Connect for iSeries Custom Protocol Implementation Guide

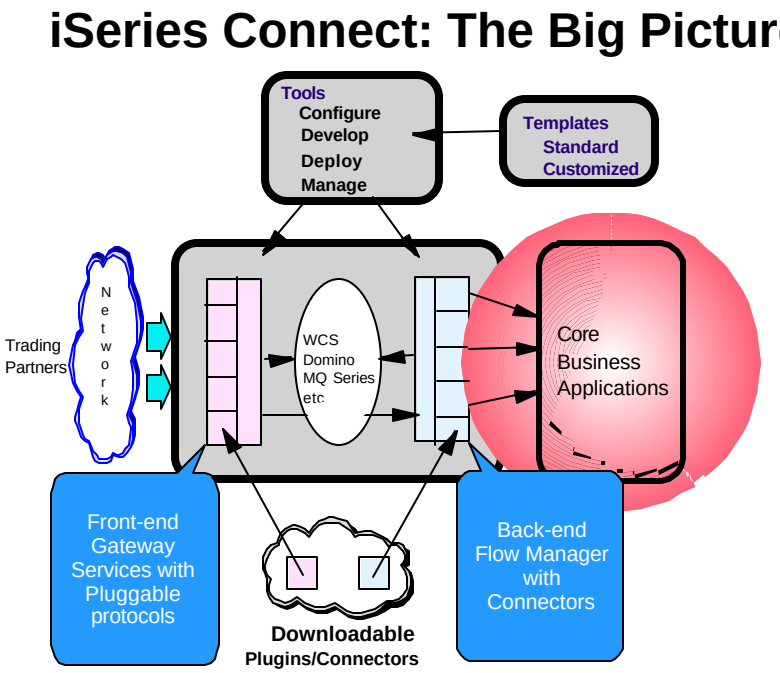
Document Version 1.1.2
September 2001

Introduction

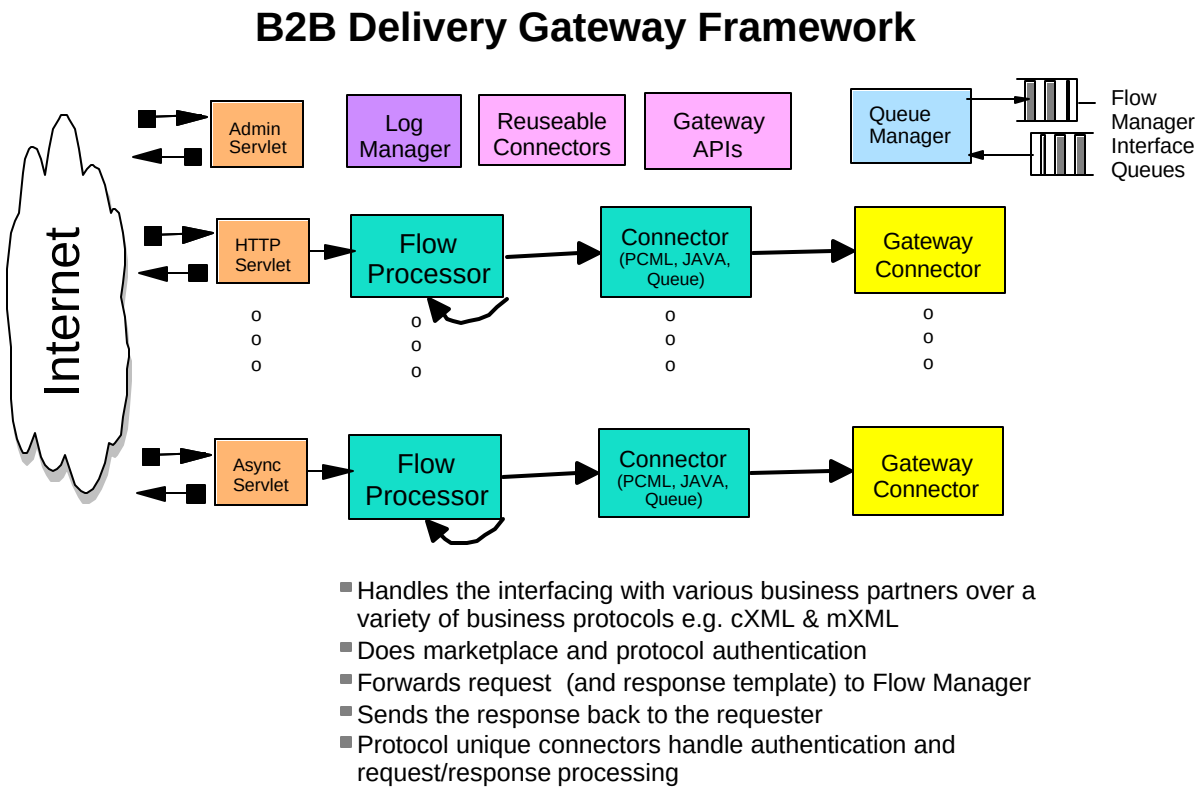
Connect for iSeries (referred to as Connect in this document) is a licensed program product that became generally available in February 2001 (version 1.0). It provides a framework to easily connect business applications with supported BtoB Marketplace protocols such as Ariba's cXML and Metiom's mXML. In Version 1.1 (Available 3Q01), function is provided for customers and ISVs to implement their own custom protocols, allowing connectivity to applications and/or marketplaces not yet supported by Connect. This document will examine the custom protocol capabilities of Connect and provide step-by-step instructions and examples on how to create and deploy a custom protocol. This document assumes the reader is familiar with internet technologies such as Java, HTTP servers, servlet engines, servlet programming, XML **and the Connect for iSeries product**.

Connect for iSeries Product Architecture

To gain understanding of custom protocols and implementing them via Connect, let's start by taking a brief look at the Connect product architecture. 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" (protocol), like cXML or mXML. (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.



This paper will concentrate on the capabilities of the Gateway and how you can use it to implement a custom protocol.



Overview of the Gateway

The purpose of the Gateway is to handle connecting to trading partners. As part of doing that, it should relieve the Flow Manager and it's connectors from knowing anything about the particular protocol that is being used.

Connect for iSeries was designed to handle BtoB protocols that use XML to describe the request. The Gateway can handle other forms of data, such as EDI, if the data is converted to an XML form once it's received, but typically BtoB protocols will use XML. The Gateway handles all aspects of the BtoB 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 BtoB request. The Gateway is implemented as a set of servlets that run in a servlet engine which can be provided by Lotus Domino or Webshpere Application Server.

Now, let's take a real simple look at what the Gateway does. BtoB requests are sent to the Gateway using the HTTP protocol. The Gateway servlets receive the BtoB requests, start up the Flow Engine

and hand off the request to a set of connectors to process the request. Once the request is processed and a response is constructed, the Gateway servlets return the response to the caller.

HTTPServlet servlet

To receive inbound requests the Gateway uses a servlet called the HTTPServlet. This servlet can be used to receive data as name-value pairs, POST data, or no data at all. Let's look at each case individually.

Name-Value Pair (nvp)

Some BtoB protocols pass data to the receiving HTTP server, in our case the Gateway, by passing name-value pairs. Typically, this is used on catalog type requests. The remote trading partner is given a URL to a general catalog and the specific supplier and buyer identities are passed to the catalog application as name-value pairs on the URL. For instance, a URL might look like this: http://btob.mybusiness.com/servlets/catalog?supplier=mybusiness&buyer=mybuyer. The URL and name-value pairs are typically configured in either the marketplace software or in the buyers procurement software. The name-value pairs are converted to nvpML by the HTTPServlet. The nvpML is then placed in the "input message byte array" for processing by the Gateway connectors.

POST data

By far the majority of BtoB requests send their data to the Gateway by using POST data and the most common form of POST data is XML. A good example of this use is cXML. When the HTTPServlet receives a cXML request it takes the POST data and copies it into the input message byte array for processing by the Gateway connectors. For an example of using POST data and XML see the cXML Users Guide available at http://www.cxml.org.

No Data

Some BtoB requests send their data in the HTTP headers. In this case the HTTPServlet doesn't do any data processing. It's left up to the Gateway connectors to access the HTTP headers by retreiving the HTTPServletRequest object and calling the appropriate access methods.

Gateway Flows

Once a request is received through the HTTPServlet and the servlet has done the necessary processing, the HTTPServlet starts the Gateway Flow Engine and intiates a Gateway Flow. A Gateway flow is a sequence of one or more Gateway connectors and/or copy/decision steps (copy/decision steps are explained in detail in the Connect for iSeries Programming Guide). The Gateway Flow Engine currently only supports Java connectors, therefore only Java connectors are being used in the Gateway throughout this paper.

A Gateway flow is called a flow cycle. The HTTPServlet is configured to start a single flow, however multiple HTTPServlets can point to the same flow. The design of the flow cycles are based on the

capabilities of procurement software used and the contents of the data that is sent to the Gateway. For instance, the procurement software 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 decide 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 flow cycle. We chose to 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.

The Gateway and Custom Protocols

Gateway Flow design guidelines for a custom protocol

Although each BtoB protocol is different and each request is unique, Gateway flow cycles should follow a basic design structure. We'll break down the Gateway flow into a sequence of typical steps and then discuss each individually. The Gateway flow cycle 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, we suggest that each of the following steps be implemented as its own connector when building your own protocol flow:

- XML validation and parsing of incoming request
- Inbound Logging
- Authentication
- Authorization
- Request DOM Header Generation (setting buyer/supplier and request information)
- Response DOM Element Priming
- Queue the request/response
- Error Checking
- Set Outbound Status
- Outbound Logging
- Error Handling
- Return to the servlet

While this seems like a large list of connectors to write, it is our intent that you shouldn't have to write them all from scratch and hopefully, use some of the connectors provided with the product as is. Later in the document as we go through an example custom protocol implementation, the connectors will be described in greater detail. This will enable you to better assess which connectors you will need to implement. It will also help you decide (for those connectors that you DO choose to implement) if they can be used as is or how they need to be modified.

Enough already!! How do we do this?

So far we've talked about Connect's architecture, how the Gateway works and also stated some generic guidelines for which steps a custom protocol must perform. These steps are the heart of the protocol where the real work gets done during runtime. To get to the point where you have written connectors and integrated them into your Connect runtime environment, you must perform a series of higher level tasks. These tasks are what will be described in this paper (among these tasks will be a description of how to write connectors). Here are the custom protocol implementation tasks that will be described in this document:

Tasks Required to Define a Custom Protocol

0) Clearly understand your objective for your new protocol.

1) Determine the name, subtype and version of your new protocol.

2) Create a new directory that will be the location for all of the artifacts you will be creating for your custom protocol.

3) Create a Document Type Definition file (DTD) to define the request/response data that you want to send/receive via your new protocol. (Obviously, this step can be skipped if you already have a DTD created)

4) Generate an example XML request/response file that adheres to the DTD created in step #3.

5) Create a Request/Response Message Format (file extension "RequestMsg") for each defined request type that describes the protocol runtime mappings.

6) For each request in your custom protocol, determine the flow steps necessary for proper handling of the request. The steps considered will be influenced by how you answered question 0 (who the trading partner(s) is (are), system topology, etc.)

7) Create an AppConnector definition file for each of the Gateway connector programs.

8) Assemble the flow steps defined in step 6 to create the ProtocolFlow file

9) Write the java connector programs to fulfill the necessary flow steps for your custom protocol.

10) Decide what data should be collected by the Buyer and Supplier GUI and create a Protocol Data Model definition file

11) Create an XML file called, ProtocolDefinition.xml. (This XML document describes the protocol that will be added to the Connect installation for use by Connect instances. The DTD for this file is the Protocol.DTD that is shipped with Connect.**)**

12) Add support for the new protocol to your Connect installation.

13) Create a new instance with the new protocol

14) Create and deploy a process flow that runs in the Flowmanager

15) Start the new instance and test it

Custom Protocol Implementation-Learning by Example

At this point, you are ready to start down the exciting road of creating your own BtoB protocol for Connect. Please note that this document is making a big assumption that you must make sure is correct. That assumption is that **you have already installed Connect on your iSeries and that you have successfully run the samples that are shipped with the product to verify that you have a solid, working installation of Connect.** It is in your best interest to make sure that this has been done. If not, it could lead you down tangents of debug effort that would be difficult, time consuming and irrelevant. Obviously, this should be avoided if at all possible, so please do not continue until you've done that. (For additional information on how to install, configure and run the samples, see the documentation for the Connect for iSeries Version 1.1 release.

In order to more clearly describe the steps involved with writing a custom protocol, an example has been created. This example will start off at the most basic level--just making something work. Following that will be some additional steps--making modifications/enhancements to what was done previously, ultimately resulting in a full custom protocol design. While these examples cannot cover all possible custom protocol designs, it is intended that there is enough information here to cover a majority of things that implementers will encounter when creating their own custom protocol. It is also intended to help the implementer become familiar enough with the Connect flow design such that they will be able to solve the things that aren't covered here.

Custom Protocol Example: A simple request/response protocol
Complexity Level: This is the "hello world"program of custom protocol creation

Following are the details behind the custom protocol steps that we took to implement our very own protocol to handle a request and provide a response. The simple request consists of querying the

amount of insurance coverage that a particular person has. For this case, the response generated by the flowmanger always returns an amount of $50,000. (the response generated by the flowmanager is uninteresting because by the time the flowmanager is involved, the gateway has done the necessary steps to support the new protocol. Since this document is specifically addressing custom protocol development, the responses generated by the flowmanager in these examples will consist of the minimum amount of function to provide meaningful responses to portray the protocol.) Keep in mind that this is a most basic example that really won't make any sense to insurance experts. However, that is not the goal of this example so it doesn't really matter. What does matter is that by going through this example the concepts are clearly portrayed such that the reader can gain a basic understanding of the steps required to successfully implement a valid custom protocol of their own. More complex examples will follow, but it is important to get the basics down first. "Walk before you run", "Rome was not built in a day", etc., etc., etc. With that, let's get started.

0. Clearly understand what your objective is for the new protocol.

It was once said *"If you can't write it down, you can't program it"*. When developing custom protocols, this is most certainly true. *(Note to reader: This may be one of the more time-consuming steps. Being absolutely sure of your environment and request/response definition will make for a better protocol design and an easier implementation.)* So, before we REALLY get into building our own protocol, it is imperative that we answer the following questions:

What is the system topology? (behind the same firewall, same company? On our own company's secure intranet? On the internet?)

The following paragraphs discuss the different topology scenarios that may exist within your target environment and what things you need to consider for each scenario.

Application to Application Within a Single System

You may require to "connect" to different applications on the same system. For example, you may want to tie your Enterprise Resource Planning application with your Supply Chain Management application. In this case, it would be beneficial to use Connect for iSeries for its data mapping functionality. You may or may not need to develop an authentication or authorization phase of the custom protocol (possibly relying on object level security)

Using Connect between applications on the same system.

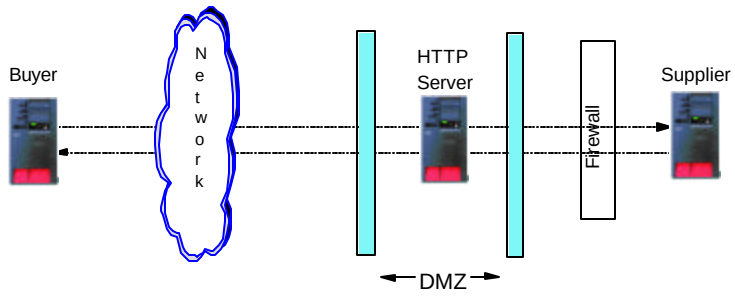
Two Systems on the Same Company Intranet

This topology might be used when you are attempting to automate a business process that will require the use of different software packages, i.e. the output from one application is input to the other. The need for authentication, authorization and a firewall depends upon your installation. It is always recommended that you use a secure communication method between systems such as TCP/IP Secure Sockets Layer (SSL).

Two Systems Over the Internet

In this example, you can assume that you must authenticate and authorize any requests coming in due to the openness of the internet. You may or may not have a firewall set up to protect your system from unwanted access.

Your business may require a topology where your business application and access to the internet are located on separate systems to further protect your core business data from unauthorized access. Connect for iSeries could be installed on both supplier systems where the Gateway and HTTP server are located outside the firewall in the Demilitarized Zone (DMZ) and the back-end flow manager would reside inside the firewall.



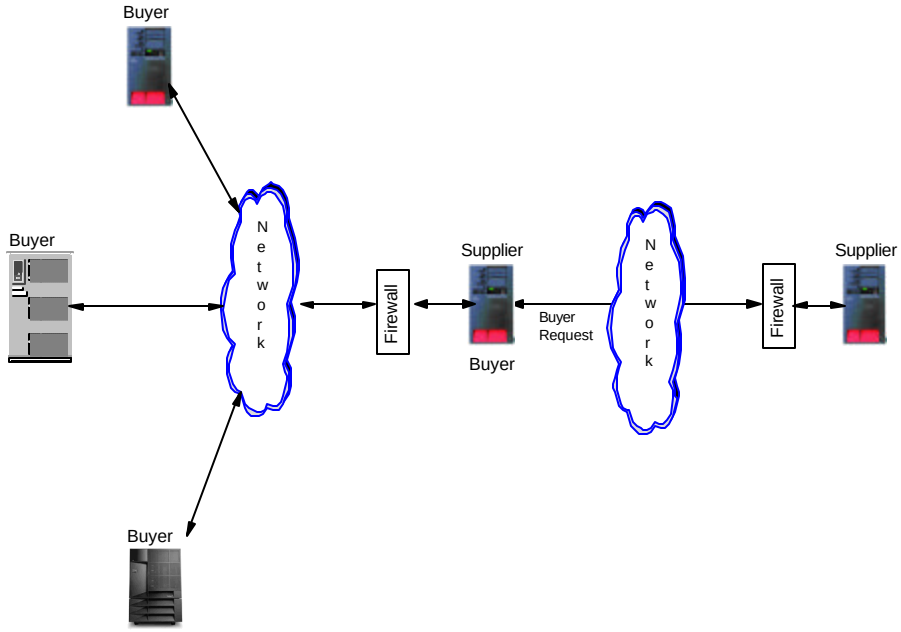
Buyer/supplier Communication with DMZ configuration with Connect.

Multiple Buyers Over the Internet

Because you are again communicating over the internet, you must authenticate and authorize any requests coming in.

Multiple Buyers and one Supplier

In many situations, a supplier, in order to fulfill an order request, also functions as a buyer requisitioning an order to another supplier.



Using Connect to connect to multiple buyers also to pass a buyer request to another supplier through the Internet.

In this case, your custom protocol can be used to communicate between you and your buyers and the business application which is fulfilling the order request can pass the request along to another iSeries that has Connect for iSeries deployed. In the case of the buyers to supplier and supplier/buyer to supplier, a custom protocol can be defined to the specifications that are required.

Our Answer for this example: Behind our own firewall.

What if it reached outside our firewall to the internet? This would change how closely you have to guard access for incoming requests. If there's a way that anyone could be coming in through a non-secure connection, you would have to provide the appropriate amount of authentication to insure the proper security. Because we are behind our own firewall, we are assured that no rogue requests will come in from an outside source.

Who are the requesters/responders in your environment (1-to-1 or 1-to-many relationship?)

You will need to set the stage for how your custom protocol solution is going to be deployed. If you are a supplier, who will you be working with to make your product (goods or services) available? Is it

single or multiple trading partners? If multiple partners, can you create just one custom protocol that will work for all of them or do you need unique ones for each partner?
The answer for this first example: A trusted 1-to-1 relationship (a single partner).
What if it was a 1-to-many relationship? A 1-to-many relationship introduces complexities that typically require authorization checking to insure that the requester coming in is truly allowed to make the requests that they are requesting. Because our example is a trusted 1-to-1 relationship, no authentication checking is done.

What types of requests/responses do you want to send and receive?
What types of request and responses are going to be passed between you and your partner? Is your business application such that you receive and fulfill orders? Or, do you receive requests for quotes or reports to be delivered via online, fax or postage mail? You will want to work out the details of the requests and responses that will flow between you and your trading partners.
Here are some example request/response pairs:

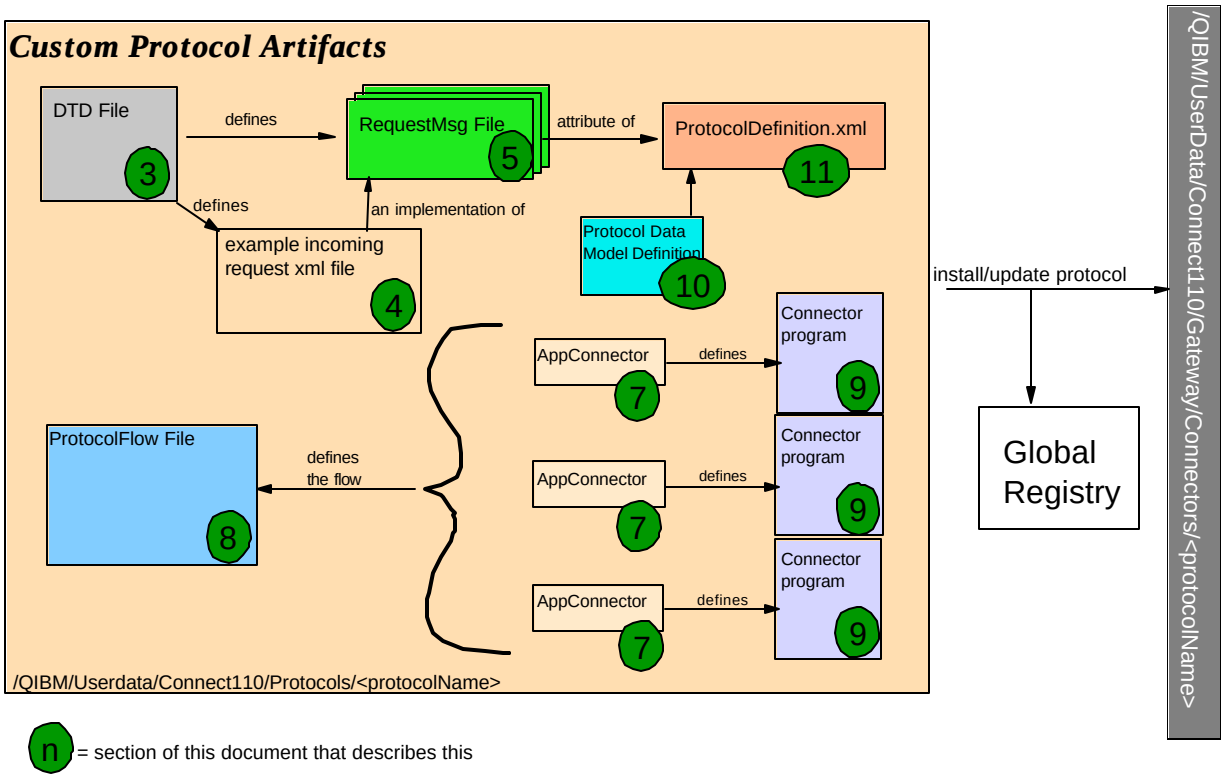
Request	**Response**
GetInsuranceQuote *pass in medical liability, deductible, comprehensive*	Insurance Quote *return cost of insurance*
GetInsuranceCoverageRequest *pass in policy number or name*	InsuranceCoverageAmount *return policy coverage amount*
GetAutomotivePartCostEstimate *pass in part ID or serial number and quantity*	PartCostEstimateResponse *return part estimate quote*
Pharmaceutical ReOrder *pass in drug name, ID, quantity*	OrderResponse *return cost, quantity, status*

Our Answer: The example in this document demonstrates an insurance application that uses an XML protocol as the communication between companies. We want to define one request/response combination as follows:
*Request/response name:**GetInsuranceCoverageRequest/GetInsuranceCoverageResponse***
Data in request:
Request type
A person's name
Data in response:
Coverage amount in dollars
Date request submitted
Response status code

Step 0 Complete. Continuing on...

Working out the details to all of these questions takes significant thought and effort. But now, after clearly answering these questions, we can continue on with a much better understanding of the solution we desire.

The ensuing steps talk about a number of different artifacts that comprise a Connect for iSeries custom protocol. It may be helpful to see the high level picture and description of what all of the artifacts are, how they relate to one another and what their purpose is. By understanding the "big picture", it will hopefully allow you to better comprehend what you are doing on each step and why. Here it is:



DTD file-Document Type Definition (DTD) file is the definition of the XML elements and attributes the protocol abides by**.** Although this picture does not show it, multiple DTD files could be created for your protocol. For example, you may have one DTD for each request/response combination, or one DTD for each request and one DTD for each response, etc. XML schemas are another way to define the XML format, but are not supported by Connect at this time.

RequestMsg file-Connect tool-generated file that defines the elements and attributes that comprise a specific request/response for the protocol. This file is used by Connect tools when mapping fields between the request/response and the connectors used to process the request. Depending on how you architect your request and responses, you may have multiples of these files with each pertaining to specific requests/responses.

example file-An example xml file that is typical of the xml requests the protocol would be receiving in production mode. This file is used for testing/development purposes only.

***ProtocolDefintion.xml*-**File that defines the configuration of the custom protocol in a manner that is consumable by Connect. The code jar files, RequestMsg files, servlet parameters/properties and buyer/supplier definition files are all specified here.

***ProtocolFlow files*-**Files that describe the order of the flow for the connector programs for your protocol.

***AppConnectors*-**Files that describe the class name, properties file and input/outputs for one protocol connector program.

***Connector*-**A program that performs a function desired as part of the protocol runtime flow. Typically this will be a Java connector.

1. Determine the name, subtype and version of your new protocol

In this example, we chose:
Name=**InsuranceXML**
Subtype=**LifePartners**
Version=**1.01**

"Name" is the formal name of the custom protocol.

"Subtype" is used to distinguish a particular implementation of the formal protocol. For example, our cXML implementation is "cXML, Ariba, 1.2". "Ariba" is used for the subtype because it has been tested against Ariba's software. Other marketplaces may choose to use cXML and could potentially implement cXML following the standard, yet be different enough from Ariba's software that connectors would have to change to be compatible.

"Version" delineates between modifications/fixes made to a particular protocol.

2. Create a new directory that will be the location for <u>all</u> of the artifacts you will be creating for your custom protocol.

We created a new directory on our iSeries called:

/QIBM/UserData/Connect110/Protocols/InsuranceXML-LifePartners-1.01

(Note to reader: The name you give your protocol should be the name of the directory you create and the directory must be under /QIBM/UserData/Connect110/Protocols)

*FYI for the future: If you plan on developing more than one protocol and ever wonder what the difference is between the protocols in the **/QIBM/UserData/Connect110/Protocols** directory and the **/QIBM/UserData/Connect110/Gateway/Connectors** directory, it is this: under the **.../Protocols** directory is where you put the things that you develop for your custom protocol. These are the new protocols that **can be** added or updated to your Connect installation. The **.../Gateway/Connectors** directory contains the working versions of the custom protocols that have **already been** deployed under your installation of Connect and are managed by the Connect administration tools.*

3. Create one or more Document Type Definition files (DTD) to define the request/response data that you want to send/receive via your new protocol. (Obviously, this step can be skipped if you already have a DTD created)

We created a file named InsuranceXML_101.DTD
Here is the DTD that we created for our example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ************************************************

    Helpful reminders:
    ? denotes element appears once or not at all
    + denotes element appears 1 or more times
    * denotes element appears 0 or more times

InsuranceXML_101.DTD defines the following hierarchy:
<InsuranceXML>
<Request>
  <GetInsuranceCoverageRequest>
  <Name>
 <Response>
  <Status>
  <Date_Time>
  <GetInsuranceCoverageResponse>
   <CoverageAmount>

********************************************************* -->
<!ENTITY InsuranceXML.version "1.01" >

<!ENTITY % InsuranceXML.requests  "GetInsuranceCoverageRequest" >
<!ENTITY % InsuranceXML.responses  "GetInsuranceCoverageResponse" >

<!ELEMENT InsuranceXML (Request*, Response*)>
<!ELEMENT Request ((%InsuranceXML.requests;))>
<!ELEMENT GetInsuranceCoverageRequest (Name)>
<!ELEMENT Name (#PCDATA) >
<!ELEMENT Response (Status?, Date_Time?,(%InsuranceXML.responses;)*)>
<!ELEMENT Status ANY>
```

```
<!ELEMENT Date_Time ANY>
<!ELEMENT GetInsuranceCoverageResponse (CoverageAmount)>
<!ELEMENT CoverageAmount (#PCDATA) >
```

This is not a paper about how to write DTDs and XML, so we won't go into the details of syntax, etc. If you are unfamiliar with DTDs and XML, then you should take the time to do some reading and understanding of them before continuing. Something that would make working with DTDs and XML easier would be to obtain an XML editor. The editor would present your XML in a more user-friendly format than a line editor. It would also use your DTD for syntax and validation checking, helping you to avoid time-consuming XML parsing errors that would be discovered during runtime. Two editors that we used were:

IBM XML and Web Services Development Environment (WSDE)

Note that WSDE not only contains an XML editor that leverages your DTD when editing your XML, it also contains a DTD creation tool that helps get you started with your DTD.

At the time of this writing, you could obtain it as follows:

http://www.alphaworks.ibm.com/tech/wsde
at the bottom of the screen, click "download"
select file xml-wsDE.zip and download it

The xml-wsDE.zip file is large (88MB) but contains much more than an XML editor, so it may be worth your while to try it out. It can also be downloaded as 9 small, individual files. It also has a 90-day evaluation period limit.

XEENA from Alphaworks

At the time of this writing, you could obtain it as follows:

http://www.alphaworks.ibm.com/tech/xeena
at the bottom of that page, click "download"
select file Xeena-1.2EA.exe and download it

run Xeena-1.2EA.exe to install it.

4. Generate an example xml request/response file that adheres to the DTD created in step #3.

This XML file is used prior to deployment of the new protocol for testing purposes. Using an XML editor would help insure that the file is valid according to the DTD. Here's a sample xml request that we wrote to do a "GetInsuranceCoverageRequest":

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE InsuranceXML SYSTEM "InsuranceXML_101.DTD">
<InsuranceXML>
```

```
 <Request>
  <GetInsuranceCoverageRequest>
   <Name>Johnny J. Johnson</Name>
  </GetInsuranceCoverageRequest>
 </Request>
</InsuranceXML>
```

Note that in this example we defined <Request> elements but not <Response> elements. The response elements in the example are only contained in the actual response. Here is an example of what the response will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE InsuranceXML SYSTEM "InsuranceXML_101.DTD">
<InsuranceXML>
   <Response>
      <Status>Successful</Status>
      <Date_Time>Mon Jul 16 15:33:08 CDT 2001</Date_Time>
      <GetInsuranceCoverageResponse>
         <CoverageAmount>$50000</CoverageAmount>
      </GetInsuranceCoverageResponse>
   </Response>
</InsuranceXML>
```

5. Create a Request/Response Message Format (file extension "RequestMsg") for each defined request type that describes the protocol runtime mappings.

A RequestMsg file is an XML document that identifies the input and output fields associated with a given request. Each type of request must have a corresponding RequestMsg file. The RequestMsg identifies the XML DTD and also defines the subset of elements in that DTD that are applicable for that request.

From a qsh screen on the iSeries, you can create a RequestMsg file by running the crtRMFFile tool that comes with the connect product. See the tools documentation for Connect Version 1.1 product for more information about this tool. Here are the commands we ran to accomplish this:

```
cd /qibm/userdata/connect110/protocols/InsuranceXML-LifePartners-1.01

/qibm/proddata/connect110/tools/runtime/crtRMFFile InsuranceXML_101.DTD
/InsuranceXML/Request/GetInsuranceCoverageRequest InsuranceXML_101.DTD
/InsuranceXML/Response GetInsuranceCoverageRequest.RequestMsg
```

Here is the file it produced, (GetInsuranceCoverageRequest.RequestMsg):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE requestmessageformat SYSTEM "RMF.DTD">
<requestmessageformat RMFVersion="1.1">
  <requestschema type="DTD">
    <field FMAccess="Read" GWAccess="Write" context="Request"
     count="one" display="yes" label="InsuranceXML" ref="/InsuranceXML">
     <field FMAccess="Read" GWAccess="Write" context="Request"
      count="multiple" display="yes" label="Request" ref="/InsuranceXML/Request">
      <field FMAccess="Read" GWAccess="Write" context="Request"
       count="one" display="yes" label="GetInsuranceCoverageRequest" ref="/InsuranceXML/Request/GetInsuranceCoverageRequest">
       <field FMAccess="Read" GWAccess="Write" context="Request"
        count="one" display="yes" label="Name" ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/Name"/>
      </field>
     </field>
    </field>
  </requestschema>
  <responseschema type="DTD">
    <field FMAccess="Write" GWAccess="Write" context="Response"
     count="one" display="yes" label="InsuranceXML" ref="/InsuranceXML">
     <field FMAccess="Write" GWAccess="Write" context="Response"
      count="multiple" display="yes" label="Response" ref="/InsuranceXML/Response">
      <field FMAccess="Write" GWAccess="Write" context="Response"
       count="one" display="yes" label="Status" ref="/InsuranceXML/Response/Status"/>
      <field FMAccess="Write" GWAccess="Write" context="Response"
       count="one" display="yes" label="Date_Time" ref="/InsuranceXML/Response/Date_Time"/>
      <field FMAccess="Write" GWAccess="Write" context="Response"
       count="one" display="yes" label="GetInsuranceCoverageResponse"
ref="/InsuranceXML/Response/GetInsuranceCoverageResponse">
       <field FMAccess="Write" GWAccess="Write" context="Response"
        count="one" display="yes" label="CoverageAmount"
ref="/InsuranceXML/Response/GetInsuranceCoverageResponse/CoverageAmount"/>
      </field>
     </field>
    </field>
  </responseschema>
</requestmessageformat>
```

6. Determine the flow steps necessary for proper handling of the requests coming into your custom protocol. The steps considered will be influenced by how you answered question 0 (who the trading partner(s) is (are), system topology, etc.)

For our simple example, we will implement a subset of the steps from the original list presented in the overview of this document (the ones we used/implemented are in large, bold characters). This is a minimal number of steps to actually get something to function (which is all we want to accomplish with this example).

- **XML validation and parsing of incoming request**
- **Inbound Logging**
- Authentication
- Authorization
- **Request DOM Header Generation (setting buyer/supplier and request information)**

- **Response DOM Element Priming**
- **Queue the request/response**
- **Error Checking**
- **Set Outbound Status**
- **Outbound Logging**
- **Error Handling**
- **Return to the servlet**

7. Create an AppConnector definition file for each of the Gateway connector programs.

Now that it's been decided what the steps are going to be for the flow, it's time to get more specific with what each step is going to be doing. This means we should start writing AppConnector files for each step. The AppConnector file is an iSeries Connect artifact that defines a connector. It states what the name of the connector program is, what (if any) the properties files are for that program, and what the input and output data is. For more information on AppConnectors, please refer to the iSeries Connect Programmer's Guide. According to the flow that was decided for our sample program, this would lead us to believe that we should create nine AppConnector files-one for each step we decided to implement. That's quite a few for a simple example, so here's some good news...we only have to create five. Here's why:

XML validation and parsing of incoming request **We are going to use the connector program provided with the Connect product. However, we must create an AppConnector for this program that specifies the correct properties file to be used, so this is AppConnector number ONE that we must write.**

Inbound Logging **We are going to use the connector program provided with the Connect product. The program and AppConnector already exist and we will use those.**

Authentication

Authorization

Request DOM Header Generation **This step is not a program call, it's a copy step, so no AppConnector needed.**

Response DOM Element Priming **This is a connector program that we are creating for our example,. It will need an AppConnector file, so we must write it.....that's TWO**

Queue the request/response **We are going to use the connector program provided with the Connect product. The program and AppConnector already exist and we will use those.**

Error Checking **We are going to use the connector program provided with the Connect product. However, we must create an AppConnector for this program that specifies the correct properties file to be used, so this is AppConnector number THREE that we must write.**

Set OutboundStatus **This is a connector program that we are creating for our example,. It will need an AppConnector file, so we must write it.....that's FOUR**

Outbound Logging **We are going to use the connector program provided with the Connect product. The program and AppConnector already exist and we will use those.**

Error Handling **We need to have a program that catches any errors that occur during the protocol flow, so we will write a simple connector program and accompanying AppConnector....that's FIVE.**

Here are the five AppConnector files that we wrote for our example:

XML Validation and Parsing

XMLToDOM.AppConnector

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Connector SYSTEM "JavaConnector.DTD">
<Connector ACDVersion="1.1" Type="%JavaACDType" Name="XMLToDOM">
  <Properties>
    <ClassName>com.ibm.connect.gateway.connector.XMLToDOMConnector</ClassName>

<PropertyFileName>gateway/connectors/InsuranceXML-LifePartners-1.01/XMLToDOMConnector.properties</PropertyFileN
ame>
    <InputFieldTemplate/>
    <OutputFieldTemplate/>
  </Properties>
  <Input/>
  <Output/>
</Connector>
```

Response DOM Element Priming

ResponseDOMPriming.AppConnector

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Connector SYSTEM "JavaConnector.DTD">
<Connector ACDVersion="1.1" Type="%JavaACDType" Name="ResponseDOMPrimingConnector">
  <Properties>
    <ClassName>InsuranceXML.ResponseDOMPrimingConnector</ClassName>
    <PropertyFileName></PropertyFileName>
    <InputFieldTemplate/>
    <OutputFieldTemplate/>
  </Properties>
  <Input>

  </Input>
  <Output>

  </Output>
</Connector>
```

Set Outbound Status

SetOutboundStatus.AppConnector

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE Connector SYSTEM "JavaConnector.DTD">
<Connector ACDVersion="1.1" Type="%JavaACDType" Name="SetOutboundStatusConnector">
  <Properties>
    <ClassName>InsuranceXML.SetOutboundStatusConnector</ClassName>
    <PropertyFileName></PropertyFileName>
    <InputFieldTemplate/>
    <OutputFieldTemplate/>
  </Properties>
  <Input>

  </Input>
  <Output>

  </Output>
</Connector>
```

Error Checking

XMLBuilder.AppConnector

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Connector SYSTEM "JavaConnector.DTD">
<Connector ACDVersion="1.1" Type="%JavaACDType" Name="XMLBuilder">
  <Properties>
    <ClassName>com.ibm.connect.gateway.connector.XMLBuilderConnector</ClassName>

<PropertyFileName>gateway/connectors/InsuranceXML-LifePartners-1.01/XMLBuilderConnector.properties</PropertyFileNa
me>
    <InputFieldTemplate/>
    <OutputFieldTemplate/>
  </Properties>
  <Input>

  </Input>
  <Output>

  </Output>
</Connector>
```

Error Handling

ExceptionHandler.AppConnector

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Connector SYSTEM "JavaConnector.DTD">
<Connector ACDVersion="1.1" Type="%JavaACDType" Name="ExceptionHandlerConnector">
  <Properties>
    <ClassName>InsuranceXML.ExceptionHandlerConnector</ClassName>
    <PropertyFileName>?</PropertyFileName>
    <InputFieldTemplate/>
    <OutputFieldTemplate/>
  </Properties>
  <Input>

  </Input>
  <Output>
```

```
  </Output>
</Connector>
```

8. Create the ProtocolFlow file

Now that you have decided what the new protocol's work flow should be, it is necessary to create a ProtocolFlow file to describe it. A ProtocolFlow file describes the sequence of connectors that need to run in order to fulfill a protocol's desired function. Here's the file we created (InsuranceCoverageRequest.ProtocolFlow), followed by explanation. (In the future, tools will generate this file for you, but at the time of this writing, these tools are not available yet. In order to create this file, we suggest using your favorite editor or one of the XML editors mentioned previously.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ProcessFlowModel SYSTEM "PFM.DTD">
<ProcessFlowModel PFMVersion="1.1">
<Protocol Protocol="InsuranceXML" ProtocolSubtype="LifePartners" ProtocolVersion="1.01"
Request="InsuranceCoverageRequest" RequestType="Get"/>
<ProcessFlow FirstStepName="XMLToDOM" FlowName="InsuranceCoverageRequest" Restartable="No">
   <Step Name="XMLToDOM" NextStepName="InboundLogger" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="XMLToDOM.AppConnector"/>
   </Step>
   <Step Name="InboundLogger" NextStepName="SetContentRequest" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="InboundLogger.AppConnector"/>
   </Step>
   <Step Name="SetContentRequest" NextStepName="ResponseDOMPriming" ErrorStepName="ExceptionHandler">
    <Copy>
     <Source>
      <Operand DataType="String" Default="GetInsuranceCoverageRequest"/>
     </Source>
     <Destination>
      <Operand Context="MessageHeader" DataType="String" Reference="com_ibm_connect_header_contentRequest"/>
     </Destination>
    </Copy>
    <Copy>
     <Source>
      <Operand DataType="String" Default="Get"/>
     </Source>
     <Destination>
      <Operand Context="MessageHeader" DataType="String" Reference="com_ibm_connect_header_contentRequestType"/>
     </Destination>
    </Copy>
   </Step>
   <Step Name="ResponseDOMPriming" NextStepName="FMComm" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="ResponseDOMPriming.AppConnector"/>
   </Step>
   <Step Name="FMComm" NextStepName="XMLBuilder" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="FMComm.AppConnector"/>
   </Step>
   <Step Name="XMLBuilder" NextStepName="SetOutboundStatus" ErrorStepName="ExceptionHandler">
```

```
    <Connector ACDRef="XMLBuilder.AppConnector"/>
   </Step>
   <Step Name="SetOutboundStatus" NextStepName="OutboundLogger" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="SetOutboundStatus.AppConnector"/>
   </Step>
   <Step Name="OutboundLogger" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="OutboundLogger.AppConnector"/>
   </Step>
   <Step Name="ExceptionHandler" NextStepName="XMLBuilderE">
    <Connector ACDRef="ExceptionHandler.AppConnector"/>
   </Step>
   <Step Name="XMLBuilderE" NextStepName="OutboundLoggerE">
    <Connector ACDRef="XMLBuilder.AppConnector"/>
   </Step>
   <Step Name="OutboundLoggerE">
    <Connector ACDRef="OutboundLogger.AppConnector"/>
   </Step>
  </ProcessFlow>
</ProcessFlowModel>
```

The first three lines of the file are "boilerplate" lines that would apply to any .ProtocolFlow file that you would create.

The next line defines your protocol and the request and request types that are handled by this flow. The protocol name, subtype and version should be what you decided they would be in an earlier step. *Request* is the name of the requests that will utilize this flow. Specific flows could be defined for different requests. Recall what was talked about in the **Gateway Flows** section earlier in this paper where it stated *"With this flexibility, we might use seperate URLs for each request or we might handle multiple requests with the same flow cycle."* This would be a key part to accomplishing that, but for our simple example, we won't delve into it at this time.

RequestType is used to categorize Requests. In this example, we set RequestType to "Get". We chose "Get" to denote all requests that are asking for information so that if we wanted to define new requests for our insurance environment, (e.g. "InsuranceCoverageRequest", "InsuranceQuoteRequest", "InsuranceClaimRequest", etc.), they could all be of request type "Get". This way, we could also define these same requests with type "Submit" or "Put" also (when these requests are sending in information). If you wanted more generic definition, you could specify a RequestType of *ALL. This means that, in our example, any insurance coverage request would be accepted by this flow.
The rest of the file defines a flow name and the steps to take and in what order. Here's a breakdown of one of the steps:

```
   <Step Name="XMLToDOM" NextStepName="InboundLogger" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="XMLToDOM.AppConnector"/>
   </Step>
```

Name-*Name of this step*
NextStepName-*the next step in the flow to execute*

ErrorStepName-*step to execute if an error is encountered during invocation of this step (this would be a connector invocation error or an unmonitored error that causes this connector to fail-not any kind of try/catch type of error that this connector would handle)*
Connector ACDRef-*the name of the file (AppConnector--we'll talk about those next!) that defines the connector program to be called during this step.*

You might notice that step3 and step (step to be added) are a little different. These are "copy" steps that don't call any connector program, but instead will copy a specified piece of information to a field in one of the structures available in the FlowState of the Connect instance (more on these structures later). In step3, for instance, the request name (GetInsuranceCoverageRequest) is being written to the MessageHeader structure. This must be done in order that the FlowManager can successfully determine the flow that it should execute once the request is passed to it. If you were to do your own simple example like this, you could do the exact same step--the only difference being you would specify your desired request value (the "Default=" value). For more complex Request DOM Header setup, another connector program could be used, but in this example, a copy step is sufficient. For more information on the copy step and other step types, please refer to the iSeries Connect Programmer's Guide. The syntax and function for ProtocolFlows is the same as that provided for ProcessFlows.

Going back to our earlier list of steps, let's re-list them here and how they correlate to the steps in our ProtocolFlow file:

- **XML validation and parsing of incoming request** **(step "XMLToDOM")**
- **Inbound Logging** **(step "InboundLogger")**
- Authentication
- Authorization
- **Request DOM Header Generation (setting buyer/supplier and request information)** **(step "SetContentRequest")**
- **Response DOM Element Priming** **(step "ResponseDOMPriming")**
- **Queue the request/response** **(step "FMComm")**
- **Error Checking** **(step "XMLBuilder")**
- **Set Outbound Status** **(step "SetOutboundStatus")**
- **Outbound Logging** **(step "OutboundLogger")**
- **Error Handling** **(step "ExceptionHandler")**
- **Return to the servlet** **(no step..happens after the last step is completed)**

9. Write the java connector programs to fulfill the necessary flow steps for your custom protocol.

The good news in the previous section was that because we used some of what was provided with Connect to support our flow, we only had to write five AppConnectors. There's also good news in this section. We only have three programs to write. Before we get to the specifics of the code that was written for our example, it would be good to establish an understanding of the basics of the architecture surrounding the connector programs. So, to do that, let's take a quick.....

🕓 *Time out!!*

Remember the earlier statement: *"If you can't write it down, you can't program it"*? We've already taken the time to write down/articulate what we want to do--a very necessary step. Now what we need to do is know how to write corresponding code that properly integrates with the Connect framework. Before we start cranking out code, we need to understand how our connector programs can access and work with the request/response data that will flow through the custom protocol. So, let's start with some basic Q&A's to help understand how the openness of the Connect framework allows us to do this.

Q: The first connector program that we need to write satisfies the step, "Response DOM element priming". We named the program for this step *ResponseDOMPrimingConnector.java* and added it to the ProtocolFlow file and wrote the corresponding AppConnector. But, how's it going to get invoked---what kind of interfaces does it need to implement?
A: Every Gateway connector program must be a Java class that implements the JavaProgramConnectorInterface interface. The method that will get called in your connector program will be:

```
JavaConnectorResult run (ProgramConnectorParm parameters)
```

At runtime, the Gateway flow engine will instantiate this class and invoke this run() method.

Q: What's this ProgramConnectorParm thingy that gets passed to my connector program?
A: The ProgramConnectorParm object that's passed in supports all of the APIs available to programmatically access the "flow state" data. The flow state data is all the data that is relevant to the Connect instance and the current request that's being handled. **It is strongly suggested that you read the Connect for iSeries programming guide to gain better understanding of the flow state and functionality therein.** It will greatly increase your proficiency in writing your connectors and understanding what you will need to do.

Q: What's that JavaConnectorResult I'm supposed to pass back on return?
A: The JavaConnectorResult object provides a standard way for the connector program to pass back result information to the Gateway flow engine. It provides the capability of setting a return code and also a string object containing other result information deemed important by the connector program. The return code helps the flow engine determine if the connector program was successful and whether or not it should continue with the flow or abort. The return code must be set. It is implied by the flow engine that a return code of zero indicates success and a non-zero return code indicates failure. The ReturnString could be anything pertinent to helping the user understand the flow activity from step to step (whether errors occur or not). JavaConnectorResult has two interfaces that support this. They are:

public void setReturnCode(int newReturnCode)
public void setReturnString(java.lang.String newReturnString)

You will see examples of how to use this in the connector source code shown in Appendix A.

End of *Time out*.

As we determined earlier (through construction of the ProtocolFlow and the writing of the AppConnector documents), we will need to provide three connector programs to support our custom protocol. Along with these programs are two property files for two existing connector programs. Here's the comprehensive list of programs and properties files needed:

XML Validation and Parsing of incoming request
- Created properties file XMLToDOMConnector.properties

(needed a properties file to state where the DTD files are that this program should be using for *request* parsing and validation)

Response DOM Element Priming
- Wrote program InsuranceXML.ResponseDOMPrimingConnector.java

Set Outbound Status
- Wrote program InsuranceXML.SetOutboundStatusConnector.java

Error Checking
- Created properties file XMLBuilderConnector.properties

(needed a properties file to state where the DTD files are that this program should be using for *response* parsing and validation)

Error Handling
- Wrote program InsuranceXML.ExceptionHandler.java

The source for these programs and the properties files are included in Appendix A of this document. It, along with all other source discussed in this document, is also available from our website at ***http://www-1.ibm.com/servers/eserver/iseries/btob/connect/.***

Compiling all connector programs and create the jar file containing them

For this example, a trivial shell script was written to call the Java compiler with the appropriate classpath values. This seemed much easier as the classpath is quite large (a "shotgun" approach was used here as more jar files were listed than required, however by doing this, the command file should work for future, more complex connector program compiling). The source for this shell script and an equivalent

command file (if you are running from mapped drives on a PC) are included in Appendix A. Once the shell script was written, here's an example of how we ran it from qsh on an iSeries:

```
> cd /qibm/userdata/connect110/protocols/InsuranceXML-LifePartners-1.01
  $
> compileconnector . ResponseDOMPrimingConnector.java
   Connector compilation finished
   $
```

Since we created our connector program to be in the "InsuranceXML" package, our compile step created a subdirectory named 'InsuranceXML' with our .class file in it. So, to create the jar file for our protocol, we issued the following command:

```
jar cvf InsuranceXML.jar InsuranceXML/*.class
```

10. Create a Protocol Data Model definition file

The Protocol Data Model definition file is an XML document that defines the text to be displayed and the data to be collected on the Supplier Marketplace Association and Buyer Marketplace Association pages for the corresponding Protocol. This is the file referenced in the TPADataModel element in the Protocol Definition. These pages are accessible under the Supplier and Buyer links in the Connect Administration GUI. The data input on these pages represent how the registered requestors (buyers) and target (supplier) will be identified in the protocol flow and optionally any logon information required to authenticate the incoming request. Because the example being illustrated here is not doing any authentication or authorization checks in the process flow (see process flow outlined previously) , this example is quite trivial, requiring no data. Here is the buyer-supplier screen model definition file (InsuranceXML-LifePartners-1.01.xml) we created for our example followed by descriptions of the fields available in a screen model definition (for a comprehensive list of fields, please see the supplierbuyerscreenmodel.dtd file shipped with the Connect Version 1.1 product):

```
<?xml version="1.0"?>
<!DOCTYPE SupplierBuyerScreenModel SYSTEM "supplierbuyerscreenmodel.dtd">
<SupplierBuyerScreenModel name="InsuranceXML-LifePartners-1.01" version="V1R1M0">
  <context name="supplierprotocol">
    <frame name="supplieraccess" order="1" title="(no supplier Information required)">
    </frame>
  </context>
  <context name="buyerprotocol">
    <frame name="buyeraccess" order="1" title="(no buyer information required)">
    </frame>
  </context>
</SupplierBuyerScreenModel>
```

The first three lines of the file are “boilerplate” lines that would apply to any data model definition file that you would create.

The root element is "SupplierBuyerScreenModel". The "version" attribute should be set to the version of Connect that
this model is to be used with. The version is "V1R1M0" for this release. The "name" attribute should be used to specify a short description of the user defined protocol that this file is used with. The suggested value is "Protocol Name - version".

The protocol data model will have two contexts. One for Supplier Marketplace Association and one for Buyer Marketplace Association. A context is defined within a "context" element. The "name" attribute specifies which context is being used. The context name "supplierprotocol" denotes this context is to be used for the Supplier Marketplace Association data model. The "buyerprotocol" context name denotes the context is for the Buyer Marketplace Association data model.

Within a context there will be a set of "frame" elements. Each "frame" element corresponds to a page in the Marketplace Association wizard for this protocol. At a minimum one frame must be included. The "name" attribute on the frame must be unique to all frames within the context. The "order" attribute specifies the order of the frames in the context. The value for the "title" attribute is the text displayed in the title area of the page.

A frame will optionally contain text fields and input fields.

A text field is defined within a "text" element. It is used to put informational text onto the page.

An input field is defined with a "field" element. A "field" element describes the data that is collected and the property name that the data is stored under.

The field element has many attributes. This is a list of the required attributes. Check the DTD for more information about additional field attributes.

"order" - Used to ensure this input field is in the correct location
on the page with relation to other input and text fields.
"varname" - The property name that the value for this field is stored
under in the protocol data base table.
"length" - The longest string of characters that is acceptable for
input to this field. Set to "-1" for no limit.
"type" - The kind of checking performed against data entered in this
field. "alphanumericsymbol" will allow the user to enter any
data as long as the length is within the value set for the
length attribute. Other values for this attribute that can be
used for data checking include "alpha", "numeric",

"alphanumeric", and "timestamp".
There are 3 values for type that cause the field to behave differently:
"password" - is a password field, text entered is "*" out.
"hidden" - the field is not displayed, but it is stored.
"noentry" - the field is displayed but the user can't change it.
The "defaultval" attribute should be set in order for the last two to have any value.

"required" - If this value is set to "yes" then the user must provide a value for this field. Set to "no" if the field is optional.

"defaultval" - If this attribute is set the value will appear in the text entry box for the field if no other value is set.

11. Create an XML file called ProtocolDefinition.xml.

This XML document describes the protocol that will be added to the Connect installation for use by Connect instances. The DTD for this file is the Protocol.DTD that is shipped with Connect.

Here is the ProtocolDefinition.XML file we created for our example followed by descriptions of each field:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Protocol SYSTEM "ignored.dtd">
<Protocol
   DeployDestination="InsuranceXML-LifePartners-1.01"
   DeployType="UserData" Name="InsuranceXML" SubType="LifePartners" Version="1.01">
   <TPADataModel>ProtocolDataModel-InsuranceXML-LifePartners-1.01.xml</TPADataModel>
   <Jars>
      <Jar>/QIBM/UserData/Connect110/Gateway/Connectors/InsuranceXML-LifePartners-1.01/InsuranceXML.jar</Jar>
   </Jars>
   <Servlets>
      <Servlet
         Class="com.ibm.connect.gateway.servlet.HTTPServlet" DefaultName="InsuranceXML-LifePartners-1.01">
         <ServletParameter Editable="No"
            Name="FlowName" Required="Yes">InsuranceCoverageRequest</ServletParameter>
         <ServletParameter Editable="No"
            Name="RequestMethod" Required="Yes">POST</ServletParameter>
         <Requests>
            <Request Name="GetInsuranceCoverageRequest" Required="NO">
               <RequestType Name="Get">
                <MessageFormat
Name="/QIBM/UserData/Connect110/Gateway/Connectors/InsuranceXML-LifePartners-1.01/GetInsuranceCoverageRequest.RequestMsg"/>
```

```
            <MessageFormat Name="/QIBM/ProdData/Connect110/Gateway/Connectors/InternalHeaderv11.RequestMsg"/>
            <MessageFormat Name="/QIBM/ProdData/Connect110/Gateway/Connectors/MessageHeaderv11.RequestMsg"/>
            <MessageFormat Name="/QIBM/ProdData/Connect110/Gateway/Connectors/BuyerSupplierv11.RequestMsg"/>
          </RequestType>
        </Request>
      </Requests>
    </Servlet>
  </Servlets>
</Protocol>
```

Heading statements:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Protocol SYSTEM "ignored.dtd">

These two lines are standard lines that belong in the front of any ProtocolDefinition.xml file. The "ignored.dtd" entry is put there because the DTD parser that's being used requires a name value to be there, but does not actually use that name. So, to satisfy the parser, a "dummy" value is put there.

Protocol

Root element for a protocol definition
Name = "value" *Name of the protocol*
SubType = "value" *Subtype of the protocol*
Version = "value" *Version identifier of the protocol*
DeployDestination = "value" *destination location of this protocol's artifacts*
DeployType = "UserData" *Identifier to indicate that this is deployed under ProdData or UserData. We strongly recommend always using UserData to insure perservation and migration of your protocol between releases. Only the protocols shipped with the Connect product should be in ProdData.*

TPADataModel

Name of the Protocol Data Model XML file that describes how buyers and supplier are identified in the protocol (usually in the terminology of the protocol, like a DUNS number or Insurance agent number, etc) and optionally if there is any logon information to be authenticated against via the incoming request. Logon information would correspond to logon id(s) and a password.

Jars

Section for the jar files required to support this protocol. These jar files will contain any specific connector programs written for this protocol.

Jar

Name of a jar file required to support this protocol. Note that this jar file and all others listed under the Jars element for your protocol will automatically be added to your classpath.

Requests

Section for the requests that this protocol supports.

DefaultFlow

FlowName = "value" *Name of the .ProtocolFlow file that supports the requests listed here. Creation of this .ProtocolFlow file was described in an earlier section of this document*

Request

Name = "value" *Name of the request (eg. "OrderRequest")*
Required = "YES" *Is this request required in all request/response flows through this protocol? Yes or no.*
FlowName = "value" *Name of the .ProtocolFlow file that supports this request. Creation of this .ProtocolFlow file was described in an earlier section of this document*

RequestType
Name = "value" *Name of the request type (eg. "create")*

MessageFormat
Name = "value" *Name of the .RequestMsg file that defines the message format used for this request*

Servlets
Section for the servlets that are defined for this protocol.

Servlet
Class = "value" *Name of the class file that is this servlet. Currently only*
com.ibm.connect.gateway.servlet.HTTPServlet *is support)*
Authentication = "BASIC" *Type of HTTP authentication to be done. BASIC or NONE. Most of the time this will be NONE because authentication will be done at the protocol level in the XML vs at the transport level by the HTTP servlet. Either way is secure, it just depends on how the protocol is defined.*
DefaultName = "value" *Default name to be associated with this servlet*

ServletParameter
Name - *Name of the parameter*
Required - *Is this parameter required, Yes or No*
Editable - *Is this parameter editable via the Connect GUI (for future use), Yes or No.*
(character data) *The actual parameter value being passed in*
When Name="RequestMethod", value is POST, NVP, GET or NONE. If the XML is POSTed by the sender it should be POST. If the sender is sending in name-value pairs it should be NVP. If the sender is doing an HTTP GET it should be GET. When XML data is sent it should be POST. NONE is also valid.

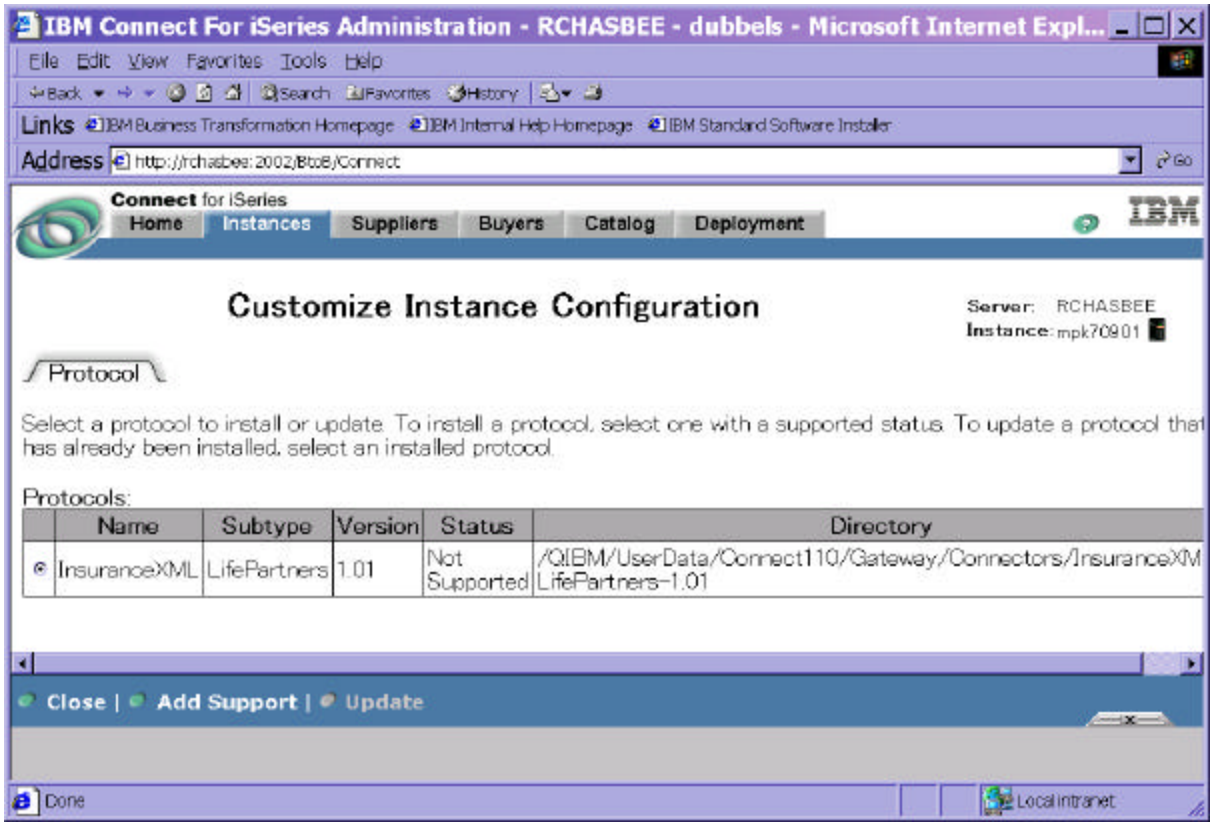
Checkpoint!! ***All the artifacts required to install the InsuranceXML-LifePartners-1.01 protocol now exist in the custom protocol directory. Here's the list:***

Directory: **/QIBM/UserData/Connect110/Protocols/InsuranceXML-LifePartner-1.01**
ExceptionHandler.AppConnector
ExceptionHandlerConnector.java
GetInsuranceCoverageRequest.RequestMsg
GetInsuranceCoverageRequest.xml
InsuranceCoverageRequest.ProtocolFlow
InsuranceXML.jar
InsuranceXML_101.DTD
ProtocolDataModel-InsuranceXML-LifePartner-1.01.xml

ProtocolDefinition.xml
ResponseDOMPriming.AppConnector
ResponseDOMPrimingConnector.java
SetOutboundStatus.AppConnector
SetOutboundStatusConnector.java
XMLBuilderConnector.AppConnector
XMLBuilderConnector.properties
XMLToDOM.AppConnector
XMLToDOMConnector.properties

12. Add support for the new protocol to your Connect installation.

In order to make our protocol useable by Connect, we have to register it as an available protocol. The adding and updating of a new BtoB protocol screen is available off of the Customize task link on the Connect Instances screen. It supports installing or updating a custom protocol and making it available for use by Connect instances. The list of protocols to work with is built internally by scanning the /QIBM/UserData/Connect110/Protocols/ subdirectory. The protocol status depends on whether the protocol has already been added. A protocol that has already been added can be updated at any time as long as all Connect instances that use that protocol are not running. If any of those instances are running, they will need to be stopped prior to updating that custom protocol. After a protocol has been added, it is then available for use by any existing Connect instances via the Edit Connect instance function. The new custom protocol can also be used when creating new Connect instances. Here is the customize screen that is displayed upon selecting “customize” from the “Instances” menu. As you can see, the new protocol, InsuranceXML, is now available to be installed:

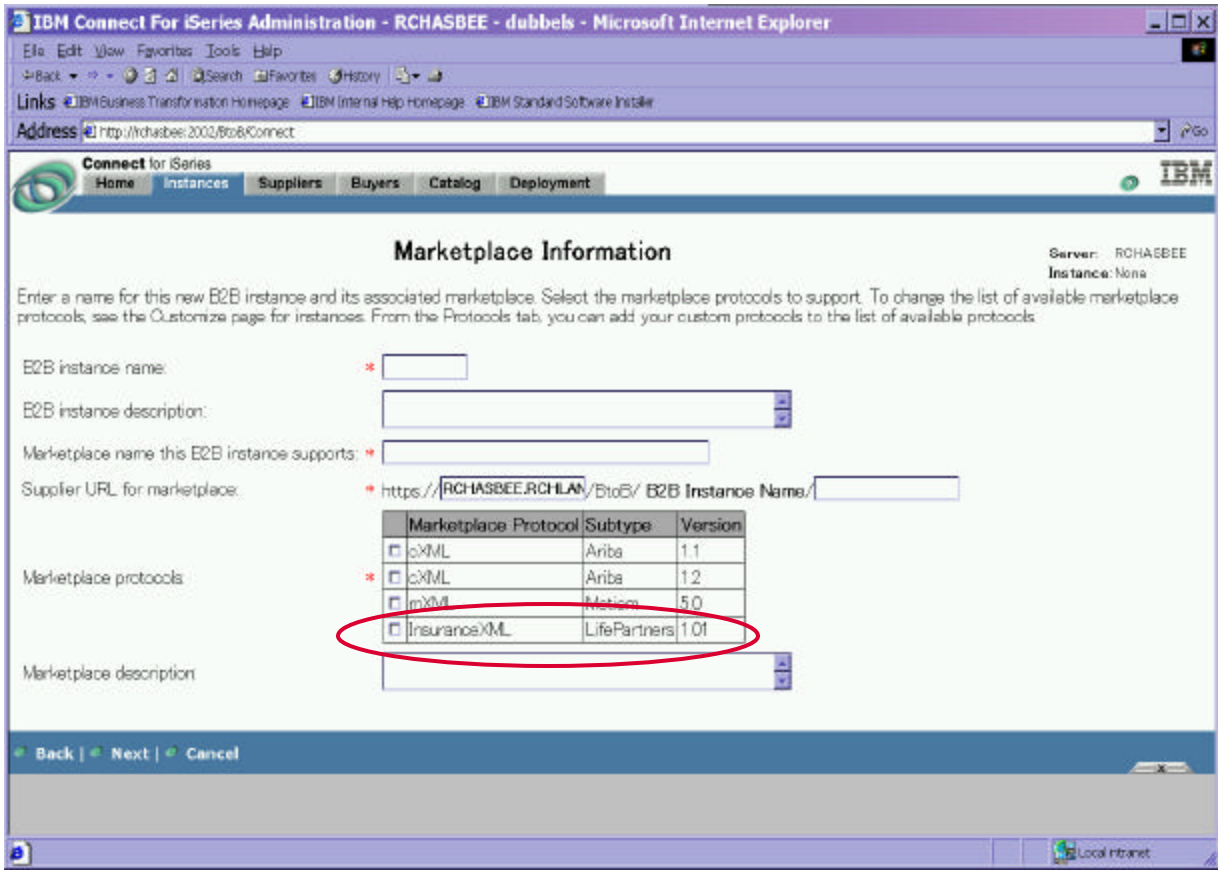


As mentioned earlier, installing a new protocol includes creating a new set of directories in /QIBM/UserData/Connect110 to contain the installed protocol. For this example, the directory structure that was created during protocol installation was:

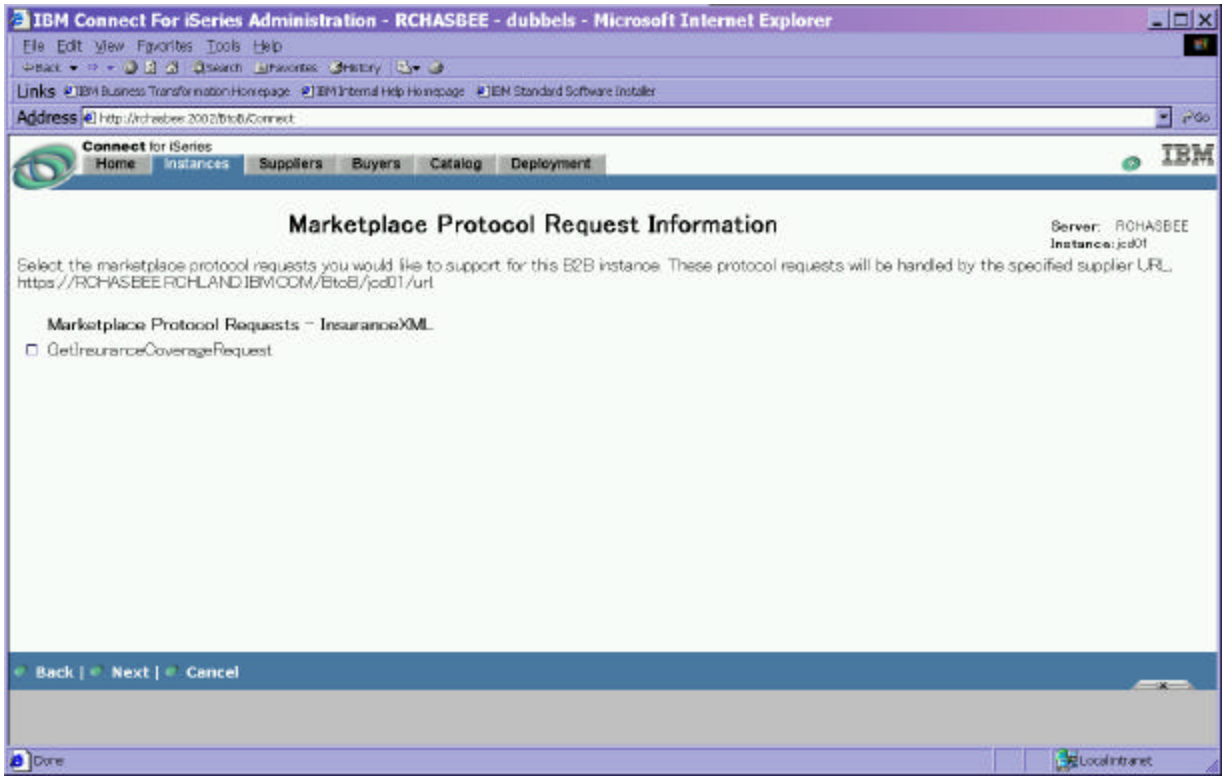
/QIBM/UserData/Connect110/Gateway/Connectors/InsuranceXML-LifePartners-1.01/

13. Create a new instance with the new protocol

(It is assumed that you are familiar with using Connect and doing instance creation.) Normal instance creation can now be done using the new protocol. A couple of screen pictures are shown here to highlight the differences you will see when using the new protocol.

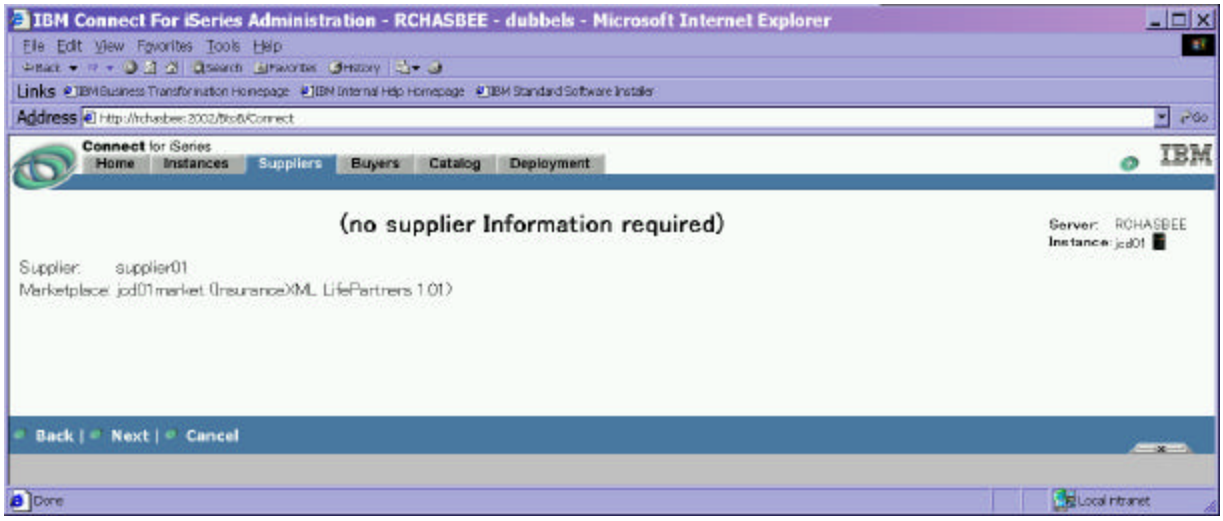


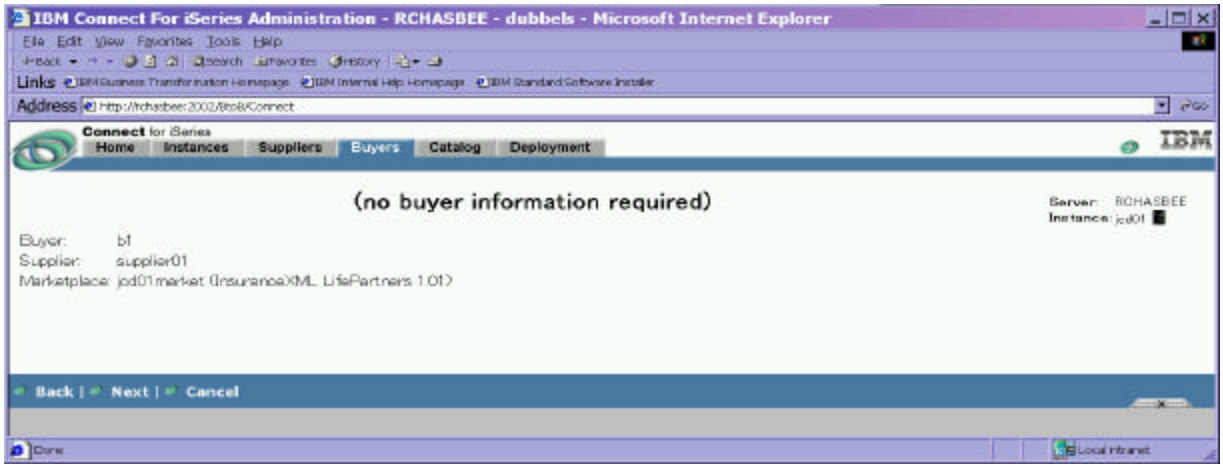
InsuranceXML is now available as a new entry in the protocol list to choose from.



When viewing/selecting Request types, you can now see the request that we defined for InsuranceXML**.**

In the next two screens (when defining buyers and suppliers), note that the screens reflect what we defined in our protocol data model file (ProtocolDataModel- InsuranceXML-LifePartners-1.01.xml). Since this example contains no authentication and authorization checking, it was not necessary to define any buyer or supplier access information. So the screens were defined to not have any.

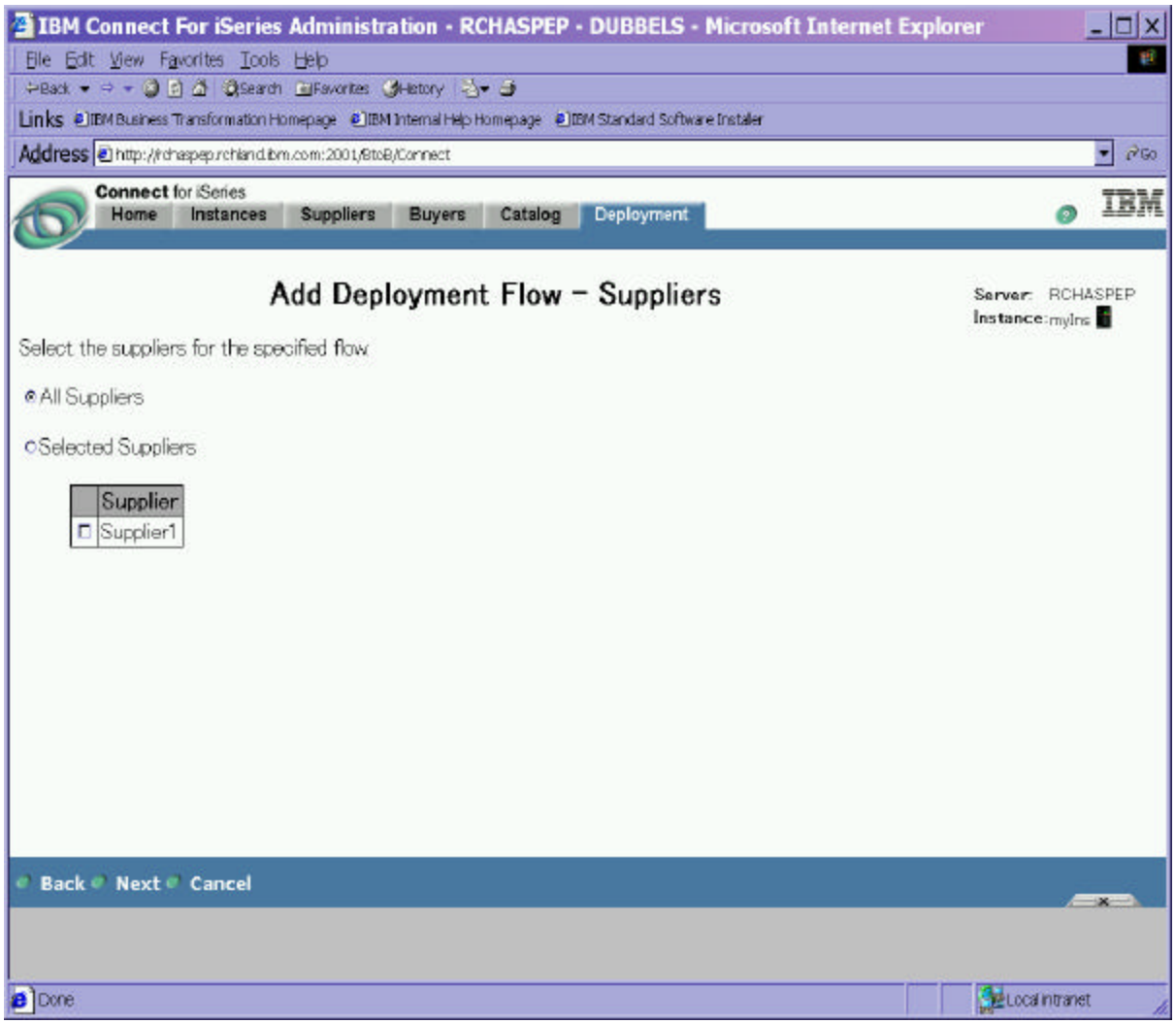




14) Create and deploy a process flow that runs in the Flowmanager

It is assumed that you are already familiar with creating and deploying process flows, so we won't go into the specifics of the connector program at this point. We basically just put together a quick program that catches the request and shoots back a canned answer (as described in our introduction to this example). The source of the connector program and the PCML file that describes it is included in appendix A. It should be noted for this example that when you create the ProcessFlow for this "quick program", you should deploy the flow for *all suppliers and all buyers* (see the following screen as an

example of where to do this). Making the modifications to the protocol connectors that cater to specific suppliers and buyers will be covered in the next example.

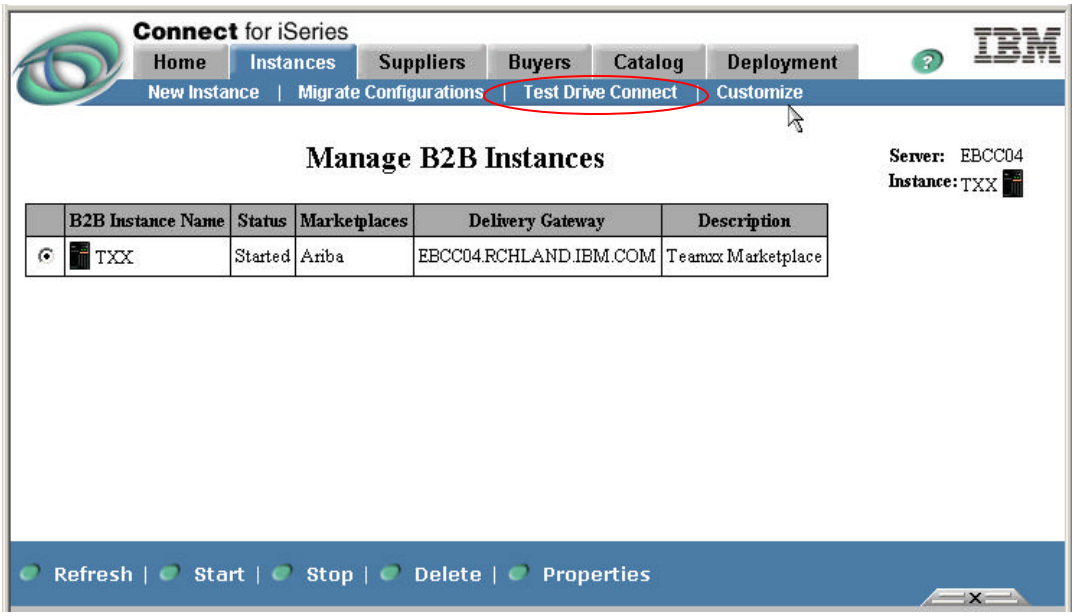


15. Start the new instance and test it.

Now that you have an instance created using the new custom protocol, you can start and test the new instance using the example XML request file you created in step 4. This can be accomplished as follows:

Go to the Connect “Instances” browser page, select your instance and click on “start” at the bottom of the page.

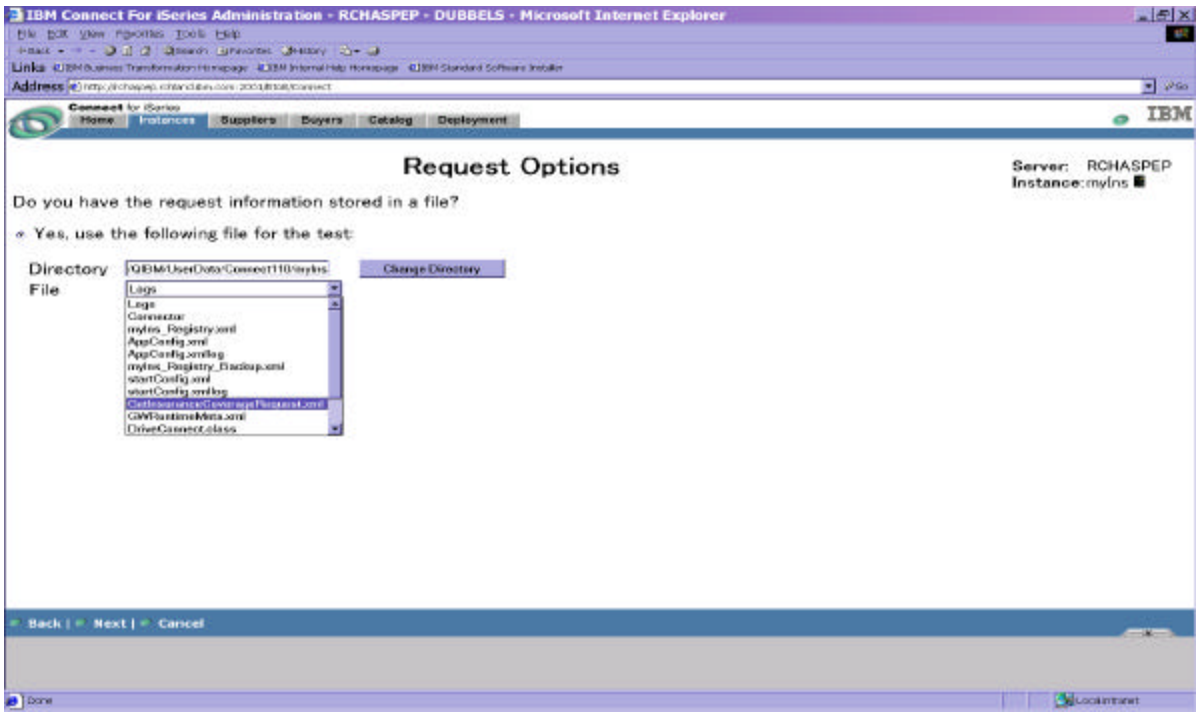
Once you've got the HTTP, web server and flowmanager started, go back to your instances page and (with your instance still selected), click on "Test Drive Connect" at the top of the page. The following screen shows where to find "Test Drive Connect" on the instances page:



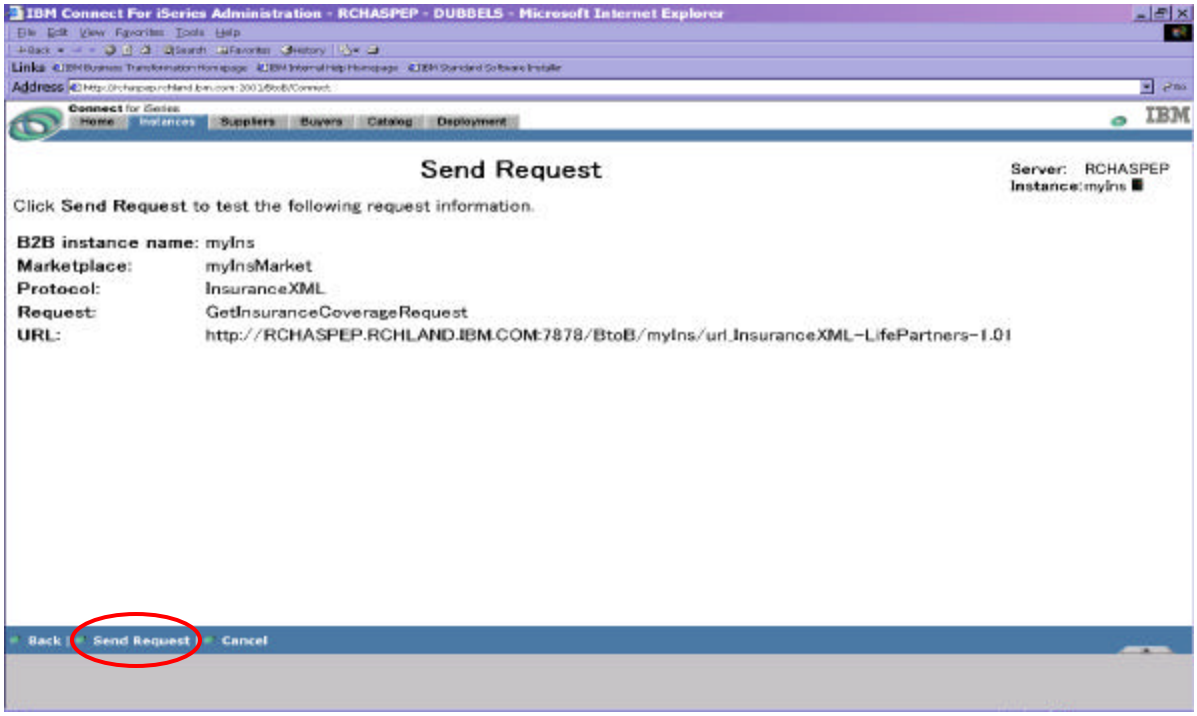
Once you have clicked on "Test Drive Connect", you should proceed through the screens in the following way (there will be a few verification screens that you need to go through also--just click "next" to navigate through those).

In this next screen, the path that you should enter should be:
/QIBM/UserData/Connect110/protocols/InsuranceXML-LifePartners-1.01

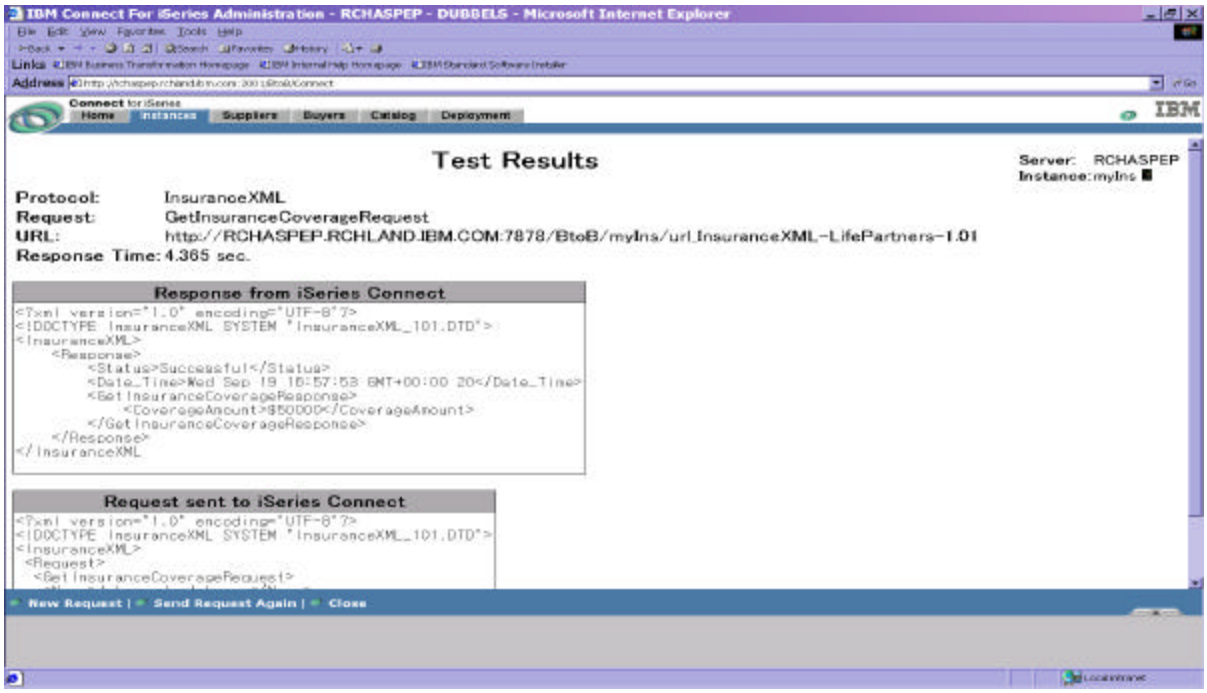
...then click "Change Directory" and then select GetInsuranceCoverageRequest.xml



Eventually you will come to this screen, at which you will click “Send Request”:



The output from this request should be equivalent to what was described in step 4 of this document, showing the coverage dollar amount response for the submitted request, as shown in this screen:



Epilog

At this point, you should now have a working custom protocol that performs a simple request/response exchange. If you have problems, please refer to the product documentation that talks about log files, tracing and debugging for assistance. You may find it helpful to copy and modify the examples given here to help save time and effort when implementing your own custom protocol. The source for these programs is included in Appendix A of this document. It, along with all other source discussed in this document, is also available from our website at ***http://www-1.ibm.com/servers/eserver/iseries/btob/connect/.***

Example 2:*Adding Authentication and Authorization*

Now that we have a working (albeit simple) protocol, let's add some function to it. In this case, we will add authentication and authorization checking as additional connector steps in the custom protocol flow. This example will be a delta based off the previous example, therefore it will consist of re-listing our original steps, but only showing the differences/additions to what was done here versus in the previous example. So, let's start through the steps necessary to build a custom protocol with this additional function, leveraging what we've already done.

Steps to Define a Custom Protocol

0) Clearly understand your objective for your new protocol.

In our first example, we stated that our deployment solution was a 1-to-1, trusted relationship within the confines of our own firewall, so no authentication or authorization needed to be done. In this case, the scenario is much more open. There is a 1-to-many relationship that is exposed outside of our firewall. Therefore we will need to guard against submitted requests that do not satisfy our security criteria plus insure that those that are allowed access stay within the guidelines of their designated authority.

1) Determine the name, subtype and version of your new protocol.

To keep our existing protocol in tact, let's create a new version of our protocol..... Name=InsuranceXML, subtype=LifePartners, version=1.02.

2) Create a new directory that will be the location for all of the artifacts you will be creating for your custom protocol.

Created directory **/QIBM/UserData/Connect110/Protocols/InsuranceXML-LifePartners-1.02**

3) Create a Document Type Definition file (DTD) to define the request/response data that you want to send/receive via your new protocol.

Created a new DTD file named InsuranceXML_102.DTD. We added the fields we determined were necessary to perform adequate authentication and authorization checking. These new fields provide identification for the requester via userID, location and password and for the target supplier via company name and insurance type. Here it is. What changed from the previous example is in bold characters:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ************************************************

    Helpful reminders:
    ? denotes element appears once or not at all
    + denotes element appears 1 or more times
    * denotes element appears 0 or more times

InsuranceXML_10**2**.DTD defines the following hierarchy:
<InsuranceXML>
<Request>
  <GetInsuranceCoverageRequest>
   <**UserID**>
   <**Domain**>
   <**SharedPassword**>
   <**InsuranceCompanyName**>
   <**InsuranceType**>
   <Name>
 <Response>
  <Status>
  <Date_Time>
  <GetInsuranceCoverageResponse>
   <CoverageAmount>

************************************************** -->
<!ENTITY InsuranceXML.version "1.02" >

<!ENTITY % InsuranceXML.requests  "GetInsuranceCoverageRequest" >
<!ENTITY % InsuranceXML.responses  "GetInsuranceCoverageResponse" >
<!ELEMENT InsuranceXML (Request*, Response*)>
<!ELEMENT Request ((%InsuranceXML.requests;))>
<**!ELEMENT GetInsuranceCoverageRequest (UserID, Location, SharedPassword, InsuranceCompanyName, InsuranceType, Name)**>
<**!ELEMENT UserID (#PCDATA)** >
<**!ELEMENT Location (#PCDATA)** >
<**!ELEMENT SharedPassword (#PCDATA)** >
<**!ELEMENT InsuranceCompanyName (#PCDATA)** >
<**!ELEMENT InsuranceType (#PCDATA)** >
<!ELEMENT Name (#PCDATA) >
<!ELEMENT Response (Status?, Date_Time?, (%InsuranceXML.responses;)*)>
```

```
<!ELEMENT Status ANY>
<!ELEMENT Date_Time ANY>
<!ELEMENT GetInsuranceCoverageResponse (CoverageAmount)>
<!ELEMENT CoverageAmount (#PCDATA) >
```

4) Generate an example XML request/response file that adheres to the DTD created in step #3.

Note the additions of the User ID, location, password and insurance company name and type elements. These elements were added to pass in the authorization and authentication data:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE InsuranceXML SYSTEM "InsuranceXML_102.DTD">
<InsuranceXML>
 <Request>
  <GetInsuranceCoverageRequest>
   <UserID>ins01</UserID>
   <Location>rochester</Location>
   <SharedPassword>secret</SharedPassword>
   <InsuranceCompanyName>Big Insurance</InsuranceCompanyName>
   <InsuranceType>life</InsuranceType>
   <Name>Johnny J. Johnson</Name>
  </GetInsuranceCoverageRequest>
 </Request>
</InsuranceXML>
```

5) Create a Request/Response Message Format (file extension “RequestMsg”) for each defined request type that describes the protocol runtime mappings.

Tool generated as in the first example, here it is:

crtRMFFile InsuranceXML_102.DTD /InsuranceXML/Request/GetInsuranceCoverageRequest

InsuranceXML_102.DTD /InsuranceXML/Response GetInsuranceCoverageRequest.RequestMsg

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE requestmessageformat SYSTEM "RMF.DTD">
<requestmessageformat RMFVersion="1.1">
  <requestschema type="DTD">
    <field FMAccess="Read" GWAccess="Write" context="Request"
     count="one" display="yes" label="InsuranceXML" ref="/InsuranceXML">
      <field FMAccess="Read" GWAccess="Write" context="Request"
       count="multiple" display="yes" label="Request" ref="/InsuranceXML/Request">
        <field FMAccess="Read" GWAccess="Write" context="Request"
         count="one" display="yes" label="GetInsuranceCoverageRequest"
ref="/InsuranceXML/Request/GetInsuranceCoverageRequest">
          <field FMAccess="Read" GWAccess="Write" context="Request"
           count="one" display="yes" label="UserID" ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/UserID"/>
```

```
        <field FMAccess="Read" GWAccess="Write" context="Request"
          count="one" display="yes" label="Location" ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/Location"/>
        <field FMAccess="Read" GWAccess="Write" context="Request"
          count="one" display="yes" label="SharedPassword"
ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/SharedPassword"/>
        <field FMAccess="Read" GWAccess="Write" context="Request"
          count="one" display="yes" label="InsuranceCompanyName"
ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/InsuranceCompanyName"/>
        <field FMAccess="Read" GWAccess="Write" context="Request"
          count="one" display="yes" label="InsuranceType"
ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/InsuranceType"/>
        <field FMAccess="Read" GWAccess="Write" context="Request"
          count="one" display="yes" label="Name" ref="/InsuranceXML/Request/GetInsuranceCoverageRequest/Name"/>
       </field>
     </field>
   </field>
  </requestschema>
  <responseschema type="DTD">
   <field FMAccess="Write" GWAccess="Write" context="Response"
    count="one" display="yes" label="InsuranceXML" ref="/InsuranceXML">
    <field FMAccess="Write" GWAccess="Write" context="Response"
     count="multiple" display="yes" label="Response" ref="/InsuranceXML/Response">
     <field FMAccess="Write" GWAccess="Write" context="Response"
      count="one" display="yes" label="Status" ref="/InsuranceXML/Response/Status"/>
     <field FMAccess="Write" GWAccess="Write" context="Response"
      count="one" display="yes" label="Date_Time" ref="/InsuranceXML/Response/Date_Time"/>
     <field FMAccess="Write" GWAccess="Write" context="Response"
      count="one" display="yes" label="GetInsuranceCoverageResponse"
ref="/InsuranceXML/Response/GetInsuranceCoverageResponse">
       <field FMAccess="Write" GWAccess="Write" context="Response"
        count="one" display="yes" label="CoverageAmount"
ref="/InsuranceXML/Response/GetInsuranceCoverageResponse/CoverageAmount"/>
      </field>
     </field>
   </field>
  </responseschema>
</requestmessageformat>
```

6) Determine the flow steps necessary for proper handling of the requests coming into your custom protocol. The steps considered will be influenced by how you answered question 0 (who the trading partner(s) is (are), system topology, etc.)

Based on our basic list of flow steps, we are adding three new connector steps to the flow. The authentication and authorization connectors are obvious choices to add (since that's the purpose of this example). So what about UniqueRefno? It is not directly related to authorization and authentication, but does make use of the same criteria that authentication and authorization uses. so it is an easy addition to make. What is UniqueRefno? If you recall in the first example, in step 14 it was stated that you should

deploy the process flow for *all suppliers and all buyers.* If you want to select specific buyers and/or suppliers to be associated with a process flow, you need to pass buyer/supplier-specific information along with the request. UniqueRefno is a generic connector provided with the Connect product that obtains this information based on mapping of what was passed in on the request. It then places this information in the request header so it's included as the request flows to the flow manager. If you don't care about designing flows for specific buyers or suppliers, then you could leave this step out. It is not absolutely necessary and could be added later.

- **XML validation and parsing of incoming request**
- **Inbound Logging**
- ***Authentication***
- ***Authorization***
- ***UniqueRefno***
- **Request DOM Header Generation (setting buyer/supplier and request information)**
- **Response DOM Element Priming**
- **Queue the request/response**
- **Error Checking**
- **Set Outbound Status**
- **Outbound Logging**
- **Error Handling**
- **Return to the servlet**

7) Create an AppConnector definition file for each of the Gateway connector programs.

No changes needed here. Using the authentication and authorization AppConnector files that shipped with the Connect product.

8) Create the ProtocolFlow file

A new ProtocolFlow file was created. The steps were added to include calling the authentication and authorization connectors.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ProcessFlowModel SYSTEM "PFM.DTD">
<ProcessFlowModel PFMVersion="1.1">
<Protocol Protocol="InsuranceXML" ProtocolSubtype="LifePartners" ProtocolVersion="1.02" Request="InsuranceCoverageRequest"
RequestType="Get"/>
<ProcessFlow FirstStepName="XMLToDOM" FlowName="InsuranceCoverageRequest" Restartable="No">
   <Step Name="XMLToDOM" NextStepName="InboundLogger" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="XMLToDOM.AppConnector"/>
   </Step>
   <Step Name="InboundLogger" NextStepName="SetContentRequest" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="InboundLogger.AppConnector"/>
   </Step>
   <Step Name="SetContentRequest" NextStepName="Authenticate" ErrorStepName="ExceptionHandler">
     <Copy>
```

```
      <Source>
        <Operand DataType="String" Default="GetInsuranceCoverageRequest"/>
      </Source>
      <Destination>
        <Operand Context="MessageHeader" DataType="String" Reference="com_ibm_connect_header_contentRequest"/>
      </Destination>
    </Copy>
    <Copy>
      <Source>
        <Operand DataType="String" Default="Get"/>
      </Source>
      <Destination>
        <Operand Context="MessageHeader" DataType="String" Reference="com_ibm_connect_header_contentRequestType"/>
      </Destination>
    </Copy>
  </Step>
  <Step Name="Authenticate" NextStepName="Authorize" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="Authentication.AppConnector">
      <MapIn Name="InstanceName">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_gatewayInstance" /></MapIn>
      <MapIn Name="Marketplace">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_marketplace" /></MapIn>
      <MapIn Name="Protocol">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_protocolType" /></MapIn>
      <MapIn Name="ProtocolSubtype">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_protocolSubType" /></MapIn>
      <MapIn Name="ProtocolVersion">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_protocolVersion" /></MapIn>
      <MapIn Name="LogonId">
        <Operator CommonName="Concat">
          <Operator CommonName="Concat">
            <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/UserID" Type="String"/>
            <Operand Default=":" Type="String"/>
          </Operator>
          <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/Location" Type="String"/>
        </Operator>
      </MapIn>
      <MapIn Name="Password">
        <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/SharedPassword" /></MapIn>
    </Connector>
  </Step>
  <Step Name="Authorize" NextStepName="UniqueRefno" ErrorStepName="ExceptionHandler">
    <Connector ACDRef="Authorization.AppConnector">
      <MapIn Name="InstanceName">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_gatewayInstance" /></MapIn>
      <MapIn Name="Marketplace">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_marketplace" /></MapIn>
      <MapIn Name="Protocol">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_protocolType" /></MapIn>
      <MapIn Name="ProtocolSubtype">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_protocolSubType" /></MapIn>
      <MapIn Name="ProtocolVersion">
        <Operand Context="MessageHeader" Reference="com_ibm_connect_header_protocolVersion" /></MapIn>
      <MapIn Name="SupplierId">
        <Operator CommonName="Concat">
          <Operator CommonName="Concat">
            <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/InsuranceCompanyName"
Type="String"/>
            <Operand Default=":" Type="String"/>
          </Operator>
```

```
        <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/InsuranceType"
Type="String"/>
       </Operator>
      </MapIn>
      <MapIn Name="BuyerId">
       <Operator CommonName="Concat">
        <Operator CommonName="Concat">
         <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/UserID" Type="String"/>
         <Operand Default=":" Type="String"/>
        </Operator>
        <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/Location" Type="String"/>
       </Operator>
      </MapIn>
      <MapIn Name="Request">
       <Operand Context="MessageHeader" Reference="com_ibm_connect_header_contentRequest" /></MapIn>
      <MapIn Name="RequestType">
       <Operand Context="MessageHeader" Reference="com_ibm_connect_header_contentRequestType" /></MapIn>
     </Connector>
    </Step>
    <Step Name="UniqueRefno" NextStepName="ResponseDOMPriming" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="UniqueRefno.AppConnector">
      <MapIn Name="InstanceName">
       <Operand Context="MessageHeader" Reference="com_ibm_connect_header_gatewayInstance" /></MapIn>
      <MapIn Name="SupplierId">
       <Operator CommonName="Concat">
        <Operator CommonName="Concat">
         <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/InsuranceCompanyName"
Type="String"/>
         <Operand Default=":" Type="String"/>
        </Operator>
        <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/InsuranceType"
Type="String"/>
       </Operator>
      </MapIn>
      <MapIn Name="BuyerId">
       <Operator CommonName="Concat">
        <Operator CommonName="Concat">
         <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/UserID" Type="String"/>
         <Operand Default=":" Type="String"/>
        </Operator>
        <Operand Context="Request" Reference="/InsuranceXML/Request/GetInsuranceCoverageRequest/Location" Type="String"/>
       </Operator>
      </MapIn>
     </Connector>
    </Step>
    <Step Name="ResponseDOMPriming" NextStepName="FMComm" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="ResponseDOMPriming.AppConnector"/>
    </Step>
    <Step Name="FMComm" NextStepName="XMLBuilder" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="FMComm.AppConnector"/>
    </Step>
    <Step Name="XMLBuilder" NextStepName="SetOutboundStatus" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="XMLBuilder.AppConnector"/>
    </Step>
    <Step Name="SetOutboundStatus" NextStepName="OutboundLogger" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="SetOutboundStatus.AppConnector"/>
    </Step>
    <Step Name="OutboundLogger" ErrorStepName="ExceptionHandler">
     <Connector ACDRef="OutboundLogger.AppConnector"/>
    </Step>
    <Step Name="ExceptionHandler" NextStepName="XMLBuilderE">
```

```
    <Connector ACDRef="ExceptionHandler.AppConnector"/>
   </Step>
   <Step Name="XMLBuilderE" NextStepName="OutboundLoggerE">
    <Connector ACDRef="XMLBuilder.AppConnector"/>
   </Step>
   <Step Name="OutboundLoggerE">
    <Connector ACDRef="OutboundLogger.AppConnector"/>
   </Step>
  </ProcessFlow>
</ProcessFlowModel>
```

9) Write the Java connector programs to fulfill the necessary flow steps for your custom protocol.

No new code needed here. The authentication, authorization, UniqueRefno and RequestToken AppConnector programs that are shipped with the Connect product are being used. You **will** have to modify XMLToDOMConnector.properties and XMLBuilderConnector.properties to point to the new protocol directory name that contains the DTDs. The updated files are shown in Appendix B.

Unrelated to the Gateway, but necessary to make the application function in the Flow Manager portion of iSeries Connect, you will also have to create the app connector and process flow and deploy it using the GenericFMJavaConnector pcml and java files listed in Appendix A.

10) Create a Protocol Data Model definition file

This file is quite different than in the previous example. In the previous example we weren't concerned with buyers and suppliers and checking out who was who and who was allowed to do what, so this file consisted of basically a couple of titles that said "no information required". In this example we now care about who is who and who is allowed to do what, so we must define the registration screens to allow input of the data required to accomplish this. *(Note: Connect is initially designed for BtoB marketplace relationships, so the paradigm of thinking is in terms of "buyers", "suppliers" and "marketplaces". Therefore, the context names of "supplierprotocol" and "buyerprotocol" are fixed and must be exactly those values in order for Connect internals to work correctly. The same is true for "Marketplace_Protocol_IDn", "Marketplace_Logon_IDn", and "Marketplace_Password". Whereever you see these values, they must be that way for your implementation too. "Marketplace_*_IDn" can be where n is a value from 1 up to 5 and must be listed in numerical order (ie. You can't have only Marketplace_Protocol_ID2...if there's only one, it must be Marketplace_Protocol_ID1). The terms don't really apply to an insurance example like this, but correlations can be made, such as supplier=insurance company and buyer=insurance agent/client. This is exactly what we are doing here. Please keep in mind that these guidelines are for the internal variables used when defining this screen. It is strongly suggested that you label the data on the screen in a manner that makes sense for the protocol being deployed (thus insulating the user from this potentially confusing correlation). In this example we have appropriate "insurance" text and title fields defined and not anything referencing buyers/suppliers.*

*One last note for clarification....**authentication** deals with Marketplace_Logon_IDn and Marketplace_Password prompts and **authorization** (identifying requestor/target relationships) deals with Marketplace_Protocol_IDn.)*

So, what we've done is defined insurance company name and coverage fields under the "supplierprotocol" context (remember that the insurance company name and coverage fields were new additions to our DTD in step 1--they will come in as part of the request. Here we are defining screens to allow them to be registered under our connect instance, such that when they DO come in on the request, we will recognize them). Under context "buyerprotocol", we've defined an insurance agent user ID, location and password (again, just as we defined in our DTD in step 1), enabling registration of allowed buyers that will be sending requests into Connect. You may notice that we are requesting agent user ID and location twice on the screen. This is due to the fact that this information is required to set up protocol ID (authorization) information ("Marketplace_Protocol_IDn") and logon (authentication) information ("Marketplace_Logon_IDn"). We are aware that this is redundant and working on a solution for it. For now, please set up your data model files in similar fashion. Please see the previous example for a descriptions of the fields and values in a screen model file.

```
<?xml version="1.0"?>
<!DOCTYPE SupplierBuyerScreenModel SYSTEM "supplierbuyerscreenmodel.dtd">
<SupplierBuyerScreenModel name="InsuranceXML-LifePartners-1.02" version="V1R1M0">
 <context name="supplierprotocol">
  <frame name="supplieraccess" order="1" title="Insurance Supplier Information">
   <text order="1" visibility="new">Insurance Company</text>
   <group order="2">
    <field order="1" varname="Marketplace_Protocol_ID1" length="30" required="yes" label="Name"/>
    <field order="2" varname="Marketplace_Protocol_ID2" length="30" required="yes" label="Coverage"/>
   </group>
  </frame>
 </context>
 <context name="buyerprotocol">
  <frame name="buyeraccess" order="1" title="Insurance Agent">
   <text order="1" visibility="new">Insurance Agent Protocol Information</text>
   <group order="2">
    <field order="1" varname="Marketplace_Protocol_ID1" length="30" required="yes" label="ID"/>
    <field order="2" varname="Marketplace_Protocol_ID2" length="30" required="yes" label="Location"/>
   </group>
   <text order="3" visibility="new">Agent Logon Information</text>
   <group order="4">
    <field order="1" varname="Marketplace_Logon_ID1" length="20" required="yes" label="ID"/>
    <field order="2" varname="Marketplace_Logon_ID2" length="30" required="yes" label="Location"/>
    <field order="3" varname="Marketplace_Password"  length="12" required="yes" type="password" label="Access
Password"/>
   </group>
  </frame>
 </context>
</SupplierBuyerScreenModel>
```

11) Create an XML file called, ProtocolDefinition.xml.

This file is the same as it was in the previous example except that the new protocol version is used throughout the file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Protocol SYSTEM "ignored.dtd">
<Protocol
   DeployDestination="InsuranceXML-LifePartners-1.02"
   DeployType="UserData" Name="InsuranceXML" SubType="LifePartners" Version="1.02">
   <TPADataModel>ProtocolDataModel-InsuranceXML-LifePartners-1.02.xml</TPADataModel>
   <Jars>
      <Jar>/QIBM/UserData/Connect110/Gateway/Connectors/InsuranceXML-LifePartners-1.02/InsuranceXML.jar</Jar>
   </Jars>
   <Servlets>
      <Servlet
         Class="com.ibm.connect.gateway.servlet.HTTPServlet" DefaultName="InsuranceXML-LifePartners-1.02">
         <ServletParameter Editable="No"
            Name="FlowName" Required="Yes">InsuranceCoverageRequest</ServletParameter>
         <ServletParameter Editable="No"
            Name="RequestMethod" Required="Yes">POST</ServletParameter>
         <Requests>
            <Request Name="GetInsuranceCoverageRequest" Required="NO">
               <RequestType Name="Get">
                 <MessageFormat
Name="/QIBM/UserData/Connect110/Gateway/Connectors/InsuranceXML-LifePartners-1.02/GetInsuranceCoverageRequest.RequestMsg"/>
                  <MessageFormat Name="/QIBM/ProdData/Connect110/Gateway/Connectors/InternalHeaderv11.RequestMsg"/>
                  <MessageFormat Name="/QIBM/ProdData/Connect110/Gateway/Connectors/MessageHeaderv11.RequestMsg"/>
                  <MessageFormat Name="/QIBM/ProdData/Connect110/Gateway/Connectors/BuyerSupplierv11.RequestMsg"/>
               </RequestType>
            </Request>
         </Requests>
      </Servlet>
   </Servlets>
</Protocol>
```

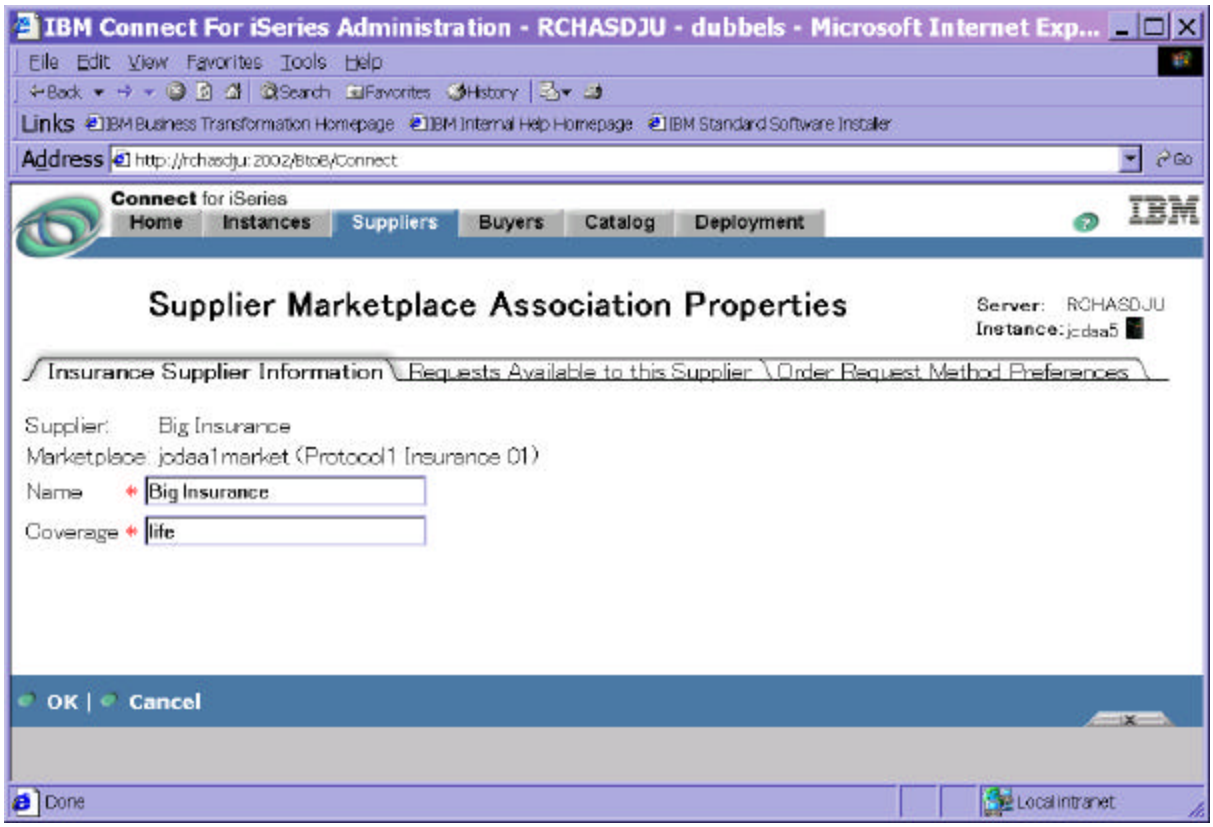
12) Add support for the new protocol to your Connect installation.

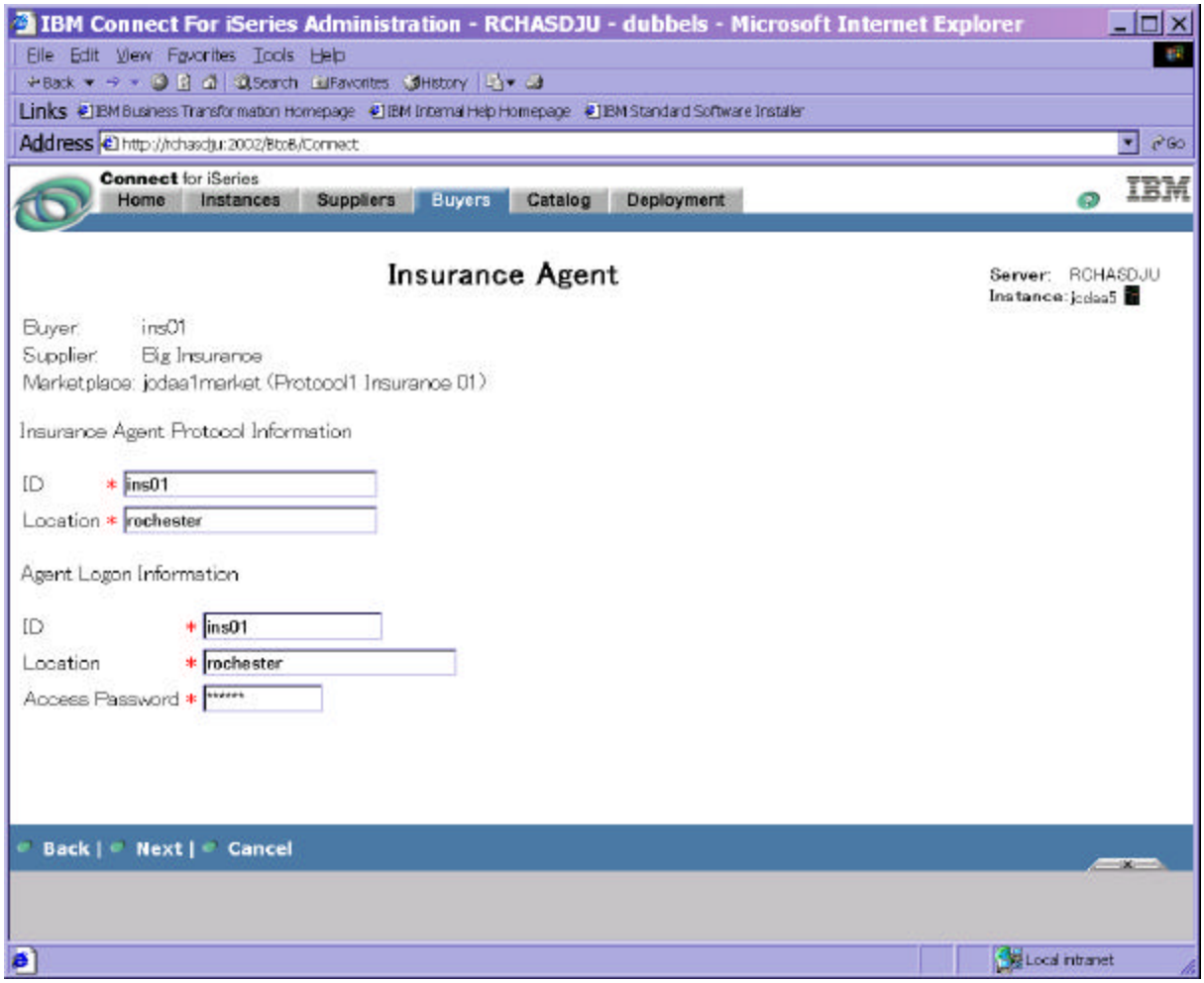
This is done exactly as it was in the previous example.

13) Create a new instance with the new protocol

Creating a new instance is basically the same, however there is a little more to do in terms of buyer/supplier registration. The difference is the specific screens we defined in our protocol data definition file that prompt for special authentication/authorization information. Here is is a look at the new

screens filled in with the content required to run the request for this example. Note that the buyer information screen has an access password field that is represented as asterisks. For this example, the value should be *secret*. This is what's also passed in on the GetInsuranceCoverageRequest.xml





14) Create and deploy a process flow that runs in the Flowmanager

This is done exactly as it was in the previous example with the exception that you *could* select specific buyers and suppliers if you chose to add in the UniqueRefno connector step as opposed to saying *all buyers and all suppliers.*

15)Start the new instance and test it

This is done exactly as it was in the previous example with the minor change of specifying InsuranceXML-LifePartners-1.0*2* instead of InsuranceXML-LifePartners-1.01 in the DriveConnect.properties file and in the java command that calls it.

Properties value:

ConnectURL=http://sssssssssss:pppp/BtoB/myInstance/url_InsuranceXML-LifePartners-1.02

Line command:

java DriveConnect ..\gateway\connectors\InsuranceXML-LifePartners-1.02\GetInsuranceCoverageRequest

Epilog

No java coding was required for this example since we used the authentication, authorization (and optionally) UniqueRefno and RequestToken connectors that are part of the Connect product. You should now have a protocol working with authentication and authorization checking enabled.

Appendix A

Source for the compileconnector shell script:

```
#!/usr/bin/qsh

if(javac -classpath
.:/qibm/proddata/connect110/classes/NCSO.jar:/qibm/proddata/connect110/classes/QueueConnector.jar:/qibm/proddata/connect110/classes/activation.jar:/qibm/proddata/connect110/classes/bridge.jar:/qibm/proddata/connect110/classes/comibmconnect.jar:/qibm/proddata/connect110/classes/command.jar:/qibm/proddata/connect110/classes/config.jar:/qibm/proddata/connect110/classes/connpool.jar:/qibm/proddata/connect110/classes/flowmanager.jar:/qibm/proddata/connect110/classes/flowmanagerapi.jar:/qibm/proddata/connect110/classes/log.jar:/qibm/proddata/connect110/classes/logging.jar:/qibm/proddata/connect110/classes/loggingapi.jar:/qibm/proddata/connect110/classes/mail.jar:/qibm/proddata/connect110/classes/queueing.jar:/qibm/proddata/connect110/classes/soap.jar:/qibm/proddata/connect110/classes/wcsconfigservices.jar:/qibm/proddata/connect110/classes/wcsiconnect.jar:/qibm/proddata/connect110/classes/wcsruntime.jar:/qibm/proddata/connect110/classes/xalan200.jar:/qibm/proddata/connect110/classes/xerces311.jar:/qibm/proddata/connect110/tools/tpa/tpa.jar:/qibm/proddata/connect110/tools/tpa/tpaapi.jar:/qibm/proddata/connect110/tools/tpa/tpagui.jar:/qibm/proddata/connect110/tools/tpa/tparuntime.jar:/qibm/proddata/connect110/tools/catalog/catalog.jar:/qibm/proddata/connect110/tools/runtime/deploy.jar:/qibm/proddata/connect110/tools/runtime/toolsmodel.jar:/qibm/proddata/connect110/tools/runtime/util.jar:/qibm/proddata/connect110/config/cwbuntpi.jar::/qibm/proddata/connect110/gateway/gateway.jar:/qibm/proddata/connect110/gateway/gatewayAPI.jar:/qibm/proddata/connect110/gateway/connectors/gwConnector.jar -d $1 $1/$2)
then
        echo Connector compilation finished
else
        echo Connector compilation failed
fi
```

Source for the CompileConnector.cmd file:

```
@rem
@rem parm1=directory program resides in and should have compiler output stored in
@rem parm2=program to compile
@if .%2 == . goto helpme
@set codeDir=%1
@set srcDir=%1
@rem ************************************************
@rem Add product jars from "classes" directory to path list
@rem ************************************************
@set jarList=\qibm\proddata\connect110\classes\NCSO.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\QueueConnector.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\activation.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\bridge.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\comibmconnect.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\command.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\config.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\connpool.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\flowmanager.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\flowmanagerapi.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\log.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\logging.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\loggingapi.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\mail.jar;
```

```
@set jarList=%jarList%\qibm\proddata\connect110\classes\queueing.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\soap.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\wcsconfigservices.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\wcsiconnect.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\wcsruntime.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\xalan200.jar;
@set jarList=%jarList%\qibm\proddata\connect110\classes\xerces311.jar;
@rem
@rem ************************************************
@rem Add product jars from "tools\tpa" directory to path list
@rem ************************************************
@set jarList=%jarList%\qibm\proddata\connect110\tools\tpa\tpa.jar;
@set jarList=%jarList%\qibm\proddata\connect110\tools\tpa\tpaapi.jar;
@set jarList=%jarList%\qibm\proddata\connect110\tools\tpa\tpagui.jar;
@set jarList=%jarList%\qibm\proddata\connect110\tools\tpa\tparuntime.jar;
@rem
@rem ************************************************
@rem Add product jars from "tools\catalog" directory to path list
@rem ************************************************
@set jarList=%jarList%\qibm\proddata\connect110\tools\catalog\catalog.jar;
@rem
@rem ************************************************
@rem Add product jars from "tools\runtime" directory to path list
@rem ************************************************
@set jarList=%jarList%\qibm\proddata\connect110\tools\runtime\deploy.jar;
@set jarList=%jarList%\qibm\proddata\connect110\tools\runtime\toolsmodel.jar;
@set jarList=%jarList%\qibm\proddata\connect110\tools\runtime\util.jar;
@rem
@rem ************************************************
@rem Add product jars from "config" directory to path list
@rem ************************************************
@set jarList=%jarList%\qibm\proddata\connect110\config\cwbuntpi.jar;;
@rem
@rem ************************************************
@rem Add product jars from "gateway" directory to path list
@rem ************************************************
@set jarList=%jarList%\qibm\proddata\connect110\gateway\gateway.jar;
@set jarList=%jarList%\qibm\proddata\connect110\gateway\gatewayAPI.jar;
@rem
@rem ************************************************
@rem Add product jars from "gateway\connectors" directory to path list
@rem ************************************************
@set jarList=%jarList%\qibm\proddata\connect110\gateway\connectors\gwConnector.jar;
@rem
@rem ************************************************
@rem Do the compile
@rem ************************************************
@javac -classpath %codeDir%;%jarList%; -d %codeDir% %srcDir%\%2
@goto endIt
:helpme
@echo You must pass in a directory name and a java file name
@echo example: CompileConnector i:\qibm\userdata\connect110\protocols\myProtocol myConnectorPgm.java
:endIt
```

Source for XMLToDOMConnector.properties

```
# there are 3 properties
# UserSpecifiedDTD -- do you want to specify a DTD, or use the one specified in the document
# if UserSpecifiedDTD = yes, then there is
# UserSpecifiedDTDType = [file | url]
# UserSpecifiedDTDLocation = [ /path/to/file | http://www.blah.com/  ]
```

```
UserSpecifiedDTD = yes
UserSpecifiedDTDType = file
UserSpecifiedDTDLocation = Gateway/Connectors/InsuranceXML-LifePartners-1.01/InsuranceXML_101.dtd
```

Source for XMLBuilderConnector.properties

```
NoValidationDTD = Gateway/Connectors/NoValidation.DTD
LocalDTDName = Gateway/Connectors/InsuranceXML-LifePartners-1.01/InsuranceXML_101.dtd
```

Source for ResponseDOMPrimingConnector.java program

```
/* ---------------------------------
This material contains IBM copyrighted sample programming source code ("Sample Code"). IBM grants you a
nonexclusive license to compile, link, execute, display, reproduce, distribute and prepare derivative works of this
Sample Code. The Sample Code has not been thoroughly tested under all conditions. IBM, therefore, does not
guarantee or imply its reliability, serviceability, or function. IBM provides no program services for the Sample Code.
All Sample Code contained herein is provided to you "AS IS" without any warranties of any kind. THE IMPLIED
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED. SOME JURISDICTIONS DO NOT ALLOW THE
EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN NO
EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER
CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE CODE INCLUDING, WITHOUT LIMITATION,
ANY LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR
INFORMATION HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.
---------------------------------- */

package InsuranceXML;

import com.ibm.connect.logging.interfaces.*;
import com.ibm.connect.flowmanager.interfaces.*;
import com.ibm.connect.gateway.interfaces.*;
import com.ibm.connect.gateway.service.*;
import com.ibm.connect.bridge.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.*;

/**
 * ResponseDOMPrimingConnector primes the ResponseDOM object with whatever information it has
 * at request time in order to offload DOM creation by the flowmanager.
 */
public class ResponseDOMPrimingConnector implements HeaderConstants,
JavaProgramConnectorInterface, GatewayConstants {

    private ProgramConnectorParm parms = null;

    /**
     * The Document form of the response document.  This is the root of the
     * response document.
     */
    protected DocumentImpl document;

    /**
     * Generic Constructor
```

```
     */
    public ResponseDOMPrimingConnector()
    {
    }

    /**
     * This method builds the <code>DocType</code> statement, then calls the
     * <code>createHeader</code> method to build the root element and a child status element.
     * The root element is then appended to the
     * <code>DocType</code> element.
     *
     * @param ProgramConnectorParm the data object passed to all connectors
     * @return JavaConnectorResult containing the return code and text
     */
    public JavaConnectorResult run(ProgramConnectorParm parms)
    {
        if (UserLogManager.isTracing) {
            UserLogManager.trace(this, "run", "Entering run method of ResponseDOMPrimingConnector.");
        }
        this.parms = parms;
        JavaConnectorResult result = new JavaConnectorResult();
        DocumentType docType = new DocumentTypeImpl(document, "ResponseDOMDoc", null,
                                        (String) parms.flowDataAreaGet(GatewayConstants.REMOTE_DTD_NAME));

        document = new DocumentImpl(docType);
        document.setErrorChecking(true);
        document.appendChild(createHeader());

        parms.setResponseDOM(document);

        /* Move outbound transport type and encoding from header to
         * TransportObject
         */
        TransportI transportObject = (TransportI) parms.flowDataAreaGet(GatewayConstants.TRANSPORT_INTERFACE);

transportObject.setOutboundContentEncoding((String)parms.sendableMessageHeaderGet(HeaderConstants.INBOUND_TRAN
SPORT_CONTENT_ENCODING));
        result.setReturnCode(EXIT_SUCCESSFUL);
        result.setReturnString("ResponseDOMPrimingConnector exited successfully");
        if (UserLogManager.isTracing) {
            UserLogManager.trace(this, "run", "Exiting run method of ResponseDOMPrimingConnector.");
        }
        return result;
    }

    /**
     * This method creates the <code>InsuranceXML</code> root element for the response.
     *
     * @return the <code>InsuranceXML</code> Element
     */
    protected Element createHeader()
    {
        Element rootElement = document.createElement("InsuranceXML");
        rootElement.appendChild(createResponseElement());
```

```
        return rootElement;
    }

    /**
    * This method creates the <code>Response</code> element.
    *
    * @return the <code>Response</code> Element
    */
    protected Element createResponseElement()
    {
        Element responseElement = document.createElement("Response");
        responseElement.appendChild(createStatusElement());
        responseElement.appendChild(createDateTimeElement());
        return responseElement;
    }

    /**
    * This method creates the <code>Status</code> element.
    *
    * @return the <code>Status</code> Element
    */
    protected Element createStatusElement()
    {
        Element statusElement = document.createElement("Status");
        return statusElement;
    }

    /**
     * This method creates the <code>Date_Time</code> element for the response.
     *
     * @return the <code>Date_Time</code> Element
     */
    protected Element createDateTimeElement()
    {
        Element dateTimeElement = document.createElement("Date_Time");
        return dateTimeElement;
    }
}
```

Source for SetOutboundStatusConnector.java program

```
/* ---------------------------------
```

```
POSSIBILITY OF SUCH DAMAGES.
------------------------------------ */

package InsuranceXML;

import com.ibm.connect.logging.interfaces.*;
import com.ibm.connect.flowmanager.interfaces.*;
import com.ibm.connect.gateway.interfaces.*;
import com.ibm.connect.gateway.service.*;

/**
 * SetOutboundStatusConnector is a Java Connector that sets a successful status code for the
 * outbound response. This connector is called near the end of the protocol flow (after all other
 * steps that may have had problems have already successfully finished). This is necessary due to
 * the return code being set to an error value by default.
 */
public class SetOutboundStatusConnector implements GatewayConstants, JavaProgramConnectorInterface {


   /**
    * Default Constructor
    */
   public SetOutboundStatusConnector()
   {
   }


   /**
   * Connector interface method implementation.
   * @param ProgramConnectorParm the data object passed to all connectors
   * @return JavaConnectorResult containing the return code and text
   */
   public JavaConnectorResult run(ProgramConnectorParm parms)
   {
      if (UserLogManager.isTracing) {
         UserLogManager.trace(this, "run", "Entering run method of SetOutboundStatusConnector.");
      }
      TransportI transportInterface      = (TransportI)
parms.flowDataAreaGet(GatewayConstants.TRANSPORT_INTERFACE);
      transportInterface.setOutboundStatus(200);

      JavaConnectorResult result = new JavaConnectorResult();
      result.setReturnCode(GatewayConstants.EXIT_SUCCESSFUL);
      result.setReturnString("SetOutboundStatus Connector exited successfully");
      if (UserLogManager.isTracing) {
         UserLogManager.trace(this, "run", "Exiting run method of SetOutboundStatusConnector.");
      }
      return result;
   }

}
```

Source for ExceptionHandlerConnector.java program

```
package InsuranceXML;

import com.ibm.connect.logging.interfaces.*;
import com.ibm.connect.flowmanager.interfaces.*;
import com.ibm.connect.flowmanager.*;
import com.ibm.connect.gateway.interfaces.*;
import com.ibm.connect.gateway.service.*;
import com.ibm.connect.bridge.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.*;
import java.util.Date;

/**
 * ExceptionHandlerConnector is a subclass of ResponseDOMPrimingConnector that gets control whenever an error occurred
 * during a flow.  This object creates a new output document consisting only of a
 * response tag followed by a Status and Date_Time tag containing error information.
 */
public class ExceptionHandlerConnector extends ResponseDOMPrimingConnector implements JavaProgramConnectorInterface
{

   /**
    * This constructor calls the super class constructor method.
    *
    */
   public ExceptionHandlerConnector()
   {
      super();
   }
   /**
    * This method looks in the FlowError and/or APPError from ProgramConnectorParm.
    * Depending upon the error, a code and text are set.
    * The super method is called to construct a Document. The Status element is modified to show the real error condition.
    * Text is appended beneath the Status element, containing the long text from the ErrorInfoString.
    * Finally, the Document constructed here is placed in the FlowDataArea object to be passed back to the client.
    *
    * @param parms ProgramConnectorParm that is passed in for each connector
    * @return JavaConnectorResult result object returned
    */
```

```
public JavaConnectorResult run(ProgramConnectorParm parms)
{
   UserLogManager.logInfoMessage(this, "run", "Entering ExceptionHandlerConnector.");
   //  The super method will build a simple response object with a Status and Date_Time field.
   //  We have to fill in the status file with the error condition
   super.run(parms);
   JavaConnectorResult result = new JavaConnectorResult();

   //Initialize the result to indicate success
   result.setReturnCode(0);
   result.setReturnString("");

   // make sure outbound status is set
   TransportI transport = (TransportI) parms.flowDataAreaGet(GatewayConstants.TRANSPORT_INTERFACE);
   transport.setOutboundStatus(200);

   // Get return code that might be set by the Connector
   int appErrorCode = 0;
   int flowErrorCode = 0;
   int errorCode = 0;
   String errorText="";
   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();
         errorCode = 500;
         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;
            }

         }
         if (UserLogManager.isTracing) {
            UserLogManager.trace(this, "run", "Error origin is Connector, code is " + appErrorCode + ", error text is
"+parms.getAPPErrorInfoString());
         }
      }
```

```
            else {
               // The failure is caused by some other flow internal error
               if (UserLogManager.isTracing) {
                  UserLogManager.trace(this, "run", "Error origin is Flow Manager, code is " + flowErrorCode + ", error text is "+
parms.getFlowErrorInfoString());
               }
            }
         }
         else {
            // flow error code is zero, should have never got here because this is an error step connector
            if (UserLogManager.isTracing) {
               UserLogManager.trace(this, "run", "Flow Error code not set, should have never got here.");
            }
         }
         // Crawl down through the response DOM hierarchy, beginning at the root,
         // to find the status and datetime fields so they can be filled in with the error information.
         Element root = document.getDocumentElement();
         Element responseElement = findElement(root, "Response");
         Element statusElem=findElement(responseElement, "Status");
         Element dateTimeElem=findElement(responseElement, "Date_Time");
         stripText(statusElem);
         stripText(dateTimeElem);
         // Create the new text value for the Status element
         Text statusText=document.createTextNode("ApplicationErrorCode="+appErrorCode+" "+"errorCode="+errorCode+"
errorText="+errorText+" "+parms.getAPPErrorInfoString()+" "+parms.getFlowErrorInfoString());
         // Create the new text value for the Date_Time element
         Date date=new Date();
         Text dateTimeText=document.createTextNode(date.toString());
         statusElem.appendChild(statusText);
         dateTimeElem.appendChild(dateTimeText);
         // set the response document
         parms.setResponseDOM(document);
         UserLogManager.logInfoMessage(this, "run", "ExceptionHandlerConnector run method finished.");
         return result;
      }
      /**
       * Given an element and a name, this routine returns the child element with that name
       *
       * @param Element
       * @param Element name
       * @return Child element
       */
      Element findElement(Element root, String desiredElementName) {
         Element desiredElement=null;
         Element currentElement=null;
         if (root.hasChildNodes()) {
            NodeList children = root.getChildNodes();
            for (int k=0; k<children.getLength(); k++) {
               if (children.item(k).getNodeType() == org.w3c.dom.Node.ELEMENT_NODE) {
                  currentElement=(Element)children.item(k);
                  if (currentElement.getNodeName() == desiredElementName) {
                     desiredElement=currentElement;
                     break;
                  }
```

```
            }
          }
        }
        return desiredElement;
    }
    /**
     * Given an element, this routine strips any text element from it
     *
     * @param Element name
     * @return void
     */
    void stripText(Element elem) {
        if (elem.hasChildNodes()) {
            NodeList children2 = elem.getChildNodes();
            for (int kk=0; kk<children2.getLength(); kk++) {
                if (children2.item(kk).getNodeType() == org.w3c.dom.Node.TEXT_NODE) {
                    elem.removeChild((Text)children2.item(kk));
                    break;
                }
            }
        }
    }
}
```

Source for GenericFMJavaConnector.pcml program definition file

```
<pcml version="1.0">
 <!-- PCML source for calling a punchout pgm -->
  <program name="GenericFMJavaConnector" path="unused">
  <struct name="InsuranceXML" usage="output">
    <struct name="Response" usage="output">
      <data name="Status" type="char" length="32" usage="output" />
      <data name="DateTime" type="char" length="32" usage="output" />
      <struct name="GetInsuranceCoverageResponse" usage="output">
        <data name="CoverageAmount" type="char" length="32" usage="output" />
      </struct>
    </struct>
  </struct>
  </program>
</pcml>
```

Source for GenericFMJavaConnector.java program

```
/* ---------------------------------
```

```
POSSIBILITY OF SUCH DAMAGES.
------------------------------------ */
import com.ibm.connect.flowmanager.interfaces.*;
import com.ibm.connect.flowmanager.metadata.Field;
import java.util.Enumeration;
import java.util.Vector;
import java.util.Date;
import com.ibm.connect.logging.interfaces.UserLogManager;


/**
 * <code>GenericFMJavaConnector</code> implements the Connector Interface to provide a simple and generic Flow Manager
 * connector that supports the full request/response flow for the Connect custom protocol examples.
 */
public class GenericFMJavaConnector implements JavaConnectorInterface {

   Field statusField = null;
   Field datetimeField = null;
   Field coverageamountField = null;
   final String statusFieldName="InsuranceXML/Response/Status";
   final String datetimeFieldName="InsuranceXML/Response/DateTime";
   final String coverageamountFieldName="InsuranceXML/Response/GetInsuranceCoverageResponse/CoverageAmount";
   ConnectorParm myParms = null;
   JavaConnectorResult result = new JavaConnectorResult();

   /**
    * <code>run</code> actual implementation of the Connector Interface that gets called by the Flow Manager
    */
   public JavaConnectorResult run(ConnectorParm parms) {
      myParms = parms;

      System.out.println("------------------------------------------------------------------------  ");
      System.out.println("In GenericFMJavaConnector code! ");
      System.out.println("------------------------------------------------------------------------  ");
      try {
         bindfields(parms.getOutputFieldList());
         parms.setFieldFromString(statusField,"Successful");
         Date date=new Date();
         parms.setFieldFromString(datetimeField,date.toString());
         parms.setFieldFromString(coverageamountField,"$50000");
         result.setReturnCode(0);
         result.setReturnString("Generic FM Connector program is now returning.");
      }
      catch (SetFieldException e) {
         e.printStackTrace();
         result.setReturnCode(01);
         result.setReturnString(e.getMessage());
      }


      //Initialize the result to indicate success
      return result;
   }
   /**
```

```
* <code>bindfields</code> is the method that gets called to do all of the field
* binding for the corresponding connector program.
*/
private void bindfields(Vector fieldList) {
   Enumeration enum=fieldList.elements();
   while (enum.hasMoreElements()) {
      bindfieldsLoop((Field)enum.nextElement());
   }
   return;
}
/**
* <code>bindfieldsLoop</code> is the method that handles the does the actual binding of the fields or does
* recursive calls to handle any structures that are encountered.
*/
private void bindfieldsLoop(Field fld) {
   try {
      if (fld.isRepeating()) {
         // Need to add code here to handle this
      }
      if (fld.getType() == ConnectorConstants.STRUCT) {
         bindfields(myParms.getOutputFieldList(fld));
      }
      if (UserLogManager.isTracing) {
         UserLogManager.logInfoMessage(this, "setfieldsLoop", "Binding field " + fld.getName() + " to cursor...");
      }
      myParms.bindOutfieldToCursor(fld);
      setSpecificField(fld);
   }
   catch (BindException e) {
      e.printStackTrace();
      result.setReturnCode(01);
      result.setReturnString(e.getMessage());
   }
   return;
}
/**
* <code>setSpecificField</code> is the method that sorts out which field is being worked with and preserves it under
* the corresponding field variable that represents it.
*/
private void setSpecificField(Field fld) {
   String fldName=fld.getName();
   if (fldName.equals(statusFieldName)) {
      statusField=fld;
   }
   else if (fldName.equals(datetimeFieldName)) {
      datetimeField=fld;
   }
   else if (fldName.equals(coverageamountFieldName)) {
      coverageamountField=fld;
   }
}
/**
* GenericFMJavaConnector constructor
*/
```

```
    public GenericFMJavaConnector() {
        super();
    }
}
```

Appendix B

Source for XMLToDOMConnector.properties for example 2

```
# there are 3 properties
# UserSpecifiedDTD -- do you want to specify a DTD, or use the one specified in the document
# if UserSpecifiedDTD = yes, then there is
# UserSpecifiedDTDType = [file | url]
# UserSpecifiedDTDLocation = [ /path/to/file | http://www.blah.com/  ]
UserSpecifiedDTD = yes
UserSpecifiedDTDType = file
UserSpecifiedDTDLocation = Gateway/Connectors/InsuranceXML-LifePartners-1.02/InsuranceXML_102.DTD
```

Source for XMLBuilderConnector.properties for example 2

```
 NoValidationDTD = Gateway/Connectors/NoValidation.DTD
LocalDTDName = Gateway/Connectors/InsuranceXML-LifePartners-1.02/InsuranceXML_102.dtd
```