

- BigDecimal
- Boolean
- Byte
- ByteArray
- Double
- Float
- Int
- Long
- Short
- Struct

These methods will find the mapping that was specified by the Business Process Workbench tools and retrieve the values by applying the mappings. The user of these methods need not be aware of the mapping specifications or where the values are retrieved from. These get methods will return the value in the data type that is specified on the method name. One way to programmatically figure out which data type to retrieve a field as is to use the Field Object getType() method. This will return the data type that a specific Field is specified in the input/output field template. Each method takes a Field object as input. The get methods return an array of values in the appropriate data type. The reason for this is that it will return a value for every mapping match that it finds. For example, if the input request has 4 elements that are called OrderNumber and each element has a value, then a call to getFieldAsInt() passing in a Field object that maps to the OrderNumber request will return an int array containing 4 elements.

The list of get methods currently supported is as follows:
getFieldAsBigDecimal(Field)
    Get of a field and return it as an array of BigDecimals.
getFieldAsBoolean(Field)
    Get of a field and return it as an array of booleans.
getFieldAsByte(Field)
    Get of a field and return it as an array of bytes.
getFieldAsByteArray(Field)
    Get of a field and return it as an array of byte[]s.

getFieldAsDouble(Field)
    Get of a field and return it as an array of doubles.
getFieldAsFloat(Field)
    Get of a field and return it as an array of floats.
getFieldAsInt(Field)
    Get of a field and return it as an array of ints.
getFieldAsLong(Field)
    Get of a field and return it as an array of longs.
getFieldAsShort(Field)
    Get of a field and return it as an array of shorts.
getFieldAsString(Field)
    Get of a field and return it as an array of Strings.
getFieldAsStruct(Field)
    Get of a field and return it as an array of cursors.

There is one special case method from the list above; the getFieldAsStruct(). This method will return an array of MapCursors. The length of the array is the number of structure items that were discovered in the mapping area. For example; let's say that the input parameters expect an order item as input (and it is a structure). In addition, the mapping specification indicates that this input structure maps to ItemOut element in the cXML Order Request. When processing each request and a getFieldAsStruct() call on the order item field is processed, the number of order items in that specific request is the length of the MapCursor array returned from getFieldAsStruct(). So if a request contains 3 ItemOut elements, then the MapCursor array will have a length of 3.

Each MapCursor object can be bound to a field so that the search for values will be relative to that MapCursor object. The ConnectorParm methods bindInfieldToCursor(Field,MapCursor) can be used for this binding. This will allow the grouping of field values relative to a specific structure.

The following is an example of how to use getFieldAsStruct(), bindInfieldToCursor() and getFieldAsInt(). It assumes that none of the input parameters are arrays. Let's say the following PCML XML snippet defines the input into your Java Connector Application.

```
<pcml version="1.0">
<!-- PCML source for calling PCMLTest -->
  <program name="pcmltest" path="/QSYS.lib/B2BTEST.lib/PCMLTEST.pgm">
    <data name="incount" type="int" length="4" usage="input" />
    <struct name="item"  usage="input" count="pcmltest.incount" >
      <data name="lineNumber" type="int" length="4" init="0" usage="input" />
      <data name="identifier" type="char" length="16" usage="input" />
      <data name="quantity" type="int" length="4" usage="input" />
      <data name="price" type="float" length="4" usage="input" />
      <data name="description" type="char" length="64" init="none" usage="input" />
    </struct>
  </program>
</pcml>
```

Here is a Java Code Snippet

```
// Declare a bunch of data elements
MapCursor[] cursorlist;
// Get the Field Objects associated with the input fields
Field itemfld = parms.getInputField("item");
Field linenum_fld = parms.getInputField("item/linenumber");
Field identifier_fld = parms.getInputField("item/identifier");
Field quantify_fld = parms.getInputField("item/quantity");
Field price_fld = parms.getInputField("item/price");
Field desc_fld = parms.getInputField("item/description");
// Declare the value arrays
int[] linenum_val;
String[] id_val;
int[] quantity_val;
float[] price_val;
String[] desc_val;

// Get the array of cursors as mapped by the structure field
cursorlist = parms.getFieldAsStruct(itemfld);

// Each array element corresponds to an order item in the request
for (int i=0;i<cursorlist.length;i++) { /*each struct element returned */

        // Get the line number for this order item
        parms.bindInfieldToCursor(linenum_fld,cursorlist[i]);
        linenum_val = parms.getFieldAsInt(linenum_fld);

        // Get the identifier for this order item
        parms.bindInfieldToCursor(identifier_fld,cursorlist[i]);
        id_val = parms.getFieldAsString(identifier_fld);

        // Get the quantity for this order item
        parms.bindInfieldToCursor(quantity_fld,cursorlist[i]);
        quantity_val = parms.getFieldAsInt(quantity_fld);

        // Get the price for this order item
        parms.bindInfieldToCursor(price_fld,cursorlist[i]);
        price_val = parms.getFieldAsFloat(price_fld);

        // Get the description for this order item
        parms.bindInfieldToCursor(desc_fld,cursorlist[i]);
        desc_val = parms.getFieldAsString(desc_fld);

        // Now that you have all the values for this order item, process them as a group
        // there should be only one value in each return array.
        ProcessOrder(linenum_val[0],id_val[0],quantity_val[0],price_val[0],desc_val[0]);

        // get the next order item
} /* end for */
```

The ConnectorParm object has corresponding methods for setting output parameter values. These methods follow a similar naming convention as the get methods; setFieldFromXXXX(), where XXXX is a supported data type. Each methods takes a Field object as input and a value in the data type of the method. The set method will perform the mapping, putting the value in the area as defined by the Business Process Workbench tools Change name?. In all the data stores except the Intermediate Data Area, the set method will perform a data type conversion from the input type into a String. Also, the set method will ensure that the XML document is properly formed and valid. The set of methods for setting values are:

setFieldFromBigDecimal(Field, BigDecimal)
  Put a BigDecimal field value
setFieldFromBoolean(Field, boolean)
  Put a boolean field value
setFieldFromByte(Field, byte)
  Put a byte field value
setFieldFromByteArray(Field, byte[])
  Put a byte[] field value
setFieldFromDouble(Field, double)
  Put a double field value
setFieldFromFloat(Field, float)
  Put a float field value
setFieldFromInt(Field, int)
  Put an int field value
setFieldFromLong(Field, long)
  Put a long field value
setFieldFromShort(Field, short)
  Put a short field value
setFieldFromString(Field, String)
  Put a string field value
setFieldFromStruct(Field)
  Put a struct field
  This will create an element node in the DOM tree or in the Intermediate Data Area and return a cursor to that newly created node.

As with the get methods, the setFieldFromStruct() method is a special case method. It will create an element node and return a MapCursor to that newly created node. This will allow the Java Connector Application to perform a bindOutfieldToCursor() and then set the member fields relative to that newly created element. This will allow control over the grouping of member fields in order to maintain proper relationships of values. This is very similar to the input field struct support. Note: the value of the node will not be set.

A sample of setting a field with a value

Let's say the following PCML XML snippet defines the output from your Java Connector Application.

<pcml version="1.0">
<!-- PCML source for calling PCMLTest -->

```
  <program name="pcmlouttest" path="/QSYS.lib/B2BTEST.lib/PCMLTEST.pgm">
     <struct name="item"  usage="output" >
     <data name="lineNumber" type="int" length="4" init="0" usage="output" />
     <data name="identifier" type="char" length="16" usage="output" />
     <data name="quantity" type="int" length="4" usage="output" />
     <data name="price" type="float" length="4" usage="output" />
     <data name="description" type="char" length="64" init="none" usage="output" />
   </struct>
  </program>
</pcml>
```

Here is a Java Code Snippet

```
// Declare a bunch of data elements
MapCursor cursor;
// Get the Field Objects associated with the output fields
Field itemfld = parms.getOutputField(“item”);
Field linenum_fld = parms.getOutputField(“item/linenumber”);
Field identifier_fld = parms.getOutputField(“item/identifier”);
Field quantify_fld = parms.getOutputField(“item/quantity”);
Field price_fld = parms.getOutputField(“item/price”);
Field desc_fld = parms.getOutputField(“item/description”);
// Declare the value
intlinenum_val;
String id_val;
Int quantity_val;
float price_val;
String desc_val;

try {
        // Create a parent element in the output DOM, returns a cursor to that new element
        cursor = parms.setFieldFromStruct(itemfld);

        // Set the line number for this order item
        Parms.bindOutfieldToCursor(linenum_fld,cursor);
        Parms.setFieldFromInt(linenum_fld, 25);  // create the element for linenum and set the
value to 25.  This will perform output mapping

        // Set the identifier for this order item
        Parms.bindOutfieldToCursor(identifier_fld,cursor);
Parms.setFieldFromString(identifier_fld, “1234”);  // create the element for identifier and set
the value to 1234.

        // Set the quantity for this order item
        Parms.bindOutfieldToCursor(quantity_fld,cursor);
        Parms.setFieldFromInt(quantity_fld, 2);  // create the element for quantity and set the
value to 2.
```

```
        // Set the price for this order item
        Parms.bindOutfieldToCursor(price_fld,cursor);
        Parms.setFieldFromFloat(price_fld, 12.56);  // create the element for price and set the
value to 12.56

        // Set the description for this order item
        Parms.bindOutfieldToCursor(desc_fld,cursor);
        Parms.setFieldFromString(desc_fld, “toothbrush”);  // create the element for
description and set the value to toothbrush


} Catch (e SetFieldException) {
        // do exception processing
} /* end try/catch */

 } /* end for */
```

There is a set of miscellaneous methods on the ConnectorParm object. The getProperty() methods allows the retrieval of Property values associated with the Connector type. For Java Connector Applications the only valid Property to retrieve is the properties file name that was entered for this Java Connector Application. The string to use for the properties file name is “propertyfilename”.

getProperty(String)

    Get a property value for a specified.

Connect generates a unique identifier for each new message that it receives. This identifier is generated and contains a timestamp (so it cannot be used for duplicate checking). The main purpose for using the identifier is for logging auditing or trace information about each message. The method getUniqueID() can be used to retrieve the unique identifier associated with the current message that is getting processed.

getUniqueId()

    Get the unique Id for this message

Connect maintains a version number. This is the current version of Connect that is running. Valid values are 1 for Connect 1.0, 2 for Connect 1.1 and 3 for Connect 2.0.

getVersion()

    Get the interface version.

3.4.1.2 Field Object

The Field Object contains all the data associated with an input or output parameter defined for the Java Connector Application. The first step in every Java Connector Application is to get the set of field objects associated with the Java Connector Application. The Java Connector Application can get this set of objects using methods on the ConnectorParm object (see below). Once the Field object is retrieved, you can use it as input into subsequent ConnectorParm methods. In addition, you can get information about the field using a set of get methods on the Field object.

There are two attributes that are associated with the Field object that require some clarification, the Name attribute and the Location attribute. The Name attribute is a '/' delimited, fully qualified, display name of the field. The value is the same as what was displayed via the Application Connection Document (ACD) editor tool (except that it is fully qualified). Fully qualified means that if the field is a child member of a structure, then all it's parent names will be part of the Name attribute. For example. If the field object represents a field named "order" and this field is a child of "item", which in turn is a child of "request", then the value for the Name attribute for this field object is "request/item/order".

The Location attribute can be used by a connector to locate the field in a target application access method. The value of the Location attribute depends on how the field is defined to the ACD editor tool. If you use PCML to define the interface to an application, then the Location value will be a '.' delimited, full qualified value. If you use XML to define the interface to an application, then the location will be of an XPATH notation. For example, if you use the following PCML to define an application interface, the Location value for "price" would be "pcmlouttest.item.price" (the Name value would be "item/price").

```
<pcml version="1.0">
<!-- PCML source for calling PCMLTest -->
 <program name="pcmlouttest" path="/QSYS.lib/B2BTEST.lib/PCMLTEST.pgm">
    <struct name="item"  usage="output" >
    <data name="lineNumber" type="int" length="4" init="0" usage="output" />
    <data name="identifier" type="char" length="16" usage="output" />
    <data name="quantity" type="int" length="4" usage="output" />
    <data name="price" type="float" length="4" usage="output" />
    <data name="description" type="char" length="64" init="none" usage="output" />
   </struct>
 </program>
</pcml>
```

If you used the following XML to define an application interface, the location value for "date" would be "order/@date" (the Name value would be "order/date").

```
<order date="x">
  <id>xxx</id>
  <price>1.1</price>
</order>
```

getCountfield()
    Returns the value of the count field attribute on this Field element.
getDefault()
    Returns the default value for this Field element.
getField(String)
    Returns the field object with the specified *name* attribute. The field identified by the String must be a child field for this field.
getFields()
    Returns the list of child field objects contained within this field.
getLength()
    Returns the int value of the length attribute on this Field element.

getLocation()
    Returns the value of the location attribute on this Field element.
getName()
    Returns the value of the name attribute on this Field element.
getPrecision()
    Returns the int value of the precision attribute on this Field element.
getRepeating()
    Returns the value of the repeating attribute on this Field element.
getSubtype()
    Returns the value of the subtype attribute on this Field element.
getType()
    Returns the int value of the type attribute on this Field element.
GetVariableLengthField
    Returns the name of another field that contains the length of this field.
IsVariableLength
    Indicates if the field is a variable length field
isRepeating()
    Indicates if the field is a repeating field.

3.4.1.3 Sample Code

Interface XML

```
<?xml version="1.0" encoding="UTF-8"?>
<order date="">
 <orderitem quantity="1" >
      <itemnum>1111</itemnum>
      <unitprice currency="USD">1.23</unitprice>
      <desc> PartA </desc>
 </orderitem>
 <orderitem quantity="21">
      <itemnum>222</itemnum>
      <unitprice >1.45</unitprice>
 </orderitem>
 <orderitem quantity="13">
      <itemnum>222</itemnum>
      <unitprice >1.45</unitprice>
      <desc> PartB </desc>
 </orderitem>
</order>
```

Request XML snippet

```
<cXML>
<Request>
<OrderRequest>
        <OrderRequestHeader  orderDate="1999-03-12" />
</OrderRequest>
<ItemOut quantity="2" requestedDeliveryDate="1999-03-25">
      ItemID>
             <SupplierPartID>1233244</SupplierPartID>
      </ItemID>
      <ItemDetail>
            <UnitPrice>
                <Money currency="USD">1.34</Money>
            </UnitPrice>
            <Description xml:lang="en">hello</Description>
       </ItemDetail>
</ItemOut>
<ItemOut quantity="5" requestedDeliveryDate="1999-03-25">
      ItemID>
             <SupplierPartID>11111</SupplierPartID>
      </ItemID>
      <ItemDetail>
            <UnitPrice>
                <Money currency="USD">10.25</Money>
            </UnitPrice>
            <Description xml:lang="en"> test part</Description>
       </ItemDetail>
</ItemOut></Request></cXML>
```

Java Connector Application

```
ConnectorParm parms; /* let's say that this is a fully initialized object */
Field orderfld = parms.getInputField("order/orderitem");
parms.bindInfieldToCursor(orderfld);
/* get a list of DOM cursors corresponding to this struct */
MapCursor orderlist = parms.getFieldAsStruct(orderfld);
if (orderlist != null) {
   Vector memberflds;
   Enumeration enum;
   Field currentmember;
   String[] values;
   /* get all the members fields name */
   memberflds = parms.getInputFieldList(orderfld);
   enum = memberflds.elements();
   /* for each struct value that is found and for each field in the struct process the
     value.  Those member fields may be struct themselves so we need to recurse */
   for (int i=0;i<orderlist.length;i++) {/*each struct element returned */
       while (enum.hasMoreElements()){ /* each member field name */
```

```
            currentmember = (Field)enum.nextElement();
            /* this will make sure we are pointing to the right spot in the input message */
            parms.bindInfieldToCursor(currentmember,orderlist[i]);
            values = getFieldAsString(currentmember);
            if (values != null) {
              System.out.println("Field {0} has value
{1}.",currentmember.getName(),values[0]);
            }/* end if*/
         } /* end while */
   } /* end for */
}/*end if */
```

3.4.2 Direct Access

The objective of providing direct access to Connect data stores is to allow the manipulation of data without requiring field by field mapping. The direct access methods provide access to the set of Connect data stores. This set contains the following:

Name	Mappabl e	Description
DOM context	Yes	This context is defined by the protocol designer and typically contains the request and or the response XML
Message Header context	Yes	This contains a set of fields that are available in a message header context
Flow Data Area	No	This contains objects that can be used for sharing information between Java Connector Applications within a specific flow instance.

At the start of the Gateway protocol flow, all these areas are empty or incomplete. The object of many of the Connectors in the protocol flow set values in these areas so that they can be used by subsequent Connectors in the Gateway protocol flow or in the Flow Manager.

3.4.2.1 ProgramConnectorParm Object

The ProgramConnectorParm object gets passed to each Java Connector Application that is invoked in the Gateway. It provides all the same methods as the ConnectorParm object described above. It also provides a set of methods that allow direct access to objects so that data doesn’t have to be processed on a field by field basis. The direct access methods are described below and are organized by methods associated with data stores.

3.4.2.2 Named DOM Context

A named DOM context is initially empty and once it contains values, it is mappable. Typically, one of the early steps in the protocol flow takes the XML byte array representation of the Request message and performs an XML Parse to generate the DOM object. The setDOM() method will take that generated DOM object and store it in a named DOM context. This will allow any subsequent connector (in both the Gateway and Flow Manager) to map fields out of the named DOM context.

A brief side note. The initialization of named DOM context with the XML string (or byte array) can be acheive throug another technique. Each DOM context has a special mappable key field that performs a specific function. For example, each DOM context has a key field called #XMLString. This key field treats DOMs as strings. So, a user can map an XML string into the #XMLString key field and Connect will parse the XML string and store it as a DOM in the DOM context. After this, the DOM context contains the fully parsed XML string and each field in the string is mappable. Conversely, at any point a user can map out of the key field #XMLString. The value returned is an XML string that represents the current state of the DOM. When use with the Copy step in the Protocol or Process flows, the utilization of mapping these key fields can be powerful and easy ways to manipulate entire XML strings (or byte arrays).

`getDOM(java.lang.String contextName)`
Return the DOM containing the xml associated with the named context provided.

`setDOM(java.lang.String contextName, org.w3c.dom.Document document)`
Set the document in the flow state for the context specified

A sample of using these methods are as follows:

```
// This sample will get the input request from the Gateway Framework, parse it and store
// the DOM in named DOM context called “InsuranceRequest”.  It will then get the DOM
// and extract some information from it.
import org.apache.xerces.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

{
Document RequestDOM;
DOMParser parser = new DOMParser();

// get the Input Byte Array from the Gateway Framework
// parms is the ProgramConnectorParm object
byte[] xmldoc =   (byte[])
                        parms.flowDataAreaGet(GatewayConstants.INPUT_BYTES_ARRAY);

// parse the XML byte  Array
 parser.parse(new InputSource(new ByteArrayInputStream(xml)));
// save the parsed DOM version of the XML byte array
 RequestDOM = parser.getDocument();

// Store it in the named DOM context associated with an Insurance protocol Request
 parms.setDOM(“InsuranceRequest”,RequestDOM);

// Get the Insurance request DOM back out and extract some information from it
Document inputDocument = parms.getDOM(“InsuranceRequest”,);
DocumentType docType = (DocumentType) inputDocument.getDoctype();
parms.flowDataAreaPut(GatewayConstants.REMOTE_DTD_NAME,docType.getSystemId(
));
}

}
```

3.4.2.3 MessageHeader Context

This context is limited in that it only accepts String data types as values. The semantics of storing/retrieving data is a name/value pair technique. You can retrieve a value by providing a name, the name of a Message Header context and the method messageHeaderGet() will return the string value associated with the name. You can store a value by providing a name, the name of a Message Header context and string value of the method messageHeaderPut(). If you wish to remove a name then pass that name and the name of a Message Header context to the messageHeaderRemove() method.

The Connect Framework has two predefined Message Header Context each with a set of predefined names. The names of the two predefined Message Header Contexts are "MessageHeader" and "InternalHeader". The "MessageHeader" message header context contains name/value pairs that are sent between the Gateway and the Flow Manager. The "InternalHeader" message header context contains name/value pairs that are scoped only to either the Gateway or the Flow Manage. The set of predefined name/value keys for each pre-defined message header context are listed in the Appendix.

New names can be added to the a particular Message Header context by simply using the messageHeaderPut() method. This will create an entry in the particular Message Header context with the new name and value.

messageHeaderGet`(java.lang.String contextName, java.lang.String key)`
Retrieve a string from the message header specified by the context name

messageHeaderPut`(java.lang.String contextName, java.lang.String key, java.lang.String value)`
Store the value at the specified key in the message header specified by the context name.

messageHeaderRemove`(java.lang.String contextName, java.lang.String key)`
Remove a string with the specified key from the the message header specified by the context name.

A sample of getting and putting a value in the sendable message header data area:

```
public JavaConnectorResult run(ProgramConnectorParm connectorParm)
   {
         // The content request value from the header, key is  "com_ibm_connect_header_contentRequest"
         String messageType =
(String)connectorParm.messageHeaderGet("MessageHeader",HeaderConstants.CONTENT_REQUE
ST);
         System.out.println("The request value for this request is " + messageType);

         // replace the message type  value in the MessageHeader
         If (messageType.equals("InputProfileRequest") {

connectorParm.messageHeaderPut("MessageHeader",HeaderConstants.CONTENT_REQUEST,
"ProfileRequest");
         }
   }
```

3.4.2.4 FlowDataArea

The purpose of the Flow Data Area is to provide Java Connector Applications an area where they can put objects or data and then retrieve them in a subsequent instance of a Java Connector Application. This means that a Java Connector Application can store an object in this area and it will reside in that area for the duration of the flow. Then in a later step a Java Connector Application can retrieve that data. The flow engine will not do any processing of the object that the Java Connector Application stores and the Java Connector Applications are responsible for name space collisions on the keys. There is no mapping support for the value stored in the Flow Data Area. The Java Connector Applications can get access to the Flow Data Area by using the following methods on the ProgramConnectorParm object.

flowDataAreaGet(String)
Return the object from the flow data area
flowDataAreaPut(String, Object)
Store an object in the flow data area at the specified key
flowDataAreaRemove(String)
Remove the object from the flow data area with the specified key

The semantics is that an object that is retrieved from the Flow Data Area and modified, the modification will not be reflected in the FlowDataArea until that object is put into the Flow Data Area. The scope of the Flow Data Area is the process flow. This means that each instance of a flow has a new Flow Data Area and the Flow Data Area is not sent between the Gateway and the Flow Manager.

A sample of getting and putting a value in the flow data area:

```
JavaConnectorResult run(ProgramConnectorParm connectorParm)

   {
         // Get the input raw message from the flowDataARea
         byte[]  input_raw_msg = (byte[])
parms.flowDataAreaGet(GatewayConstants.INPUT_BYTES_ARRAY);

         // Set the DTD name in the flowDataArea
         parms.flowDataAreaPut(GatewayConstants.REMOTE_DTD_NAME, “cXML.dtd”);

   }
```

3.4.2.5 Error handling and other Miscellaneous APIs.

The Connect flow engine has the ability to do error processing for Connector steps that fail. In developing a flow, a flow designer has the ability to point to an alternative next step to take if the step returns an error while processing a step. If after processing a step, the flow engine determines an error occurred, then it will invoke the step that is defined as the error step. If none is defined then the flow is terminiated. To indicate to the flow engine that an error has occur in a connector step, the application must return a non-zero return code. Once the flow engine determines that the connector step set the error code (StepApplicationErrorCode), it will store these values and make them available to the error step via the following set of methods. These values are also available via mapping of the InternalHeader message header context.

This set of methods provide convenience in getting information from the flow engine. Many of the methods will be used in error steps to determine which step failed and how did that step fail.

getAPPErrorInfoCode()
    Return the Application error code that was set in the previous step.
getAPPErrorInfoString()
    Return the Application error string that was set in the previous step.
getCurrentStepName()
    Return the current step name
getFailedStepName()
    Return the failed step name
getFlowErrorInfoCode()
    Return the Flow error code
getFlowErrorInfoString()
    Return the Flow error string
getUniqueId()
    Get the unique identifier for this request message
isInErrorPath()
    Return true if this flow hit an error that forced it into the error path.
isRestart()
    Return true if this flow was being restarted.

Sample using the methods

```
int appErrorCode = 0;
int flowErrorCode = 0;
flowErrorCode = parms.getFlowErrorInfoCode();
if (flowErrorCode != 0) {
        // Something failed
        if (flowErrorCode == FlowManagerCodes.ERROR_JAVA_CONNECTOR) {
                // The failure is set by the java connectors
                appErrorCode = parms.getAPPErrorInfoCode();
                int errorCode = 500;
                String errorText = "Internal Server Error";
                switch (appErrorCode) {
                        case
GatewayConstants.ERROR_GW_INBOUND_MESSAGE_PARSING_FAILED: {
                                errorCode = 400;
                                errorText = "Bad Request";
                                break;
                        }
                        case GatewayConstants.ERROR_GW_AUTHENTICATION_FAILED: {
                                errorCode = 401;
                                errorText = "Unauthorized";
                                break;
                        }
                        case GatewayConstants.ERROR_GW_AUTHORIZATION_FAILED: {
                                errorCode = 403;
```

```
                                        errorText = "Forbidden";
                                        break;
                              }
                    }
                    System.out.println( "Error origin is Connector, code is " + appErrorCode + ", error
text is  "+parms.getAPPErrorInfoString());
            }
```

3.5 Logging Interfaces

User-written Java code is allowed to log trace and message information into the same trace and message files used by iSeries Connect. Logging functions for user-written Java programs are provided by the Java class, `UserLogManager`. The `UserLogManager` class will provide methods for tracing and sending messages. There is one instance of the `UserLogManager` class per JVM. The `UserLogManager` instance is created and initialized when iSeries Connect logging support is started.

3.5.1 User Tracing

User tracing is controlled by the same iSeries Connect configuration tool tracing controls that are used for the Delivery Gateway and the Flow manager. That is, if tracing is active for the Delivery Gateway or the Flow manager, then user tracing is active for any user-written software running in the same JVM.

User trace entries are logged to the same files used by the Delivery Gateway and the Flow manager. The trace files for the Delivery Gateway and the Flow manager are written to /qibm/userdata/connect200/Commerce/<instance name>/Logs. The trace files are named GW_Trace1.log to GW_Trace*n*.log for the Delivery Gateway and FM_Trace1.log to FM_Trace*n*.log for the Flow manager, where *n* is the number of trace file configure on the iSeries Connect configuration tool tracing screen.

The following UserLogManager trace methods will be provided:

setTracing(boolean)
    Sets the static tracing indicator that is used to guard tracing in the user-written application code.
trace(Object, String, byte[])
    Traces an array of bytes.
trace(Object, String, String)
    Traces a text string.
trace(Object, String, Throwable)
    Traces an exception.

Each method includes in its parameters the name of the class and method which is logging the trace point. In all cases, the Object loggingClass parameter may be a String naming the logging class. In instance (non-static) methods, loggingClass may be *this* (the logging object). Using *this* is a convenience to the programmer, as the class name can be derived from any Object. In static methods, an object does not exist, so *this* cannot be used. This use of the *this* pointer versus a String also applies to the message methods in the next section. In additional to the parameters on the trace methods, each trace entry also has the date, timestamp, and the name of the current thread.

This is a sample trace point:

```
public class MyClass {
   public void myMethod {
         /* Note that the "this" pointer is used for the logging class */
         UserLogManager.trace(this, "myMethod", "This is a trace example");
   }
}
```

This is the resulting trace file entry:

```
2001.03.07 10:15:04.769 com.ibm.connect.MyClass myMethod Thread-0 This is a trace
example
```

To minimize the performance impact when tracing is inactive, the UserLogManager class includes a static Boolean that can be used to check whether trace is active.

isTracing

Boolean used to guard tracing in calling code.

This Boolean is set true when iSeries Connect tracing is active, and false when it is not. For example:

```
if (UserLogManager.isTracing)
   UserLogManager.trace(...);
```

3.5.2 User Messages

Messages logged by user-written Java applications will be written to the same locations as iSeries Connect Messages.

- B2B Instance Message Queue. A message queue is created for each instance, and is located in the instance library. The message queue name and library name are the same as the instance name.
- B2B Instance Message File. The same messages that are written to the B2B instance message queue are also written to an integrated file system file named FM_Messages_<date>.log in the /qibm/userdata/connect200/Commerce/<instance name>/Logs directory or GW_Messages_<date>.log in the /qibm/userdata/connect200/Commerce/<instance name>/Logs directory. The message file name has a date and time stamp, and is switched at midnight.

The following methods will be provided to allow users to log 3 types of messages: Informational, Warning, and Error.

logErrorMessage(Object, String, String)

Logs an error message.

logInfoMessage(Object, String, String)

Logs an informational message.

logWarningMessage(Object, String, String)

Logs a warning message

If translation of messages into different languages is desired, the user-written application must perform its own translation prior to passing the text into the `UserLogManager` message methods.

This is a sample of the logging of a message:

```
public class MyClass {
   public static void myStaticMethod {
          /* Note that a string is used since there is no "this" pointer */
          UserLogManager.logsendInfoMessage("MyClass", "myStaticMethod", "This
is a                          message example");
   }
}
```

This is the resulting IFS message file entry:

```
2001.03.07 10:15:04.769 MyClass myStaticMethod Thread-0 This is a message example.
```

Each message logged to the IFS file is also mapped to one of the following AS/400 messages and sent to the instance message queue.

3.5.2.1 Message CON0350 - Information Message

```
:MSGL.
:SUBTYPE.I
:CCSID.*JOB
:USER.PGMR
:SEVERITY.00
:LOGPRB.No
:ALRTFLG.*NO
:ALRTIDX.
:MSG.iSeries Connect user application information:  &1 &2
:EMSG.
:XPL.Additional information:  &3
:EXPL.
:URESP.This information is from an iSeries Connect user-written application.
:EURESP.
:PRESP.
:EPRESP.
:GROUP IDF='REPLVAR' LEN='3 32 20' COLHDR1='Symbol' COLHDR2='Description'
COLHDR3='Format'.
:ITEM1.&1
:ITEM2.Class name
:ITEM3.*CHAR *VARY 2
:ITEM1.&2
:ITEM2.Method name
:ITEM3.*CHAR *VARY 2
:ITEM1.&23
:ITEM2.Message text
:ITEM3.*CHAR *VARY 2
:EGROUP.
:PROGNOTE.This message can be sent by an iSeries Connect user-written
application.
:EPROGNOTE.
:TRANNOTE.Do not translate iSeries Connect.
:ETRANNOTE.
:EMSGL.
```

3.5.2.2 Message CON0351 - Warning Message

```
:MSGL.
```

```
:SUBTYPE.I
:CCSID.*JOB
:USER.PGMR
:SEVERITY.10
:LOGPRB.No
:ALRTFLG.*NO
:ALRTIDX.
:MSG.iSeries Connect user application warning:  &1 &2
:EMSG.
:XPL.Additional warning information:  &3
:EXPL.
:URESP.This warning information is from an iSeries Connect user-written
application.
:EURESP.
:PRESP.
:EPRESP.
:GROUP IDF='REPLVAR' LEN='3 32 20' COLHDR1='Symbol' COLHDR2='Description'
COLHDR3='Format'.
:ITEM1.&1
:ITEM2.Class name
:ITEM3.*CHAR *VARY 2
:ITEM1.&2
:ITEM2.Method name
:ITEM3.*CHAR *VARY 2
:ITEM1.&23
:ITEM2.Message text
:ITEM3.*CHAR *VARY 2
:EGROUP.
:PROGNOTE.This message can be sent by an iSeries Connect user-written
application.
:EPROGNOTE.
:TRANNOTE.Do not translate iSeries Connect.
:ETRANNOTE.
:EMSGL.
```

3.5.2.3 Message CON0352 - Error Message

```
:MSGL.
:SUBTYPE.I
:CCSID.*JOB
:USER.PGMR
:SEVERITY.20
:LOGPRB.No
:ALRTFLG.*NO
:ALRTIDX.
:MSG.iSeries Connect user application error:  &1 &2
:EMSG.
:XPL.Additional error information:  &3
:EXPL.
:URESP.This error information is from an iSeries Connect user-written
application.
:EURESP.
:PRESP.
:EPRESP.
:GROUP IDF='REPLVAR' LEN='3 32 20' COLHDR1='Symbol' COLHDR2='Description'
COLHDR3='Format'.
:ITEM1.&1
:ITEM2.Class name
:ITEM3.*CHAR *VARY 2
:ITEM1.&2
:ITEM2.Method name
```

```
:ITEM3.*CHAR *VARY 2
:ITEM1.&23
:ITEM2.Message text
:ITEM3.*CHAR *VARY 2
:EGROUP.
:PROGNOTE.This message can be sent by an iSeries Connect user-written
application.
:EPROGNOTE.
:TRANNOTE.Do not translate iSeries Connect.
:ETRANNOTE.
 :EMSGL
```

4.0 XML Protocol Customization

4.1.1 Delivery Gateway Foundation Overview

At a very high-level, the Connect Delivery Gateway can be thought of as the gatekeeper between partners (either end users or applications) which want to communicate with back-end applications, via some networking technology (either internet or intranet). There has not been a single protocol which has been widely adopted and used by companies for carrying out their business over the net. For the time being, there are several protocols for carrying out commerce related transactions, and there are also a number of protocols which various companies or business sectors have developed outside of any standards bodies. Since the first release of the Connect product, implementations for the cXML protocol, which provides supplier enablement to communicate with partners which have adopted procurement solutions from Ariba, have been provided.

The Gateway handles all aspects of the XML request from receiving the request to sending the response. The Gateway's design is very similar to the Flow Manager in that they share a common Flow Engine. Both use a series of connectors to process the XML request. The Gateway is implemented as a set of servlets that run in a servlet engine. For Connect 2.0, Websphere Application Server is the only servlet engine which the Gateway will support.

The Gateway also provides support for allowing applications to originate messages and send them to business partners. This support provides a mechanism for providing true peer-to-peer capabilities between business partners. This support is provided by a function within the Gateway called the Outbound Message Handler (OMH), and will be explained in detail later in this document.

4.1.2 An overview of defining a new protocol

Tools are provided to create and edit custom protocols in Connect. The protocol tools present a set of menu options for guiding you through creating a custom protocol from start to finish. In this section we'll go through the main menu of items and explain their use and meaning. This will give you an overview of the capabilities of the Gateway and the process to follow to write a custom protocol. As you read through this section, it might be helpful to create a temporary protocol and follow along in the screens as each item is described. The later sections, will describe the Gateway's use and interfaces in much more detail.

After creating a new protocol instance and giving it a name the tools will present to you the following list of options to edit;

- Protocol Information
- Requests
- Contexts
- Protocol Data
- Protocol Data Presentation
- Application Connectors
- Request Groups
- Protocol Classpaths

The order of these items suggests the order in which the information for the protocol should be specified. The sections below will give an introduction to each item to be completed. In the last section we'll describe how to publish a protocol once you're finished developing it.

4.1.2.1 Protocol Information

In this section you can give a description of the protocol. The description can be entered directly into the description field or it can be optionally contained in a resource bundle and referenced from the description field by entering the resource bundle key. The description is optional, and a brief description is sufficient.

4.1.2.2 Requests

This is where the requests for the protocol are defined. For each request in your protocol, you'll enter the name of the request and optionally give it a description.

4.1.2.3 Contexts

In this section you'll define the data contexts that the protocol will use. A data context stores information for mapping between connectors. The contexts can be used in just the Gateway or just the Flow Manager or shared between the two.

There are three different types of contexts for storing different types of data. First is the DOM context type for storing XML data. Next is the Header type for storing name value pairs (or Properties type data). Finally, there is the Intermediate type for storing structured data. That is, for storing repeating fields (arrays), and structures (data records). The Intermediate context type is very similar in functionality to the DOM type but it is designed to be more generic than just XML data. See the section on Contexts for more information.

If a DOM type of context is chosen, you can specify the level of validation that should take place on the XML document when it stored in the context. A full description of the values and the meaning of validation values is given in the more detailed sections below.

Once the Context type is set you can choose whether it is shared with the Flow Manager, whether it should be returned to the Gateway (containing any modifications the Flow Manager process might have made) and the level of access each has to the context.

Contexts store data to be mapped by the flow processor but before we can use them for this purpose, we first must define the fields that are in the context. To define the fields we use a "Request Message Format" file (RMF file for short). The name originated from the DOM contexts that were first supported in Connect. Multiple RMF files can be used to define the fields for a context and the RMF files can be scoped at different levels. That is, an RMF file can be specified to be in effect for the entire protocol, or for specific requests. If an RMF is specified at the protocol level that means all the fields defined by the RMF can be mapped during any request in the protocol. If an RMF is specified at the request level, then those fields are only visible for mapping by the protocol flow that is handling that request. We'll explain protocol flows in a section below. This scoping is very useful when your contexts are storing both common and request specific fields. For instance, if all of the requests in your protocol contain the sender and recipient's address, those fields could be stored in the protocol level RMF. Whereas, the body of the request (a purchase order for instance) could have it's fields specified in a request specific RMF. Requests can actually have two RMFs specified. One that is used when the request handled in the normal way, and one that is used when a "rebound" message is received. We'll explain what a rebound flow or message is later, but briefly, it's when a message is returned to Connect in response to a message that was sent to a remote trading partner.

Request Message Format files can be created from either a Document Type Definition (DTD), XML, ACDFieldSet or PCML file. It can also can be created from scratch, without an input source. Once the RMF is created the fields defined in the RMF can be edited, added, or deleted through the tools.

You'll notice DOM contexts have three additional fields added to the RMF when it is being edited or when mapping to or from the contexts. These fields are #XMLByteArray, #ValidationLevel, and #XMLString. They are used to set and retrieve the entire contents of the context. The value mapped into #ValidationLevel controls the amount of validation done on the XML when the contents of the context is set.

4.1.2.4 Protocol Data

This section allows you to define data fields that the protocol requires to correctly process the protocol, or a request in the protocol. Later, in the next section, you can specify how these data fields should be presented in the protocol, provider, and partner configuration screens.

By utilizing the protocol data field capabilities of Connect you can design powerful and flexible protocol implementations. For instance, you can create a protocol data field that the Connect administrator must set for each trading partner configured. This is how you would implement a password authentication model in a protocol. You would create a user identification and password data field that the administrator would set when the partner is configured. Furthermore, you can create data fields that are configured at the protocol level but can be overridden for particular partners or providers.

Each field that is defined is available for mapping in the Gateway's flow processor. These fields are available as "read-only" to the process flows that run in the Flow Manager.

4.1.2.5 Protocol Data Presentation

This section allows you define how the protocol data is presented to the user. For each field that was defined through the protocol data screens you can define the type of input field that is presented, default values, it's size, range checking, a description of the field, and whether it is required to be set.

Once all the field's presentation is setup, you can organize the fields into pages, and define whether those pages should be presented during protocol, partner or provider configuration.

4.1.2.6 Application Connectors

This section allows you to define new application connectors to be used in the protocol flows of the protocol. An application connector defines the input and output fields that are expected by the connector, as well as the class name of the connector itself. Although the Flow Manager supports a variety of connector types, the Gateway only supports Java connectors.

If you are writing custom java code to implement the protocol, you would create an application connector for each Java class you expect to call in the protocol flow.

4.1.2.7 Request Groups

In this section, you create groups of requests that will be implemented as a single protocol flow. There are some important things to consider when deciding what requests should be grouped together. First, it is a convenient way for requests that might have similar semantics to be processed by a single protocol flow. For instance, if all your request have the same authentication and authorization steps, maybe they should be grouped together. Then the protocol flow would be common for all requests and could simply branch out to request specific steps once the authentication and authorization are completed.

The second thing to note about request groups is that each group gets its own unique servlet URL for receiving the requests. Since the Gateway protocol flow processor does not inspect the incoming data, the only way it has to know which protocol flow should be run is to have each group have a unique URL. This is important when you consider how your trading partner is going to interface with your protocol implementation. If the trading partners software is not flexible enough to configure different URLs for each request, you will be forced to group all requests together.

Once a strategy for the grouping of requests is complete, you can set whether this group is for inbound or outbound requests. Outbound requests are described later in this document. If this is an inbound request group you can set how the data is to be received and passed to the protocol flow. Also in this section, is the screen where you can define which requests are in this group. You are presented with a list of all the requests and you simple check off any of the requests that you want to be in this group, that are not already in a different group. A request can only be a member of one group.

Finally, you can specify and edit the protocol flows that should be used for this group. Two protocol flows are available to be set. One to use when data is received in the normal manner (called the “normal” flow), and one for when data is received back from the trading partner after calling the SendMessage connector (called the “rebound” flow.)

4.1.2.8 Protocol Classpath

This section allows you to add directories, jar and zip files to be used by the Gateway. Any directories that are specified are added to the classpath of the JVM that the Gateway runs in. Jar files are added to the J2EE War file. These two classpaths use different class loaders and can present some problems. Any classes in the jar files will be able to call classes found in the directories, but the opposite is not true. Classes found in directories cannot call classes in Jar files, because the JVM class loader does not know about the class loader used by WebSphere.

If you’re creating custom Java connectors to implement your protocol, or any Java code that references classes supplied by Connect, you must package them in a uniquely named Jar file, copy that Jar file to the directory where the protocol files are kept (/QIBM/Userdata/Connect200/Protocols/MyProtocol) and use this section to point to that Jar file.

You can add directories to point to Java class files, but these class file must not be dependent upon any classes in the jar or zip files.

4.1.2.9 Publishing a protocol

Once the protocol is finished you can publish it by returning to the main “Protocols” page and selecting the Publish button. Publishing a protocol will take a snap shot of the protocol and write it into the /QIBM/Userdata/Connect200/Commerce/Protocols directory. Once there you can use the “Manage User-Defined Protocols” screen to add support for the protocol into Connect. If you wanted to use this protocol on a different system you can copy (by using FTP or some other means) the contents of the protocols directory in QIBM/Userdata/Connect200/Commerce/Protocols to the other system and use the “Manage User-Defined Protocols” screen on the other system to add support for the protocol there. When you add support for a protocol, Connect will find all remote systems used for gateway processing of instances on this system and automatically copy and add support for the protocol to those remote systems. When you create an instance to a new gateway system, the current set of supported protocols will be copied to the new system

That completes the overview of the steps and screens that it takes to create a custom Gateway protocol. The sections below give a more detailed explanation of Gateway features and topics.

4.1.3 Gateway data contexts

This section will describe the data contexts used by the Delivery Gateway. These data contexts are used to store information required to process requests within the Connect framework. There are different types of contexts used within the Delivery Gateway.

The protocol designer can use the mapping technology built into the flow engine or programmable access to modify or access fields in most of these data contexts. When describing these fields, there is a 'constant' name as well as the 'Mapping' name of the field. The constant name is what the protocol designer must use when accessing a field using programmable access. The mapping name is the name of the field as it will be displayed when accessing fields using mapping technology. Both of these names will be provided for completeness.

There is also a column in the tables called usage. This column will provide information describing which portion of the Gateway or Flow Manager sets or uses the field. The legend for this column is as follows:

Key	Description
SetBy	Indicates which component sets a particular field. Note that if a field indicates that it is set by PROT, and that it is used by FM or DGI, then the protocol designer must ensure this field gets set during the running of the protocol flow.
UsedBy	Indicates that this field is used by and possibly required by some component within Connect in order to perform its function.
DGAccess	Contains values of None, Read, or R/W This field indicates how protocol flows in the Gateway can access this field via mapping.
FMAccess	Contains values of None, Read, or R/W This field indicates how process flows in Flow Manager can access this field via mapping.
DGI	Delivery Gateway Infrastructure - value used in SetBy and UsedBy. If DGI is specified, then this indicates the DGI sets or uses the field to perform its function.
FM	Flow Manager - value used in SetBy and UsedBy
FENG	Flow Engine - This is the common flow engine support which provides the underlying functions required for either protocol flows or process flows to run.
PROT	Protocol Flow - value used in SetBy and UsedBy
PROC	Process Flow - value used in SetBy and UsedBy

The purpose of the usage column will be to specify how various fields are intended to be used and to explicitly spell out fields set by the Delivery Gateway and Flow Manager. Even if fields are not explicitly called out as SetBy or UsedBy, protocol flow or process flow, as long as the process flow or protocol flow has access to a particular field, it can use that field for whatever purpose it needs to fulfill its task.

When discussing the fields in the data contexts, it mentions whether a field is set or used by the Delivery Gateway (DG) infrastructure. This means that the field is directly used or modified by the DG infrastructure itself (e.g. The code which processes requests prior to invoking the protocol flows, or after control has been returned to the DG infrastructure after having run a protocol flow.) Many additional fields are set/used within these data contexts in the implementation of the protocols.

4.1.3.1 MessageHeader

This context contains a set of information that is sent back and forth between the Gateway and the Flow Manager. Some of the fields in this header are eventually used by the Flow Manager for performing flow selection, thus it becomes the responsibility of the protocol flow designer to ensure these fields get set prior to passing the request over to the Flow Manager over the MQ series queue. This data context supports both programmable and mappable access.

NOTE: The MessageHeader and InternalHeader are describing header type data contexts which are used by the Connect infrastructure and these context names are reserved by Connect. Protocol Flow designers can define additional Header type contexts (and specify whether those headers are available to both Gateway and FlowManager or if the header is available only to Gateway or to FlowManager).

Connect Infrastructure Message Header section

FIELD NAME (CONSTANT)	FIELD NAME (MAPPING)	USAGE	DESCRIPTION
GATEWAY_INSTANCE	Gateway Instance	SetBy - DGI UsedBy - FM DGAccess - Read FMAccess - Read	Connect instance name.
PROTOCOL_GROUP	Protocol Group	SetBy - DGI UsedBy - FM DGAccess - Read FMAccess - Read	ProtocolGroup (formerly marketplace name).
PROTOCOL	Protocol	SetBy - DGI UsedBy - FM DGAccess - Read FMAccess - Read	This field is used to hold the data that uniquely identifies the protocol being used.
REQUEST	Request	SetBy - PROT UsedBy - FM DGAccess - R/W FMAccess - Read	The request.
AUDIT_RECEIVED_TIMESTAMP	Audit Received Timestamp	SetBy - DGI UsedBy - DGI DGAccess - Read FMAccess - Read	This field is used to record when the message being processed by the Delivery Gateway was received.
AUDIT_UNIQUE_ID	Audit Unique ID	SetBy - DGI UsedBy - DGI, FM, PROT DGAccess - Read FMAccess - Read	This field contains a unique ID for the request being processed. This allows the various audit records which get logged due to this single request to be correlated with each other. Refer to the Gateway Auditing section

			for a description of how this field is used.
REQUEST_TOKEN	Request Token	SetBy - PROT UsedBy - PROC DGAccess - R/W FMAccess - Read	This field contains a token which can be used to represent the key parameters used for identification. The instance name, protocol group, protocol, ProviderId, and PartnerId are considered in this unique identification. This token can be used in conjunction with the OMH support and can be generated by using the RequestToken utility connector provided by the Delivery Gateway support.
FLOW_INDEX_AUDIT_POINT_INDEX	Flow Index and Audit Point Index	SetBy - DGI UsedBy - DGI, FM DGAccess - Read FMAccess - Read	This field is used for audit logging purposes by the Delivery Gateway and FlowManager infrastructure components. Protocol Flow designers should not need to set or use this parameter. For more information on how DG and FM use this field refer to the Gateway Auditing Changes section of this document.
APPLICATION_TOKEN	Application Token	SetBy - DGI UsedBy - PROT DGAccess - RW FMAccess - RW	Provides a way to pass data between the application and a process flow. The value is set by the delivery gateway from the data passed in on the OutboundRequest API.

Common Protocol Enablement Message Header section

FIELD NAME (CONSTANT)	FIELD NAME (MAPPING)	USAGE	DESCRIPTION
INBOUND_TRANSPORT_URL	Inbound Transport URL	SetBy - PROT UsedBy - PROC DGAccess - R/W FMAccess - Read	This field can be used by a protocol flow designer to provide the URL that a back-end application can send subsequent/related requests into the

			Delivery Gateway. This is used in the Connect cXML implementation to notify the back-end application of the NewQuote URL which can be invoked after completing the B2B shopping experience.
OUTBOUND_TRANSPORT_URL	Outbound Transport URL	SetBy - PROC UsedBy - PROT DGAccess - R/W FMAccess - R/W	This field can be used by a protocol flow designer to allow a re-direct of the URL. For example, a protocol flow designer can allow a back-end application to provide a URL to redirect to as opposed to simply sending the response back directly to the partner that sent in the original request. This can be useful in commerce type applications where a redirect is needed to the URL which must be invoked to initiate a B2B shopping experience.
PARTNER_SESSION_ID	Partner Session ID	SetBy - PROT UsedBy - PROC DGAccess - R/W FMAccess - Read	This field contains an identifier which the partner provides to identify the transaction as it pertains to their own application.
PARTNER_TRANSPORT_MSG_ID	Partner Transport Message ID	SetBy - PROT UsedBy - PROC DGAccess - R/W FMAccess - Read	This field contains the message identifier that the partner system has placed within the received request. If the partner system ever resubmits the same message with the same message ID, this field can be used by a back-end application to identify

			this condition and act appropriately.
AUDIT_SENT_TIMESTAMP	Audit Sent Timestamp	SetBy - PROT UsedBy - DGAccess - R/W FMAccess - Read	This field is used by protocol flow designers to log the timestamp of when the request was sent by the partner system. Some protocols provide this information in the XML messages which are sent between systems. This is an optional field for protocol flow designers to set. If set, this field along with the rest of the MessageHeader will appear in the audit log.
MESSAGE_ID	Message ID	SetBy - PROT UsedBy - PROC DGAccess - R/W FMAccess - Read	This field contains the message identifier that the partner system has placed within the received request. If the partner system ever resubmits the same message with the same message ID, this field can be used by a back-end application to identify this condition and act appropriately. This field contains the same information as the PARTNER_TRANSPORT_MSG_ID field, but is being left in to accomodate process flows which may have used either field.
REQUISITIONER_ID	Requisitioner ID	SetBy - PROT UsedBy - PROC DGAccess - R/W FMAccess - Read	.This field contains the identifier of the requisitioner within the partner organization. This can be used to allow back-end

			applications to differentiate between requisitioners that belong to the same partner organization. This field is optional and would only need to be filled in by protocol if the process flows require it.
PARTNER_ORG_NAME	Partner Organization Name	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	This field contains the unique name of the partner organization. A unique name may consist of several distinct subfields. If the name consists of multiple subfields, the subfields must be separated with a colon ":" character. For example, in the case of cXML, the fields used for identification are the domain and ID. If domain were "DUNS" and ID was "123456789" , then this field must contain "DUNS:123456789".
PROVIDER_ORG_NAME	Provider Organization Name	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	This field contains the unique name of the provider organization. A unique name may consist of several distinct subfields. If the name consists of multiple subfields, the subfields should be separated with a colon ":" character. For example, in the case of cXML, the fields used for identification are the domain and ID. If domain were "DUNS" and ID was "555666777" , then this field should contain "DUNS:555666777".

			This field is not set by the DG infrastructure.

CXML Extensions (protocol specific) to the MessageHeader

The following table shows the extensions that have been provided to the Message Header in support of the cXML protocol implementation that is shipped with Connect. None of these fields are set or used by the DG infrastructure, but are set by the cXML protocol implementation in the Delivery Gateway. The majority of these fields are copied from the cXML header information by the DG cXML protocol implementation.

FIELD NAME (CONSTANT)	FIELD NAME (ACTUAL)	USAGE	DESCRIPTION
AGENT_ORG_NAME	Agent Org Name	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	Agent organization name. The cXML implementation sets this to a concatenation of the AGENT_ORG_ID and AGENT_ORG_DOMAIN using a colon ":" as the separator character.
AGENT_ORG_ID	Agent Org ID	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	Agent organization ID. This this will contain the partner ID.
AGENT_ORG_DOMAIN	Agent Org Domain	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	Agent organization domain. This will contain the partner domain.
AGENT_USERID	Agent Userid	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	Agent userid. This will contain the provider ID.
AGENT_DOMAIN	Agent Domain	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	Agent domain. This will contain the provider domain.

AGENT_AUTH_TYPE	Agent Auth Type	SetBy - PROT UsedBy - PROT DGAccess - R/W FMAccess - Read	Agent authentication type. For cXML this will be set to “PW” to indicate that authentication is handled via userid and password verification.
PARTNER_ORG_ID	Partner Organization ID	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	The partner (formerly buyer) organization ID. This parameter is typically used in the identification of the partner node. For example “123456789” which represents a DUNS number of the partner.
PARTNER_ORG_DOMAIN	Partner Organization Domain	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	The partner organization domain. For example “DUNS” . Used to qualify what type of information the PARTNER_ORG_ID represents.
PROVIDER_ORG_ID	Provider Organization ID	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	The provider ID (formerly supplier organization ID). This parameter is typically used to identify the provider on the local system. For example “555666777” which represents a DUNS number of the provider.
PROVIDER_ORG_DOMAIN	Provider Organization Domain	SetBy - PROT UsedBy - PROT, PROC DGAccess - R/W FMAccess - Read	The provider organization domain. For example “DUNS” . Used to qualify what type of information the PROVIDER_ORG_ID represents.

Retired Fields

Due to some restructuring of the Connect product, fields used to identify protocol subtype, protocol version, and request type have been retired. The protocol information is now contained within the single field, PROTOCOL. The request information is now contained within the single field, REQUEST. Also, some unused fields have been retired as well. The retired fields from the MessageHeader are:

Retired Field	Comments
PROTOCOL_SUB_TYPE	Consolidated into PROTOCOL field
PROTOCOL_VERSION	Consolidated into PROTOCOL field
CONTENT_REQUEST_TYPE	Consolidated into CONTENT_REQUEST
BUYER_ORG_TOKEN	Replaced by the PARTNER_ORG_REFNUM field of the Partner data context.

SUPPLIER_ORG_TOKEN	Replaced by the PROVIDER_ORG_REFNUM field of the Provider data context.
AUDIT_REQUEST_TYPE	Unused
REQUISITIONER_NAME	Unused
AUDIT_BUYER	Unused
AUDIT_SUPPLIER	Unused
AUDIT_PROTOCOL	Unused
AUDIT_MARKETPLACE	Unused
AUDIT_REQUEST	Unused
BUYER_DEPT_KEY	Unused
BUYER_DEPT_TYPE	Unused
AGENT_ORG_TOKEN	Unused
AGENT_PASSWORD	Moved to PASSWORD in NonSendableMessageHeader

Deprecated Fields

Due to some renaming of various entities within the Connect product, the following fields have been deprecated. The internal constant values for these fields still exist. The following chart shows the name of the field that is deprecated, and the name of the field which replaced it.

Deprecated Field	Replacement Field
MARKET_PLACE	PROTOCOL_GROUP
BUYER_ORG_ID	PARTNER_ORG_ID
BUYER_ORG_DOMAIN	PARTNER_ORG_DOMAIN
SUPPLIER_ORG_ID	PROVIDER_ORG_ID
SUPPLIER_ORG_DOMAIN	PROVIDER_ORG_DOMAIN
BUYER_SESSION_ID	PARTNER_SESSION_ID
BUYER_TRANPORT_MSG_ID	PARTNER_TRANSPORT_MSG_ID
PROTOCOL_TYPE	PROTOCOL
CONTENT_REQUEST	REQUEST

4.1.3.2 InternalHeader

This context contains a set of information that is only used within the Delivery Gateway. This context is available for both programmable access and mapping.

FIELD NAME (CONSTANT)	FIELD NAME (ACTUAL)	USAGE	DESCRIPTION
GATEWAY_TYPE	Gateway Type	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field is used to represent the Gateway type. This field will be set to a value of “Connect” by the DG infrastructure.

GATEWAY_VERSION	Gateway Version	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field is used to represent the version of the Gateway that is running. This field is set to “2.0.0” by the DG infrastructure in Connect 2.0.
TRACE_ENABLED	Trace Enabled	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates whether logging to the trace files has been enabled for this instance. Valid values are “true” or “false”. This parameter is set by the DG infrastructure by copying the value configured in the instance registry. Modifying this parameter in the InternalHeader has no effect on whether or not tracing is performed.
MAX_TRACE_FILE_SIZE	Max Trace File Size	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates the maximum trace file size for the trace files associated with this Gateway instance. This parameter is set by the DG infrastructure by copying the value configured in the instance registry. Modifying this parameter in the InternalHeader has no effect.
MAX_NUMBER_OF_TRACE_FILES	Max Number of Trace Files	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates the maximum number of trace files available for use by this Gateway instance. This parameter is set by the DG infrastructure by copying the value configured in the instance registry. Modifying this parameter in the InternalHeader has no effect.
MSG_QUEUE_TIMEOUT	Message Queue Timeout	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates the number of seconds that the Delivery Gateway will wait for a response from the Flow Manager before considering the message

			as timed out and returning control back to the protocol flow. This value is copied from the Gateway properties in the instance registry.
PRODDATA_PATH	ProdData Path	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field contains the ProdData directory path for the Connect instance which is currently processing this request. For Connect 2.0, this will contain the value "/QIBM/ProdData/Conne ct200".
USERDATA_PATH	UserData Path	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field contains the UserData directory path for the Connect instance which is currently processing this request. For Connect 2.0, this will contain the value "/QIBM/UserData/Conne ct200".
FLOW_NAME	Flow Name	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates the protocol flow which will process the current request.
SERVLET_NAME	Servlet Name	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field provides the name of the servlet which was invoked to initiate the current request. This field will be set by the DG infrastructure when the HTTP transport is being used for inbound requests.
DEPLOY_DESTINA TION	Deploy Destination	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates the directory in which this protocol has been deployed in. For example, this field is set to cXML12Ariba when running the cXML 1.2 implementation which is shipped with the Connect product.

DEPLOY_TYPE	Deploy Type	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates whether this protocol has been deployed into “ProdData” or “UserData”.
DEPLOY_THEME	Deploy Theme	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field indicates which theme this protocol has been deployed into. For Connect 2.0, this field will contain “Commerce”.
DEPLOY_DIRECTORY	Deploy Directory	SetBy - DGI UsedBy - DGAccess - Read FMAccess - None	This field contains the fully qualified directory where this protocol has been deployed. For example, if the DEPLOY_DESTINATION field is set to “cXML12Ariba” , DEPLOY_TYPE field is set to “ProdData”, DEPLOY_THEME field is set to Commerce, then this field will contain: “/QIBM/ProdData/Connect200/Commerce/Gateway/Connectors/cXML12Ariba”.
REQUEST_METHOD	Request Method	SetBy - DGI UsedBy - DGI DGAccess - Read FMAccess - None	This indicates whether the byte stream received from the partner is a byte array or if it should be processed as name-value pairs, and converted into byte array containing an nvpML XML document. Valid values are POST or NVP.
PASSWORD	Password	SetBy - PROT UsedBy - PROT DGAccess - R/W FMAccess - None	This is the shared secret used between the provider and the partner (or between provider and the agent that is working on behalf of the partner).
VALIDATE_INPUT	Validate Input	SetBy - DGI UsedBy - DGAccess - Read FMAccess - Read	This field indicates whether validation of inbound XML documents

			should be performed. This field is copied from the Gateway properties of the instance registry by DG infrastructure. This parameter could be used in conjunction with the XMLValidator connector described later.
VALIDATE_OUTPUT	Validate Output	SetBy - DGI UsedBy - DGAccess - Read FMAccess - Read	This field indicates whether validation of outbound XML documents should be performed. This field is copied from the Gateway properties of the instance registry by DG infrastructure. This parameter could be used in conjunction with the XMLValidator connector described later

Retired Fields

New Transport contexts, one for input and one for output, are being used to consolidate the information involved in processing transport related data in protocol flows. Due to this restructuring, the following fields are being retired from the InternalHeader. The fields being retired are:

Retired Field	Comments
INBOUND_TRANSPORT_CONTENT_TYPE	Information accessible via the TransportInput context
INBOUND_TRANSPORT_CONTENT_ENCODING	Information accessible via the TransportInput context
INBOUND_TRANSPORT_SENDER_INFO	Information accessible via the TransportInput context
INBOUND_TRANSPORT_URL_NAME	Unused
OUTBOUND_TRANSPORT_CONTENT_TYPE	Information accessible via the TransportOutput context
OUTBOUND_TRANSPORT_CONTENT_ENCODING	Information accessible via the TransportOutput context
OUTBOUND_TRANSPORT_SENDER_INFO	Information accessible via the TransportOutput context
OUTBOUND_TRANSPORT_URL_NAME	Unused

4.1.3.3 TransportInput

The TransportInput context holds the request data that was received. The recieved data is stored in the TransportInput according to the Request Group settings. No programable access is provided for these fields. Protocol flow designers will need to access this data context using mapping technology.

The context contains the following fields:

Name	Type	Description
Headers	repeating String	Contains the values of the main headers for the received request. Each String would appear like the following: `<headerName>: <headerValue>` This field will be set if Type specifies byteArray -- it may be set if Type specifies MIME
Type	String	Contains one of the following values: • *byteArray* -- the ByteArray field contains only the body of a request • *MIME* -- the ByteArray field contains a complete MIME document, including the beginning headers
ByteArray	byte[]	Request contents

This context is never sent to the Flow Manager.

4.1.3.4 Transport Output

The TransportOutput context holds the data to be sent. The protocol implementation would use this context to control what data is sent and how it is to be sent to the trading partner. No programable access is provided for these fields. Protocol flow designers will need to access this data context using mapping technology.

The context contains the following fields:

Name	Type	Description
StatusCode	int	Status code that should be returned to the partner system. This field should contain an HTTP status code value.
Headers	repeating String	Contains the values of the main headers for the data to be sent. Each String would appear like the following: `<headerName>: <headerValue>` This field must be set if Type specifies byteArray -- it may be set if Type specifies MIME, but it will be ignored.
Type	String	Contains one of the following values: • *byteArray* -- the ByteArray field contains only the body of a request • *MIME* -- the ByteArray field contains a complete MIME document, including the beginning headers
ByteArray	byte[]	Response/acknowledgement contents

4.1.3.5 Provider

The Provider context holds business data which needs to be maintained on a provider basis. Protocol flows do not have access to this business data. It is only available to the Flow Manager. However a protocol implementation must set the #Provider_Refno field in order for the Flow Manager to have access to the data. The #Provider_Refno is the key field for this context and the Protocol context. The PartnerOrProviderResolver utility connector (described later) can be used to set the key field.

4.1.3.6 Partner

The Partner context holds business data which needs to be maintained on a partner basis. Protocol flows do not have access to this business data. It is only available to the Flow Manager. However a protocol implementation must set the #Partner_Refno field in order for the Flow Manager to have access to the data. The #Partner_Refno is the key field for this context and the Protocol context. The PartnerOrProviderResolver utility connector (described later) can be used to set the key field.

4.1.3.7 Protocol

The Protocol context holds protocol specific data which needs to be collected as part of the protocol configuration process. This data is available and intended to be used by the protocol implementation.

This context can contain three levels of information for each data field present. The levels are the protocol configuration, the provider, and the partner, in that order. The protocol designer can determine which levels are appropriate for the specific data being collected. The levels are from least specific (protocol) to most specific (partner). The level of information that is returned from a mapping is dependent on what key fields have been set in the Provider and Partner contexts. Generally, if neither the #Provider_Refno or #Partner_Refno have been set, then the Protocol level of data will be returned. If the #Provider_Refno has been set, but not the #Partner_Refno, then the provider level is returned. Finally, if the #Partner_Refno has been set then the partner level of information is returned. This assumes that data is present at all levels. If data is not available at the expected level then a lower level of data will be returned if it's available. That is, if the #Partner_Refno is set but the data is only available at the protocol level, then the protocol level of data is returned.

4.1.4 Defining the flow to process the protocol

4.1.4.1 Protocol Flows

Once a request is received by the Delivery Gateway, the flow engine is started and the correct protocol flow is initiated. A protocol flow is a sequence of one or more Gateway connectors and/or copy/decision steps. The Gateway Flow Engine only supports Java connectors.

A given request will cause a single flow to start. However, it should be noted that multiple requests can point to the same protocol flow. The design of the protocol flows are based on the expected choreography of the messages which flow between trading partners. For instance, the trading partner's software package may allow one URL to be configured for each request or it may place restrictions on the configuration and only allow a single URL to be configured for all requests. These restrictions or flexibility will dictate how the protocol may be implemented in the Gateway. Let's look at cXML as an example. cXML defines the ProfileRequest that is called to return the URLs that handle all the other cXML requests. With this flexibility, we might use separate URLs for each request or we might handle multiple requests with the same protocol flow. We implement cXML in Connect for iSeries as a single URL because the XML structure shares a common XML header across all requests, each request was easily determined by the XML header and the way we processed each request would be very similar.

4.1.5 Gateway Flow design guidelines for protocols

Although each protocol is different and each request is unique, Gateway protocol flows should follow a basic design structure. We'll break down the Gateway protocol flow into a sequence of typical steps. The Gateway protocol flow could be implemented as one big Java connector that performs all of the flow steps by itself. However, it's more desirable to implement each step as a separate connector, looking for commonality and reusing connectors across different requests and across different protocols.

As a general guideline, it is suggested that each of the following steps be implemented as its own connector when building protocol flows for standalone protocols. Typical steps which need to be addressed when defining standalone protocol flows are:

- XML validation and parsing of incoming request
- Authentication
- Authorization
- Inbound Logging
- Request DOM and Header Generation (setting partner/provider and request information)
- Queue the request/response to the FlowManager
- Error Checking
- Set Outbound Status
- Outbound Logging
- Error Handling
- Response Step (provide response data to the transport layer)

Many of these connectors are provided by Connect. They are implemented in a generic fashion so protocol flow designers can avoid writing a lot of code. Each of the supplied connectors are described later in the documentation.

One aspect of protocol flow development for which there are no utility connectors provided is error handling. This is because every protocol will have different requirements for how they process different types of errors. Protocol flow designers are able to define 'error steps'. These error step specifications within a protocol flow allow the protocol flow designer to call a connector which can be designed for handling error processing. There can be different error connectors defined for different steps within a flow, or the same error connector can be specified for all the steps. Anytime that a connector within the protocol flow returns a non-zero return code or throws an exception, that error step will be invoked.

The protocol flow designer must decide the appropriate steps to take based upon the error. There are a number of fields, mappable from the InternalHeader, which provide information regarding what type of error was encountered in the flow, and which step of the flow encountered the error. This error information as well as the protocol designers knowledge of the step which failed can be used to determine what their error processing connector should do. For instance, if the error occurred during some step of the flow which is considered optional or non-critical, then the protocol flow designer could choose to log some additional information that may assist in analyzing the error, but continue on in the processing of the protocol flow. However, if the error occurred in a step of the flow which is deemed critical (such as failure to authenticate the partner or failure to communicate with the FlowManager), then the error processing connector will need to take steps to construct an error response, and to either invoke the response step or the send message connector to cause the 'error' response to get sent back to the partner system.

4.1.6 Utility Connectors for Protocol Flow Development

The following utility connectors are intended for use by protocol flow designers. The utility connectors which the Delivery Gateway component ships with the product rely heavily on the mapping technology provided by the flow engine. One would expect some of these utility connectors to be used once in a protocol flow, whereas other connectors could be utilized multiple times within a flow.

Each of the utility connectors has input parameters, output parameters, and connector return codes section. The input parameters are describing the fields defined in the AppConnector which need to have mapin specifications (or possibly string constants) provided for them in the protocol flow which uses the connector. The output parameters are describing the fields defined in the AppConnector which need to have mapout specifications provided for them in the protocol flow. The connector return codes are defined in the java interface class called GatewayConstants, which is located in the GatewayAPI.jar file. Any protocol specific connector which has a requirement to interrogate the return code provided by these utility connectors should refer to the constants (the RETURN CODE column in the tables below) and not the actual values described here.

4.1.6.1 XML Validator (XML validation)

Name: XMLValidator

Description: This connector can be used to validate an XML byte array. The XML byte array is parsed into an XML document, but the XML document is not retained when using this connector. One use of this connector is to validate an XML byte array which has been built by the Connect infrastructure, to ensure that the system is sending valid XML to a trading partner.

The XML validation function will use the Connect Global entity resolver to locate XML validation documents (DTDs).

Input Parameters:

Name	Type	Required	Description
XMLInput	byte[]	Yes	Input byte array containing xml data.
ValidationMethod	String	No	Validation method to be used. Valid values for this field are: "Full", "EntityResolver", or "None".

Output Parameters:

Name	Type	Description
N/A		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the XML validation completed successfully if validation was requested.
ERROR_GW_XML_VALIDATIO N_FILES_NOT_FOUND	953	This error indicates that XML validation failed because a DTD or XML Schema file required for validation could not be located.
ERROR_GW_XML_VALIDATIO N_ERROR	954	This error indicates that XML validation failed due to some error in the XML document.
.ERROR_GW_EXCEPTION_TH ROWN	901	An exception was taken when attempting to access the XML byte array, or when validating the document. See the DG Message log files for additional information.

4.1.6.2 Inbound Logging

Name: InboundLoggerConnector

Description: Creates an audit log entry based upon the contents of the inbound message and the MessageHeader. This connector will use the input parameter NamedContexts to determine which named DOM or header contexts that it should include in the audit log entry. It also allows an additional byte array and byte array name to be specified which will be added to the audit log entry.

Input Parameters:

Name	Type	Required	Description
NamedContexts	String	No	Names of the contexts which the protocol designer chooses to dump into the audit log. This string may contain a list of named DOM and named Header contexts separated by the SeparatorCharacter. For example, if the protocol designer chooses to log named

			DOM contexts called Request and Response, then this field needs to contain the following string: “Request , Response” assuming the comma is being used as the separator character. Note: Under all conditions, the MessageHeader context will always be dumped by the InboundLogger. Any other contexts which the protocol designer chooses to dump out must be specified in this parameter. Also, any unrecognized contexts or any contexts besides DOM or Header contexts will be ignored by this connector, and not dumped into the log.
SeparatorCharacter	String	No	This field is used to specify the separator character used to delineate the request names within the NamedContexts String. The AppConnector will provide a default value of comma “,” .
ByteArray	byte array	No	Byte array to be added to the audit log entry
ByteArrayName	String	No	Name of byte array -- both ByteArray and ByteArrayName must be specified for the byte array to be added to the audit log entry. The ByteArrayName should be a valid XML element name.

Output Parameters:

Name	Type	Description
N/A		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the inbound message information was logged successfully.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken when attempting to access information or when dumping information into the audit log. See the DG trace log files for additional information.

4.1.6.3 Authentication Connector

Name: AuthenticationConnector

Description: This connector uses the supplied input parameters and invokes the Connect B2B basic authentication function to determine if the partner has provided the appropriate credentials to prove their identity. This basic authentication verifies that the password supplied to this connector matches the password which was configured for the LogonId that was configured into the Partner/Provider DB. If authentication is successful, then a successful return code is provided by this connector. If authentication fails, a non-zero return code is returned by the connector.

Input Parameters:

Name	Type	Required	Description
LogonId	String	Yes	The logon ID is a string made up of 1 or more substrings used to uniquely identify the partner being authenticated. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the uniqueId for the partner is a domain name of "DUNS" and Id Number of "123456798", then the LogonId should be "DUNS:123456789" .
Password	String	Yes	This is a password, or shared secret between the local system and the partner system.
Provider	String	Yes	The provider ID
Partner	String	Yes	The partner ID

Output Parameters:

Name	Type	Description
N/A		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the LogonId is known by the Connect B2B authentication function and that the Password supplied matches the password which was configured for this LogonId.
ERROR_GW_AUTHENTICATION_FAILED	942	The Connect B2B authentication function failed this request. This could indicate that the LogonId specified has not been configured in the Partner/Provider DB or that the password supplied does not match the password that is configured for this LogonId.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken when attempting to invoke the Connect B2B authentication function. See the DG Message log files for additional information.

4.1.6.4 Authorization Connector

Name: AuthorizationConnector

Description: This connector uses the supplied input parameters and invokes the Connect B2B authorization function to determine if the partner, identified by the ProviderId / PartnerId pair, is allowed to issue the Request which has been received. If authorization is successful, then a successful return code is provided by this connector. If authorization fails, a non-zero return code is returned by the connector.

Input Parameters:

Name	Type	Required	Description
Request	String	Yes	Request (such as "OrderRequest" for cXML).
ProviderId	String	Yes	The ProviderId is a string made up of 1 or more substrings used to uniquely identify the provider (or supplier) involved in the request. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the provider/supplier are "DUNS" and Id Number of "123456798", then the ProviderId should be "DUNS:123456789" .
PartnerId	String	Yes	The PartnerId is a string made up of 1 or more substrings used to uniquely identify the partner (or

			buyer) involved in the request. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the partner/buyer are "DUNS" and Id Number of "555555555", then the PartnerId should be "DUNS:555555555" .

Output Parameters:

Name	Type	Description
N/A		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the partner that is authorized to issue the request to the provider specified. This has been validated using the Connect B2B authorization function.
ERROR_GW_AUTHORIZATIO N_FAILED	938	The Connect B2B authorization function failed this request. This could indicate that the Partner or Provider specified has not been configured properly in the Partner/Provider DB, or that this request is not listed as one of the valid requests which can be processed by this provider from this partner.
.ERROR_GW_EXCEPTION_TH ROWN	901	An exception was taken when attempting to invoke the Connect B2B authorization function. See the DG Message log files for additional information.

4.1.6.5 Partner Or Provider Resolver

Name: PartnerOrProviderResolver

Description: This connector is used to determine the unique reference numbers which represent the provider (supplier) and/or the partner (buyer) involved in this request. This unique reference number is maintained in the Partner/Provider database, and the reference number which is retrieved can be used by subsequent connectors to access information regarding this Partner or Provider. This connector allows either the partner, provider, or both reference numbers to be retrieved. The connector maps out these reference numbers to the location controlled by the mapout specification. A typical use of this connector is to retrieve these reference numbers and mapout the results into the key fields maintained in the Partner and Provider data contexts (specifically the #Partner_Refno and #Provider_Refno keys.)

Input Parameters:

Name	Type	Required	Description
ProviderId	String	Yes	The ProviderId is a string made up of 1 or more substrings used to uniquely identify the provider (or supplier) involved in the request. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the provider/supplier are "DUNS" and Id Number of "123456798", then the ProviderId should be "DUNS:123456789" .
PartnerId	String	Yes	The PartnerId is a string made up of 1 or more substrings used to uniquely identify the partner (or buyer) involved in the request. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the partner/buyer are "DUNS" and Id Number of "555555555", then the PartnerId should be "DUNS:555555555" .

Output Parameters:

Name	Type	Description
ProviderRefNum	String	The unique reference number of the provider.
PartnerRefNum	String	The unique reference number of the partner.

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the unique partner and/or provider reference numbers were successfully retrieved and the reference numbers were successfully mapped out to the fields provided in the mapout specifications.
ERROR_GW_PROVIDER_REFNUM_RETRIEVE_FAILED	950	An error occurred while attempting to retrieve the unique provider reference number. This error most likely indicates that the ProviderId supplied as input to this connector does not match any of the providers configured in the Partner/Provider DB for this instance.
ERROR_GW_PARTNER_REFNUM_RETRIEVE_FAILED	951	An error occurred while attempting to retrieve the unique partner reference number. This error most likely indicates that the PartnerId supplied as input

		to this connector does not match any of the partners configured in the Partner/Provider DB for this instance.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken while attempting to process this request. See the DG Message log files for additional information.

4.1.6.6 Request Token Generation

Name: RequestToken

Description: This connector is used to generate a request token. This request token can be used by process flow designers to supply identification type parameters required by the Gateway to process a subsequent Outbound Message. These outbound messages are initiated by a connector within a process flow or by a back-end application. Refer to the Outbound Message Handler section of this document for additional information regarding the use of a request token. The parameters supplied to this connector are used as input in creating a string representation of this request token.

Suggested use of this connector is to mapout the results into the REQUEST_TOKEN field of the MessageHeader context.

Input Parameters:

Name	Type	Required	Description
ProviderId	String	Yes	The ProviderId is a string made up of 1 or more substrings used to uniquely identify the provider (or supplier) involved in the request. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the provider/supplier are "DUNS" and Id Number of "123456798", then the ProviderId should be "DUNS:123456789" .
PartnerId	String	Yes	The PartnerId is a string made up of 1 or more substrings used to uniquely identify the partner (or buyer) involved in the request. The separator characters for the substrings must be a colon ":" . For example, if the substrings making up the partner/buyer are "DUNS" and Id Number of "555555555", then the PartnerId should be "DUNS:555555555" .
Request	String	No	The request is a string that specifies the request which is

			associated with the Request Token. This can either be the request currently being processed or the request associated with the outbound request that will eventually be processed.

Output Parameters:

Name	Type	Description
RequestToken	String	RequestToken which can be used by back-end application or process flows to invoke the OMH function with the appropriate identification parameters required to initiate OMH flows.

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the request token was successfully retrieved and placed into the field based upon the mapout specification.
ERROR_GW_REQUEST_TOKEN_CREATION_FAILED	952	The connector issued the command to have a request token generated, but no request token was generated. Verify that the input parameters being passed into this connector have valid data.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken when attempting to create the request token or setting the request token into the field specified by the mapout specification. See the DG Message log files for additional information.

4.1.6.7 FlowManager Communication Connector

Name: FMComm

Description: This connector handles sending requests from the Gateway to the FlowManager, using the MQ Series queue for the inter-process communication. It also handles the response received from the FlowManager. This connector uses the information contained in the flow to determine which of the data contexts to serialize and send to the FlowManager. The MessageHeader context is always included, and other data contexts are provided based upon the selections made by the protocol flow designer. When the response is received from the FlowManager, the data returned is deserialized and placed back into the appropriate data context to allow the protocol flow to continue with processing of the response.

Input Parameters:

Name	Type	Required	Description
N/A			

Output Parameters:

Name	Type	Description
N/A		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the request was sent to the FlowManager, response data was received from the FlowManager, and the response data was successfully transferred back into the flow data area of the flow that is currently running.
ERROR_GW_TRANSPORT_INIT	903	An error occurred while gathering up the information into an object for transport to the FlowManager. See the DG trace log files for additional information.
ERROR_GW_TRANSPORT	904	An error occurred in the transport of the message either from the DG to the FlowManager, or from the FlowManager back to the DG. There was no response data available when the DG regained control. See the DG trace log files for additional information.
ERROR_GW_TRANSPORT_TIMEOUT	905	The DG sent the message to the FlowManager, but no response was received from the FlowManager prior to the timeout value expiring. The timeout value is controlled by the MSG_QUEUE_TIMEOUT value in the NonSendable header. Verify that this timeout value is long enough to complete requests and that the FlowManager process is started for this Connect instance.
**** OTHER *****	xxx	Either the FlowManager or back-end application has encountered an error. The error code provided by this connector is the same error code that was returned by the FlowManager. See the DG or FM trace log files for additional information.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken when attempting to send the request or get the response from the FlowManager. See the DG Message log files for additional information.

4.1.6.8 Outbound Logging

Name: OutboundLoggerConnector

Description: Creates an audit log entry based upon the contents of the outbound message and the MessageHeader context. This connector will use the input parameter NamedContexts to determine which named DOM or header contexts that it should include in the audit log entry. If one of the named contexts is not available, then this connector will still proceed with dumping out the MessageHeader and any of the specified name contexts which are available. If this condition occurs or if some other error occurs during the logging process, an error code will be returned by this connector. If logging is successful, then a good return code will be provided.

An additional byte array and byte array name may also be specified which will be added to the audit log entry.

Input Parameters:

Name	Type	Required	Description
NamedContexts	String	Yes	Names of the contexts which the protocol designer chooses to dump into the audit log. This string may contain a list of named DOM and named Header contexts separated by the SeparatorCharacter. For example, if the protocol designer chooses to log named DOM contexts called Request and Response (which is what our cXML implementation will map into this connector), then this field needs to contain the following string: “Request , Response” assuming the comma is being used as the separator character. Note: Under all conditions, the MessageHeader context will always be dumped by the InboundLogger. Any other contexts which the protocol designer chooses to dump out must be specified in this parameter. Also, any unrecognized contexts or any contexts besides DOM or Header contexts will be ignored by this connector, and not dumped into the log.

SeparatorCharacter	String	N	This field is used to specify the separator character used to delineate the request names within the NamedContexts String. The AppConnector will provide a default value of comma “,” .
ByteArray	byte[]	No	Byte array to be added to the audit log entry
ByteArrayName	String	No	Name of byte array -- both ByteArray and ByteArrayName must be specified for the byte array to be added to the audit log entry. The ByteArrayName should be a valid XML element name.

Output Parameters:

Name	Type	Description
N/A		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the outbound message information was logged successfully.

4.1.6.9 Request Mapper Connector

Name: RequestMapper

Description: This connector is used to verify that a request entering the Gateway via OMH is supported. The incoming request must either match a protocol specific request which is expected by this flow, or a generic request. It will also mapout the protocol specific request into the mapout ProtocolSpecificRequest mapout parameter. The reason that the protocol flow designer can specify a generic request as well as the protocol specific request is that the back-end application initiating this outbound message could be written in a protocol independent manner, thus the back-end application may not know (or care) what protocol is actually being used to communicate between the partners.

This connector makes use of several input parameters. The first parameter contains the request to compare. The second input parameter is used to provide the list of protocol specific requests that are supported by this protocol flow. The third parameter, if provided, lists the generic requests which correspond to the list of protocol specific requests. If the generic list is provided, the number of generic requests must be the same as the number of protocol specific requests, otherwise an error code will be returned by this connector.

The suggested use of this connector is to specify the Request field of the MessageHeader for both the RequestToCompare mapin parameter and the ProtocolSpecificRequest mapout parameter. This will allow the MessageHeader to get filled in properly so that the FlowManager can perform flow selection against a valid request supported by this protocol.

Input Parameters:

Name	Type	Required	Description
RequestToCompare	String	Y	This field contains the request field to compare against, to determine if this request is either a Protocol Specific or Generic request supported by this protocol flow.
ProtocolRequestsSupported	String	Y	Protocol specific names of the requests which this flow is capable of processing. This string contains a list of these request names, separated by the SeparatorCharacter. For example, if this flow can process the requests of StatusUpdateRequest, ConfirmationRequest, and ShipNoticeRequest, and the SeparatorCharacter is a comma, then the value passed into this field must be: “StatusUpdateRequest,ConfirmationRequest,ShipNoticeRequest”
GenericRequestsSupported	String	N	Generic names of the requests which this flow is capable of processing. The number of generic requests must match the number of protocol specific request names supported. This string contains a list of these generic request names separated by the SeparatorCharacter. For example, if this flow supported the generic requests of Status Update, OrderConfirmation, and ShipNotice, and the corresponding protocol specific requests are: “StatusUpdateRequest,ConfirmationRequest,ShipNoticeRequest” Then this GenericRequestsSupported field should be set to:

			“StatusUpdate,OrderConfirmation,ShipNotice”
SeparatorCharacter	String	N	This field is used to specify the separator character used to delineate the request names within the ProtocolRequestsSupported and the GenericRequestsSupported parameters. The AppConnector will provide a default value of comma “,” .

Output Parameters:

Name	Type	Description
ProtocolSpecificRequest	String	If this connector is successful, this field will contain the value of the RequestToCompare field if the RequestToCompare was found in the ProtocolRequestsSupported list. If the RequestToCompare field is found in the GenericRequestsSupported list, then the correponding ProtocolRequestsSupported entry will be mapped out.

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates that the request passed in either matched one of the allowable protocol specific or generic requests being compared against. The ProtocolSpecificRequest was successfully set in the field specified by the mapout specification.
ERROR_GW_UNSUPPORTED_MESSAGE_TYPE	948	The RequestToCompare that was passed in was not listed in either the ProtocolRequestsSupported or the GenericRequestsSupported list.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken when attempting to perform the request comparison or when mapping out the results. See the DG Message log files for additional information.

4.1.6.10 Response step

The Response step is used in the Gateway protocol flows to signal a response is ready to be sent from the flow. The Response step can be used by protocol flows started by either an inbound request, or an outbound request. When the Response step is encountered the Delivery Gateway will collect the response from the TransportOutput context and deliver it.. All output data to be returned to the caller must be set before calling the Response step.

Even though the Response step returns data to the caller, the flow continues with the next step after the Response step, if available. By calling the Response step as early as possible in the protocol flow, the caller is able to handle the response sooner while the protocol flow is still able to continue with the next step and handle the request. This is especially useful when a protocol request just needs an acknowledgment, like “200, Okay”, without the entire request being processed. Performance wise, it is a good idea to issue the Response step as soon after the response data is available. If a protocol flow needs to send a message to a remote partner after the Response step is called the SendMessage connector should be used.

A Response step will always be called in a Protocol flow. If no Response step is encountered before the end of the Gateway protocol flow then the Delivery Gateway will treat this as though an implicit response step was provided and the data in the transport output context will be returned to the partner as a response.

4.1.6.11 SendMessage Connector

Name: Sendmessage

Description: This connector is used to send a message to a remote partner from a protocol flow. The SendMessage Connector uses the Outbound Message Handler to deliver the message. See the Outbound Message Handler section for more information.

Input Parameters:

Name	Type	Required	Description
MessageId	String	Yes	Message Identifier. This must be a unique message identifier for the protocol.
Message	Byte []	Yes	The message to send.
MessageType	String	Yes	Type of message input, either “ByteArray” or “MIME”.
Headers	Repeating String	No	The headers that should be sent along with the message.
ProviderRefNum	String	No.	ProviderRefNum The unique reference number of the provider
PartnerRefNum	String	No.	PartnerRefNum The unique reference number of the partner

TransportType	String	No	Transport to use. Valid types are HTTP, MAILBOX, *PARTNERPROFILE Default: *PARTNERPROFILE
DefaultProperties	String	No	Default properties to use to send the message. These properties can be overridden by the partners properties stored in the partners profile. String format is “key=value:key=value” Where “:” is the separator character.
OverrideProperties	String	No	Override properties will override all other properties contained in either the default properties or the partners profile properties. These are useful to override certain properties for a particular message. For example you can override the URL for HTTP if an inbound message contained a “reply-to” URL. String format is “key=value:key=value” Where ” :” is the separator character
SeparatorCharacter	String	No	The separator character used to separate the values of the previous two parms. This value defaults to “,” if not specified.

Output Parameters:

Field	Type	Description

Properties for SendMessage connector:

Key	Value	Description
SynchronousDelivery	Yes, No Default: No	If synchronous delivery is “Yes” then the send message connector will not return until the message is either delivered or an error is encountered. Otherwise the message will be saved to a persistent database and attempts to deliver the message will continue until the retry count expires
RetryCount	0-999, Config Default: Config	Controls the number of retries for a message. If “Config” is used, the send message connector will use the number

		of retries specified in the Gateway properties.
RetryInterval	1-999, Config Default: Config	Controls the interval of retries in minutes. If “Config” is used, the send message connector will use the retry interval specified in the Gateway properties
ReboundFlow	Flowname, Config Default: Config	Controls what flow will be called if data is received back while deliverying the outbound message. If “Config” is specified the rebound flow defined for the current flow definition is called.
DeliverySuccess	Transport, Protocol Default: Transport	Controls who decides if the message was delivered successfully. Some protocols use acknowledgement messages within the protocol to signal when a message was successfully received. The default is for the transport protocol to decide when a message was successfully delivered.

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully. This indicates either the message was sent sucessfully or the message was saved to be sent at a later time.
.ERROR_GW_EXCEPTION_THROWN	901	An exception was taken when attempting to access information. See the DG Message log files for additional information.
.ERROR_GW_MESSAGE_FAILED	922	The message could not be sent.

Transport data used by the OMH

The OMH retrieves delivery information data from the protocol partner registry to deliver an outbound message. When a Gateway protocol is written the data model and screen model for the protocol partner registry must contain this data that is required by the delivery transports for the transports that are supported by the Gateway protocol. Below is a list of transports and the data that needs to be collected.

HTTP

Partner Profile Field	Properties Example	Description
URL	URL=http://example.com	Location to send the request
Security	Security=*NO	Indicator whether to use a security mechanism. Values: *SSL or *NO
LogonID	LogonID=myID	If this URL uses basic authentication a logon ID can be supplied
LogonPassword	LogonPassword=secret	If this URL uses basic authentication, a password can be supplied.

MailBox Database

None.

This data may be associated with either the target or the partner depending on the needs of the protocol. If the protocol specifies that the partners communicate with each other directly then the data should be associated with the partner. If communications go through a central hub or marketplace (like Ariba and it's Commerce Services Network does) then the data should be set once for the target, since it doesn't vary per partner.

Rebound Flows

When an outbound message is sent to a remote partner, data may be received on the connection if the transport supports response data. For instance, if a message is sent via an HTTP POST then the remote partner may send data back on the same connection. If this happens the Outbound Message Handler will call a Gateway flow to process the response data if one is available. This type of flow is called a "rebound" flow. Each Gateway flow defined in a protocol implementation can have a rebound flow associated with it. The outbound message handler will locate the flow that initiated the outbound message and determine if a rebound flow is defined and if so, it will then pass the data to the flow and start the flow.

4.1.6.12QueryMessage Connector

The QueryMessage connector can be used to find messages that were stored in the Message database.

Input Parameters:

Field	Type	Required	Description
Key	String	Y	Message selection key.
Value	String	Y	Key value.

Output Parameters:

Field	Type	Description

MessageRefNum	Long []	Message Reference Number

Selection Keys:

Key	Value example	Description
PartnerRefNum	22	Partners reference number
ProviderRefNum	34	Providers reference number
ProtocolMsgId	Order123	Protocol specific message identifier

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully.
ERROR_GW_EXCEPTION_THROWN	901	An exception was thrown.

4.1.6.13RetrieveMessage Connector

The RetrieveMessage connector can be used to retrieve messages that were stored in the Message database.

Input Parameters:

Field	Type	Required	Description
MessageRefNum	Long	Y	Message Reference Number

Output Parameters:

Field	Type	Description
Message	byte []	Message
Headers	String []	Message Headers
Format	String	Format of the message. Either “MIME” or “ByteArray”

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully.
ERROR_GW_EXCEPTION_THROWN	901	An exception was thrown.

4.1.6.14UpdateMessageDeliveredStatus Connector

The UpdateMessageDeliveredStatus connector can be used to set the delivered status of a message that was stored in the Message database. The message may have been stored by the OMH processing.

Input Parameters:

Field	Type	Required	Description
MessageRefNum	Long	Y	Message Reference Number
Status	Int	N	Valid values are 0 and 1. Default value is 0.

Output Parameters:

Field	Type	Description
None		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully.
ERROR_GW_EXCEPTION_THROWN	901	An exception was thrown.

4.1.6.15UpdateMessageProcessingStatus Connector

The UpdateMessageProcessStatus connector can be used to set the processing status of a message that was stored in the Message database.

Input Parameters:

Field	Type	Required	Description
MessageRefNum	Long	Y	Message Reference Number
Status	String	Y	Any value that is meaningful to the protocol may be stored.

Output Parameters:

Field	Type	Description
None		

Connector Return Codes:

RETURN CODE	VALUE	Description
EXIT_SUCCESSFUL	0	The connector completed successfully.
ERROR_GW_EXCEPTION_THROWN	901	An exception was thrown.

4.1.7 Enhanced XML Validation

A protocol designer can determine whether XML documents need to validated. This validation is performed by the XML parser. There are three 'validation methods' which will be supported by the system. They are:

- Full Validation
- Partial Validation with Entity Resolution
- No Validation

Full validation means that all of the XML validation files will be located, and full XML validation will be performed. This method provides the most assurance that the XML document is both syntactically and semantically correct, but is the most expensive from a performance standpoint.

Partial Validation with Entity Resolution still verifies that an XML document is well formed and it also requires locating the XML validation files. This method requires the validation files because XML instance documents may be using entities, and these entities are defined in the XML validation document. This entity resolution can be thought of as 'macro substitutions' which must occur in the XML document. This option will perform faster than full validation. This is the default value configured in the Gateway instance properties.

No Validation means the system will not use any XML validation file. This is the fastest option from a performance standpoint. The XML parser will be supplied with an 'empty' XML validation file. When this option is selected, the XML parser only assures that the XML document is well formed. This option is not recommended. If an XML instance document contained any entity references, these would not be resolved. Erroneous results could occur later in the processing of the parsed XML document. This option should only be considered if the protocol designer knows there will never be entities used within their XML instance documents and if performance is a critical factor.

As an XML document is being parsed, it locates the XML DTD documents that define the rules that govern whether a given XML document is valid or not. For simplicity, we will refer to these XML DTD documents as XML validation documents. The validation of a single XML document may require multiple XML validation documents if multiple XML validation documents are referenced from within a single XML document. The XML parser provides a mechanism to allow the 'caller' of its function to assist in locating these XML validation documents. The Connect for iSeries framework has a global entity resolver which will handle locating the required XML validation documents required for validating XML documents..

There is three ways that the Global Entity Resolver will locate XML validation documents. They are:

1. Use XML validation documents that are provided by the Connect administrator, which are located on the local file system.
2. Use XML validation documents that were previously located in the network, that have been cached locally by the Connect Global Entity Resolver.

3. Use XML validation documents that are located in the network. After locating these XML validation documents, they will be cached for future use.

The three methods of locating these XML validation documents are the priority order in which the Connect Global Entity Resolver will attempt to locate the documents. Each of these techniques will be explained in greater detail.

4.1.7.1 XML validation documents provided by the Connect Administrator, which are located on the local file system

There are several key points that need to be understood regarding this topic. These points are 1) where should the files be located , 2) What should the files be called, and 3) How will the system identify the correct file to use during validation.

File Location

The location of the files can be anywhere, but it should be noted that all instances that need a particular file must have read access to the file. The instances will also need execute authority on all of the directories in the path, which will allow the path to be searched. The '/QIBM/UserData/Connect200/XMLValidation' directory can be used for storing these files. All the Connect instances will be granted 'read' authority to this directory, and execute authority on all of the directories in the path. For files located in other directories, the administrator will be responsible for granting the proper authority to any Connect instance user profiles which requires these other files for performing XML validation.

Note: In a split system configuration an administrator will need to keep the files between the FlowManager and Gateway system synchronized manually.

File Name

The files can be named anything that a protocol designer chooses. The one restriction is that all of the files placed into the /QIBM/UserData/Connect200/XMLValidation directory must be uniquely named.

How the system identifies the correct file to use during validation

The Connect administrator must provide mappings. There is a table which the administrator modifies using the Connect for iSeries configuration GUI which controls what local file is to be used for validation purposes. This table will be used to map the 'location' (which is found in the DOCTYPE element of an XML document being validated) to the file name which corresponds to the XML validation document located on the local file system. This table will be made up of a series of URILocation and LocalFileID pairs. The 'URILocation' will be the location parameter as it appears in the XML documents. The 'LocalFileID' will be the local file which contains the XML validation document. The value can be just a filename. In that case, it is assumed the XML validation file is located in the /QIBM/UserData/Connect200/XMLValidation directory. The value can also be a fully qualified path and filename. It is important to note that whatever the 'LocalFileID' parameter points to, the Connect instance that requires the use of the file for validation must have read authority to the directories and file.

The 'URILocation' field will also support the use of an '*' as a wildcard character. This wildcard character can be used to allow 'partial' locations to be specified. This is useful for cases when the partner system is specifying a slight revision to the XML validation document, but the protocol designer wants to continue using the local copy of the XML validation document. The precedence is that the Connect global entity resolver will first attempt to find a direct match for the URILocation. If no direct match is found, then the first entry encountered in the table which would provide a match based upon the wildcard character will be selected.

Following is a sample set of URILocation and LocalFileID pairs which could be stored in the global registry, and an explanation of how the Connect Global Entity resolver will behave based upon various 'locations' encountered in received XML documents.

URILocation	LocalFileID
http://xml.Widgets.org/schemas/Widgets/1.2.3/Widgets.dtd	Widgets123.dtd
http://xml.Widgets.org/schemas/Widgets/*/WidgetShipNotice.dtd	WidgetShipNoticeAny.dtd

Case 1 - XML document received specifies the following for location:

http://xml.Widgets.org/schemas/Widgets/1.2.3/Widgets.dtd

In this case, Widgets123.dtd will be used for validation.

Case 2 - XML document received specifies the following for location:

http://xml.Widgets.org/schemas/Widgets/1.2.7/Widgets.dtd

In this case, There is no match found in the file, thus the system will first attempt to locate the dtd in the 'cache', and if not found, it will search the network.

Case 3 - XML document received specifies the following for location:

http://xml.Widgets.org/schemas/Widgets/1.3.8/WidgetShipNotice.dtd

In this case, the system will use WidgetShipNoticeAny.dtd for validation.

Case 4 - XML document received specifies the following for location:

http://xml.Widgets.org/schemas/Widgets/1.3.9/WidgetShipNotice139.dtd

In this case, There is no match found in the file, thus the system will first attempt to locate the dtd in the 'cache', and if not found, it will search the network.

NOTE: There are several DTD and schema files which are shipped with the Connect product in support of cXML. The mapping table described above will have mapping entries for the appropriate URIs and files filled in automatically as part of the install process. These cXML files will be shipped in Proddata.

4.1.7.2 XML validation documents that were previously located in the network, that have been cached locally by the Connect Global Entity Resolver

A protocol designer may choose not to supply any XML validation documents, and they may rely on having the system retrieve the XML validation documents from the network. In this case, the system will cache the XML validation documents in the /QIBM/UserData/Connect200/CachedXMLValidation directory.

Besides having the URILocation and LocalFileID mapping pairs stored in the global registry, there is also a configurable property called **CachedEntityResolverExpiration**. This property specifies the 'number of days' that cached files found by the global entity resolver will continue to be used. If a file has been in the cache longer than this value, then the next time this XML validation file is required, the file will be retrieved from network to refresh the cache. The system supplied default for this parameter is 14.

Besides placing the XML validation files in the directory above, there will also be some additional files that are placed in this directory for keeping track of these cached file mapping (e.g. Similar to how the URILocation to LocalFileID mapping works), and for keeping track of when to refresh the cached XML validation files.

4.1.7.3 XML validation documents that are located in the network

The system will also have support for retrieving the files using either http or ftp, based upon the location information provided in the XML document. An important note regarding the retrieval of XML documents over the network is that in order for it to work, the URL provided in the XML document must point to a valid URL, and that URL must be reachable by the local system. If the site specified by the URL is down, or behind some firewall which the local system cannot reach, then this option cannot be used. In those cases, the XML validation documents need to be made available locally using the first technique which was described earlier.

5.0 eCatalog Interfaces

The e-Catalog administration tool was an original part of iSeries Connect's administration servlet. It is used to maintain product and pricing information and for the publishing of this information to individual buyers and/or public e-marketplaces like Ariba.

Customers/suppliers can use the e-Catalog tool in one of two ways. They can maintain the product and price information for their catalog in the e-Catalog tool itself or maintain it in their legacy backing store, such as DB2, WebSphere Commerce Suite, or Domino.

Information in the e-Catalog tool is then published to a format acceptable to a register marketplace. This information can then be sent to the marketplace for use by potential buyers. If a private format is desired, then the catalog information can be exported to a standard XML format. Customers/suppliers can then use an open-standard translator, such as Xalan, to convert the markup to their private format.

Since product and pricing information is dynamic in nature, it is often necessary to maintain this information by repeatedly using the Connect Administration GUI to refresh the product/price information from the souce DB, and then publish or export it.

5.1.1 Public APIs for Refresh/Publish

Three new APIs will be provided to facilitate programmatic invocation of certain e-Catalog functions that are periodic in nature. These functions are refresh, publish, and export. Invocation of these APIs is performed by calling main() on the desired Java class, with the specified string parameters:

```
com.ibm.connect.tools.catalog.ExternalRefresh( instanceName,

                                                catalogName )


com.ibm.connect.tools.catalog.ExternalPublish( instanceName,

                                                catalogName )
```

`instanceName` is the name of the B2B instance for the catalog.

`catalogName` is the name of the target catalog.

Callers (USRPRFs) of these APIs will be required to possess the same authority as that of a valid Connect Administration GUI user.

Implicit parameters (those not supplied to the API) for the publishing of a catalog via the API will be taken from the current registry information for that catalog.

Any error messages will be translated via resource bundles and directed to standard output.

A CL *PGM object can be created that refreshes and publishes an input catalog. A sample CLSRC for this *PGM, that takes two parameters, instanceName and catalogName, is shown below:

```
PGM (&INSTANCE &CATALOG)

DCL VAR(&INSTANCE) TYPE(*CHAR) LEN(10)

DCL VAR(&CATALOG) TYPE(*CHAR) LEN(32)

ADDENVVAR  ENVVAR(CLASSPATH) +

VALUE('/tobrien:+

/QIBM/PRODDATA/CONNECT200/TOOLS/CATALOG/CATALOG.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/CONFIG.JAR:+

/QIBM/PRODDATA/CONNECT200/TOOLS/TPA/TPA.JAR:+

/QIBM/PRODDATA/CONNECT200/TOOLS/RUNTIME/util.jar:+

/QIBM/PRODDATA/CONNECT200/CLASSES/WCSCONFIGSERVICES.JAR:+

/QIBM/PRODDATA/JAVA400/JT400NTV.JAR:+

/QIBM/PRODDATA/OS400/JT400/LIB/JT400NATIVE.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/XERCES321.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/SOAP.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/XALAN220.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/LOGGING.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/LOG.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/LOGGINGAPI.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/NCSO.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/ACTIVATION.JAR:+

/QIBM/PRODDATA/CONNECT200/CLASSES/MAIL.JAR:+
/QIBM/PRODDATA/CONNECT200/CLASSES/COMIBMCONNECT.JAR') +

REPLACE(*YES)

JAVA       CLASS(com.ibm.connect.tools.catalog.ExternalRefresh) +

PARM(&INSTANCE &CATALOG)

JAVA       CLASS(com.ibm.connect.tools.catalog.ExternalPublish) +

PARM(&INSTANCE &CATALOG)

ENDPGM
```

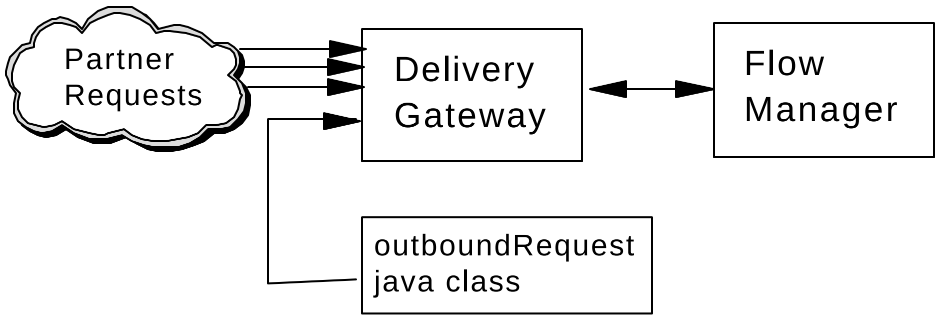
This *PGM can be called directly via the CALL cmd, or can be scheduled via SBMJOB with the SCDDATE and SCDTIME parameters or via the Advanced Job Scheduler (5722-JS1).

6.0 Sending Outbound Messages

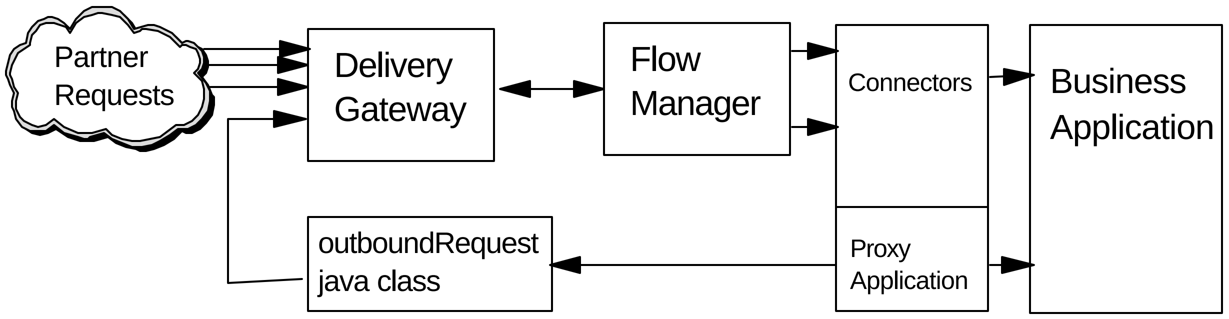
Besides receiving messages from remote trading partners the Delivery Gateway is also capable of originating messages. The Outbound Message Handler (OMH) is used to send messages to remote partners. The OMH supports sending messages via HTTP, and also supports storing messages in a mailbox database. To send an outbound message the protocol writer would call the SendMessage connector from a protocol flow. The connector can be called from a protocol flow that is being run from the result of an incoming request from a partner or a protocol flow can be initiated by using the outboundRequest Java class.

6.1 Initiating an Outbound Message

The outboundRequest Java class is used to initiate a protocol flow to send an outbound message. It works by calling into the Delivery Gateway and acting as an inbound request. Once the protocol flow is initiated it can perform any processing necessary for the outbound message. This would include queuing the request to the Flow Manager so that the backend application can complete the outbound request. What is traditionally a response to a trading partner initiated request is now the outbound message instead.



The outboundRequest API can be called from a Java Connector or from a separate application. The separate application is referred to as a “proxy application” because it most likely not part of the business application you’re connecting to and would be written by an ISV with detailed knowledge of the business application’s capabilities and interfaces. The proxy application has two functions; one is to extract information from the business application and to call the outboundRequest APIs to originate outbound requests. The second part of the proxy application is to implement a connector to extract information from the backend application to complete the outbound request.



The proxy application that initiates the outbound request by calling the API could be a stand alone application or it could be a step in a flow manager flow.

6.2 OutboundRequest Java class

The java connector or proxy application will construct an OutboundRequest object, set any required fields and call the initiateRequest() method to send the outbound message. The OutboundRequest API needs to know enough information about the request to call the correct Delivery Gateway instance and run the proper Gateway protocol flow for the protocol. It uses the Connect configuration registry to locate the Gateway so the API must be run on a system where either the Delivery Gateway or the Flow Manager resides. The following fields are required;

Field	Example	Description
Instance	"Acme_Corp"	Connect Instance
Protocol Group	"Ariba_CSN"	Connect Protocol Group (formerly called Marketplace)
Protocol	"BusinessXML"	Protocol
Protocol Subtype (deprecated)		Protocol Subtype This field is deprecated and if present will be concatenated with Protocol to form a Protocol field name in the format of: Protocol:ProtocolSubtype:ProtocolVersion
Protocol Version (deprecated)		Protocol Version This field is deprecated and if present will be concatenated with Protocol to form a Protocol field name in the format of: Protocol:ProtocolSubtype:ProtocolVersion
Request	"Invoice"	Protocol Request

The following fields are optional on the API but may be required by the protocol flow;

Field	Example	Description
Request Type (deprecated)		Request Type This field is deprecated and if present will be concatenated with Request to form a Request field name in the format of: Request:RequestType
Provider	"123456789:DUNS"	Provider (formerly called Supplier) If a "Supplier" field is supplied it will be used for the "Provider"
Partner	"987654321:DUNS"	Partner (formerly called Buyer) If a "Buyer" field is supplied it will be used for the Partner.

Finally, the following field is for use by the java connector or proxy application. It can contain any string data that may be needed to communicate between the proxy application and the flow manager connectors.

Field	Example	Description
Application Token	"Invoice #1234567"	Application Token This string is saved in the MessageHeader context as 'com_ibm_connect_header_appToken', or Java constant HeaderConstants.APP_TOKEN.

There are two methods the caller can use to set the required fields. One method is to set the fields individually. The other method is to use a Request Token. The fields can be set individually on the object constructor, or through individual set methods or through the use of a Properties object.

Using a Request Token can greatly simplify the API call. A Request Token is usually generated on an incoming request during the Gateway protocol flow and is stored in the MessageHeader context. When the Request Token is generated it saves the following fields; Protocol Group, and Protocol. It will also save the Provider, Partner and request information if it is available.

A Flow Manager connector can use or save away the value of the Request Token to make calling the OutboundRequest easier. If a Request Token is used then just the Instance, Request Token, Request and any optional fields need to be specified.

If the deprecated methods are called or a properties object provided on the setRequestFields() method contains ProtocolSubtype, and ProtocolVersion, then these fields will be concatenated with the Protocol value separated by a colon. Similarly, if RequestType is provided it will be concatenated with Request. Any value provided as the Buyer or Supplier will become the Partner and Provider respectfully.

A few of the discussed methods are presented below. See the Javadoc for the complete documentation of the object and all of it's methods.

public OutboundRequest()

```
(deprecated)public OutboundRequest(String instance,
                        String marketplace,
                        String protocol,
                        String protocolSubtype, /* will be concatenated with protocol */
                        String protocolVersion, /* will be concatenated with protocol */
                        String request,
                        String requestType,
                        String buyer,   /* will be treated as the Partner */
                        String supplier,  /* will be treated as the Provider */
                        String applicationToken)
```

```
(deprecated)public OutboundRequest(String instance,
                        long requestToken,
                        String request,
                        String requestType,
                        String applicationToken)
```

public void setRequestFields(Properties requestFields)

6.2.1 Authority Required

The caller will need *RX access to all the directories in the path to the jar file and *R authority to the jar file itself.

The caller of the API must be authorized to the "Connect Outbound Messages" program function or have *ALLOBJ authority on the Delivery Gateway system. To authorize the user profile to have "Connect Outbound Messages" program function use Operations Navigator.

6.2.2 Requests

There are a set of predefined protocol independent requests the caller can use to set the request value. The caller can also set a protocol specific request value if none of the defined generic requests are similar. The predefined set allows the caller to be protocol independent yet still send outbound requests like "ShipNotice". The Delivery Gateway protocol connectors will intercept the predefined set of request and turn them into a protocol specific request. For example, the cXML protocol flow connector that is processing the outbound request will see the "ShipNotice" request and turn that into a cXML ShipNoticeRequest request. Protocol writers should see the RequestMapper connector for information on how the request mapping is done. If the Delivery Gateway protocol connector does not support the request an error is returned to the proxy application (BAD_REQUEST).

Since these predefined requests are translated to actual protocol specific requests before the Flow Manger is called, the generic requests don't need to be deployed or have RequestMessage files associated with them.

Request	Description
Invoice	Order Invoice
ShipNotice	Advance Ship Notice
OrderConfirmation	Order Confirmation

6.2.1 Classpath requirements

/QIBM/ProdData/Connect200/gatewayAPI.jar

/QIBM/ProdData/Connect200/gateway.jar

/QIBM/ProdData/Connect200/bridge.jar

/QIBM/ProdData/Connect200/Classes/xerces321.jar

/QIBM/ProdData/Connect200/Classes/xalan220.jar

/QIBM/ProdData/Connect200/Classes/config.jar

/QIBM/ProdData/Connect200/Tools/Runtime/util.jar

/QIBM/ProdData/Connect200/Classes/loggingapi.jar

/QIBM/ProdData/Connect200/Classes/logging.jar

/QIBM/ProdData/Connect200/Classes/log.jar

/QIBM/ProdData/Connect200/Classes/comibmconnect.jar

/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar

6.2.2 Sample Code:

```
public class MyClass {

    void MyClass() {
    };

    public void sendInvoice() {

        try {

            OutboundRequest orq = new OutboundRequest();

            Properties prop = new Properties();
            FileInputStream propFile = new
             FileInputStream("OutboundRequest.properties");
            prop.load(propFile);

            /*
               Properties object contains values for Instance, Protocol Group,
               Protocol, Protocol Version.
            */
            orq.setRequestFields(prop);
            orq.setRequest(OutboundRequest.INVOICE_REQUEST);
            orq.setApplicationToken("INVOICE,123456");

            int rc = orq.initiateRequest();

        }
      catch (Exception ex)

          System.err.println( "Exception " + ex.getClass().getName() + "  " +
ex.getMessage() );
          ex.printStackTrace();
          System.exit(1);
      }
    }

    public void sendInvoice(String invoiceNumber, String partner) {

        try {
            OutboundRequest orq = new OutboundRequest();

            orq.setInstance("AcmeCorp");
            orq.setProtocolGroup("Ariba CSN");
            orq.setProtocol("cXML:Ariba:1.2");
            orq.setRequest(OutboundRequest.INVOICE_REQUEST);

            orq.setTarget("123456789:DUNS");
            orq.setPartner(partner);

            orq.setApplicationToken("INVOICE,"+invoiceNumber);

            int rc = orq.initiateRequest();
        }
      catch (Exception ex)
      {
          System.err.println( "Exception " + ex.getClass().getName() + "  " +
ex.getMessage() );
          ex.printStackTrace();
          System.exit(1);
      }

    }

}
```

6.3 Sendmessage Connector

The Sendmessage connector can be used to send outbound messages to remote partners from a protocol flow. See the SendMessage connector specification for details.

6.4 Mailbox Cleanup

Expired messages that are still in the message database can be cleaned up by the MailboxCleanup java class. The clean() method can be called to delete all expired messages for all the configured Connect instances. Additionally, the MailboxCleanup class contains a main() method that directly calls the cleanup() method so that the class can be called from a command line or from Qshell.

```
JAVA CLASS('com.ibm.connect.gateway.interfaces.MailboxCleanup')
      CLASSPATH('/QIBM/Proddata/Connect/Gateway/gatewayapi.jar:
      /QIBM/Proddata/Connect200/classes/config.jar:
      /QIBM/Proddata/Connect200/classes/xerces321.jar:
      /QIBM/Proddata/Connect200/tools/tap/tparuntime.jar:
      /QIBM/Proddata/Connect200/classes/connpool.jar:
      /QIBM/Proddata/Connect200/classes/logging.jar')
```

It can also be schedule to run periodically using OS/400 job scheduling. For example the command below will schedule the cleanup to run every night at 1:00 AM.

```
ADDJOBSCDE JOB(CONNECTDG)
      CMD(JAVA CLASS(com.ibm.connect.gateway.interfaces.MailboxCleanup)
      CLASSPATH('/QIBM/Proddata/Connect/Gateway/gatewayapi.jar:
      /QIBM/Proddata/Connect200/classes/config.jar:
      /QIBM/Proddata/Connect200/classes/xerces321.jar:
      /QIBM/Proddata/Connect200/tools/tpa/tparuntime.jar:
      /QIBM/Proddata/Connect200/classes/connpool.jar:
      /QIBM/Proddata/Connect200/classes/logging.jar'))
      FRQ(*WEEKLY) SCDATE(*NONE) SCDDAY(*ALL) SCDTIME(0100)
      TEXT('Connect for iSeries Mailbox cleanup')
```

7.0 Buyer/Supplier Interfaces

The purpose of the Connect partner Supplier Registry is to allow for the authentication and authorization of requests coming from a marketplace or private protocol exchange and to facilitate the mapping of the identities in those requests to corresponding backend application entities. In each request there is an identified target (the provider) and an identified requester (the partner). The Connect product, through the partner Supplier Registry interfaces, not only provides the registry for authenticating the requests and identifying the partner and provider but also allows for the synchronization and mapping between the provider and partner entities as identified in an 'external' protocol request and the corresponding entity as it is known to the 'internal' backend application.

There are two approaches available within the partner provider Registry support in Connect to allow the exchange of information between partner provider entities and backend application entities. The first approach is a registered exit program approach. In this approach, you implement a java class that implements a specific interface where the various methods map to exit points that correspond to specific requests made in the partner provider Administration GUI. These exit points correspond to the create of a partner or provider, the edit of a partner or provider, or the delete of a partner or provider. There are also exit points to allow for the select of backend application entities for the purpose of populating the properties of a partner or provider with that entity's known information. The exit program approach requires that you implement either a provider Exit program, a partner Exit program, or both and then register that exit program for a given instance in the partner provider Administration GUI.

The second approach is a set of APIs that are distinct from the actions taken in the partner provider Administration GUI. These APIs are available to standalone programs or in exit points in backend application code, to effect the partner provider registry under Connect. These APIs allow you to create a provider, create a partner, edit the properties of a provider or partner, retrieve the individual properties of a provider or partner, retrieve a list of partner or suppliers and more. Note that you can also use theses APIs in a registered exit program to make additional changes to the properties of a partner or provider based on how you want that partner or provider to relate to a backend entity.

To develop either a registered exit program, implementing a specific set of interfaces, or to create a standalone java program that issues APIs to alter the contents of the partner provider registry, you must understand that a partner or provider is implemented as a set of properties. Documented below are two key classes, the SupplierProperties class and the BuyerProperties class. These two classes inherit from the java properties object,see java.util.Properties class, and represent a set of name, value pairs. These property classes are used as both input and output in the various methods and are used to exchange information about a provider or partner. The actual data about a registered partner or provider is stored in the partner provider Registry database.

What follows is a discussion of the two approaches.

Registered Exit program:

There are a number of reasons why you might want to implement and register a provider or partner Exit program with the partner provider Administration GUI.

- To create a corresponding backend entity object (a customer, shopper or store) to reflect the newly created partner or provider and store the relationship key in the partner or provider properties
- To do additional data integrity checking on the input of various partner or provider properties via the partner provider Administration GUI

- To map a partner or provider object to an existing backend entity object such as a Store, Merchant, Shopper, Customer, Agent, etc, and to set additional related information in the properties of a partner or provider to reflect that mapping
- To select from and populate a partner or provider from a list of known backend application entities
- To shadow or copy partner or provider data to another instance

As part of the partner provider Registration exit process, which allows for real time exit calls out of the partner provider Administration GUI, we have provided two interface classes: BuyerExit and SupplierExit. The input and output parameters of the methods in these two class interfaces are directly related to the exit points they plug in to in the partner provider Administration GUI. The exit points in the partner provider Administration GUI are the following:

- After the create of a partner or provider, when using the New partner or New provider wizard and pressing the FINISH button.
- After the update of a partner or provider, when doing an edit Properties of a registered partner or provider and pressing OK button.
- Before the delete of a partner or provider, when doing a DELETE of a select partner or provider.
- Prior to moving through the New partner or New provider wizard, if the Select from Other Source has been enabled. This function will call the select exit point, allowing the presentation of a list of backend entities to choose from and then will invoke the populate exit point, so that the exit can set various properties with the content of the backend entity.

Your exit program can choose to implement some or all of these exit points. For example, you may want to implement create, update and delete, but not select and populate from backend entity. For those exit call that you don't implement, you need only provide the appropriate return value, a null pointer or void or boolean.

Once you have created your exit program routine, either partner or provider, you need to register it in the provider partner Administration GUI using the Registered Exit Program link on the sub-task bar. What you will provide is the full package and class name. In order for the Administration GUI, which runs under the Apache server, to successfully load your class, it must be able to find it. This means that your class or corresponding jar file must be found at load time. If you only have a class file, you must store it in the following path:

/QIBM/UserData/HTTPA/admin/webapps/BtoB/web-inf/classes

If you have a jar file, you would put the jar file in the following path or add symbolic link to your jar file in the following path:

/ QIBM/UserData/HTTPA/admin/webapps/BtoB/web-inf/lib

If your class or jar file is put in the appropriate one of these two paths, the class will be found when doing a load during the registration of the exit under the partner provider Administration GUI. Note that you may need to restart the administration server before doing a register of your exit as that is when the classpath for apache is determined.. If you do not choose to put your class or jar in one or the other of these two locations, then you must manually insure that the apache classpath is updated to reflect your class. See the /qibm/proddata/httpa/adminj/conf/workers.properties file, the worker.jni.class_path property for update. If the partner provider Administration GUI can successfully load your class, then the description of your implementation (see getDescriptor) will be presented. If it could not be loaded, registration will fail with the appropriate message.

Provided as one of the interface methods is the isImplemented method. This method is used to interrogate the implementation class to determine if the class has an implementation behind the particular exit points. This is primarily of use with the select and populate, so the partner provider Administration GUI can best determine if those screens should be presented.

The following classes and interfaces can be found in the tpaapi.jar in the following path: /qibm/proddata/connect110/tools/tpa file **(note**: All constants associated with these classes are found in the Appendix):

-com.ibm.connect.tools.tpa.SupplierProperties:

- **Purpose:** This class represents a provider as a set of properties a helper class and extends the **java.util.Properties** class. It is used as input and output in the registered exit interfaces and Supplier Buyer API interfaces. The objective of this class is to make Supplier data available to the Backend Application connectors and also allow Supplier data to be populated from Backend application data.

- The supplier properties object allows for two sets of properties, data and password. The data properties are stored in the Supplier Registry database "as is". The password properties are not stored in the Supplier Registry database, but rather are stored as tokens in the database and it is up to the exit program to cache the real password property value into a secure store. For example, your exit code could use a validation list. You can use any other property information as a key to get to the password you have stored away, but you will not be able to get the real password property from the SupplierProperties object except as input on the create or update Supplier exit, and only then when it is actually input new via the GUI. In all other cases, you get a token password value for that named password property and this value can be used as an indication that the password property was set at one time via the GUI. See the PASSWORD_SET_TOKEN in the appendix. This is the value that will be returned as the named password property value on all retrieves. As password properties are distinct from data properties, there are distinct methods to get, remove or set the property. You can also get a list of the password properties names and process by returned names.

SupplierProperties()

getPasswordPropertiesGetPasswordProperties()

Get a vector of password property names (Strings).

getPasswordProperty(String)

removePasswordProperty(String)

setPasswordProperty(String, String)

-com.ibm.connect.tools.tpa.BuyerProperties:

- **Purpose:** This class represents a buyer as a set of properties a helper class and extends the **java.util.Properties** class. It will be used by the Registered Exit class that implements the BuyerOrgExit interface. This class is used to input Buyer data to or return Buyer data from the B/S Administration GUI. The objective of this class is to make Buyer data available to the Backend Application connectors and also allow Buyer data to be populated from Backend application data.

•The buyer properties object allows for two sets of properties, data and password. The data properties are stored in the Buyer Registry database "as is". The password properties are not stored in the Buyer Registry database, but rather are stored as tokens in the database and it is up to the exit program to cache the real password property value into a secure store. For example, your exit code could use a validation list. You can use any other property information as a key to get to the password you have stored away, but you will not be able to get the real password property value from the BuyerProperties object except as input on the create or update Buyer exit, and only then when it is actually input new via the GUI. In all other cases, you get a token password value for that named password property and this value can be used as an indication that the password property was set at one time via the GUI. See the PASSWORD_SET_TOKEN in the appendix. This is the value that will be returned as the named password property value on all retrieves. As password properties are distinct from data properties, there are distinct methods to get, remove or set the property. You can also get a list of the password properties names and process by returned names.

BuyerProperties()

getPasswordProperties()

Gets a vector of password property names (Strings).

getPasswordProperty(String)

removePasswordProperty(String)

setPasswordProperty(String, String)

-com.ibm.connect.tools.tpa.api.SupplierExit:

•**Purpose:** This class represents an **interface** class. Implementations of this interface are registered under the Buyer Supplier Administration GUI and result in real time exit calls to interface with backend application entities. Backend applications will ship classes that implement this interface in order to provide real time information about Connect Suppliers to the backend application entities.

•**Throws : SupplierBuyerException**

Contains message data to be retrieved and any additional Exception object caught by the implementing routine. Any SupplierBuyerException thrown by an exit program will result in the rollback of the requested GUI operation. So, for example, if the createSupplier method throws the SupplierBuyerException, the newly created Supplier via the GUI will be deleted.

void **createSupplier**(SupplierProperties properties)
This method gets invoked when Finish is pressed during the New Supplier wizard after the supplier has been successfully created.

void **deleteSupplier**(SupplierProperties properties)
This method gets invoked when OK is pressed during the Delete Supplier before the delete supplier is actually performed.

java.lang.String **getDescriptor**()
Get supplier exit program description

void **init**(java.lang.String instanceName, java.lang.String propertiesFile)
Initializes supplier exit with Connect instance name and optional properties file name.

boolean **isImplemented**(int exitType)
Test supplier exit to determine if exit type is fully Implemented.

SupplierProperties **populateSupplierFromBackendEntity**(java.lang.String name)
Populate Supplier properties from named backend business application entity.

java.lang.String[] **selectFromBackendEntities**()
Select from backend business application entities.

void **updateSupplier**(SupplierProperties propertiesNew, SupplierProperties propertiesOld)
This method gets invoked when OK is pressed during the Edit properties of Supplier.

-com.ibm.connect.tools.tpa.api.BuyerExit:

- **Purpose:** This class represents an **interface** class. Implementations of this interface are registered under the Buyer Supplier Administration GUI and result in real time exit calls to interface with backend application entities. Backend applications will ship classes that implement this interface in order to provide real time information about Connect Buyer Organizations and the backend application entities.

- **Throws : SupplierBuyerException**

  Contains message data to be retrieved and any Exception object caught by implementing routine. Any SupplierBuyerException thrown by an exit program will result in the rollback of the requested GUI operation. So, for example, if the createBuyer method throws the SupplierBuyerException, the newly created Supplier via the GUI will be deleted.

void **createBuyer**(BuyerProperties properties)
This method gets invoked when Finish is pressed during the New Buyer wizard, after the buyer has been successfully created.

void **deleteBuyer**(BuyerProperties properties)
This method gets invoked when OK is pressed during the Delete Buyer request before the delete buyer is actually performed.

java.lang.String **getDescriptor**()
Get buyer exit program description

void **init**(java.lang.String instanceName, java.lang.String propertiesFile)
Initializes buyer exit with specific Connect instance name and optional properties file name.

boolean **isImplemented**(int exitType)
Test buyer exit to determine if exit type is fully Implemented.

BuyerProperties **populateBuyerFromBackendEntity**(java.lang.String name)
Populates buyer properties from named backend business application entity.

java.lang.String[] **selectFromBackendEntities**()
Select from backend business application entities.

void **updateBuyer**(BuyerProperties propertiesNew, BuyerProperties propertiesOld)
This method gets invoked when OK is pressed during the Edit Properties of Buyer.

Additional Supplier /Buyer Java APIs for External access:

You can use the Supplier Buyer APIs in a number of ways.

- To make additional changes to a registered Buyer or Supplier's information via a Buyer or Supplier registered exit program
- To prime the buyers or suppliers based on information in a backend application database
- In exit programs associated with a backend application entities to keep Buyer or Supplier data in sync with registered entities in the backend application (like customers, shoppers, merchants, etc)

The following is a set of Supplier/Buyer APIs. The purpose of these APIs is to allow priming of Buyer Organization and Supplier data from existing Backend Application databases or to enable exits in the BackEnd application to update the Supplier/Buyer registration information in real time. Just as in the case of the exit APIs, these APIs only deal with partner data and do NOT relate to protocol information.

The following are the necessary classes, they can be found in the tpaapi.jar in /qibm/proddata/connect110/tools/tpa.

-com.ibm.connect.tools.tpa.SupplierProperties: as referred to in the Register Exit section

-com.ibm.connect.tools.tpa.BuyerProperties: as referred to in the Register Exit section

-com.ibm.connect.tools.tpa.api.SupplierBuyerAPIs:

•**Purpose:** This class represents the Supplier r/Buyer API manager class. Use this class to create, edit or delete Supplier or Buyers organizations. This class deals in partner data only, not with protocol data.

•**Throws : SupplierBuyerException**

Contains message data to be retrieved and any Exception object caught by the implementing routine. Example would be if a supplier name was a duplicate or if the instance name was incorrect. SQL errors.

SupplierBuyerAPIs(java.lang.String instance)
Initiate a Supplier Buyer API manager with a Connect instance name.

void **closeConnection**()
Close Supplier Buyer API manager connection with a Connect instance.

void **createBuyer**(BuyerProperties buyerProperties)
Creates a buyer organization.

void **createSupplier**(SupplierProperties supplierProperties)
Creates a Supplier.

void **deleteBuyer**(java.lang.String buyerName)
Delete registered buyer organization by Name.

BuyerProperties **getBuyerDetails**(java.lang.String buyerName)
Get details about a registered buyer organization.

java.lang.String[] **getBuyerNames**()
Get list of registered buyer organization names.

java.lang.String **getBuyerProperty**(java.lang.String buyerName, java.lang.String propertyName)
Get specific buyer property Value.

SupplierProperties **getSupplierDetails**(java.lang.String supplierName)
Get details about a registered supplier.

java.lang.String[] **getSupplierNames**()
Get list of registered suppliers by name.

java.lang.String **getSupplierProperty**(java.lang.String supplierName, java.lang.String propertyName)
Get a specific supplier property value from named (registered) supplier.

void **removeBuyerProperty**(java.lang.String buyerName, java.lang.String propertyName)
Remove specific buyer property from named (registered) buyer organization.

void **removeSupplierProperty**(java.lang.String supplierName, java.lang.String propertyName)
Remove specific supplier property from named (registered) supplier.

void **updateBuyer**(BuyerProperties buyerProperties)
Update buyer identified within buyer properties object.

void **updateBuyer**(java.lang.String buyerName, BuyerProperties buyerProperties)
Update buyer identified by buyer name parameter.

void **updateBuyerProperty**(java.lang.String buyerName, java.lang.String propertyName, java.lang.String propertyValue)
Update or add specific buyer property to named (registered) buyer organization.

void **updateSupplier**(java.lang.String supplierName, SupplierProperties supplierProperties)
Update supplier identified by supplier name parameter.

void **updateSupplier**(SupplierProperties supplierProperties)
Update supplier identified within supplier properties object itself.

void **updateSupplierProperty**(java.lang.String supplierName, java.lang.String propertyName, java.lang.String propertyValue)
Update or add specific supplier property to named (registered) supplier.

SupplierBuyerException

SupplierBuyerException()

SupplierBuyerException(String)

SupplierBuyerException(String, boolean)

SupplierBuyerException(String[])

SupplierBuyerException(Throwable)

SupplierBuyerException(Vector)

appendMessages(String[])

appendMessages(SupplierBuyerException)

getLocalizedMessage()

getLocalizedMessages()

The returned array may be larger than the number of messages.

isFatal()

printStackTrace()

printStackTrace(PrintStream)

printStackTrace(PrintWriter)

setFatal(boolean)

setLocalizedMessage(String)

toString()

7.1.1 Create Buyer Sample Code

This is a sample program which uses JDBC to read a "CUSTOMER " DB2 table, and creates a buyer for each corresponding customer record. This is a standalone java program that uses the SupplierBuyerAPIs class to interface with Connect Buyer Supplier registry.

```
import com.ibm.connect.tools.tpa.api.*;
import com.ibm.connect.tools.tpa.SupplierProperties;
import com.ibm.connect.tools.tpa.BuyerProperties;
import java.lang.*;
import java.util.*;
import java.sql.*;

public class createBuyers {

   private String InstanceName;
   private final String nativeurl="jdbc:db2:*LOCAL";
   private final String nativeDriver="com.ibm.db2.jdbc.app.DB2Driver";

   public createBuyers(String b2bInstance) {

   ResultSet srs = null;

   InstanceName = b2bInstance;
   try {

     // init a JDBC request to local host database
     Class.forName(nativeDriver);
     Properties p = new Properties();
     p.put("transaction isolation","read uncommitted");
     Connection db2Conn = DriverManager.getConnection(nativeurl,p);

    // The supplier buyer api manager class gives you a connection to the Supplier Buyer
Registry
    // for the named instance
    SupplierBuyerAPIs mgr = new SupplierBuyerAPIs(InstanceName);

    java.sql.PreparedStatement stmt=db2Conn.prepareStatement("SELECT CUSTNA, CUSTNO,
REPNO, CPHONE, CFAX, CADDR, CSTATE, CCITY, CCOUNT, CZIP  FROM  CUSTOMER for fetch
only");
    // Query the customer table
    srs= stmt.executeQuery();
    while (srs.next()==true) {
      // for each customer record, create a buyer properties object
      BuyerProperties buyerProp = new BuyerProperties();
      // Fill in the properties making each customer a buyer, note only BUYER_NAME is required
property
```

```
        // Specify the properities as name, value pairs
        // Set buyer name to customer name
        buyerProp.put((String)BuyerProperties.BUYER_NAME,srs.getString("CUSTNA").trim());
        // Set buyer phone to customer phone
        buyerProp.put((String)BuyerProperties.BUYER_PHONE,srs.getString("CPHONE").trim());
        // set buyer fax to customer fax
        buyerProp.put((String)BuyerProperties.BUYER_FAX,srs.getString("CFAX").trim());
        // set buyer address (first line) to customer address
        buyerProp.put((String)BuyerProperties.BUYER_ADDRESS1,srs.getString("CADDR").trim());
        // set buyer city to customer city
        buyerProp.put((String)BuyerProperties.BUYER_CITY,srs.getString("CCITY").trim());
        // set buyer state to customer state
        buyerProp.put((String)BuyerProperties.BUYER_STATE,srs.getString("CSTATE").trim());
        // set buyer country to customer country
        buyerProp.put((String)BuyerProperties.BUYER_COUNTRY,srs.getString("CCOUNT").trim());
        // set buyer postal code to customer zip
        buyerProp.put((String)BuyerProperties.BUYER_POSTAL,srs.getString("CZIP").trim());
        // Provide for two unique properties, customer number and representative number
        // These properties would surface in the Buyer GUI if the Customize option was used to
        // add these two fields to the screens (see Customize functions, add page/field on GUI
        // note property names are case sensitve and what is specified here must match the screen
    property
        buyerProp.put("Customer_Number",srs.getString("CUSTNO").trim());
        buyerProp.put("Representative_Number",srs.getString("REPNO").trim());

        // Create the buyer with the provided properties
        mgr.createBuyer(buyerProp);
      }

      // close local JDBC handles and connection
      srs.close();
      stmt.close();
      db2Conn.close();

      // close connection to Supplier Buyer API
      mgr.closeConnection();

     }
     catch (Exception e) {
        System.out.println("Exception occurred: " + e.getLocalizedMessage());}
     }
     public static void main(String args[]) {
        // provide instance name as input
        // to execute:  java createBuyers <instancename>
        createBuyers doBuyers = new createBuyers(args[0]);

     }
  }
```

7.1.2 Shadow Buyer Exit Program Sample Code

This is a BuyerExit sample, which when registered as a Buyer Registered Exit program would shadow the create of buyers in one instance to another instance.   The shadow target instance name is provided on the register of the exit, as part of the exit properties file name.   This example uses BuyerExit interface and shows how SupplierBuyerAPIs class can also be used in a exit program.

```
import com.ibm.connect.tools.tpa.api.*;
import com.ibm.connect.tools.tpa.*;
import java.io.*;
import java.lang.*;
import java.util.*;

public class ShadowBuyerExit implements BuyerExit {

   // this class implements the BuyerExit interface and should only be used as a registered exit
for the buyer screens.
   // this class demonstrates how you could use an exit to shadow registered buyers from a
master instance to a shadow instance.
   // the primary purpose of this class is to illustrate the apis/interfaces available

   String currentInstance = null; // this is the instance that the exit was registered in (from the
init method).
   String shadowInstance = null;  // this is the instance that all operations are shadowed to.
   SupplierBuyerAPIs sba = null;  // an instance of the supplier buyer apis class. this is
connected to the "shadowInstance" instance (all instances must be on the same system in the
same connect installation)

   public void init(String instanceName, String propertiesFile) throws SupplierBuyerException {
      // instanceName is the name of the instance this exit was just registered in.
      // copy that to the currentInstance instance variable to keep track of it throughout
      // the lifetime of the instance of this class.
      currentInstance = instanceName;

      // propertiesFile holds the value of the "Properties file name" field on the exit
      // registration page.  You can use it as a name of an external properties file or
       // in this case, it is the name of the instance property itself, that all of the
      // operations are being shadowed to.

       // The value of propertiesFile will be copied to
      // the shadowInstance instance variable to keep track of it throughout the lifetime
      // of the instance of this class.
      shadowInstance = propertiesFile;

      // The SupplierBuyerAPIs class will be used to make a connection to the target shadow
instance.
      try {
         sba = new SupplierBuyerAPIs(shadowInstance);
      } catch(Exception e) {
         // SupplierBuyerAPIs could not be initialized. This is an example of sending a
SupplierBuyerException
          // This error message would show up on the Buyer registration page if target instance
did not exit.
         throw new SupplierBuyerException("Could not connect to shadow instance:
"+shadowInstance+". Make sure this instance exists.");
      }
```

```
    }

    public boolean isImplemented(int exitType) {
        // isImplemented answers whether or not a specific interface is fully implemented. It is
called at exit program load.
        // Given the following setup this exit should be called for all operations except select and
populate. If an exit
        // writer doesn't want to implement specific operations this method should return false for
those operations.
       // Alternatively, the exit would be called with a noop being performed (return of void or
null).
        boolean rv = false;
        switch(exitType) {
        case BUYER_EXIT_SELECT : rv = false; break;
        case BUYER_EXIT_POPULATE : rv = false; break;
        case BUYER_EXIT_CREATE : rv = true; break;
        case BUYER_EXIT_UPDATE : rv = true; break;
        case BUYER_EXIT_DELETE : rv = true; break;
        default: rv = false;
        }
        return rv;
    }

    public String[] selectFromBackendEntities() throws SupplierBuyerException {
        // not implemented for this example. notice that isImplemented will return
        // false for BUYER_EXIT_SELECT, which corresponds to this method.
        return null;
    }

    public BuyerProperties populateBuyerFromBackendEntity(String name) throws
SupplierBuyerException {
        // not implemented for this example. notice that isImplemented will return
        // false for BUYER_EXIT_POPULATE, which corresponds to this method.
         // note that the populate exit always coincides with a previous exit call to select.
        return null;
    }

    public void createBuyer(BuyerProperties properties) throws SupplierBuyerException {
        // this method will be called to shadow the creation of a new buyer from current instance
to the shadow instance.
        // this operation will fail if the buyer being registered has the same name (which is the
assumed error condition here)
        // as one that already exists in the shadow instance.
        try {
            sba.createBuyer(properties);
        } catch(Exception e) {
            String buyerName = null;
            if(properties!=null) {
                    buyerName = (String)properties.get(BuyerProperties.BUYER_NAME);
            }
            throw new SupplierBuyerException("Buyer "+buyerName+" already exists in instance
"+shadowInstance);
        }
    }
    public void updateBuyer(BuyerProperties propertiesNew, BuyerProperties propertiesOld)
throws SupplierBuyerException {
       // updateBuyer may attempt to update a buyer that doesn't exist in the shadow instance.
       // this is not a problem.

       // (might want to actually check to see if it does exist or not, there may be real errors ? )
       String origBuyerName = (String)propertiesOld.get(BuyerProperties.BUYER_NAME);
       try {
          sba.updateBuyer(origBuyerName, propertiesNew);
       } catch(Exception e) {
          String msg[]=new String[2];
          msg[0]=e.getLocalizedMessage();  // get error from updateBuyer request
          msg[1]="Update of shadowed buyer to instance "+ shadowInstance + " failed.";  //
indicate failure on shadow
         throw new SupplierBuyerException(msg);  //  resulting failure messages will show in
Buyer Supplier GUI
       }
    }

    public void deleteBuyer(BuyerProperties properties) throws SupplierBuyerException {
       // deleteBuyer will attempt to delete the buyer in the shadow instance. it may not
       // exist so all errors are ignored.
       String origBuyerName = (String)properties.get(BuyerProperties.BUYER_NAME);
       try {
          sba.deleteBuyer(origBuyerName);   // delete buyer by required property of Buyer_Name
       } catch(Exception e) {
          // not handling anything.
       }
   }

    public String getDescriptor() throws SupplierBuyerException {
       // This method is called to generate the description string used on the registered exit
screens.
       // it can return a null if no information is needed. but the person writing the exit may want
       // to put some information in here that explains what is setup and whether or not the
       // exit program is in a functioning state and what the exit program interfaces with.

       StringBuffer sb = new StringBuffer();
       sb.append("Current Instance = "+currentInstance+"\n");
       sb.append("Shadow Instance = "+shadowInstance+"\n");
       return sb.toString();
    }

}
```

7.1.3 Supplier Populate Exit Sample Code

This is a SupplierExit sample, which when registered as a Supplier Registered Exit program would show how the select and populate from backend application entity would work.   This example uses SupplierExit interface.

```
import java.lang.*;
import java.util.*;
import com.ibm.connect.tools.tpa.*;
import com.ibm.connect.tools.tpa.api.*;

public class SupplierPopulateExit implements SupplierExit {
   //uses Java 1.2 functions

   private Hashtable allSuppliers = null;
   private String[] supplierNames = null;

   public String getDescriptor() {
      // just return a short message explaining this exit program.
      return "This sample class shows how select and populate work.";
   }

   public void init(String instanceName, String propertiesFile) {
      // initialize our internal data store.
      allSuppliers = new Hashtable();

      // our hashtable will store SupplierProperties objects keyed by the supplier name
       // Note that SUPPLIER_NAME and SUPPLIER_DUNS are required for a Supplier, all others
are
       // optional.   Also, you can add any other private properties (name, value) that you want.
       // To view private properties via the GUI, you would customize the Buyer/Supplier GUI
       // screens and add a page with any additional supplier properties.
       // Note property names are case sensitive.

       // Contents is hard coded here but a real example would be to seed this data from some
       // backend application database using JDBC access.
      SupplierProperties supProp = new SupplierProperties();
      supProp.put(SupplierProperties.SUPPLIER_NAME, "Carl's DX");
      supProp.put(SupplierProperties.SUPPLIER_DUNS, "000000001");
       supProp.put(SupplierProperties.SUPPLIER_CONTACT_FIRSTNAME,"Carl");
      allSuppliers.put(supProp.get(SupplierProperties.SUPPLIER_NAME), supProp);

      supProp = new SupplierProperties();
      supProp.put(SupplierProperties.SUPPLIER_NAME, "Joe's Muffler and Brakes Shop");
      supProp.put(SupplierProperties.SUPPLIER_DUNS, "000000002");
       supProp.put(SupplierProperties.SUPPLIER_CONTACT_FIRSTNAME,"Joe");
      allSuppliers.put(supProp.get(SupplierProperties.SUPPLIER_NAME), supProp);

      supProp = new SupplierProperties();
      supProp.put(SupplierProperties.SUPPLIER_NAME, "Small Engine Repair");
      supProp.put(SupplierProperties.SUPPLIER_DUNS, "000000003");
      allSuppliers.put(supProp.get(SupplierProperties.SUPPLIER_NAME), supProp);

      Set keys = allSuppliers.keySet();

      supplierNames = new String[keys.size()];
      supplierNames = (String[])keys.toArray(supplierNames);
   }

   public SupplierProperties populateSupplierFromBackendEntity(String name) {
      // name is the value selected in the GUI on the Populate from Other Source page.
      // it will be one of the values in the supplierNames String array.
      // since the values in supplierNames were gathered from the
      SupplierProperties properties = null;

      properties = (SupplierProperties)allSuppliers.get(name);

      // check that a supplier was found.
      // return an empty SupplierProperties object if no supplier was found.
      if(properties == null) properties = new SupplierProperties();
      return properties;
   }

   public String[] selectFromBackendEntities() {
      return supplierNames;
   }

   public boolean isImplemented(int exitType) {
      // the only methods implemented are selectFromBackendEntities and
     // populatesupplierFromBackendEntity.  return true for those exit points.
      boolean implemented = false;

      if(exitType == SUPPLIER_EXIT_SELECT || exitType == SUPPLIER_EXIT_POPULATE)
implemented = true;

      return implemented;
   }

   public void updateSupplier(SupplierProperties propertiesNew, SupplierProperties
propertiesOld) {
      // not implemented for this sample
   }

   public void createSupplier(SupplierProperties properties) {
      // not implemented for this sample
   }

   public void deleteSupplier(SupplierProperties properties) {
      // not implemented for this sample
   }
}
```

A.1 Connector Constants

BIG_DECIMAL
: int representation of Big Decimal field type.

BOOLEAN
: int representation of boolean field type.

BYTE
: int representation of byte field type.

BYTE_ARRAY
: int representation of byte[] field type.

CHAR
: int representation of char field type. **Deprecated.**

DOUBLE
: int representation of double field type.

FLOAT
: int representation of float field type.

INT
: int representation of int field type.

LONG
: int representation of long field type.

PACKED
: int representation of packed field type. **Deprecated.**

SHORT
: int representation of short field type.

STRING
: int representation of string field type.

STRUCT
: int representation of struct field type.

ZONED
: int representation of zoned field type. **Deprecated.**

A.2 Header Constants

`AGENT_AUTH_TYPE`
The agent's authentication type.

`AGENT_DOMAIN`
The agent domain.

`AGENT_ORG_DOMAIN`
The agent's organization domain.

`AGENT_ORG_ID`
The agent's organization ID.

`AGENT_ORG_NAME`
The agent's organization.

`AGENT_ORG_TOKEN`
The agent's organization token.

`AGENT_USERID`
The agent's userid.

`APP_ERROR_INFO_CODE`
Error information code for application.

`APP_ERROR_INFO_STRING`
Error information string for application.

`APP_TOKEN`
The application token which can be used to keep track of information related to an outbound message.

`AUDIT_ASSOCIATED_UNIQUE_ID`
Unique ID of a request associated with this request NOTE: This constant is added as a value for field PROPERTIES_MESSAGE_NAME in table QABET_OMH_MESSAGE_PROPERTIES, so it is limited to a length of 40 bytes

`AUDIT_BUYER`
Deprecated. *Use AUDIT_PARTNER*

`AUDIT_ERROR_OCCURRED`
When set to a non-null value, will cause an indication to be set in the AuditHeader to indicate that an error has occurred.

AUDIT_FLOWMANAGER_CALLS
The value specified for this field should be numeric -- it will be set in the AuditHeader to indicate how many times the flow manager has been called for this unique ID

AUDIT_MARKETPLACE
DEPRECATED -- The audit value for the marketplace.

AUDIT_PARTNER
The audit value for partner

AUDIT_PROTOCOL
The audit value for the protocol.

AUDIT_PROVIDER
The audit value for the provider.

AUDIT_RECEIVED_TIMESTAMP
The timestamp used to record when the value was received.

AUDIT_REQUEST
The audit value for the request.

AUDIT_REQUEST_TYPE
The audit value for the request type.

AUDIT_SENT_TIMESTAMP
The timestamp used to record when the value was sent.

AUDIT_SUPPLIER
Deprecated. *Use AUDIT_PROVIDER*

AUDIT_UNIQUE_ID
A unique ID used for auditing.

BUFFER_FORMAT_IN
&&&& NEED DESCRIPTION &&&&&

BUFFER_FORMAT_OUT
&&&& NEED DESCRIPTION &&&&&

BUYER_DEPT_KEY
The buyer's dept.

BUYER_DEPT_TYPE
The buyer's dept.

BUYER_ORG_DOMAIN
Deprecated. *Use PARTNER_ORG_DOMAIN*

BUYER_ORG_ID
Deprecated. *Use PARTNER_ORG_ID*

BUYER_ORG_NAME
Deprecated. *Use PARTNER_ORG_NAME*

BUYER_ORG_TOKEN
Deprecated. *Now stored in partner context*

BUYER_SESSION_ID
Deprecated.

BUYER_TRANSPORT_MSG_ID
Deprecated.

CONTENT_REQUEST
The content request.

CONTENT_REQUEST_TYPE
Deprecated. *Now part of CONTENT_REQUEST*

CURRENT_STEP_NAME
Current step name.

DEPLOY_DESTINATION
&&&& NEED DESCRIPTION &&&&&

DEPLOY_DIRECTORY
&&&& NEED DESCRIPTION &&&&&

DEPLOY_THEME
The theme name.

DEPLOY_TYPE
&&&& NEED DESCRIPTION &&&&&

FAILED_STEP_NAME
Step name which failed.

FLOW_INDEX_AUDIT_POINT_INDEX
The following should be added by the flow manager to all of its audit point values when creating audit records.

FLOW_NAME
The name of the flow currently running

FLOW_NUMBER
Flow number value

FM_ERROR_INFO_CODE
Flow engine error information code.

FM_ERROR_INFO_STRING
Flow engine error information string.

GATEWAY_INSTANCE
The gateway instance name.

GATEWAY_TYPE
The gateway type.

GATEWAY_VERSION
The gateway version.

HEADER_BASE
IN_ERROR_PATH
Flow engine error path key.

INBOUND_TRANSPORT_CONTENT_ENCODING
The inbound transport content encoding value.

INBOUND_TRANSPORT_CONTENT_TYPE
The inbound transport content type.

INBOUND_TRANSPORT_SENDER_INFO
The inbound transport sender information.

INBOUND_TRANSPORT_URL
The inbound transport URL.

INBOUND_TRANSPORT_URL_NAME
The inbound transport URL name.

MARKETPLACE
Deprecated. *Use PROTOCOL_GROUP*

MAX_NUMBER_OF_TRACE_FILES
Maximum number of trace files available for use by this gateway instance

MAX_TRACE_FILE_SIZE
Maximum trace file size for the trace files associated with this instance.

MESSAGE_DELIVERY
The message delivery.

MESSAGE_HANDLER_ID
The message handler ID.

MESSAGE_ID
The unique valued used for item potentcy.

MSG_QUEUE_TIMEOUT
&&&& NEED DESCRIPTION &&&&&

MSG_RETENTION_PERIOD
Message retention period (in days)

OMH_RETRIES
Number of OMH retries `static`

OMH_RETRY_INTERVAL
OMH retry interval (in seconds)

OUTBOUND_TRANSPORT_CONTENT_ENCODING
The outnbound transport content encoding value.

OUTBOUND_TRANSPORT_CONTENT_TYPE
The outbound transport content type.

OUTBOUND_TRANSPORT_SENDER_INFO
The outbound transport sender information.

OUTBOUND_TRANSPORT_URL
The outbound transport URL.

OUTBOUND_TRANSPORT_URL_NAME
The outbound transport URL name.

PARENT_CONTENT_REQUEST
Request being processed in parent flow

PARENT_PROTOCOL
Protocol of parent flow

PARENT_REQUEST_TOKEN
Request token of request in parent flow

PARTNER_ORG_DOMAIN
The partner organization domain.

PARTNER_ORG_ID
The partner organization ID.

PARTNER_ORG_NAME
The partner organization.

PARTNER_SESSION_ID
The partner's session ID.

PARTNER_TRANSPORT_MSG_ID
The partner's transport message ID.

PASSWORD
Password

POSTBACK_URL
The postback URL.

PRODDATA_PATH
&&&& NEED DESCRIPTION &&&&&

PROTOCOL
The protocol.

PROTOCOL_GROUP
The protocol group.

PROTOCOL_SUBTYPE
Deprecated. *Now part of PROTOCOL*

PROTOCOL_TYPE
Deprecated. *Use PROTOCOL*

PROTOCOL_VERSION
Deprecated. *Now part of PROTOCOL*

PROVIDER_ORG_DOMAIN
The provider organization domain.

PROVIDER_ORG_ID
The provider organization ID.

PROVIDER_ORG_NAME
The provider organization.

REQUEST_METHOD
&&&& NEED DESCRIPTION &&&&&

REQUEST_TOKEN
The request token which represents several values including protocol, subtype, version, marketplace, etc.

REQUISITIONER_ID
The requisitioner's ID.

REQUISITIONER_NAME
The requisitioner.

RESPONSE_SENT
Flow engine response sent key.

RESTART
Flow engine restart key.

SERVLET_NAME
&&&& NEED DESCRIPTION &&&&&

STEP_APP_ERROR_INFO_CODE
Error information code for application step.

STEP_APP_ERROR_INFO_STRING
Error information string for application step.

SUPPLIER_ORG_DOMAIN
Deprecated. *Use PROVIDER_ORG_DOMAIN*

SUPPLIER_ORG_ID
Deprecated. *Use PROVIDER_ORG_ID*

SUPPLIER_ORG_NAME
Deprecated. *Use PROVIDER_ORG_NAME*

SUPPLIER_ORG_TOKEN
Deprecated. *Now stored in provider context*

TRACE_ENABLED
Indicates whether logging to the trace files has been enabled for this instance.

USERDATA_PATH
&&&& NEED DESCRIPTION &&&&&

`VALIDATE_INPUT`

Indicates whether validation of inbound XML documents should be performed.

`VALIDATE_OUTPUT`

Indicates whether validation of outbound XML documents should be performed

A.3 Buyer/Supplier Constants

A.3.1.1 BuyerExit

BUYER_EXIT_CREATE

Constant used to test if Create Buyer exit is fully implemented

BUYER_EXIT_DELETE

Constant used to test if Delete Buyer exit is fully implemented

BUYER_EXIT_POPULATE

Constant used to test if Populate from Backend Application Entity is fully implementation

BUYER_EXIT_SELECT

Constant used to test if Select from Backend Application Entity is fully implemented

BUYER_EXIT_UPDATE

Constant used to test if Update Buyer exit is fully implemented

A.3.1.2 SupplierExit

SUPPLIER_EXIT_CREATE

Constant used to test if Create Supplier exit is fully implemented

SUPPLIER_EXIT_DELETE

Constant used to test if Delete Supplier exit is fully implemented

SUPPLIER_EXIT_POPULATE

Constant used to test if Populate from Backend Application Entity is fully implementation

SUPPLIER_EXIT_SELECT

Constant used to test if Select from Backend Application Entity is fully implemented

SUPPLIER_EXIT_UPDATE

Constant used to test if Update Supplier exit is fully implemented

A.3.1.3 BuyerProperties

BUYER_ADDRESS1

BUYER_ADDRESS2

BUYER_ADDRESS3

BUYER_BILLTO_ADDRESS1

BUYER_BILLTO_ADDRESS2

BUYER_BILLTO_ADDRESS3

BUYER_BILLTO_CITY

BUYER_BILLTO_CODE

BUYER_BILLTO_COUNTRY

BUYER_BILLTO_COUNTRY_ISO

BUYER_BILLTO_NAME

BUYER_BILLTO_POSTAL

BUYER_BILLTO_STATE

BUYER_CITY

BUYER_CONTACT_EMAIL1

BUYER_CONTACT_EMAIL2

BUYER_CONTACT_FAX

BUYER_CONTACT_FIRSTNAME

BUYER_CONTACT_LASTNAME

BUYER_CONTACT_MIDDLENAME

BUYER_CONTACT_PHONE1

BUYER_CONTACT_PHONE2

BUYER_CONTACT_TITLE

BUYER_COSTCENTER_ADDRESS1

BUYER_COSTCENTER_ADDRESS2

BUYER_COSTCENTER_ADDRESS3

BUYER_COSTCENTER_CITY

BUYER_COSTCENTER_CODE

BUYER_COSTCENTER_COUNTRY

BUYER_COSTCENTER_COUNTRY_ISO

BUYER_COSTCENTER_NAME

BUYER_COSTCENTER_POSTAL

BUYER_COSTCENTER_STATE

BUYER_COUNTRY

BUYER_COUNTRY_ISO

BUYER_DESCRIPTION

BUYER_DUNS

BUYER_EMAIL

BUYER_FAX

BUYER_LAST_UPDATE

BUYER_NAME

BUYER_PHONE

BUYER_POSTAL

BUYER_REFNO

BUYER_SHIPTO_ADDRESS1

BUYER_SHIPTO_ADDRESS2

BUYER_SHIPTO_ADDRESS3

BUYER_SHIPTO_CITY

BUYER_SHIPTO_CODE

BUYER_SHIPTO_COUNTRY

BUYER_SHIPTO_COUNTRY_ISO

BUYER_SHIPTO_NAME

BUYER_SHIPTO_POSTAL

BUYER_SHIPTO_STATE

BUYER_STATE

PASSWORD_SET_TOKEN

Password properties can be input on create or update requests and cached away by custom exit routines but, to maintain security, they cannot be returned directly as properties of the Buyer.

WCS_SHOPPER_NAME

A.3.1.4 SupplierProperties

PASSWORD_SET_TOKEN

Password properties can be input on create or update requests and cached away by custom exit routines but, to maintain security, they cannot be returned directly as properties of the Supplier.

SUPPLIER_ADDRESS1

SUPPLIER_ADDRESS2

SUPPLIER_ADDRESS3

SUPPLIER_CITY

SUPPLIER_CONTACT_EMAIL1

SUPPLIER_CONTACT_EMAIL2

SUPPLIER_CONTACT_FAX

SUPPLIER_CONTACT_FAX_AREA

SUPPLIER_CONTACT_FAX_CTRY

SUPPLIER_CONTACT_FAX_MAIN

SUPPLIER_CONTACT_FIRSTNAME

SUPPLIER_CONTACT_LASTNAME

SUPPLIER_CONTACT_MIDDLENAME

SUPPLIER_CONTACT_PHONE1

SUPPLIER_CONTACT_PHONE1_AREA

SUPPLIER_CONTACT_PHONE1_CTRY

SUPPLIER_CONTACT_PHONE1_MAIN

SUPPLIER_CONTACT_PHONE2

SUPPLIER_CONTACT_PHONE2_AREA

SUPPLIER_CONTACT_PHONE2_CTRY

SUPPLIER_CONTACT_PHONE2_MAIN

SUPPLIER_CONTACT_TITLE

SUPPLIER_COUNTRY

SUPPLIER_COUNTRY_ISO

SUPPLIER_CURRENCY

SUPPLIER_DESCRIPTION

SUPPLIER_DUNS

SUPPLIER_EMAIL

SUPPLIER_FAX

SUPPLIER_FAX_AREA

SUPPLIER_FAX_CTRY

SUPPLIER_FAX_MAIN

SUPPLIER_LAST_UPDATE

SUPPLIER_NAME

SUPPLIER_PHONE

SUPPLIER_PHONE_AREA

SUPPLIER_PHONE_CTRY

SUPPLIER_PHONE_MAIN

SUPPLIER_POSTAL

SUPPLIER_REFNO

SUPPLIER_STATE

SUPPLIER_UNSPSC

SUPPLIER_URL

WCS_MERCHANT_NAME

WCS_MERCHANT_REFNO

A.4 Custom Protocol Sample

The Custom Protocol Sample is available in the Connect Web page at the following URL.

http://www-1.ibm.com/servers/eserver/iseries/btob/connect/devtools.htm

A.5 Connect for iSeries Jar files

All jar files that contain the classes for the Connect for iSeries APIs are available in the directory where the product is installed. The path where to find the jar file is:

/QIBM/Proddata/Connect200

The jar file names in the table below are relative to the above path.

Jar file names	Classes
Classes/flowmanagerapi.jar	com/ibm/connect/flowmanager/interfaces/B2BExit.class com/ibm/connect/flowmanager/interfaces/BindException.class com/ibm/connect/flowmanager/interfaces/ConnectorConstants.class com/ibm/connect/flowmanager/interfaces/ConnectorParm.class com/ibm/connect/flowmanager/interfaces/FMOperatorInterface.class com/ibm/connect/flowmanager/interfaces/GetFieldException.class com/ibm/connect/flowmanager/interfaces/JavaConnectorInterface.class com/ibm/connect/flowmanager/interfaces/JavaConnectorResult.class com/ibm/connect/flowmanager/interfaces/JavaProgramConnectorInterface.class com/ibm/connect/flowmanager/interfaces/MapCursor.class com/ibm/connect/flowmanager/interfaces/OperandParm.class com/ibm/connect/flowmanager/interfaces/OperatorException.class com/ibm/connect/flowmanager/interfaces/ProgramConnectorParm.class com/ibm/connect/flowmanager/interfaces/SetFieldException.class com/ibm/connect/flowmanager/metadata/Field.class
Classes/loggingapi.jar	com/ibm/connect/logging/interfaces/UserLogManager.class
Classes/config.jar	com/ibm/connect/config/B2BException.class com/ibm/connect/config/B2BMarketplaceObject.class com/ibm/connect/config/B2BMarketplacesObject.class com/ibm/connect/config/B2BProtocolObject.class com/ibm/connect/config/B2BProtocolsObject.class com/ibm/connect/config/B2BRequestObject.class com/ibm/connect/config/B2BRequestsObject.class com/ibm/connect/config/B2BServletObject.class com/ibm/connect/config/B2BServletParameterObject.class com/ibm/connect/config/B2BServletsObject.class

	com/ibm/connect/config/IB2BPublicInstanceRegistry.class com/ibm/connect/config/B2BPublicInstanceRegistry.class
Gateway/gatewayAPI.jar	com/ibm/connect/gateway/interfaces/ConnectGatewayException.class com/ibm/connect/gateway/interfaces/GatewayConstants.class com/ibm/connect/gateway/interfaces/InvalidFieldException.class com/ibm/connect/gateway/interfaces/MailboxCleanup.class com/ibm/connect/gateway/interfaces/OutboundMessageException.class com/ibm/connect/gateway/interfaces/OutboundRequest.class
Tools/TPA/tpaapi.jar	com/ibm/connect/tools/tpa/BuyerProperties.class com/ibm/connect/tools/tpa/SupplierProperties.class com/ibm/connect/tools/tpa/api/BuyerExit.class com/ibm/connect/tools/tpa/api/SupplierBuyerAPIs.class com/ibm/connect/tools/tpa/api/SupplierBuyerException.class com/ibm/connect/tools/tpa/api/SupplierExit.class

"END OF DOCUMENT"