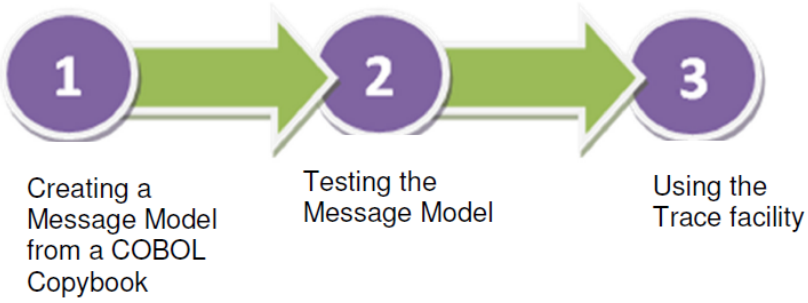
Lab 4 Modeling fixed-length data using a COBOL copybook

4.1 Introduction

In this lab, you will create a message model from a COBOL Copybook. Then you will use the DFDL Test Tooling to test parse valid and malformed data files. In the last part of the lab you will investigate the trace facility.



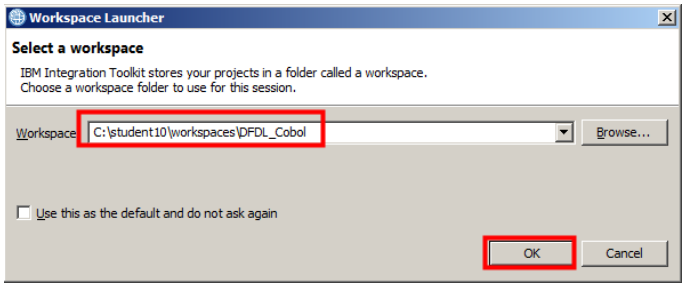
4.2 Creating a message model from a COBOL Copybook

This lab shows you how to create a message model based on a fixed length COBOL Copybook format. You will create the message model using the message model wizard and providing a *.cpy file as input.

__1. Start the IBM ACE Toolkit (if not already started).

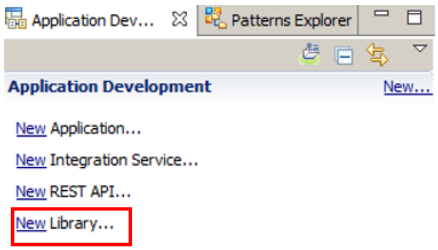
You are prompted to select an Eclipse Workspace

__1. Enter the workspace named **C:\student10\workspaces\DFDL_Cobol** (or alternative location of your choice). Click **OK**.



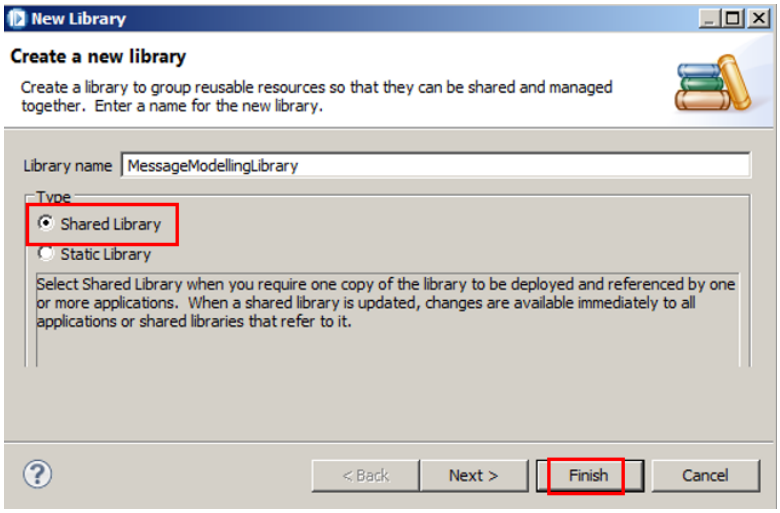
Create a new library named “MessageModellingLibrary”.

Click **New library.**

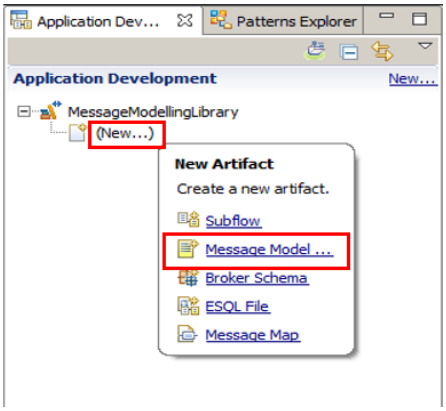


__2. You will see radio buttons to choose “Shared Library” or “Static Library”. For this lab you will use a shared library.

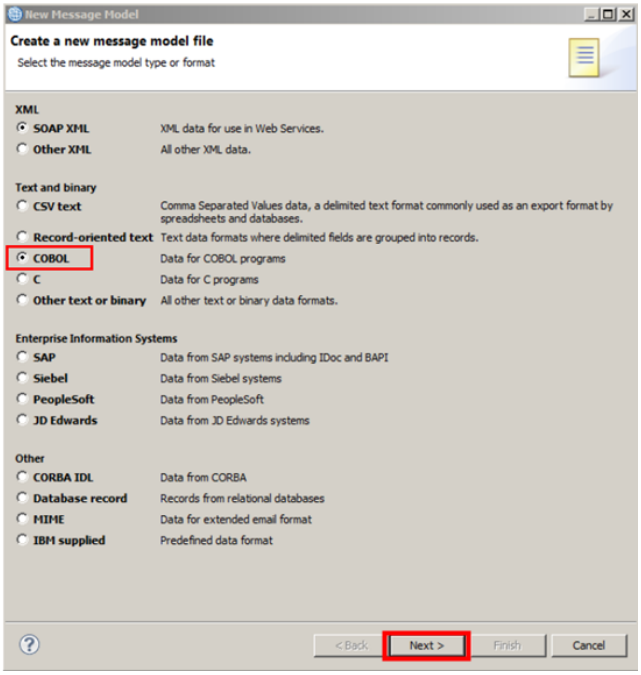
Click **Finish**.



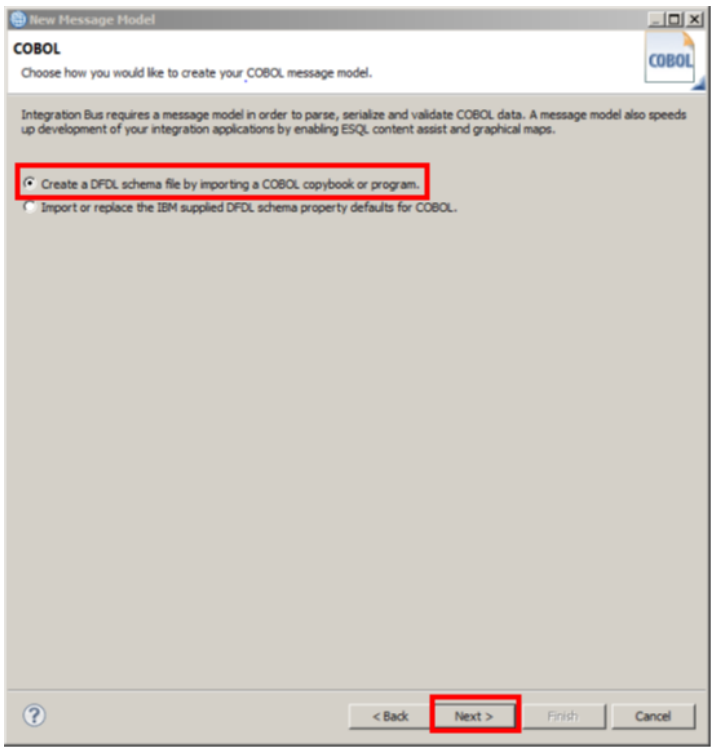
__3. In this library, click **New** and select **Message Model**.



__4. In the message model wizard, select **COBOL** and click **Next**.

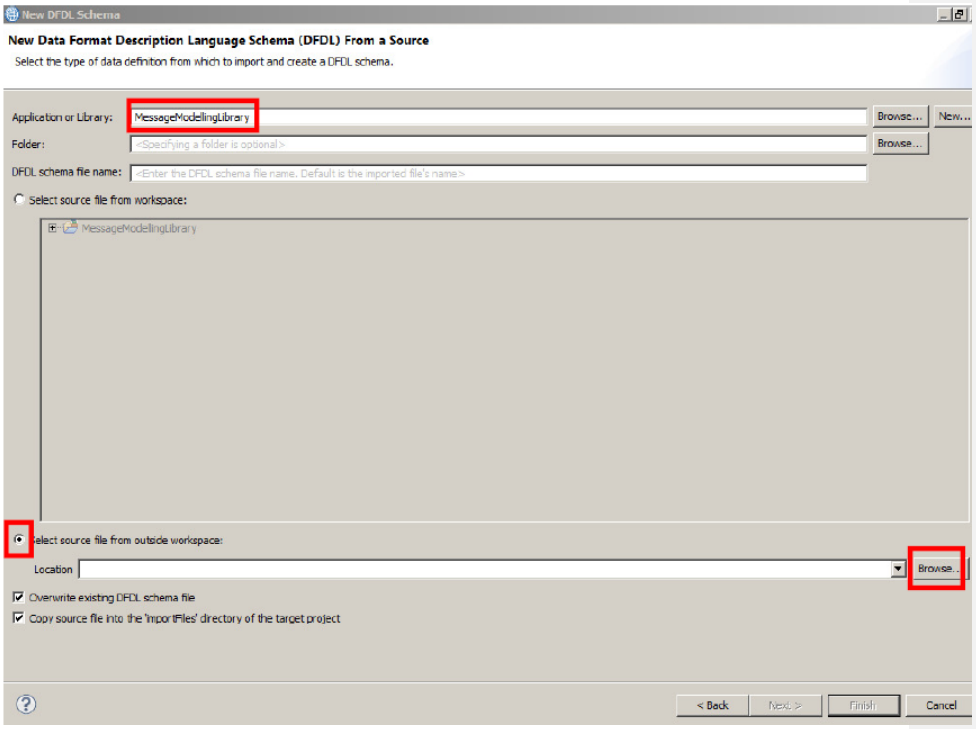


__5. Leave the default option and click **Next**.



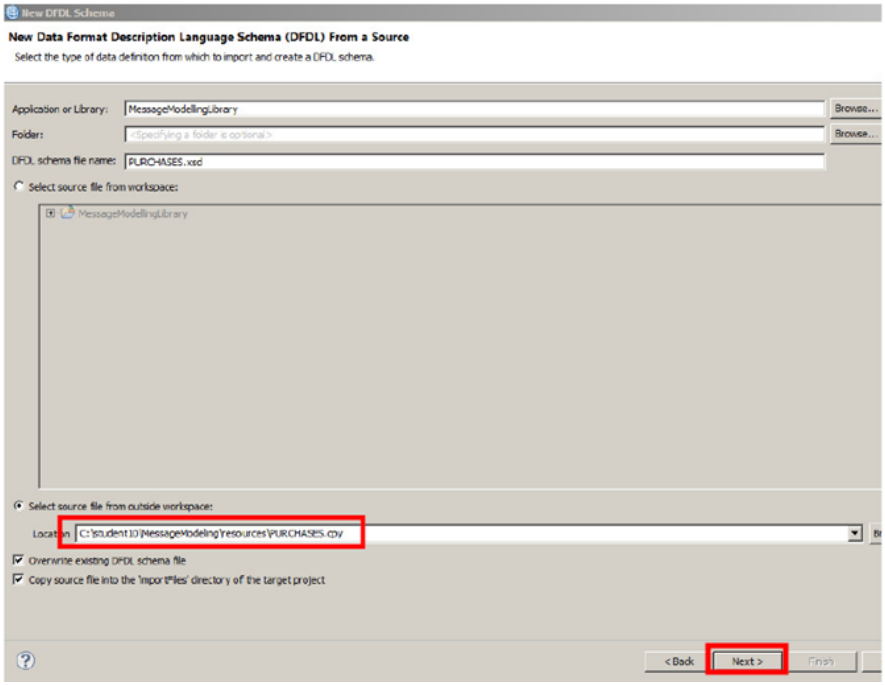
__6. Use the **Browse** button to set the **Application or Library** to “MessageModellingLibrary”.

Select **Select source file from outside workspace**. Click **Browse**.



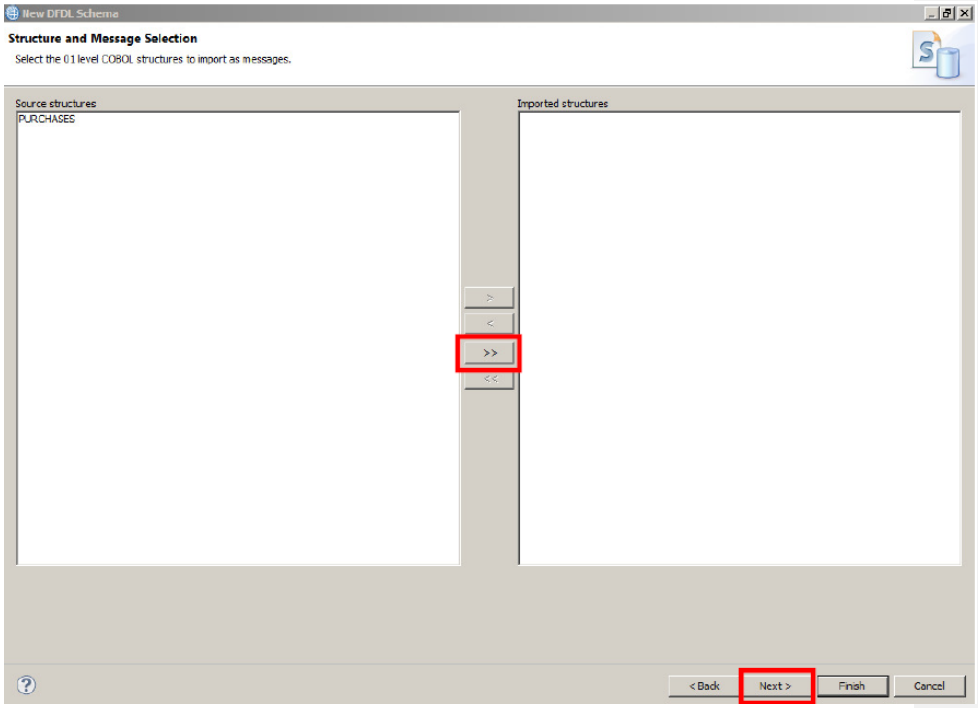
__7. Browse in **C:\student10\MessageModeling\resources** and select the file **PURCHASES.cpy**.

Click **Next**.



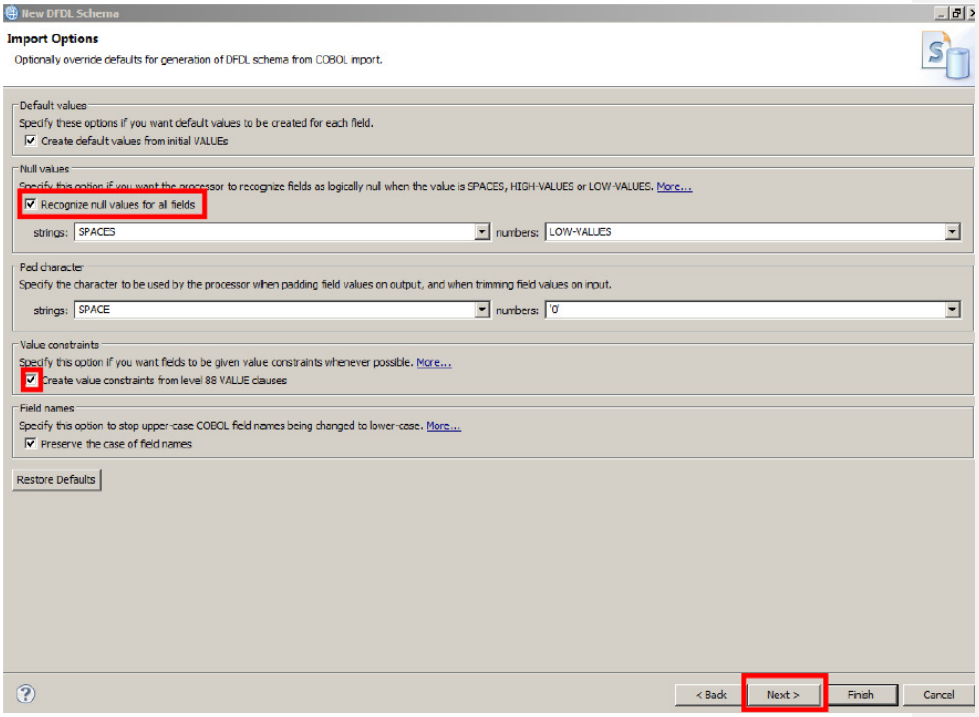
__8. Click the **>>** button to select all found objects (just one in this case) and click **Next**.

Do **NOT** click Finish.

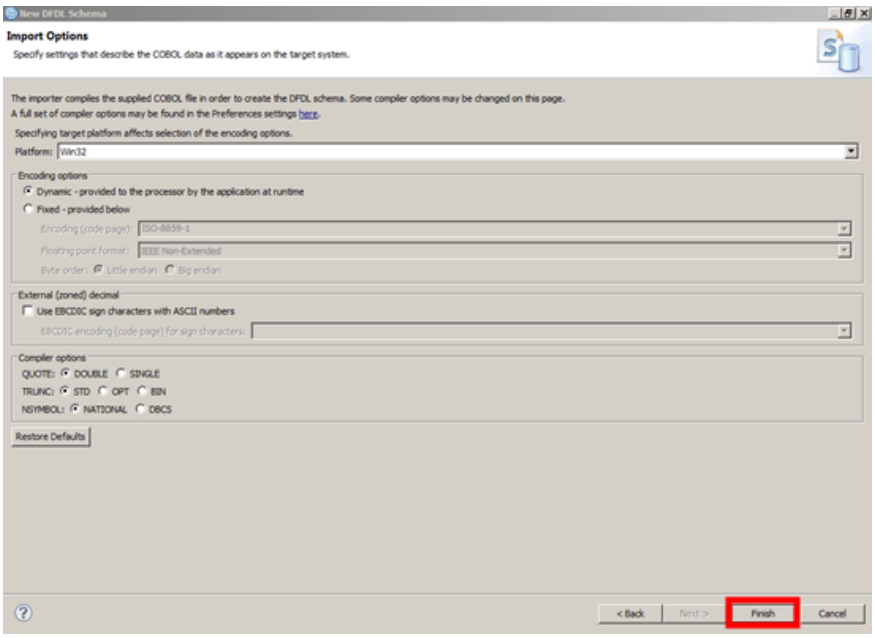


__9. Leave most of the default values, but select **Recognize null values for all fields** and **Create value constraints from level 88 VALUE clauses**.

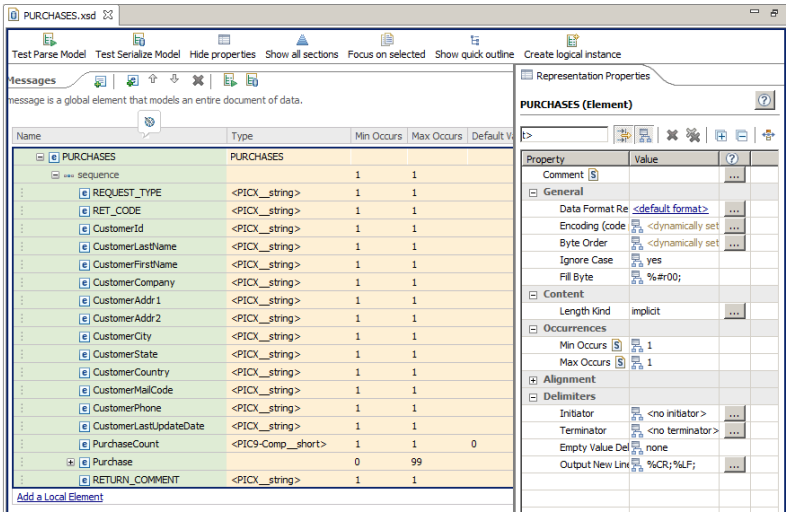
Click **Next**.



__10. Leave all the default values, and click **Finish**.

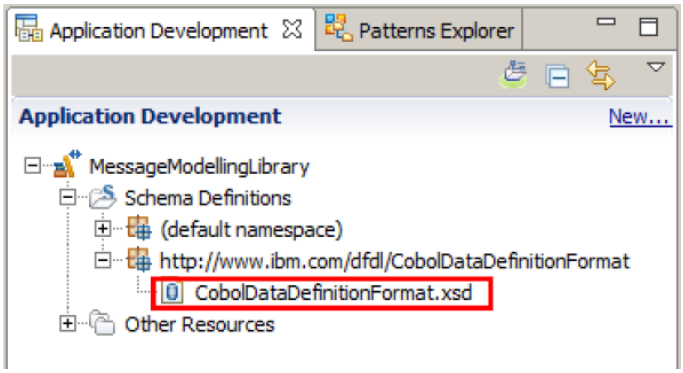


__11. The DFDL editor opens with the newly created DFDL message model called PURCHASES.xsd.



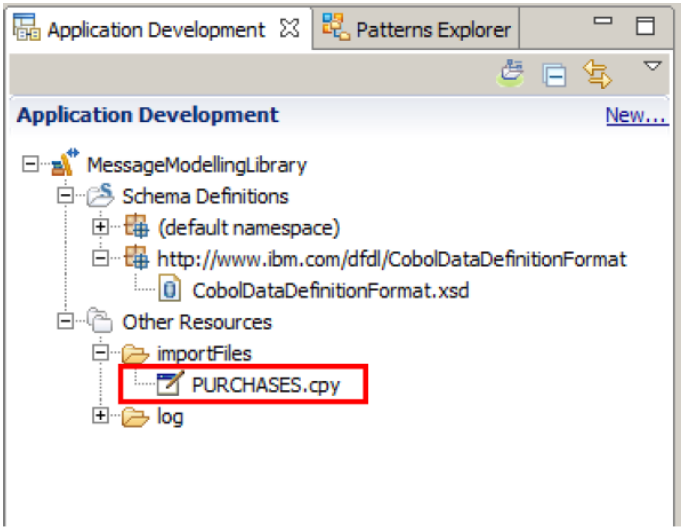
__12. Note that the wizard automatically added a file called "CobolDataDefinitionFormat.xsd" under the Schema Definitions in MessageModellingLibrary.

This file is referenced by PURCHASES.xsd as a schema import, and it contains COBOL-specific defaults for all the DFDL properties, and some pre-defined simple types.



__13. Expand the **Other Resources** folder under the MessageModelingLibrary library.

Expand the **importFiles** folder and you will see the PURCHASES.cpy file that the wizard has automatically imported.



__14. Double-click **PURCHASES.cpy** to open it in the editor.

```
01  PURCHASES.
    03  REQUEST-TYPE                PIC X.
    03  RET-CODE                    PIC XX.
    03  CustomerId                  PIC X(8).
    03  CustomerLastName            PIC X(20).
    03  CustomerFirstName           PIC X(20).
    03  CustomerCompany             PIC X(30).
    03  CustomerAddr1               PIC X(30).
    03  CustomerAddr2               PIC X(30).
    03  CustomerCity                PIC X(20).
    03  CustomerState               PIC X(20).
    03  CustomerCountry             PIC X(30).
    03  CustomerMailCode            PIC X(20).
    03  CustomerPhone               PIC X(20).
    03  CustomerLastUpdateDate      PIC X(8).
    03  PurchaseCount               PIC 9(3) USAGE COMP.
    03  Purchase OCCURS 0 TO 99 TIMES
        DEPENDING ON PurchaseCount.
      04  PurchaseId                PIC 9(5).
      04  ProductName               PIC X(30).
      04  Amount                    PIC 9(2).
      04  Price                     PIC 9(8)V99.
    03  RETURN-COMMENT              PIC X(50).
```

This is a simple copybook with:

- 14 string fields
- PurchaseCount: binary field with the number of the Purchase structure occurrences

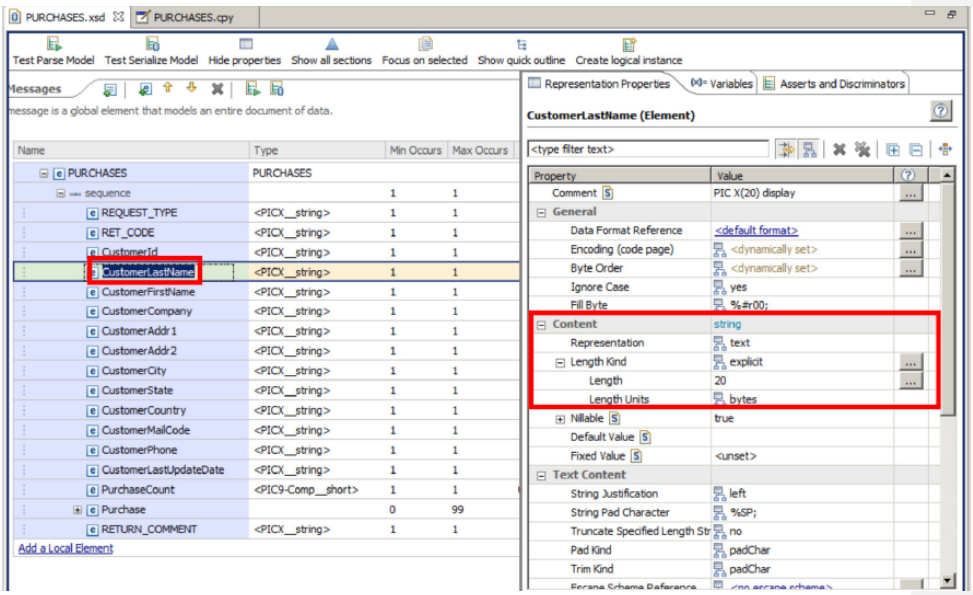
- Purchase: Repeating structure
  - PurchaseId, Amount: numeric fields
  - Price: numeric field with 2 decimal places

__15. Switch to the DFDL editor. For the PURCHASES.xsd this shows the string fields, defined as "PICX_string" by the import wizard:

Name	Type	Min Occurs	Max Occurs
PURCHASES	PURCHASES		
sequence		1	1
REQUEST_TYPE	<PICX__string>	1	1
RET_CODE	<PICX__string>	1	1
CustomerId	<PICX__string>	1	1
CustomerLastName	<PICX__string>	1	1
CustomerFirstName	<PICX__string>	1	1
CustomerCompany	<PICX__string>	1	1
CustomerAddr1	<PICX__string>	1	1
CustomerAddr2	<PICX__string>	1	1
CustomerCity	<PICX__string>	1	1
CustomerState	<PICX__string>	1	1
CustomerCountry	<PICX__string>	1	1
CustomerMailCode	<PICX__string>	1	1
CustomerPhone	<PICX__string>	1	1
CustomerLastUpdateDate	<PICX__string>	1	1
PurchaseCount	<PIC9-Comp__short>	1	1
Purchase		0	99
RETURN_COMMENT	<PICX__string>	1	1

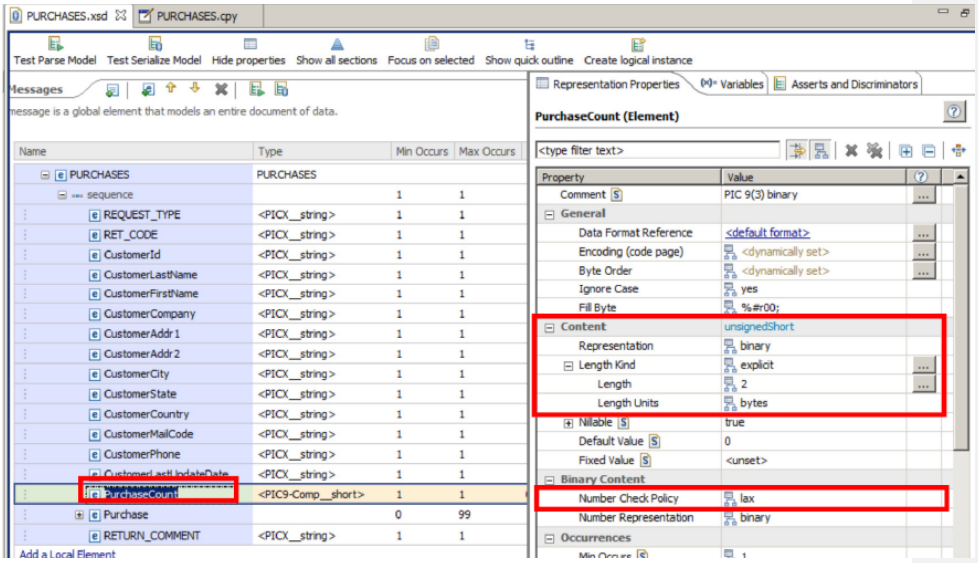
Add a Local Element

__16. In the DFDL Editor click the **CustomerLastName** field to see its properties:



In the properties view, look for the **Content** section. Note that the field was modeled as "text" representation, with a fixed (explicit) length of 20 bytes, because the cpy file defined it as a "PIC X(20)".

__17. In the DFDL Editor, click the **PurchaseCount** field to see its properties:



This field, which was defined as binary in the copybook file ("PIC 9(3) USAGE COMP"), was created as "PIC9_Comp_short" by the Import wizard.

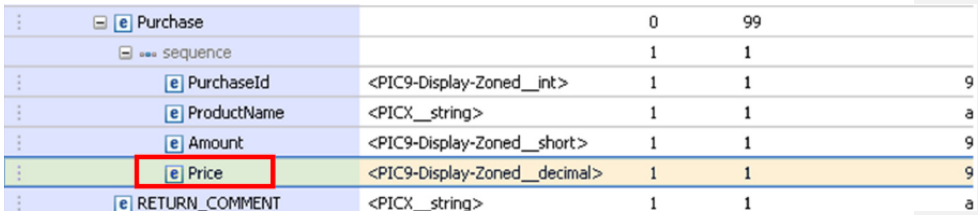
You can see the details of this field in the properties view, where its length is set to "2", its length units to "bytes" and its representation to "binary".

Also, in the "Binary Content" section, its binary number representation is set to binary. This property can take four different values:

- packed: represented as a packed decimal. Each byte contains 2 decimal digits except for the least significant byte, which contains a sign in the least significant nibble.
- bcd: represented as a binary coded decimal with 2 digits per byte
- binary: represented as 2' complement for signed types and unsigned binary for unsigned types
- ibm4690Packed: used by the IBM 4690 retail store devices

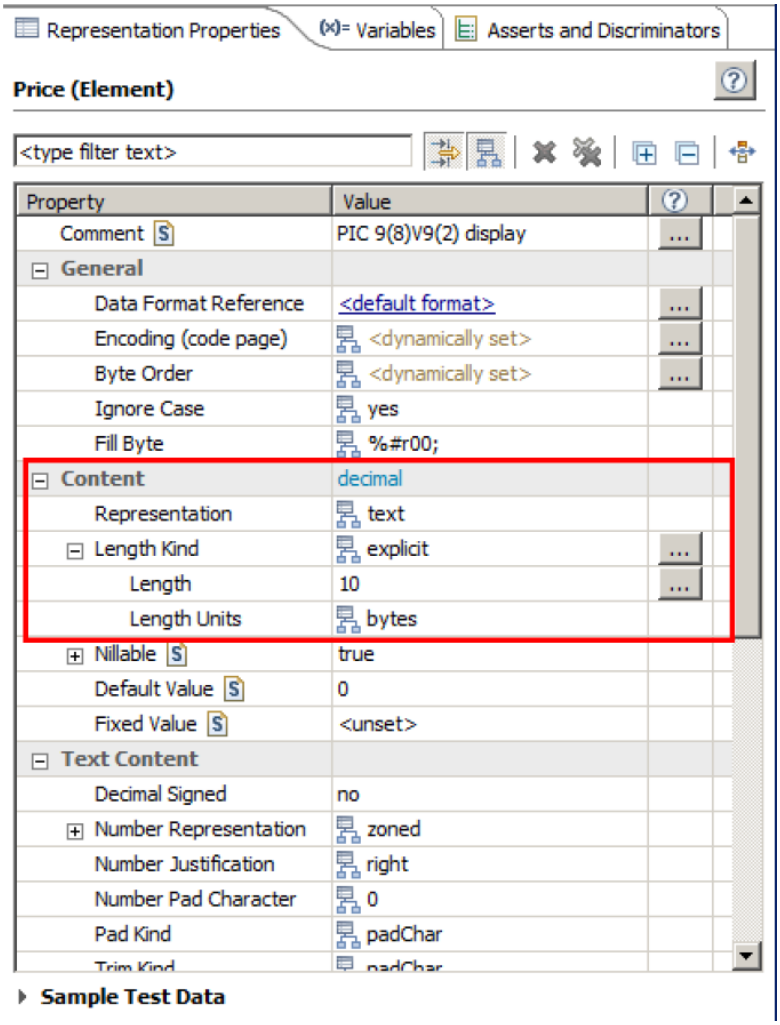
Commented [EL1]: Is this standard usage?

__18. Now click the **Price** field, in the Purchase structure.



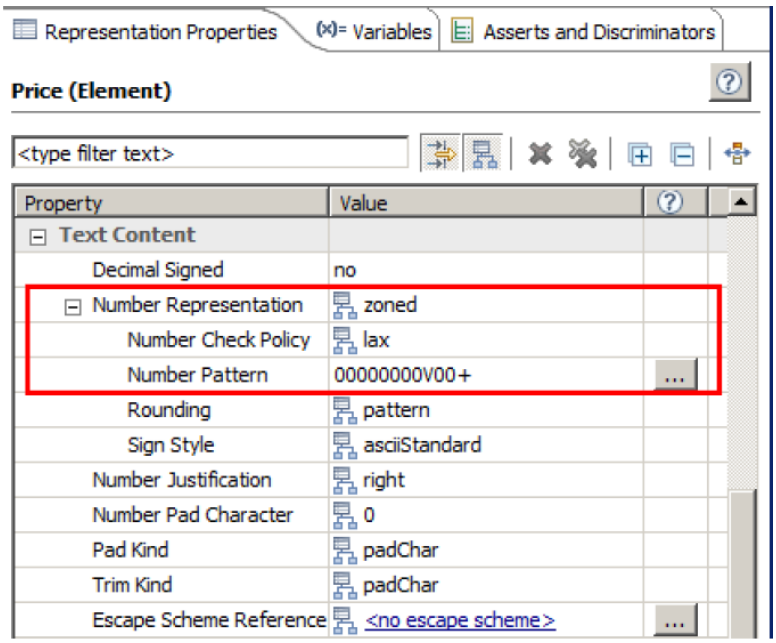
Purchase		0	99	
sequence		1	1	
PurchaseId	<PIC9-Display-Zoned__int>	1	1	9
ProductName	<PICX__string>	1	1	a
Amount	<PIC9-Display-Zoned__short>	1	1	9
Price	<PIC9-Display-Zoned__decimal>	1	1	9
RETURN_COMMENT	<PICX__string>	1	1	a

__19. In the properties view, look at the **Content** section.



Note that it is defined as a decimal field, with text representation and a length of 10 bytes (8 integers and 2 decimal places).

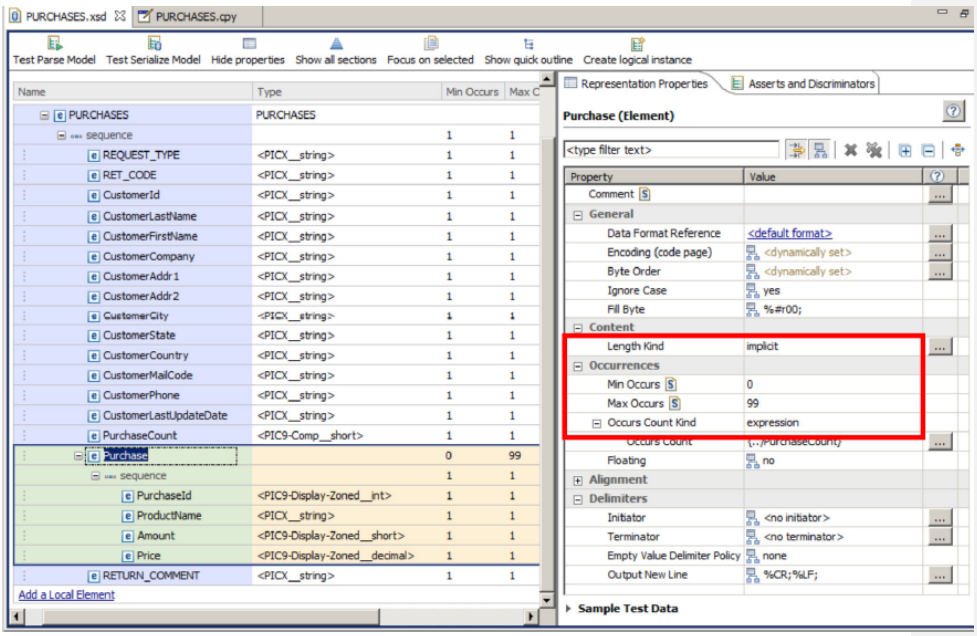
__20. Look at the **Text Content** section of the properties view.



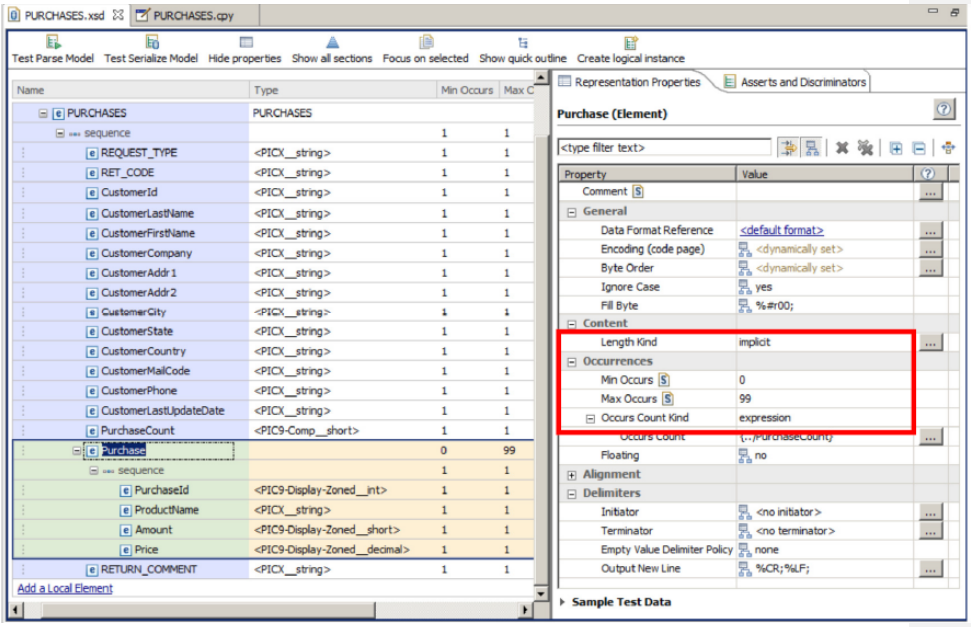
Note that the "Number Representation" is defined as "zoned", with a pattern of 8 integer numbers and 2 decimal places.

The letter “V” in the Number Pattern is an implied decimal point (common in COBOL copybooks).

__21. Click the **Purchase** element to open its properties.



__22. Look for the **Occurrences** section inside the properties view, and expand the **Occurs Count Kind** property.



This property, as defined by the DFDL specification, can take different values:

- fixed: uses the "maxOccurs" property
- expression: uses the value defined by the expression in "occursCount" property
- parsed: the number of occurrences is determined by normal speculative parsing
- implicit: uses “minOccurs” and “maxOccurs” properties with speculative parsing

In this case, the "OccursCountKind" property is set to "expression", and "occursCount" is set to point to the "PurchaseCount" element. This means that the number of occurrences of the "Purchase" repeating structure will be defined by the PurchaseCount element.

This was defined by the Import wizard to reflect the cpy file, which stated:

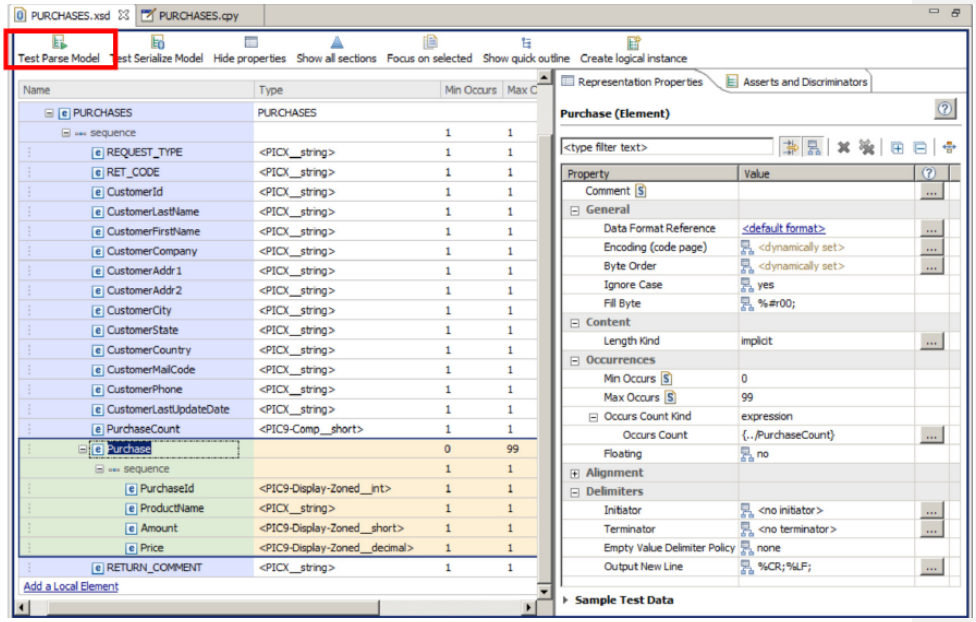
```
Purchase OCCURS 0 TO 99 TIMES DEPENDING ON PurchaseCount.
```

Also notice that the MinOccurs property is set to "0" and the MaxOccurs property is set to "99", as the cpy file stated.

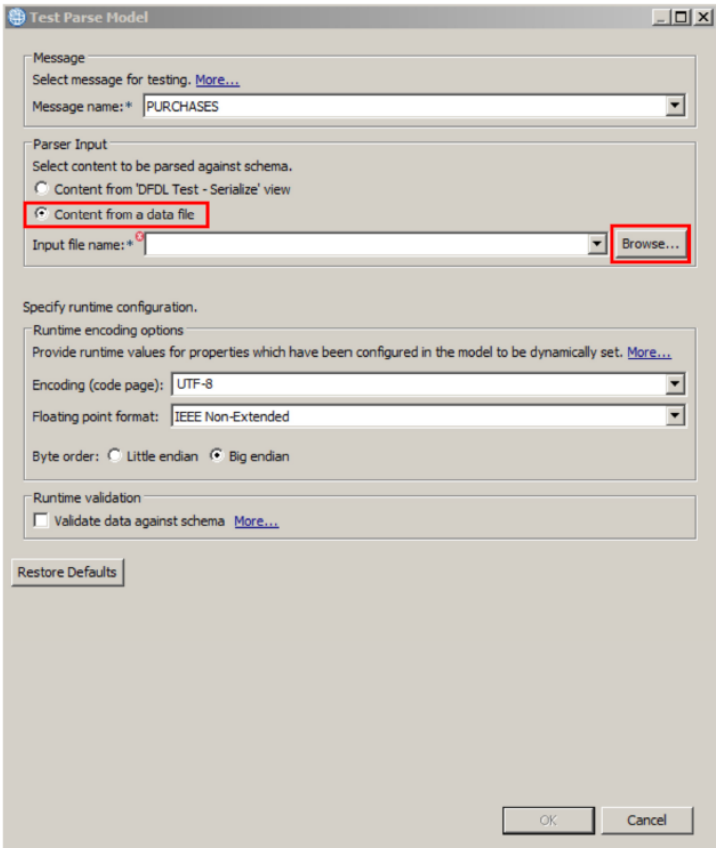
__23. Save your message model (PURCHASES.xsd) by pressing **Ctrl+S**, or **File > Save**.

4.3 Testing the message model

__24. Now you will use the DFDL test tooling to validate that the message model correctly models the COBOL data. Click the **Test Parse Model** icon.

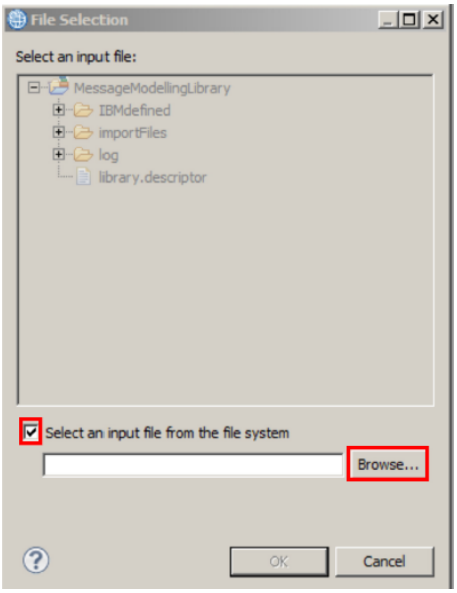


__25. In the Parser Input section, select **Content from a data file** and click **Browse**.



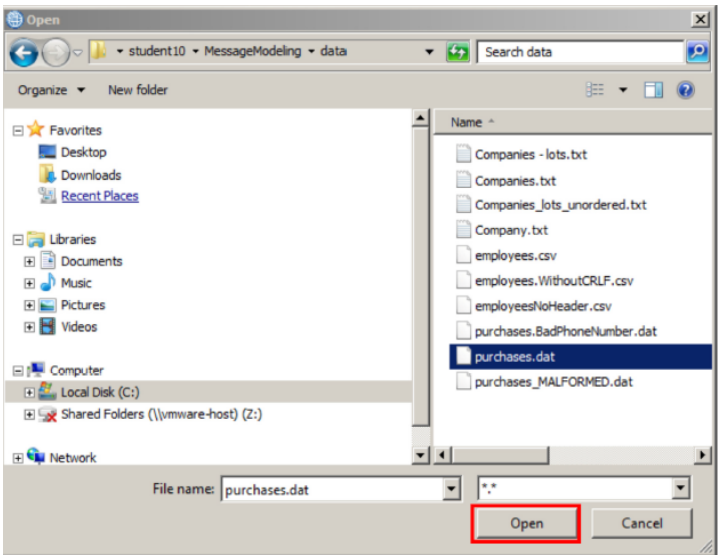
__26. In the File Selection window, select **Select an input file from the file system**.

Click **Browse**.

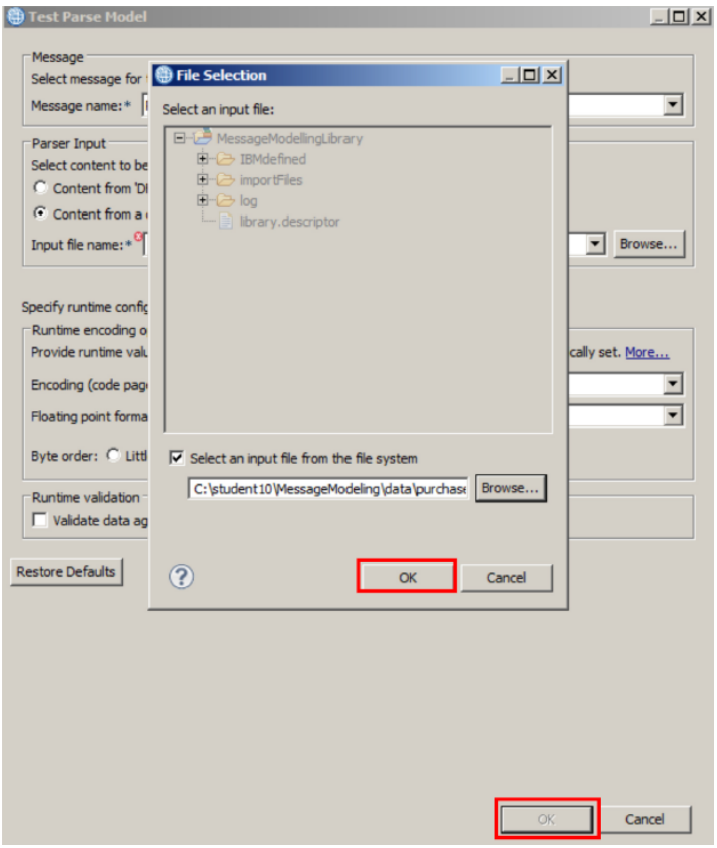


__27. Navigate to **C:\student10\MessageModeling\data** and select the **purchases.dat** file.

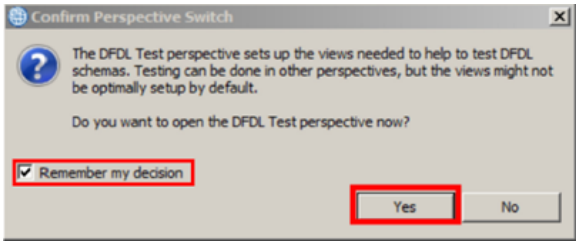
Click **Open**.



__28. Click **OK** on both windows.



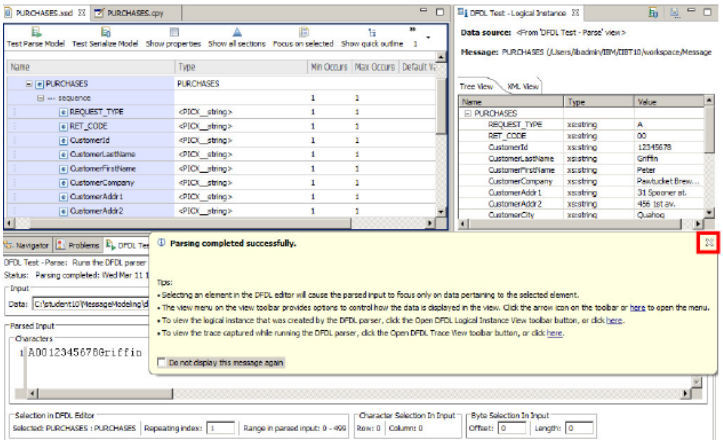
__29. Select **Remember my decision**, and click **Yes**.



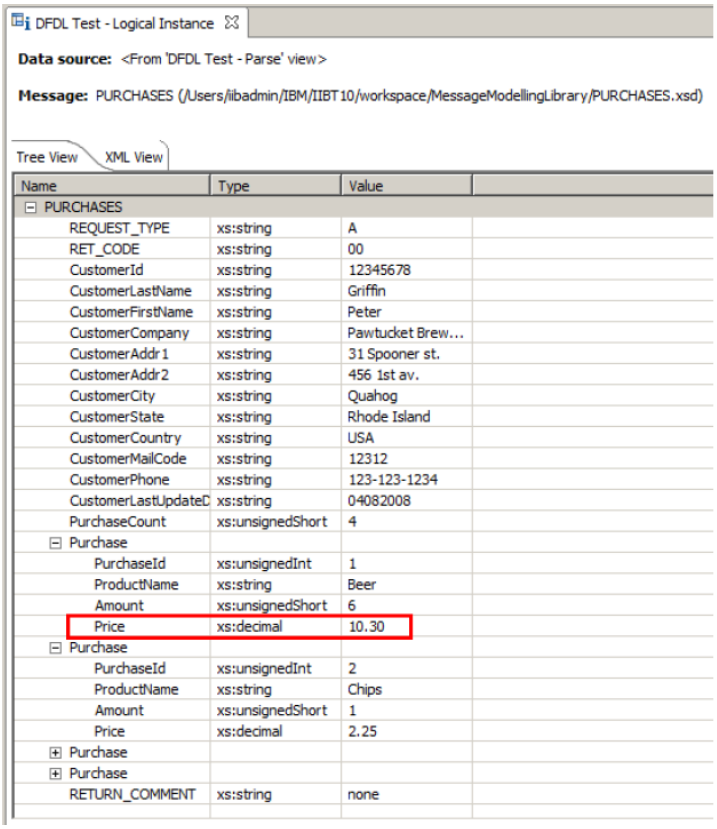
__30. The DFDL Test perspective will open, with the Test Parse view in focus.

A message balloon will appear, indicating the parsing was successful.

Close it by clicking on the **x**, or by clicking anywhere else in the workbench.



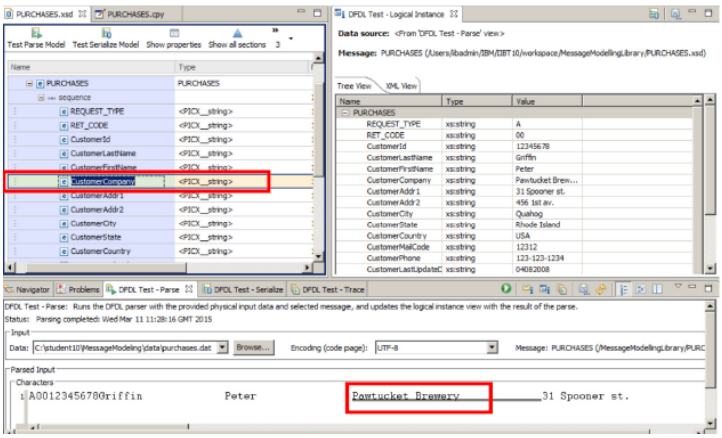
__31. Inspect the **Test - Logical Instance** view. Navigate through the message tree parsed from the input file.



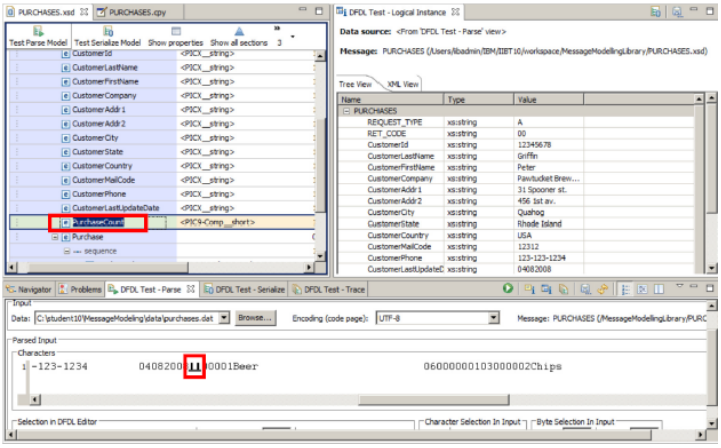
Name	Type	Value
PURCHASES		
REQUEST_TYPE	xs:string	A
RET_CODE	xs:string	00
CustomerId	xs:string	12345678
CustomerLastName	xs:string	Griffin
CustomerFirstName	xs:string	Peter
CustomerCompany	xs:string	Pawtucket Brew...
CustomerAddr1	xs:string	31 Spooner st.
CustomerAddr2	xs:string	456 1st av.
CustomerCity	xs:string	Quahog
CustomerState	xs:string	Rhode Island
CustomerCountry	xs:string	USA
CustomerMailCode	xs:string	12312
CustomerPhone	xs:string	123-123-1234
CustomerLastUpdateD	xs:string	04082008
PurchaseCount	xs:unsignedShort	4
Purchase		
PurchaseId	xs:unsignedInt	1
ProductName	xs:string	Beer
Amount	xs:unsignedShort	6
Price	xs:decimal	10.30
Purchase		
PurchaseId	xs:unsignedInt	2
ProductName	xs:string	Chips
Amount	xs:unsignedShort	1
Price	xs:decimal	2.25
Purchase		
Purchase		
RETURN_COMMENT	xs:string	none

Note that the parser shows "10.30" (2 decimal places) because the COBOL field was defined as PIC 9(8)V99.

__32. In the DFDL Editor, click any element on the message model and you will see the relevant data underlined in the input text below:

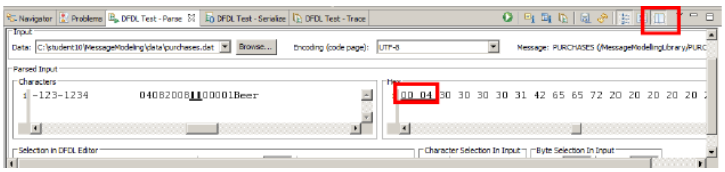


__33. Now click the **PurchaseCount** element.



Because it is a binary field, the highlighted value is not readable with this editor.

__34. Now click the **Show hex** button (top right of the lower pane, as highlighted below).



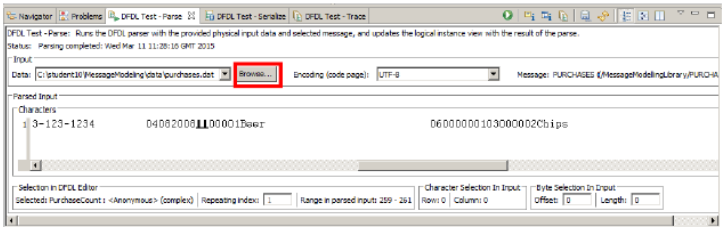
Note that the binary field is now readable, and has a value of "00 04" which corresponds to the 4 occurrences of the "purchase" element.

Click the **Show hex** button again to revert to normal display.

4.4 Using the Trace facility

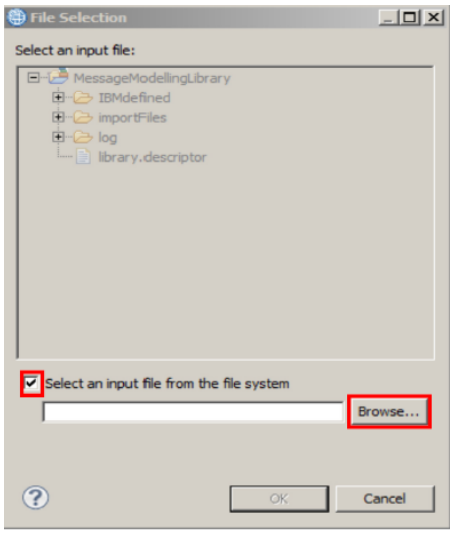
__35. Now you will test the message model using a malformed message.

In the DFDL Test perspective, "DFDL Test - parse" view, click **Browse**.

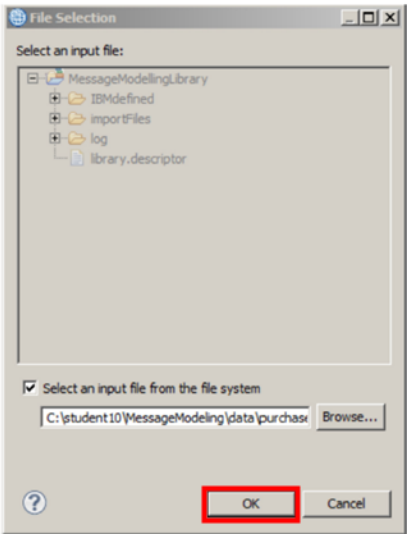


__36. In the File Selection window, select **Select an input file from the file system**.

Click **Browse**.



__37. Navigate to the **C:\student10\MessageModeling\data** directory and select the **purchases_MALFORMED.dat** file.



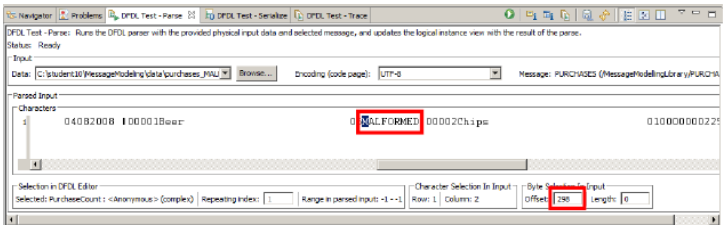
Click **OK**.

__38. In the **Offset** textbox, enter "298", and scroll right to find the highlighted character (byte 298 will be highlighted in blue).

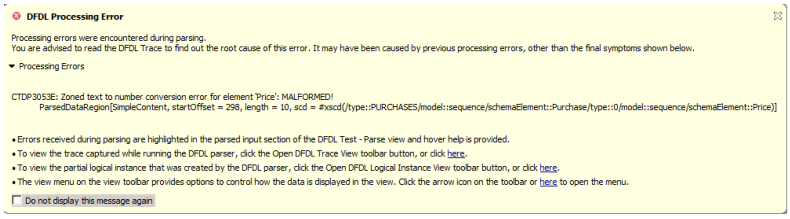
Note that at this position (the "Price" element position) there is a string "MALFORMED!" instead of the expected decimal number.

Now click the **Run parser** button to test the message model (green arrow shown below).

Commented [EL2]: It's not highlighted in this screenshot



__39. An error message will appear with the cause of the failed parsing.



__40. Inspect the **DFDL Test - Logical Instance**; you will see that the parsed tree is not complete.

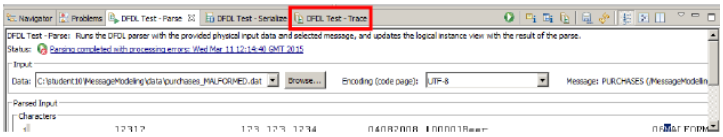
Go to the **Purchase** element, expand it, and check that it was correctly parsed until the "Amount" element. The following field "Price" is empty.

Tree View | XML View

Name	Type	Value
⊟ PURCHASES		
REQUEST_TYPE	xs:string	A
RET_CODE	xs:string	00
CustomerId	xs:string	12345678
CustomerLastName	xs:string	Griffin
CustomerFirstName	xs:string	Peter
CustomerCompany	xs:string	Pawtucket Brew...
CustomerAddr1	xs:string	31 Spooner st.
CustomerAddr2	xs:string	456 1st av.
CustomerCity	xs:string	Quahog
CustomerState	xs:string	Rhode Island
CustomerCountry	xs:string	USA
CustomerMailCode	xs:string	12312
CustomerPhone	xs:string	123-123-1234
CustomerLastUpdateD	xs:string	04082008
PurchaseCount	xs:unsignedShort	8196
⊟ Purchase		
PurchaseId	xs:unsignedInt	1
ProductName	xs:string	Beer
Amount	xs:unsignedShort	6

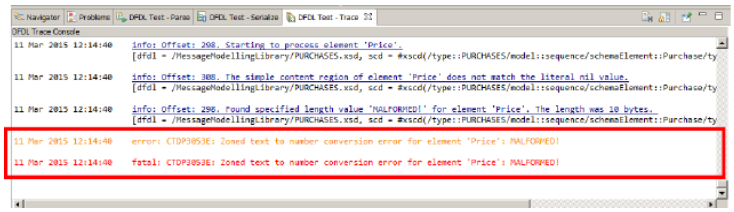
__41. Now you will use the **DFDL Test - Trace** view, to better understand the problem.

Click the **DFDL Test - Trace** view.

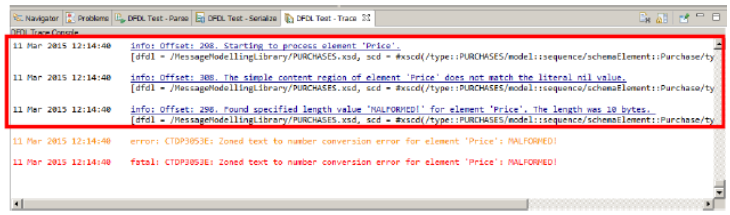


__42. In the **DFDL Test - Trace** view, you will find an execution log of the parsing activities.

At the end of the trace, there are colored lines with the found error.



__43. Look at the lines before the error:



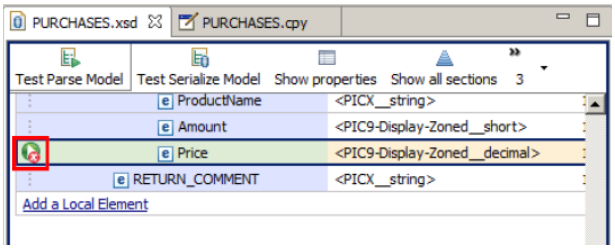
The first line states that it is starting to process the Price element.

In the third, it has found a string "MALFORMED!" as the value of the element.

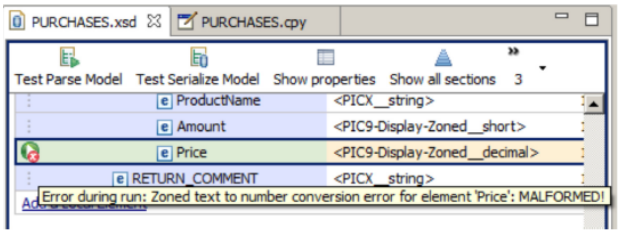
Then the parser tries to convert the string to a decimal number, and an error appears.

__44. Back in the DFDL Editor, scroll to the **Price** element.

Note that it has an error icon next to its name.



__45. Place the cursor on the error icon and a message will appear, showing the same error displayed in the trace view.



END OF LAB 4

Lab 5 Creating a REST API service

ACE V11 includes a type of project, or container, called a REST API. A REST API in App Connect Enterprise is a specialized application that allows you to expose a set of resources as a RESTful web service.

To create a REST API in App Connect Enterprise, you must first define the resources and operations that are available in the new REST API. Those definitions must be specified in a Swagger document, an open specification for defining REST APIs.

To create a REST API in App Connect Enterprise, you use the Create a REST API wizard in the IBM ACE Toolkit to import a Swagger document. You must then create the REST API sub-flows and other project artifacts that are required to implement and deploy the new REST API.

Operations defined in the REST API are implemented as normal sub-flows.

The REST API container automatically takes care of the routing of inbound HTTP requests to the correct sub-flow for the operation being called and processing of input parameters.

You simply need to connect the dots between the Input and Output nodes in each operation's sub-flow! Catch, Failure and Timeout handlers are implemented as sub-flows as well; ACE handles the HTTP response code automatically.

Commented [EL3]: There are three cases of "sub-flowsub-flows" here - this seems wrong and I assume it should just be "sub-flows", but please confirm

REST APIs support all of the ACE's features that you can use with applications (such as shared libraries, monitoring, activity log), and all message flow nodes can be used within a REST API.

A REST API application is deployed like existing applications. There are new artifacts, such as the Swagger JSON document, but ACE handles these automatically and you simply drag and drop the REST API application to deploy it from the Toolkit.

All administrative and operational controls that are available for applications in ACE are also available for REST API applications, this includes command-line programs. The ACE Web UI has been extended to provide information about resources, operations and parameters that are available in a deployed REST API application.

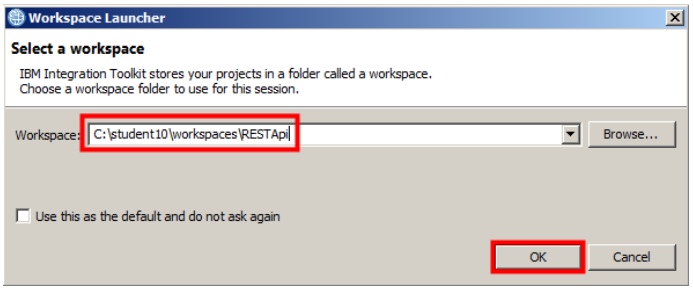
During this lab you will explore a Swagger JSON document that defines the EmployeeServiceV1 REST API, use the Create a REST API wizard to import the document and then implement five of the operations defined in the EmployeeServiceV1 REST API. You will then deploy and test the EmployeeServiceV1 REST API.

The REST API lab requires the EmployeeService SOAP web service that is implemented and deployed in Lab 6, "Implementing a service interface". If you did not perform the "Implementing a service interface" lab prior to this lab then follow the instructions in the following sections to import the necessary projects to your workspace and deploy the necessary artifacts to the myaceworkdir integration server.

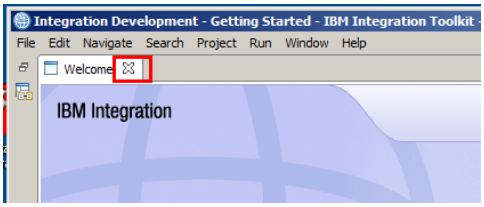
5.1 REST API lab workspace

__1. Open the ACE Toolkit, if not already open.

To start a new workspace browse to **C:\student10\workspaces\RESTApi** (or a directory of your choice) and click **OK**:

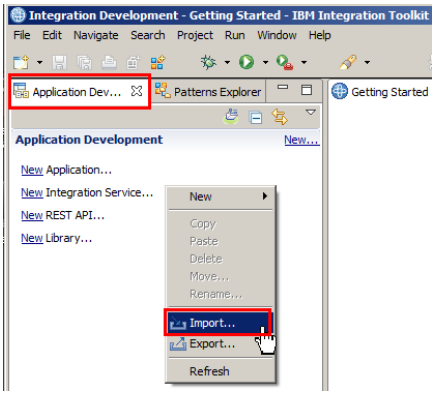


Close the **Welcome** view:

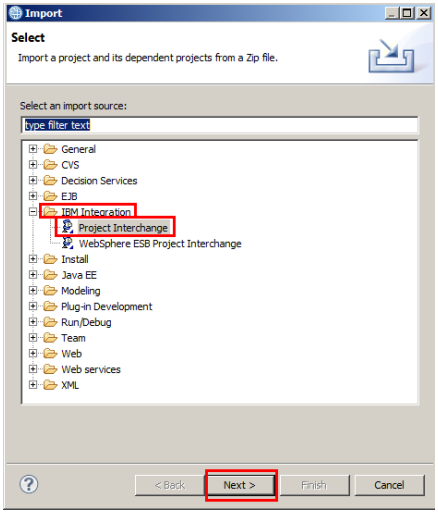


Since you have not completed the “Implementing a service interface” lab as part of the PoT then you must import the “Implementing a service interface” projects into your workspace along with the REST API projects. You must deploy the “Implementing a service interface” libraries to the myaceworkdir integration server.

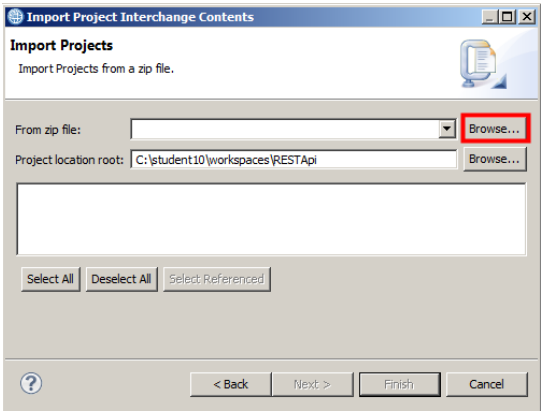
__2. Right-click in the **Application Development** view and select **Import**.



__3. Expand **IBM Integration**, select **Project Interchange** and click **Next**.

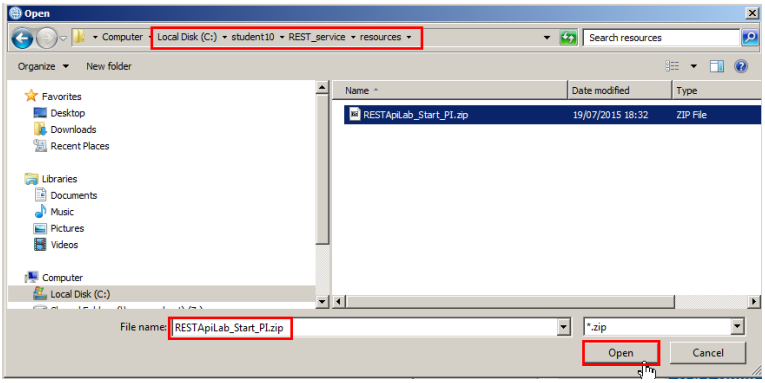


__4. Click **Browse**:



__5. Select the following Project Interchange file and click **Open**:

C:\student10\REST_service\resource\RESTApi_Start_PI.zip



__6. Select the following projects and click **Finish**:

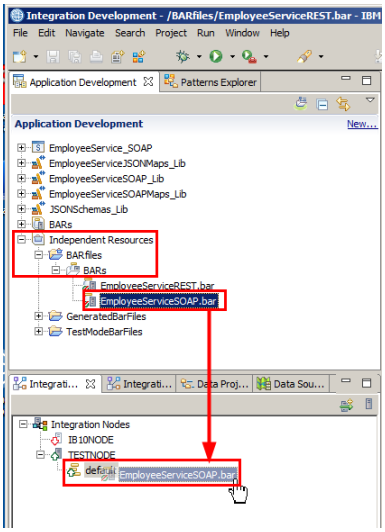
- BAR files
- EmployeeServiceJSONMaps_Lib
- EmployeeServiceSOAP_Lib
- EmployeeServiceSOAPMaps_Lib
- EmployeeService_SOAP
- JSONSchemas_Lib

If prompted, click **OK** to confirm the overwrite of BAR files.

5.2 REST API lab deployment

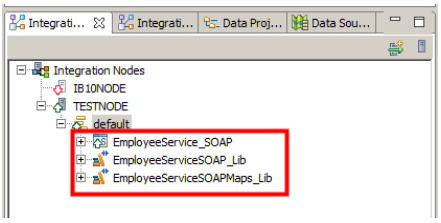
The REST API lab requires that the EmployeeService SOAP web service is deployed as well as REST API lab libraries. This SOAP service is already prepared for you and all you need to do is deploy it to the integration server.

In the Application Development view expand **Independent Resources > BAR files > BARs** and select **EmployeeServiceSOAP.bar**. Drag and drop the **EmployeeServiceSOAP.bar BAR** file to the **myaceworkdir** integration server:

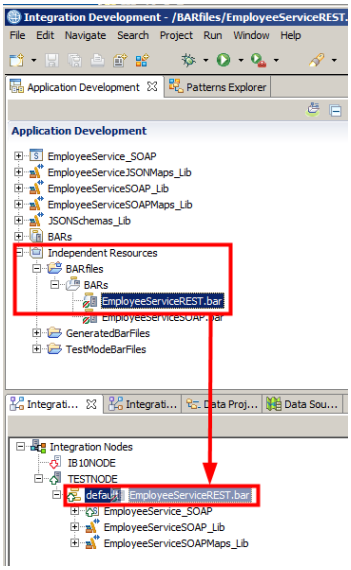


__7. You should see the following libraries and service interfaces deployed to the myaceworkdir integration server:

- EmployeeService_SOAP
- EmployeeServiceSOAP_Lib
- EmployeeServiceSOAPMaps_Lib

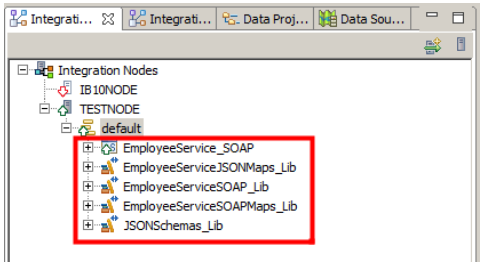


__8. In the Application Development view, expand **Independent Resources > BAR files > BARs** and select **EmployeeServiceREST.bar**. Drag and drop the **EmployeeServiceREST.bar** BAR file to the **myaceworkdir** integration server:



__9. You should see the following libraries and service interfaces deployed on the **myaceworkdir** integration server:

- EmployeeService_SOAP
- EmployeeServiceJSONMaps_Lib
- EmployeeServiceSOAP_Lib
- EmployeeServiceSOAPMaps_Lib
- JSONSchemas_Lib



5.3 Explore the EmployeeServiceV1 JSON document

In this section the EmployeeServiceV1.json document will be explored. The EmployeeServiceV1.json document defines the REST API that will be implemented in this lab. Although this lab will not cover the creation of the EmployeeServiceV1.json file, it is beneficial to understand the contents and structure of the document.

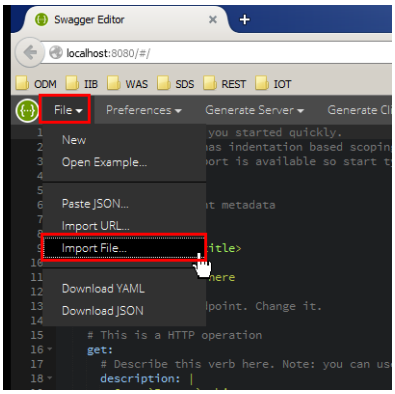
5.4 Using the Swagger editor

The goal of Swagger is to define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code or documentation. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

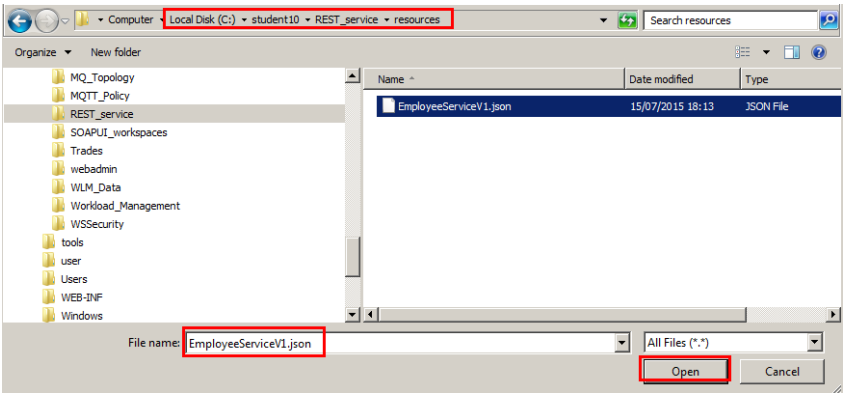
Swagger is more than just a way to describe REST APIs; it is also a powerful set of tools including an editor (Swagger editor) and UI (swagger-ui). The Swagger editor will be explored in this section of the lab.

__10. In your web browser, navigate to https://editor.swagger.io/

__11. You will now open the EmployeeServiceV1.json document. Click the **File** menu item and select **Import File**:



__12. Browse to **C:\student10\REST_service\resources**, select **EmployeeServiceV1.json** and click **Open**:



__13. Take a couple of minutes to explore the defined **paths** and **model definitions**. Observe that the associated YAML code is expanded in the editor in the left pane.

__14. Observe all of the JSON objects that are defined for the REST API.

__15. Expand the **tags/paths** section and note that there are many REST operations (GET, PUT, POST, DELETE) available, for **employees** and **departments.**

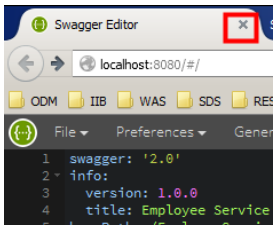
__16. Expand the **models section:**

The following models are defined:

- Department/Departments
- Employee/Employees
- DeptManager
- PredictedSalary

Observe the definition of these objects and arrays in the document. Later in the lab you will utilize XML Schemas and the Graphical Data Mapper to map these message models.

__17. Close the **Swagger Editor** tab in browser.



5.5 Create the REST API service

In this section you will create a new REST API service. This scenario will be based on the EmployeeServiceV1.json Swagger document that was explored in the previous lab section.

The wizard will be used to implement the EmployeeService REST API. You will also use the REST API project which contains and organizes the API assets in a context that makes implementation easy and structured while applying ACE design and development best practices.

Using the REST API wizard and project eliminates a large number of development tasks and allows the developer to focus on implementing the service interface.

In this lab you will implement **two** operations defined in the EmployeeServiceV1.json Swagger document. These operations will wrap operations that are implemented by the EmployeeService SOAP web service. If you performed the “Implementing a service interface” lab you are familiar with this web

service. If not, feel free to explore the project definitions and understand the implementation of SOAP services.

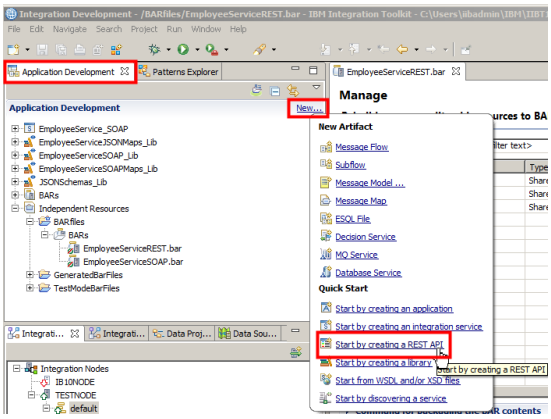
Each operation requires two maps – SOAP Request, SOAP Response – that have been pre-built to allow the lab to focus on the REST API development process. You will explore a few of these maps to observe how REST input parameters, JSON objects and arrays are mapped.

The REST API wizard does not generate schemas for models defined in the Swagger document, but as an alternative, XML schemas can be defined to model JSON objects and arrays. The models in the EmployeeServiceV1.json Swagger document are defined by the schemas in the JSONSchemas_Lib Library. This allows powerful GUI mapping for the REST API implementation.

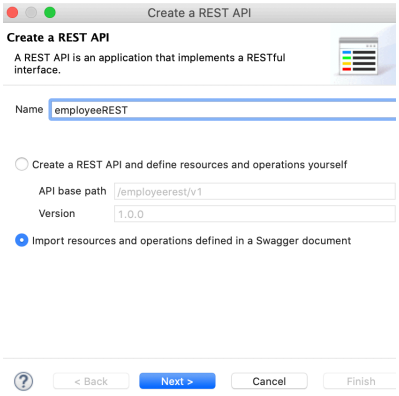
You will implement the following operations defined in the EmployeeServiceV1.json Swagger document:

- getDepartment (query parameter, returns JSON object)
- getDepartmentEmployees (path parameter, returns JSON array)

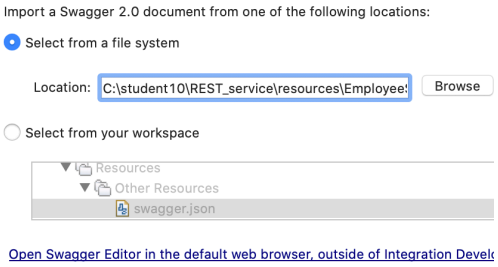
__18. In the Application Development view, select **New > Start By Creating a REST API**.



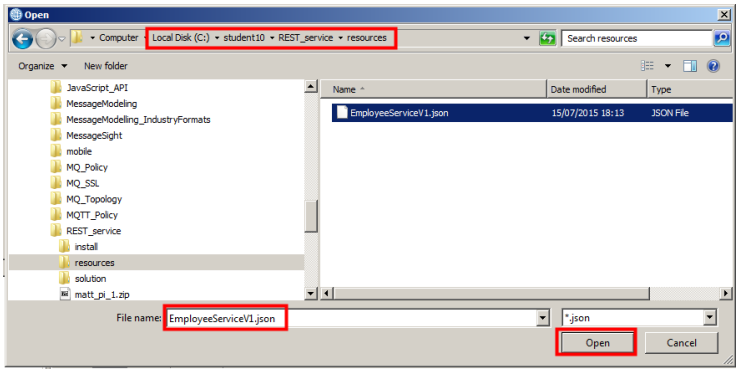
__19. Enter "**EmployeeService_REST**" for the name, choose the option to "**Import resources and operations…**" and click **Next.**



__20. Select **Select from a file system** and click **Browse**.

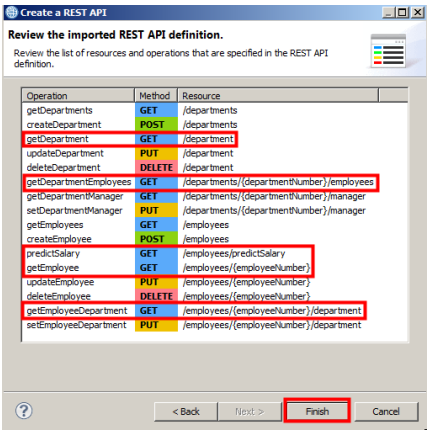


__21. Select **C:\student10\REST_service\resources\EmployeeServiceV1.json** and click **Open**:



__22. Click **Next** to see a list of the operations that are defined in the Swagger document:

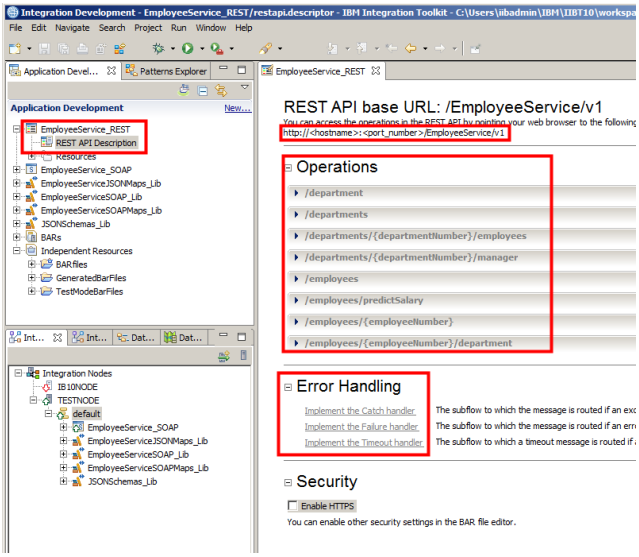
__23. Observe the operations that are defined. Two of these operations will be implemented in this lab. Click **Finish**:



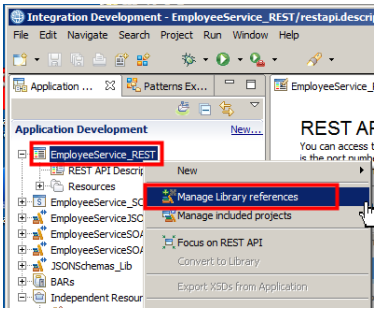
__24. Observe the REST API Project is created and the REST API Description is open.

Note the base URL for the REST API: http://hostname:portnumber/EmployeeService/v1.

Note the list of operations that are defined for the REST API and the Error Handling handlers that have been created. You will implement the **Catch** error handler during this lab.

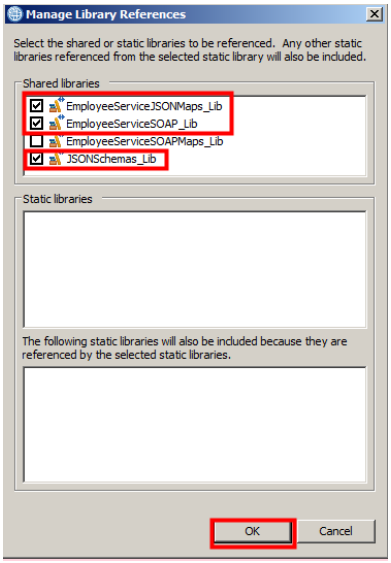


__25. Right-click **EmployeeService_REST** and select **Mange Library References**:

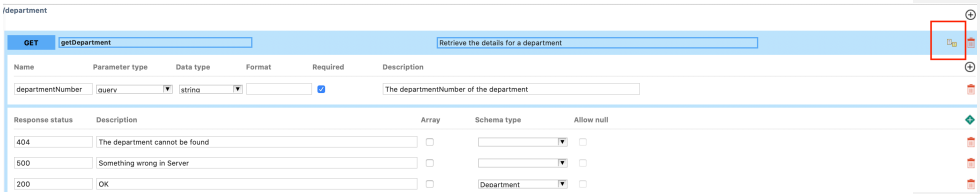


__26. Select the following libraries and click **OK**:

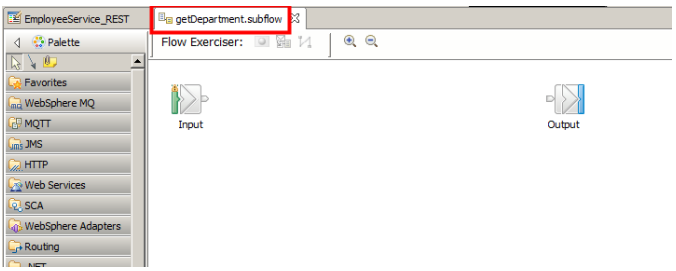
- EmployeeServiceJSONMaps_Lib
- EmployeeServiceSOAP_Lib
- JSONSchemas_Lib



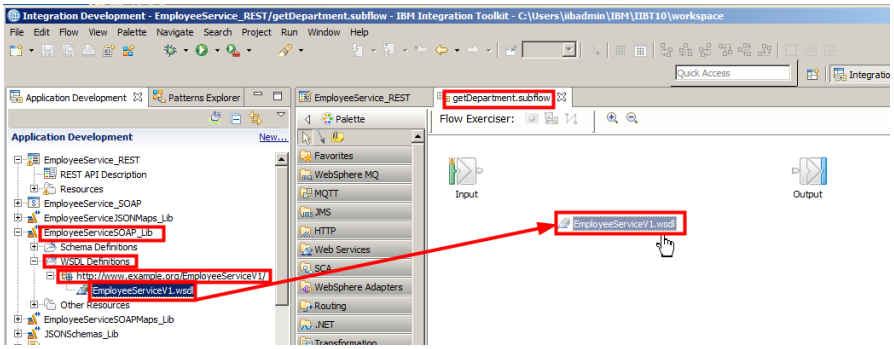
__27. You will now implement the getDepartment operation. Expand the **/department** path under **Operations**. Click **the highlighted icon** for the getDepartment operation:



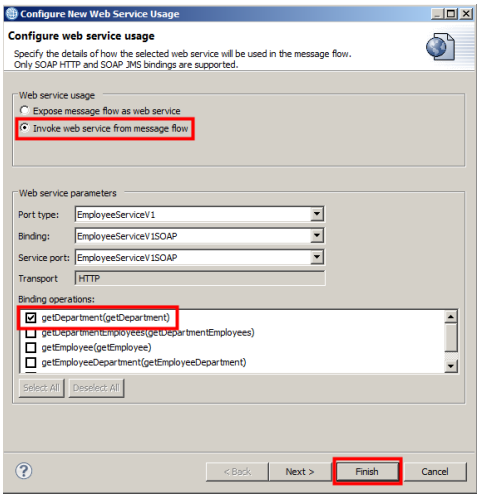
__28. The **getDepartment** sub-flow is opened in the Message Flow Editor:



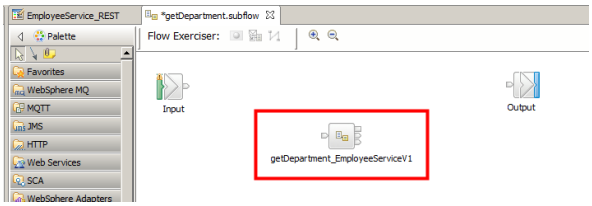
__29. The getDepartment REST operation will access the EmployeeService SOAP web service for the implementation. To make a SOAP request to the EmployeeService web service, expand **EmployeeServiceSOAP_Lib > WSDL Definitions > http://www.example.org/EmployeeServiceV1**, select **EmployeeServiceV1.wsdl** and drag and drop it in the **getDepartment** sub-flow between the Input and Output nodes:



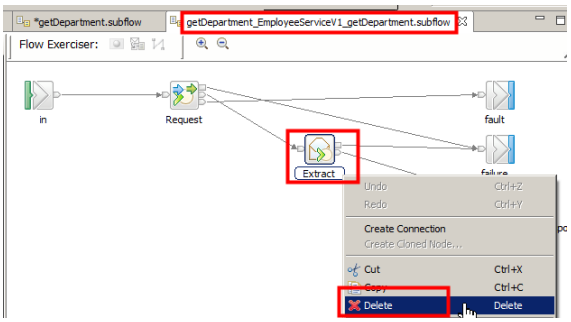
__30. Select **Invoke web service from message flow**, check **getDepartment(getDepartment)** and click **Finish**:



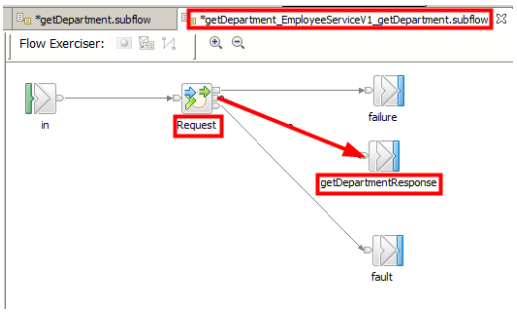
__31. Double-click the **getDepartment_EmployeeServiceV1** sub-flow node to open the **getDepartment_EmployeeServiceV1_getDepartment** sub-flow:



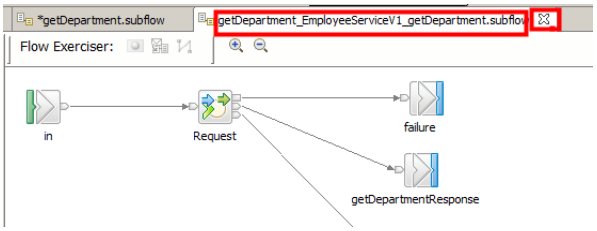
__32. The sub-flow contains a SOAPExtract node to remove the SOAP Envelope from the web service response. The map for the REST API response accepts a SOAP Envelope so it can handle valid responses and SOAP Faults, so the SOAPExtract node must be removed from the sub-flow. Right-click the **Extract** node and select **Delete**.



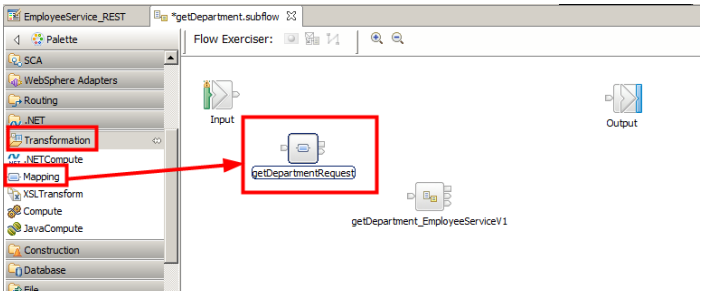
__33. Connect the Out terminal of the **Request** node to the In terminal of the **getDepartmentResponse** node:



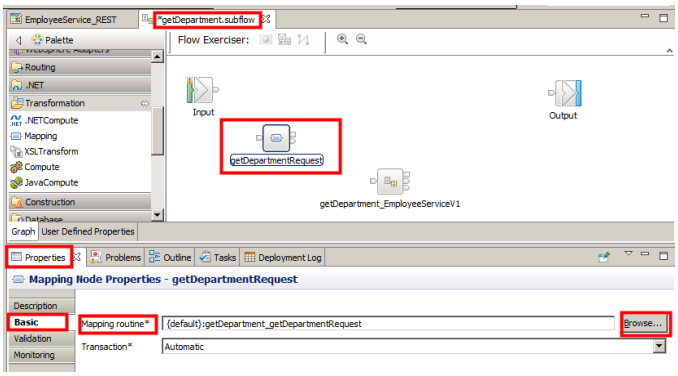
__34. Enter **CTRL+S** to save the **getDepartment_EmployeeServiceV1_getDepartment** sub-flow. Click the **X** on the getDepartment_EmployeeServiceV1_getDepartment tab to close the sub-flow:



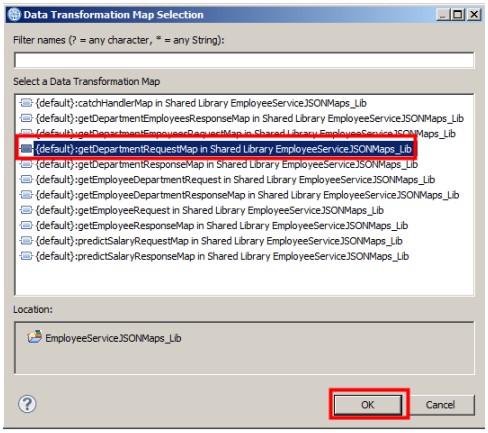
__35. You will now add a mapping node to map the REST request to the SOAP request. Expand the **Transformation** node category, select the **Mapping** node and left-click in the **getDepartment** sub-flow between the Input and getDepartment_EmployeeServiceV1 nodes. Rename the node to “getDepartmentRequest”:



__36. Highlight the **getDepartmentRequest** node and select the **Basic** properties page. Click the **Browse** button for the **Mapping routine*** property:



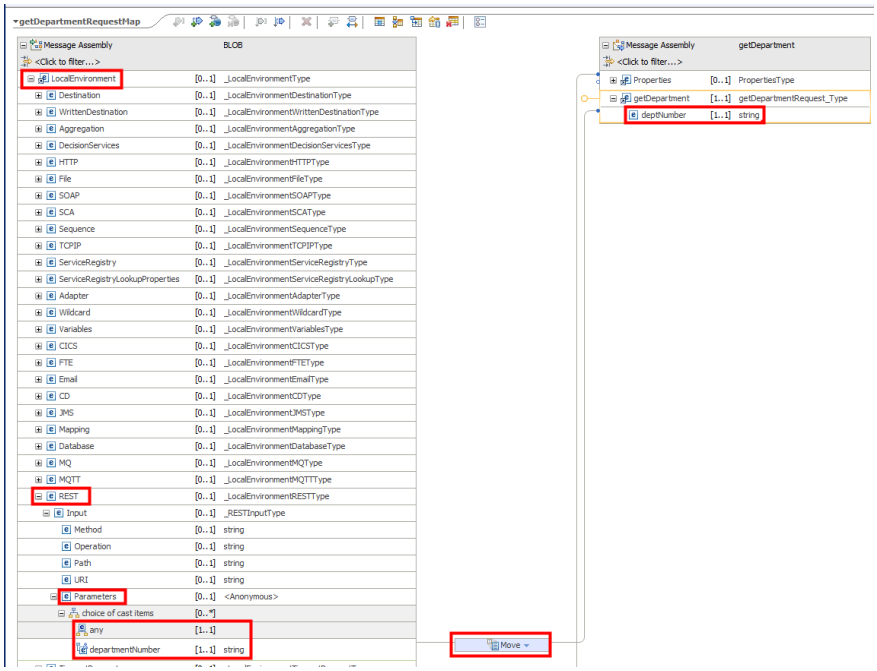
__37. Select the **{default}:getDepartmentRequestMap** map and click **OK**:



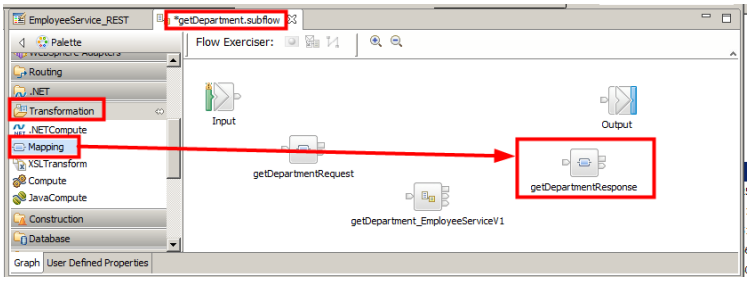
__38. Double-click the **getDepartmentRequest** node to open the **getDepartmentRequestMap** map.

The getDepartment operation accepts a query input parameter named “departementNumber”. The REST API application makes REST API parameters available in a well-defined location under “LocalEnvironment/REST/Input/Pararmeters/any”. The “any” element can be Cast to any type or “user-defined” elements can be added. The other REST input parameter types of “path” and “header” will also made available at “LocalEnvironment/REST/Input/Pararmeters/any”.

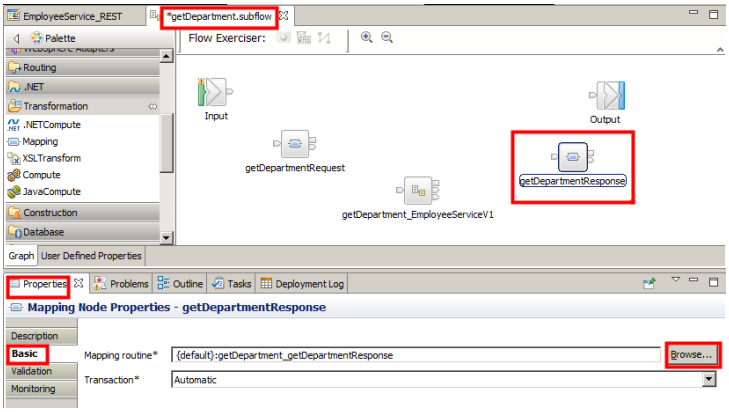
Click the **Move** transform; this will expand the **LocalEnvironment** tree and show the REST API parameter mapping. Here you see that the “departmentNumber” “User Defined Type” was added to “LocalEnvironment/REST/Input/Pararmeters/any”. Note, the name added here must match the parameter name in the Swagger document exactly:



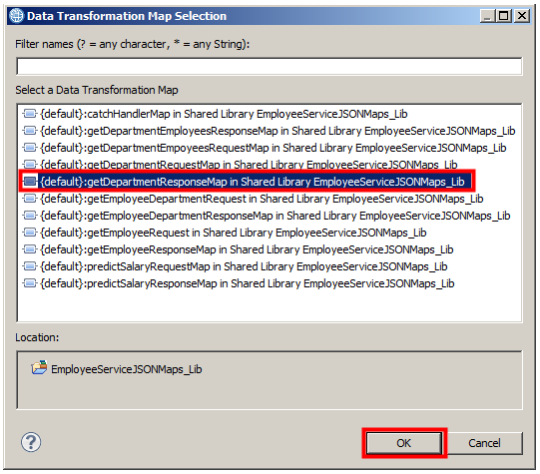
__39. You will now add a mapping node to map the web service SOAP response to the REST response. Expand the **Transform** node category, select the **Mapping** node and left-click in the sub-flow between the getDepartment_EmployeeServiceV1 and Output nodes. Rename the node "getDepartmentResponse".



__40. Highlight the **getDepartmentResponse** node, select the **Basic** properties tab and click the **Browse** button for the **Mapping routine*** properties:



__41. Select the **{default}:getDepartmentRepsonseMap** map and click **OK**:



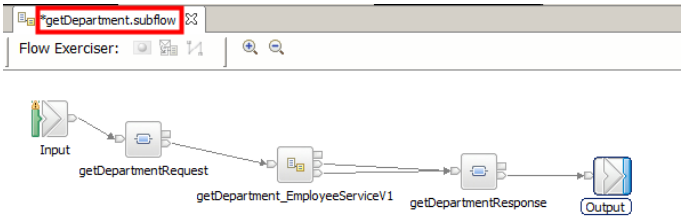
__42. Wire the Out terminal of the **Input** node to the In terminal of the **getDepartmenRequest** node.

Wire the Out terminal of the **getDepartmentRequest** node to the In terminal of the **getDepartment_EmployeeServiceV1** node.

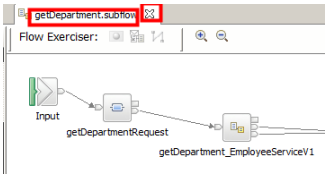
Wire the getDepartmentResponse terminal of the **getDepartment_EmployeeServiceV1** node to the In terminal of the **getDepartmentResponse** node.

Wire the fault terminal of the **getDepartment_EmployeeServiceV1** node to the In terminal of the **getDepartmentResponse** node.

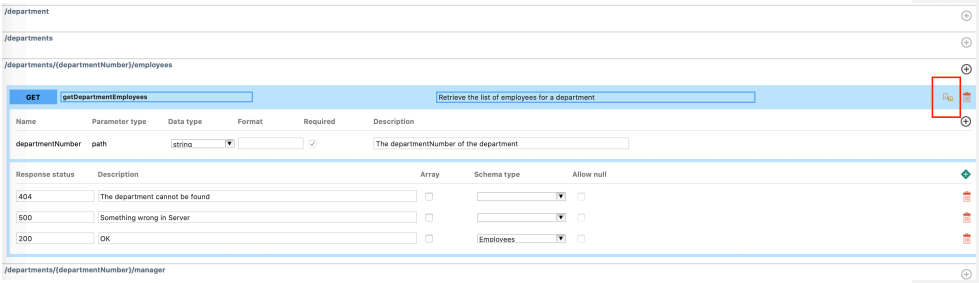
Wire the Out terminal of the **getDepartmentResponse** node to the In terminal of the **Output** node.



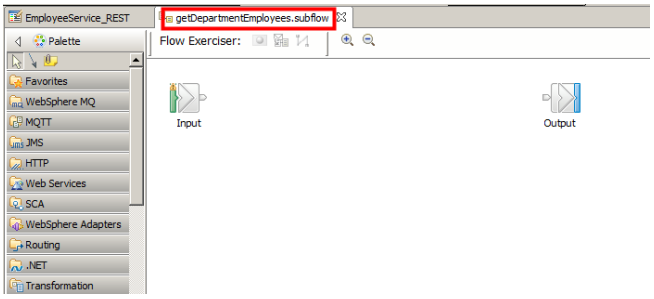
__43. Enter **CTRL+S** to save the **getDepartment** sub-flow. Click the **X** on the getDepartment.sub-flow tab to close the sub-flow:



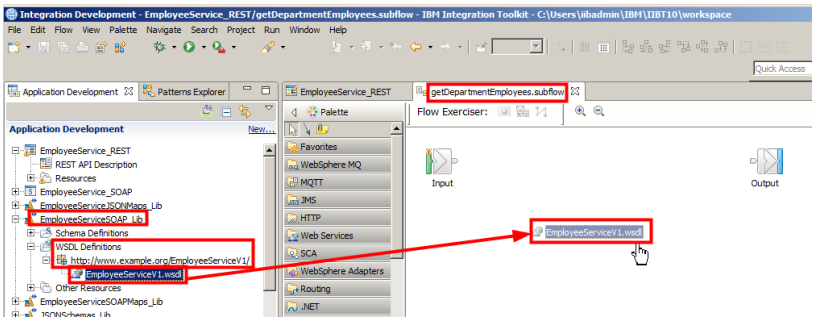
__44. Now, you will implement another operation.
Select the **EmployeeService_REST** tab. Under **Operations** expand the **/departments/{departmentNumber}/employees** path. Click **the highlighted icon** for the getDepartmentEmployees operation:



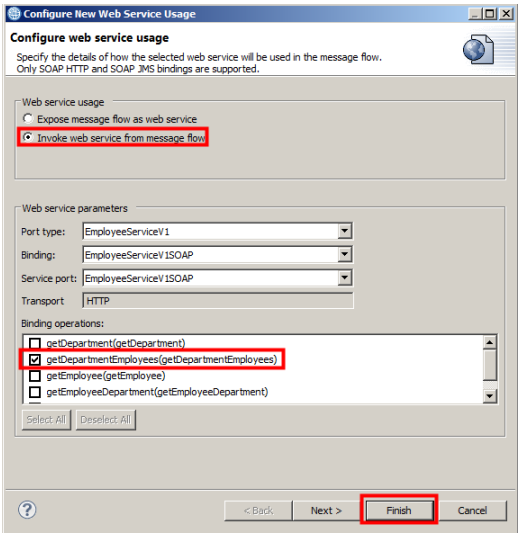
__45. The getDepartmentEmployees sub-flow is opened in the Message Flow Editor:



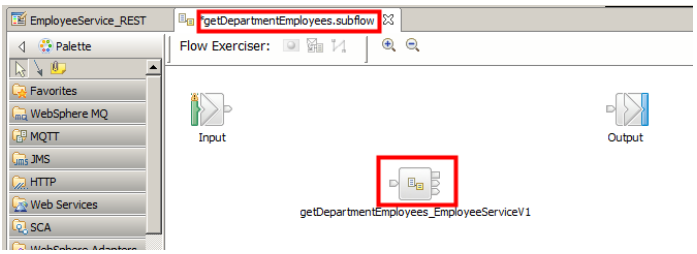
__46. The getDepartmentEmployees REST operation will access the EmployeeService SOAP web service for the implementation. To make a SOAP request to the EmployeeService web service, expand **EmployeeServiceSOAP_Lib > WSDL Definitions > http://www.example.org/EmployeeServiceV1**, select **EmployeeServiceV1.wsdl** and drag and drop it in the **getDepartmentEmployees** sub-flow between the Input and Output nodes:



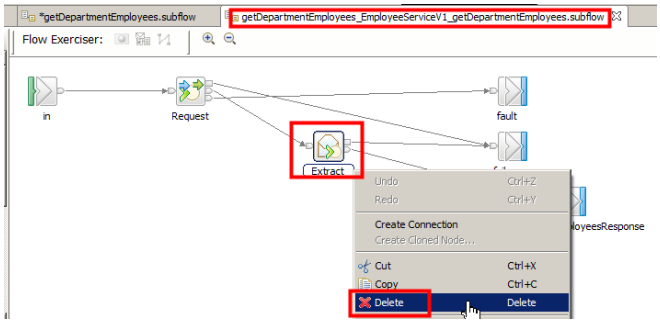
__47. Select **Invoke web service from message flow**, select **getDepartmentEmployees(getDepartmentEmployees)** and click **Finish**:



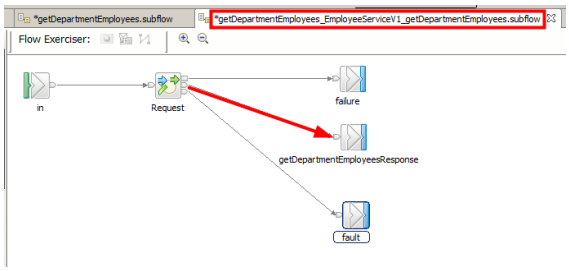
__48. Double-click the **getDepartmentEmployees_EmployeeServiceV1** sub-flow node to open the getDepartmentEmployees_EmployeeServiceV1_getDepartmentEmployees sub-flow:



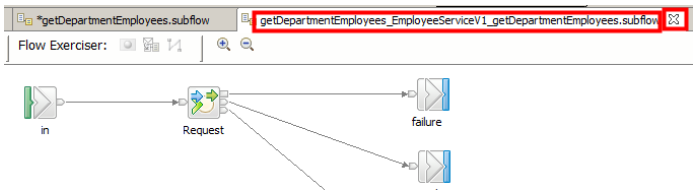
__49. The sub-flow contains a SOAPExtract node to remove the SOAP Envelope from the web service response. The map for the REST API response accepts a SOAP Envelope so it can handle valid responses and SOAP Faults, so the SOAPExtract node must be removed from the sub-flow. Right-click the **Extract** node and select **Delete**.



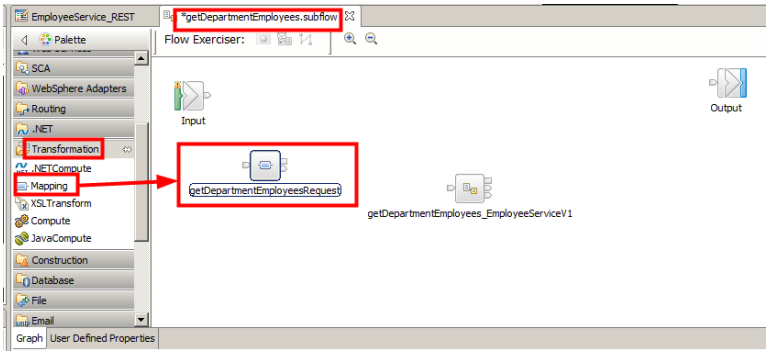
__50. Connect the Out terminal of the **Request** node to the In terminal of the **getDepartmentEmployeesRepsonse** node:



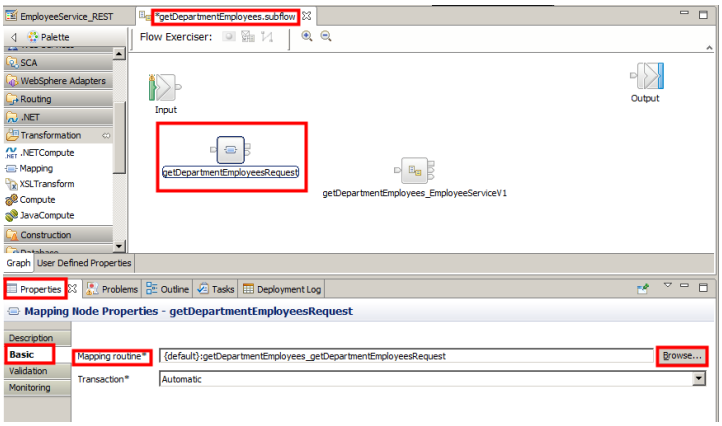
__51. Enter **CTRL+S** to save the **getDepartmentEmployees_EmployeeServiceV1_getDepartmentEmployees** sub-flow. Click the **X** on the getDepartmentEmployees_EmployeeServiceV1_getDepartmentEmployees tab to close the sub-flow:



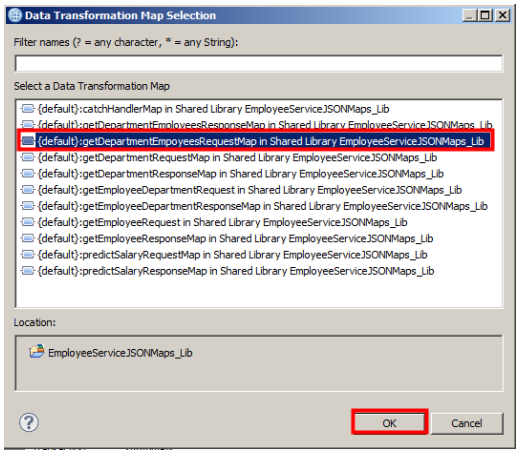
__52. You will now add a mapping node to map the REST request to the SOAP request. Expand the **Transformation** node category, select the **Mapping** node and left-click in the **getDepartmentEmployees** sub-flow between the Input and getDepartmentEmployees_EmployeeServiceV1 nodes. Rename the node to "getDepartmentEmployeesRequest":



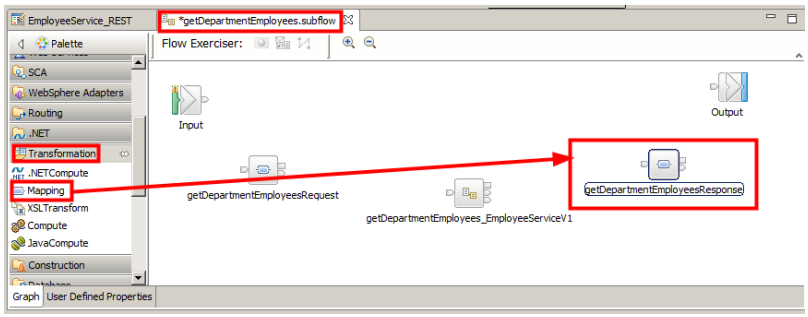
__53. Highlight the **getDepartmentEmployeesRequest** node and select the **Basic** properties page. Click the **Browse** button for the **Mapping routine*** property:



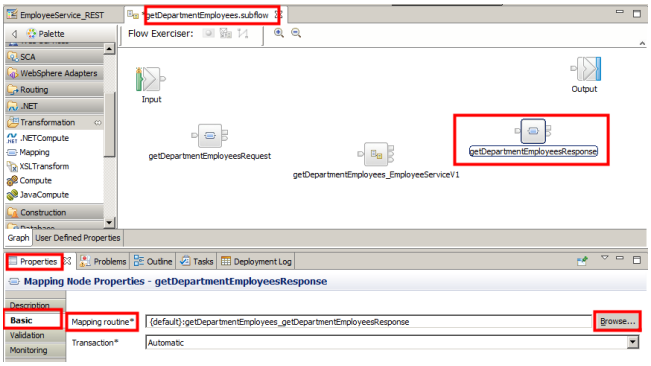
__54. Select the **{default}:getDepartmentEmployeesRequestMap** map and click **OK**:



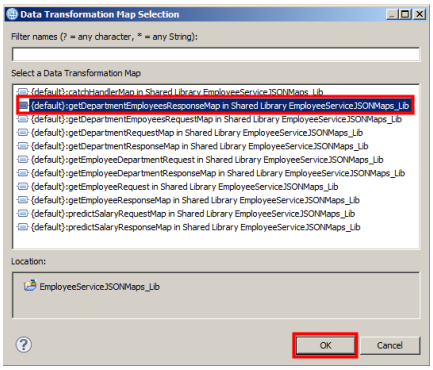
__55. You will now add a mapping node to map the web service SOAP response to the REST response. Expand the **Transform** node category, select the **Mapping** node and left-click in the sub-flow between the getDepartmentEmployees_EmployeeServiceV1 and Output nodes. Rename the node “getDepartmentEmployeesResponse”.



__56. Highlight the **getDepartmentEmployeesResponse** node, select the **Basic** properties tab and click the **Browse** button for the **Mapping routine*** property:

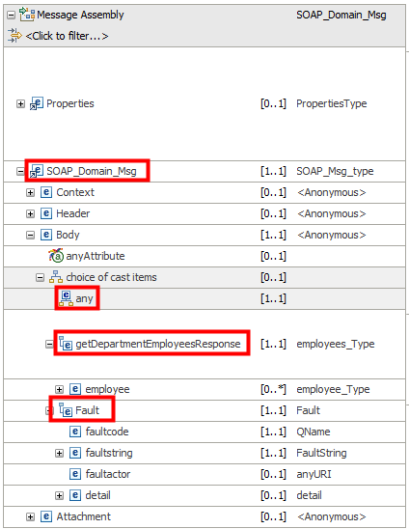


__57. Select the **{default}:getDepartmentEmployeesRepsonseMap** map and click **OK**:



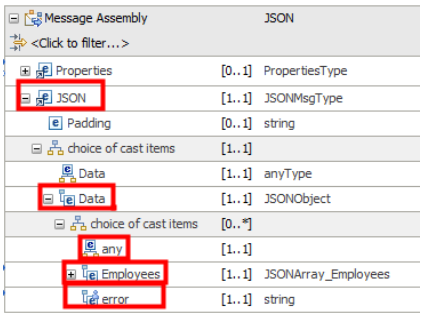
__58. Double-click the **getDepartmentEmployeesResponse** node to open the **getDepartmentEmployeesResponseMap** map.

Observe the map source is a SOAP Domain message where the Body of the SOAP Envelope has been cast to a “getDepartmentEmployeesResponse” type and a “fault” type. This allows the operation to easily handle valid and fault responses from the web service. If a getDepartmentEmployeesResponse message is received an “Employees” JSON array is returned. If a fault message is received an “error” object is returned.



Observe that the map output uses the JSON parser. The “EmployeeServiceV1” Swagger document defines that the “getDepartmentEmployees” operation returns an “Employees” JSON array. While the “Employees” model is defined in the Swagger document the current tooling does not generate JSON schemas that can be used by the Graphical Data Mapper. An alternative is to use XML schemas to implement the JSON schemas that are defined in the Swagger document. The schemas in the JSONSchemas_Lib library implement all of the models in the Swagger document and are used in all of the JSON mappings.

Note that the “any” element under JSON/Data has been cast to an “Employees” type and a User Defined type of error has been added. This allows the operation to return either a JSON array of “Employees” or an “error” object, and offers easy GUI mapping of JSON objects and arrays:



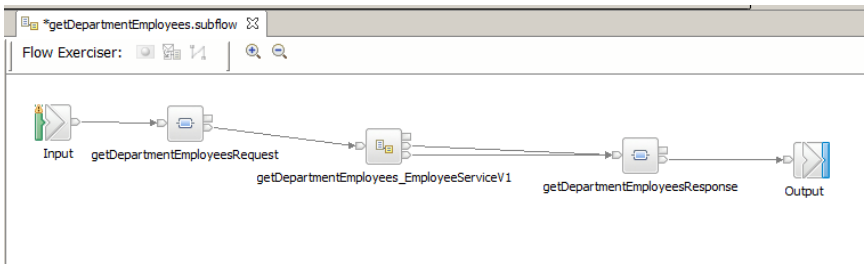
__59. Wire the Out terminal of the **Input** node to the In terminal of the **getDepartmenEmployeesRequest** node.

Wire the Out terminal of the **getDepartmentEmployeesRequest** node to the In terminal of the **getDepartmentEmployees_EmployeeServiceV1** node.

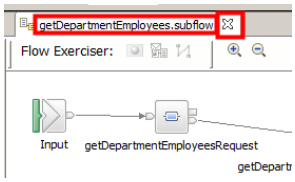
Wire the getDepartmentEmployeesResponse terminal of the **getDepartmentEmployees_EmployeeServiceV1** node to the In terminal of the **getDepartmentEmployeesResponse** node.

Wire the fault terminal of the **getDepartmentEmployees_EmployeeServiceV1** node to the In terminal of the **getDepartmentEmployeesResponse** node.

Wire the Out terminal of the **getDepartmentEmployeesResponse** node to the In terminal of the **Output** node.

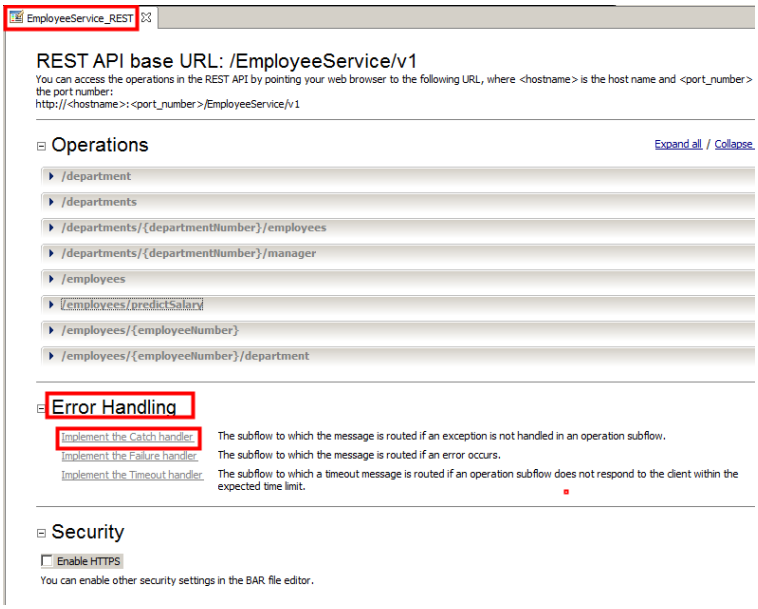


__60. Enter **CTRL+S** to save the **getDepartmentEmployees** sub-flow. Click the **X** on the getDepartmentEmployees.sub-flow tab to close the sub-flow:

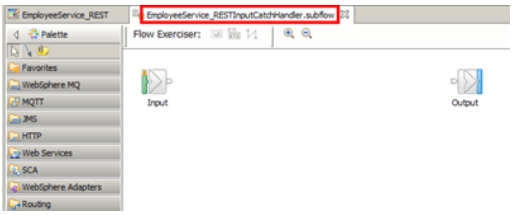


__61. Next, you will implement the flow to handle errors in our REST service.

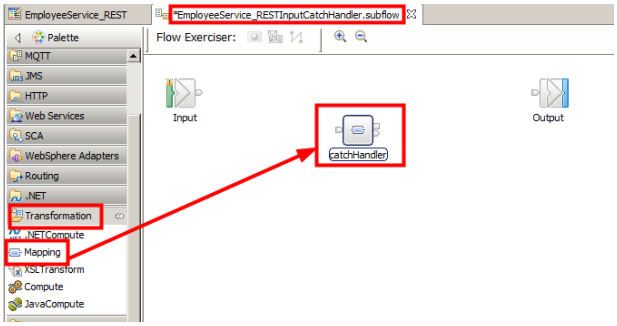
Select the **EmployeeService_REST** tab. Under **Error Handling** click **Implement the Catch handler**:



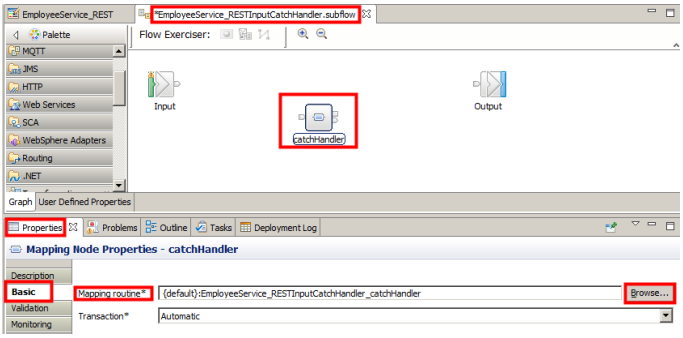
__62. The EmployeeService_RESTInputCatchHandler sub-flow opens in the Message Flow Editor:



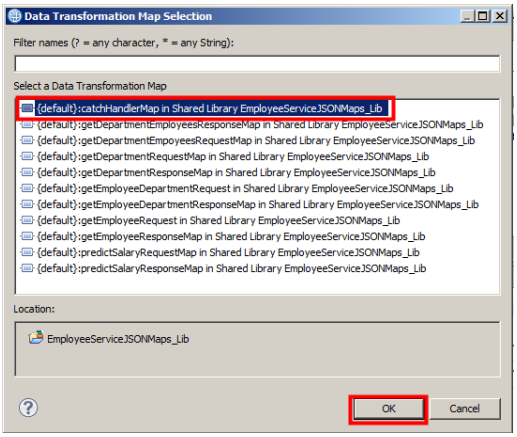
__63. You will now add a mapping node to create a simple JSON error message for exceptions that occur during message flow processing. Expand the **Transform** node category and select the **Mapping** node, then left click in the sub-flow between the Input and Output nodes. Rename the Mapping node to “catchHandler”:



__64. Highlight the **catchHandler** node and on the **Basic** properties page click the **Browse** button for the **Mapping routine*** property:

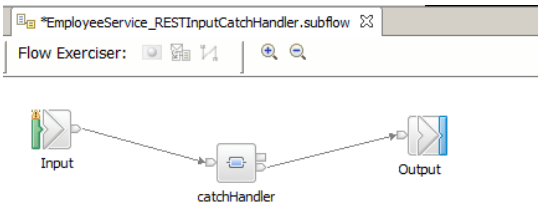


__65. Select the **{default}:catchHandlerMap** map and click **OK**:

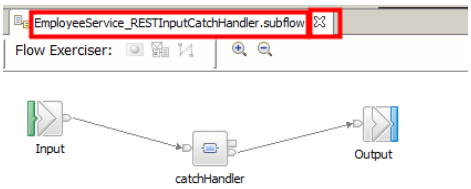


__66. Wire the Out terminal of the **Input** node to the In terminal of the **catchHandler** node.

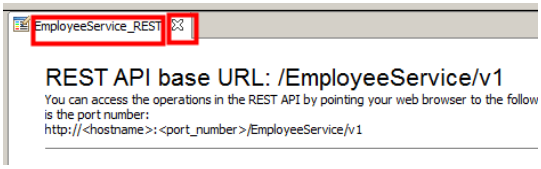
Wire the Out terminal of the **catchHandler** node to the In terminal of the **Output** node.



__67. Enter **CTRL+S** to save the **EmployeeService_RESTInputCatchHandler** sub-flow. Click the **X** on the EmployeeService_RESTInputCatchHandler.sub-flow tab to close the sub-flow:



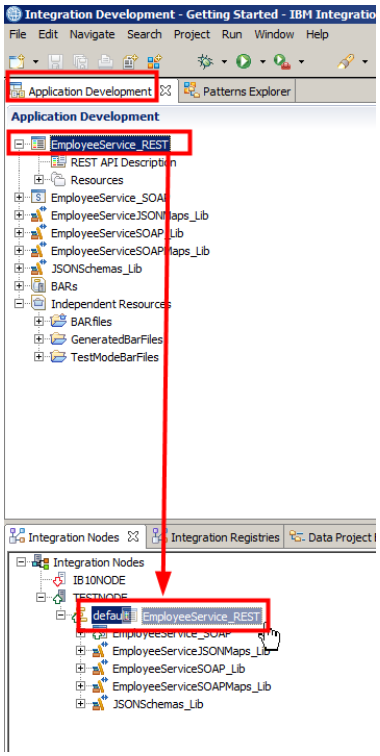
__68. Click the **X** in the EmployeeService_REST tab to close the REST API:



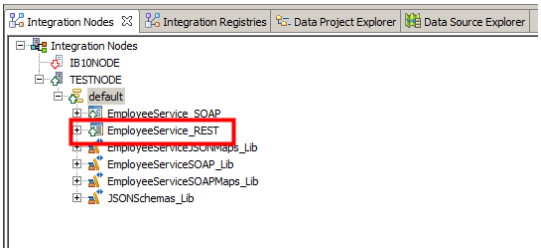
5.6 Deploy the EmployeeService_REST REST API

You will now deploy the EmployeeService_REST REST API to the myaceworkdir integration server.

__69. In the Application Development view, select the **EmployeeService_REST** REST API project and drag and drop it onto the **myaceworkdir** integration server:



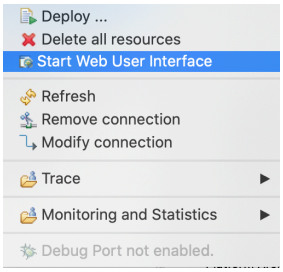
__70. Observe that the EmployeeService_REST REST API has been successfully deployed:



5.7 Testing the EmployeeService_REST REST API

This section will show you how to use the SwaggerUI tool to send a REST request into the REST API service that you have just created.

__71. Open the ACE Web UI by right-clicking the **myaceworkdir** integration server and selecting **Start Web User Interface**:



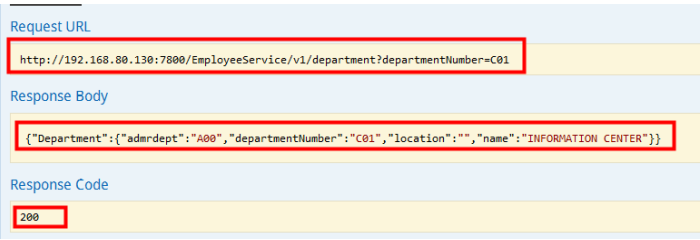
__72. When the Web User Interface opens, click on **EmployeeService_REST** service and then click **OpenAPI document** link to open it in a browser. You will see the swagger document, representing the details of your implemented REST API.

__73. Now, using a browser, you will test the REST service you have just implemented.

Modify the swagger link to look like the following, to call the get Department method, using the "C01" as the department parameter:

http://localhost:7800/EmployeeService/v1/department?departmentNumber=C01

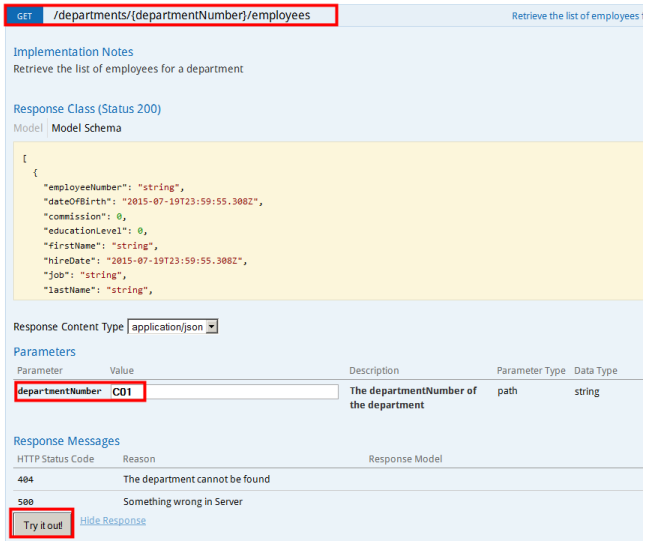
Observe the Request URL and the response body for the HTTP 200 return code:



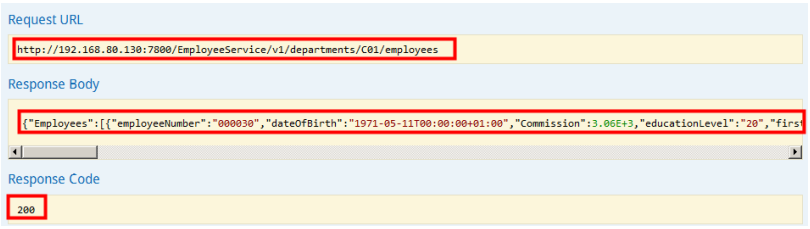
__74. Now, let's test the other method you have implemented.

__75. Enter http://localhost:7800/EmployeeService/v1/departments/C01/employees in your browser.

You should get a HTTP 200 and a JSON response from the service, like the one on the image below:



__76. Observe the Request URL and the Response Message for a HTTP return code of 200:



Note: A JSON array of Employee objects has been returned.

__77. You have completed the REST API lab.

5.8 Summary

During the “Creating a REST API” lab you explored the contents of a Swagger document that defined the EmployeeServiceV1 REST API using the Swagger editor.

The Create REST API wizard was used to import the EmployeeServiceV1 Swagger document and create a REST API application. You observed the structure, organization and contents of the REST API application and how they offer ACE design and implementation best practices.

You used the REST API application tooling to quickly implement two operations and a Catch handler, and then deployed the REST API application to a runtime Integration Server. The two operations exchanged JSON Objects/Arrays and demonstrated the following input parameter types – path and query.

- The Web UI was used to observe the REST API extensions.
- The EmployeeServiceV1 REST API was tested.

END OF LAB 7

Lab 6 Exception handling and the interactive Flow Debugger

Commented [EL4]: There was some inconsistency in how this feature was referred to - both "integrated" and "interactive" and the word "flow" wasn't always used. I've tried to standardize this throughout the document, as an analogue to the Flow Exerciser

6.1 Overview

Murphy's law states that "Anything that can go wrong, will go wrong".

All programs should be designed with that adage in mind, and include exception handling of some sort. Integration solutions developed for App Connect Enterprise are no different—a well-designed message flow will have some form of exception handling.

In this lab you will see an example of how exceptions can be thrown, caught, and handled directly in your message flow. You will also be introduced to the interactive Flow Debugger and see how that can be used in addition to the Flow Exerciser in diagnosing problems.

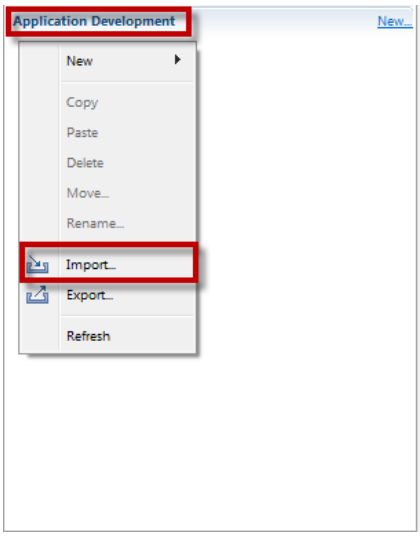
6.2 Lab setup

You will use the Lab 2 solution as the starting point for this lab.

If you completed Labs 1 and 2, and still have the IntroLab application and IntroLab_Lib library in your workspace, you can proceed to section **8.3**.

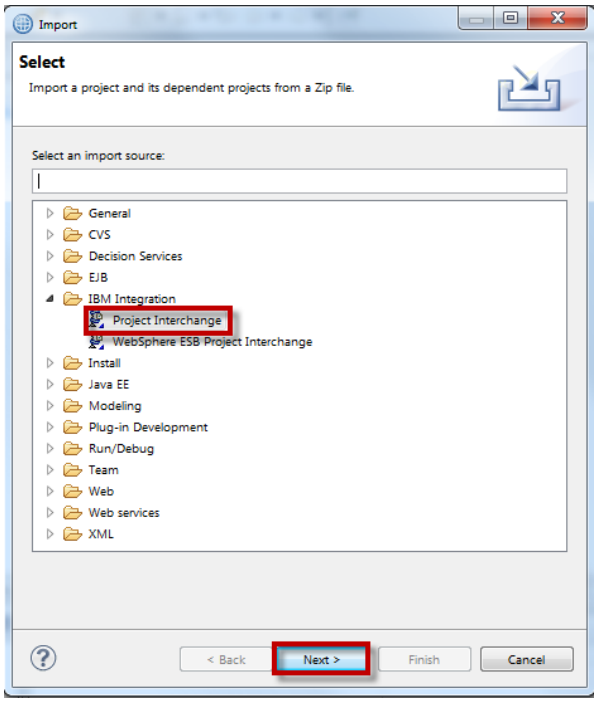
If you did not complete both Labs 1 and 2, or are now using a different workspace, you can import the completed Lab 2 solution into your workspace using the following steps.

__1. In the whitespace of the Application Development view, right-click and select **Import.**

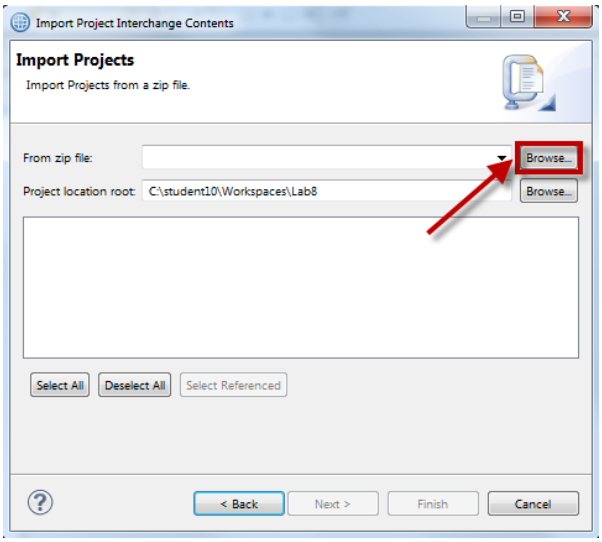


__2. In the Import wizard, expand the **IBM Integration** folder if necessary, and select **Project Interchange**.

Click **Next**.

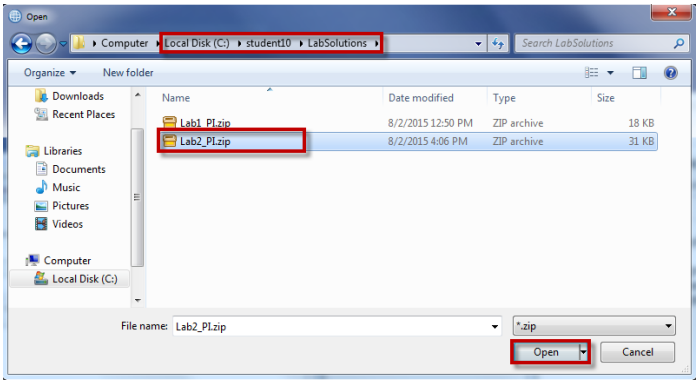


__3. Click the **Browse** button next to the **From zip file:** entry field.



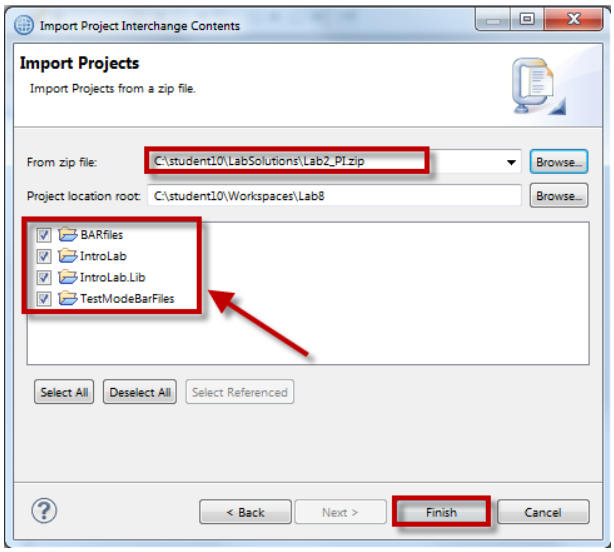
__4. In the Open dialog, navigate to **C:\student10\LabSolutions**.

Select **Lab2_PI.zip** and click **Open**.

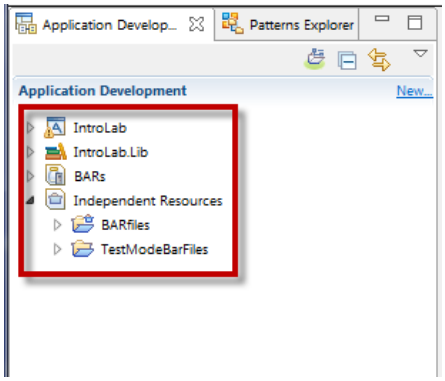


__5. Ensure that all four project folders shown are checked.

Click **Finish**.



__6. In the Application Development view, you should now see the imported project folders (and perhaps others as well).



6.3 Understanding exceptions in App Connect Enterprise

The IntroLab message flow had no exception handling of any kind included. In this lab you will extend the IntroLab message flow to both throw user exceptions in certain situations, as well as be able to catch and handle both User and Recoverable system exceptions, regardless of where in the flow these occur.

6.3.1 Key Idea: Catch Terminals and TryCatch nodes

Key Idea: Catch Terminals and TryCatch nodes.

Read the section below for additional information about using Failure and Catch Terminals, TryCatch nodes, or both to catch and handle exceptions in App Connect Enterprise.

With App Connect Enterprise there are two general approaches to handling errors in a message flow:

Failure checking

Most nodes have a Failure terminal. Wire this terminal to explicitly check for any errors that occur *within that node.* Sometimes node failures are tolerable, or recoverable. If you do not want a node failure to result in the flow as a whole aborting, you can wire the Failure terminal to nodes that can deal with (or ignore) the failure.

Catching exceptions

If you do not wire a Failure terminal, a node failure is converted into an exception which is thrown from the node. Any changes that were made to the inflight message before the exception was thrown are reversed. Message flow nodes that represent the start of a transaction typically have a Catch terminal. Use the Catch terminal to handle any exceptions that are thrown "downstream".

Note this important difference!

A message is propagated to a node's *Failure* terminal if an exception occurs *inside that node*. A message is propagated to a node's *Catch* terminal only if it has first been propagated beyond the node (for example, to the nodes connected to the Out terminal) and an *unhandled* exception occurs *downstream* from the node.

In both cases, an Exception List tree is created, which contains diagnostic information about the error that has occurred. If you wire the node's Failure terminal, or wire an upstream node's Catch terminal, the message assembly, including the Exception List, will be propagated down that path. Along that path in the message flow you can include nodes to handle such exceptions as you see fit. Examples of what you might do include logging exception data, saving the message or event being processed in a file or undeliverable message queue, or generating an alert.

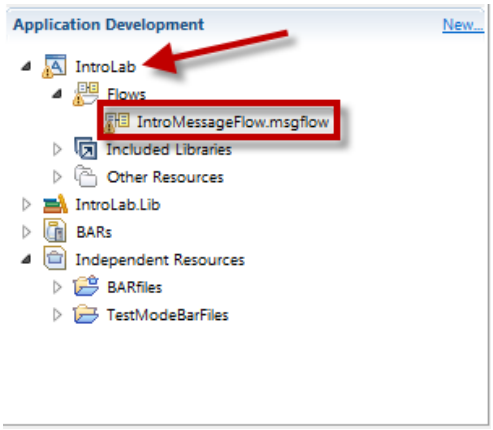
For complex message flows, a TryCatch node can be used to break the large message flow into multiple "TryCatch blocks". This can enable more granular exception handling to be performed. The TryCatch node itself does not process a message in any way; it merely represents a decision point in a message flow. When the TryCatch node receives a message, it propagates it to the Try terminal (a "try subflow"). If an exception occurs on that path, control will return to the TryCatch node, which will then propagate it to the Catch terminal (a "catch subflow") that can then attempt to handle the exception.

See the App Connect Enterprise documentation for more detailed information on failure and exception handling in message flows.

In this section you will see a simple example of how exception handling can be incorporated into a message flow.

Note:
The following should not be viewed as an example of robust exception handling! The example is very simple, and intended merely to illustrate some of the options available to catch and handle exceptions in App Connect Enterprise.

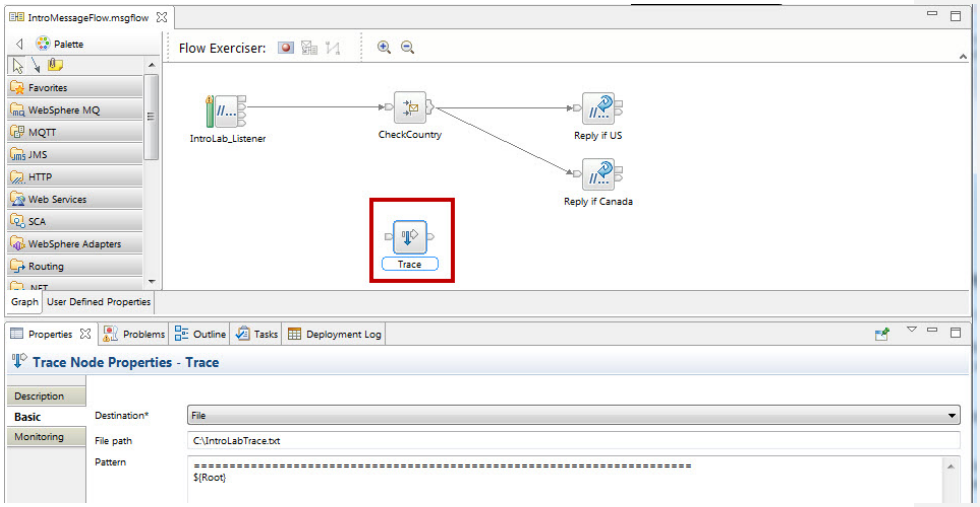
__7. In the Application Development view, expand the **IntroLab** application and **Flows** folder, and double-click **IntroMessageFlow.messageflow**.



__8. Locate the **Trace** node that you left unwired at the end of Lab 2.

You will be repurposing this node to gather additional information when exceptions occur.

Click the **Trace** node and review its properties.



__9. Leave the Destination and File path the same.

In the **Pattern** box, add the following additional lines:

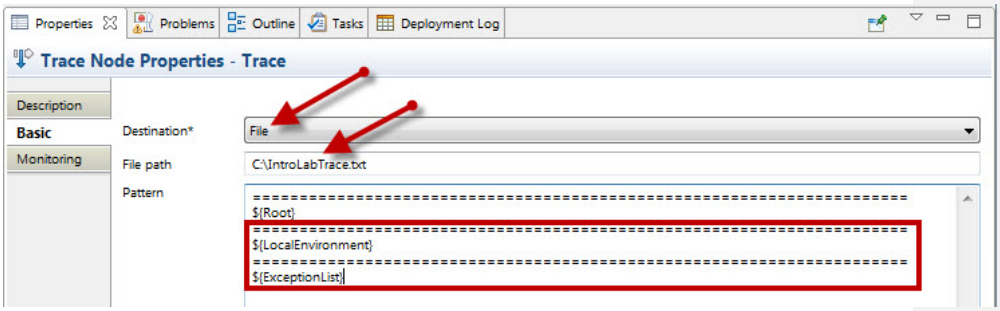
```
==========
${LocalEnvironment}
==========
${ExceptionList}
```

Enter the string **${LocalEnvironment}** *exactly as indicated* – this tells the trace node to dump out the entire contents of the LocalEnvironment tree.

Enter the string **${ExceptionList}** *exactly as indicated* – this tells the trace node to dump out any Exception data if present.

Note:
The pattern uses **curly braces**, *not* parentheses!
The pattern is *case sensitive*!

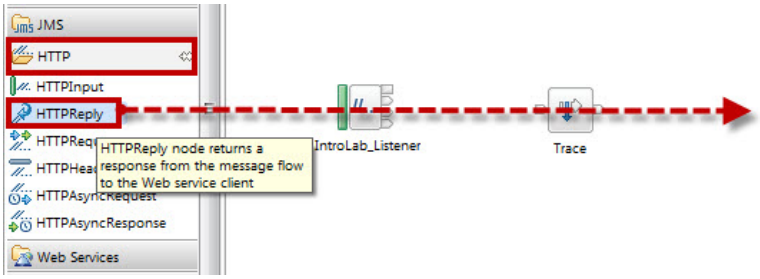
__10. The trace node properties should look like this when complete:



__11. On the Palette, open the **HTTP** drawer.

Select an **HTTPReply** node from the drawer.

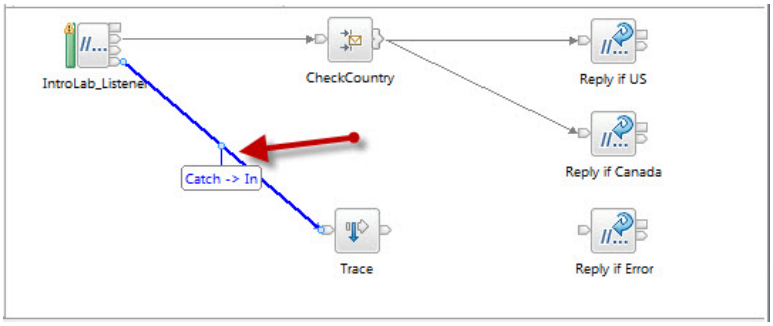
Place it to the right of the trace node.



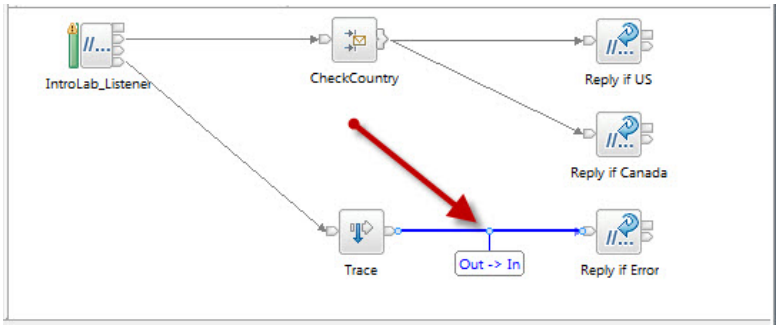
__12. Rename the node “Reply if Error”.

Press the **Enter** key to accept the changed name.

__13. Wire the Catch terminal (at the bottom) of the **IntroLab_Listener** input node to the in terminal of the **Trace** node.



__14. Wire the Out terminal of the **Trace** node to the In terminal of the **Reply if Error** node.

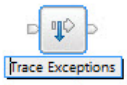


__15. Click the node, then click one more time on the node name. It should then look like the following:

Commented [EL5]: Deleted the "Rename" line because it is a bit premature - the renaming occurs in the next step, below.

__16. Change the name of the node to “Trace Exceptions”.

Press the **Enter** key to accept the changed name.



The changes are nearly complete. But there is one more thing to take into account.

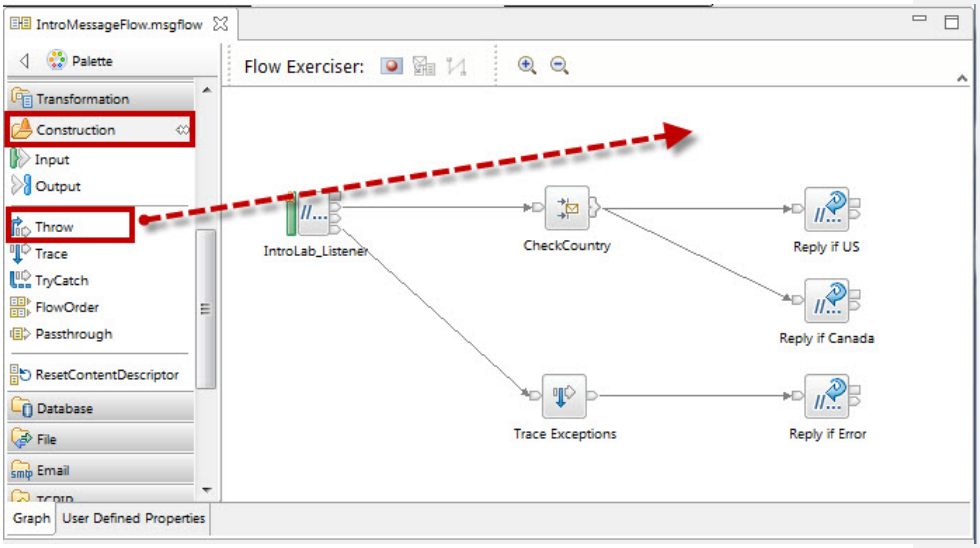
Remember that, in Lab 2, you ran one test with MX for customerCountry. When that test was run, no failure was generated – the flow simply ended. As discussed in that lab, the reason was that, in a Route node, a no-match condition is not considered a failure.

For purposes of this lab, you are going to consider a no-match condition to be a failure. So you will now extend the flow further, to throw a User Exception when there is not a valid value for customerCountry.

__17. On the Palette, find the **Construction** drawer and open it

Select a **Throw** node from the drawer.

Place it to the right and slightly above the CheckCountry node.

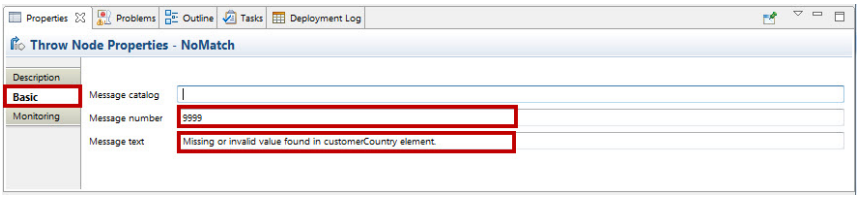


__18. Change the name of the node to “NoMatch”.

Press the **Enter** key to accept the changed name.

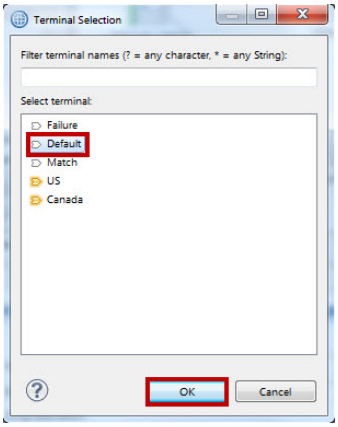
__19. Click the **NoMatch** (previously, “Throw”) node and look at its Properties.

On the **Basic** tab, change **Message number** to “*9999*”, and **Message text** to “Missing or invalid value found in customerCountry element”.



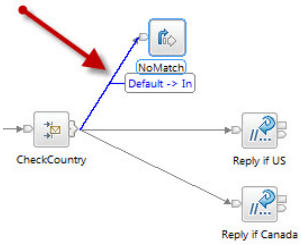
__20. On the **CheckCountry** route node, click the group of output terminals.

__21. Select **default** from the list. Click **OK**.



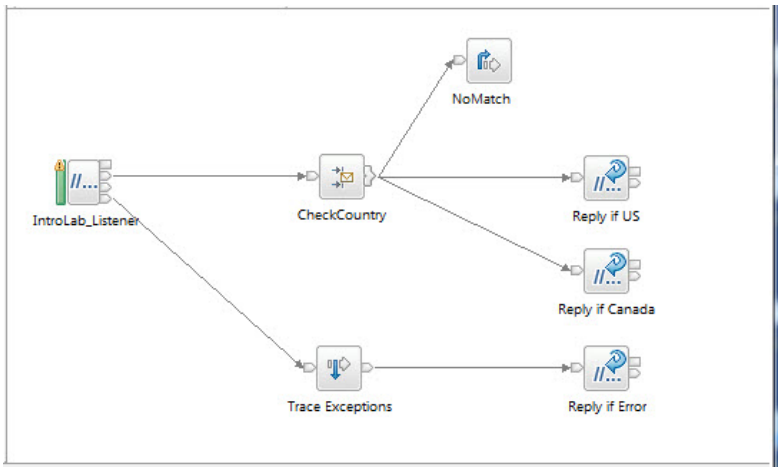
__22. Connect the Default terminal of the **CheckCountry** node to the In terminal of the **NoMatch** throw node.

Verify that the correct terminals are wired.



__23. The changes should be complete.

Your flow should look like this:



__24. 

Save the message flow.

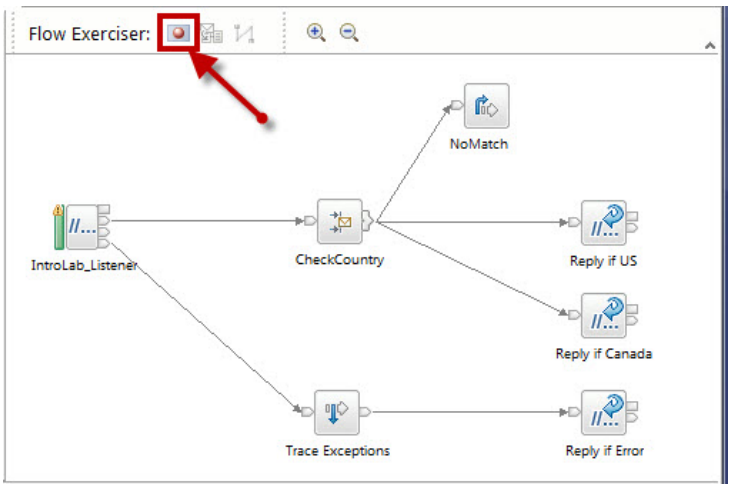
You will now repeat some of the tests you ran in Lab 2.

You will first retest the US and Canada paths in the flow, to ensure they still work as expected.

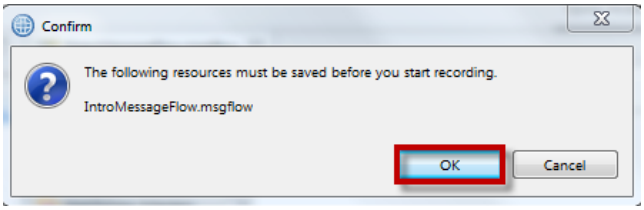
You will then retest sending an unexpected customerCountry code, to see if the logic you added handles that correctly.

Finally, you will send in a malformed XML document, to see how system exceptions are caught and handled.

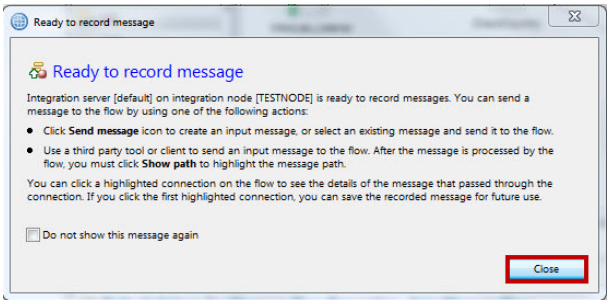
__25. Start the Flow Exerciser.



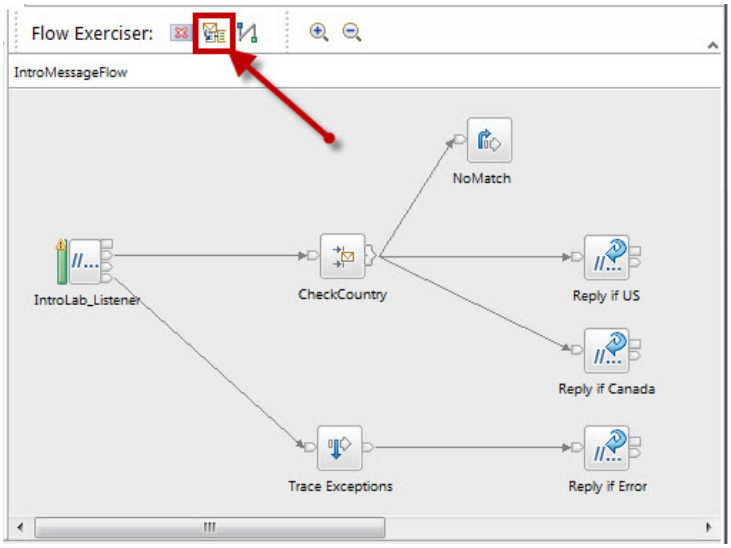
__26. If prompted to save the message flow, click **OK**.



__27. Click **Close** to start recording.

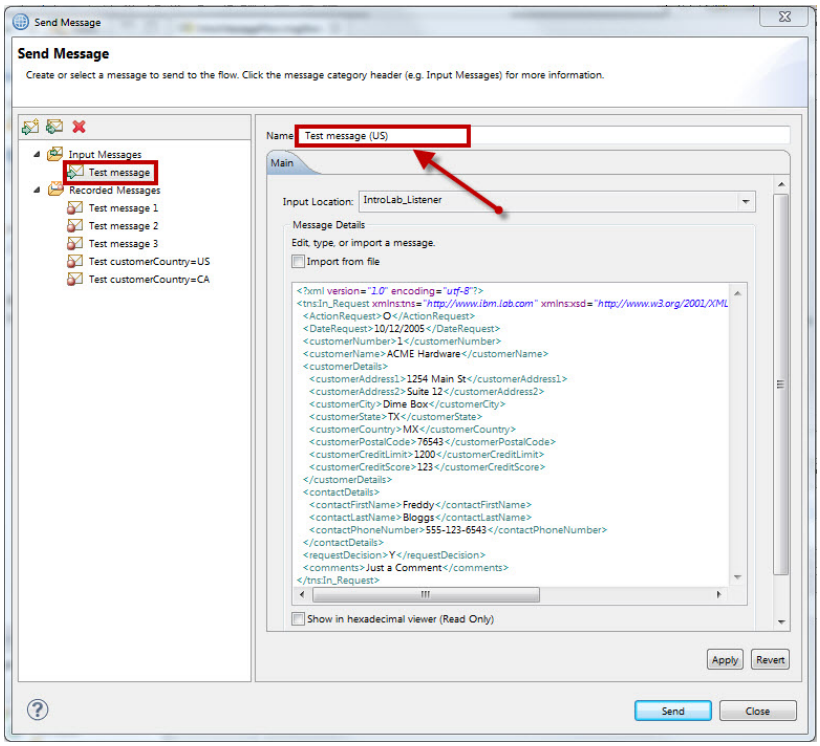


__28. Click the **Send** message icon to configure a test message.



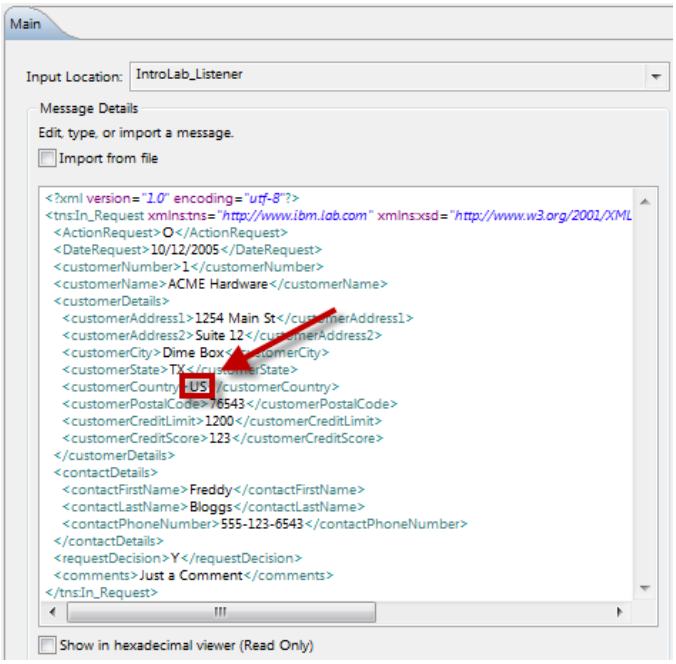
__29. Click **Test message.**

Change the name to “Test message (US)”.



__30. Recall that in Lab 2, you had changed customerCountry to “MX”. You may see that value there now.

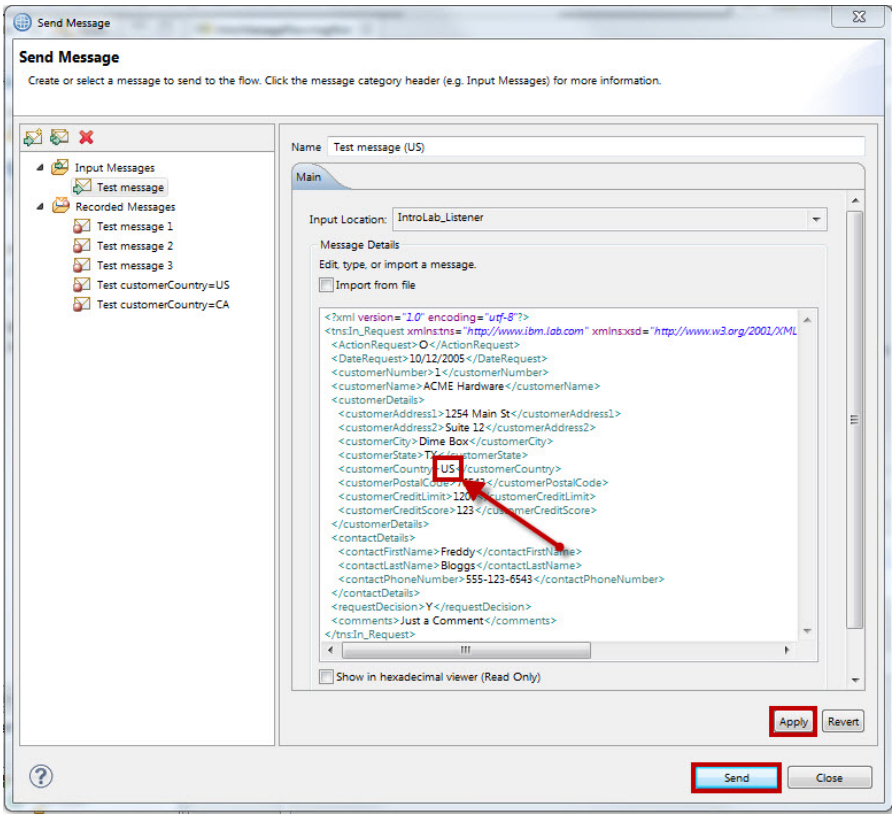
You want to retest the US path through the flow, so change **customerCountry** back to “US”.


Main
Input Location: IntroLab_Listener
Message Details
Edit, type, or import a message.
Import from file
<?xml version="1.0" encoding="utf-8"?>
<tns:In_Request xmlns:tns="http://www.ibm.lab.com" xmlns:xsd="http://www.w3.org/2001/XML
<ActionRequest>O</ActionRequest>
<DateRequest>10/12/2005</DateRequest>
<customerNumber>1</customerNumber>
<customerName>ACME Hardware</customerName>
<customerDetails>
<customerAddress1>1254 Main St</customerAddress1>
<customerAddress2>Suite 12</customerAddress2>
<customerCity>Dime Box</customerCity>
<customerState>TX</customerState>
<customerCountry>US</customerCountry>
<customerPostalCode>76543</customerPostalCode>
<customerCreditLimit>1200</customerCreditLimit>
<customerCreditScore>123</customerCreditScore>
</customerDetails>
<contactDetails>
<contactFirstName>Freddy</contactFirstName>
<contactLastName>Bloggs</contactLastName>
<contactPhoneNumber>555-123-6543</contactPhoneNumber>
</contactDetails>
<requestDecision>Y</requestDecision>
<comments>Just a Comment</comments>
</tns:In_Request>
Show in hexadecimal viewer (Read Only)

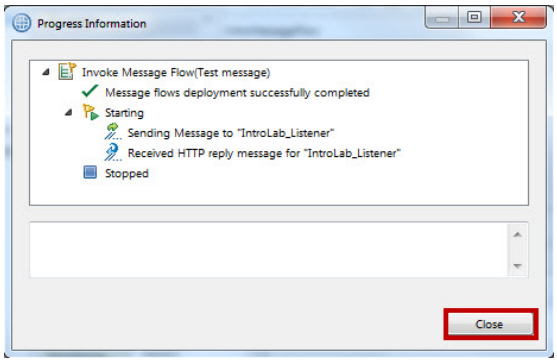
__31. Click **Apply** to 

the change.

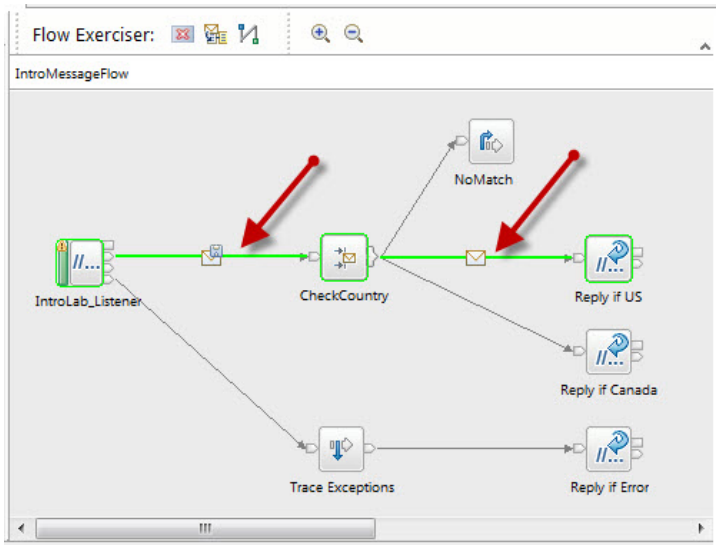
Click **Send** to send the message to the flow.



__32. The Flow Exerciser will run. Click **Close** after it stops.

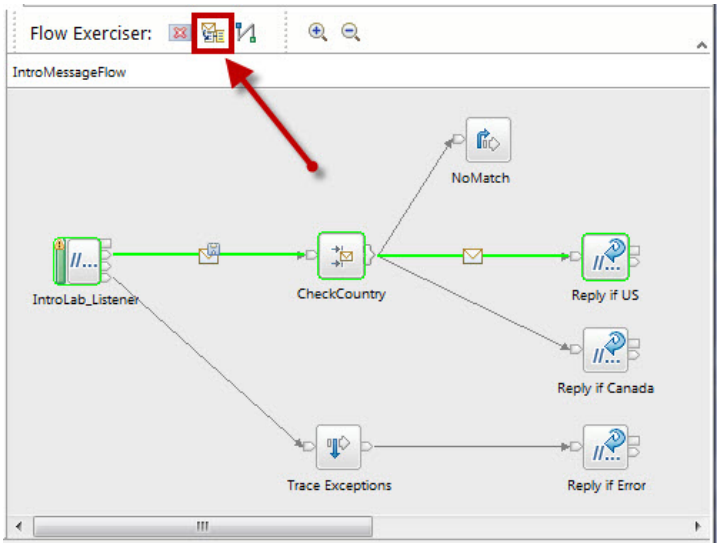


__33. The Flow Exerciser shows the path the **US** message took through the flow.

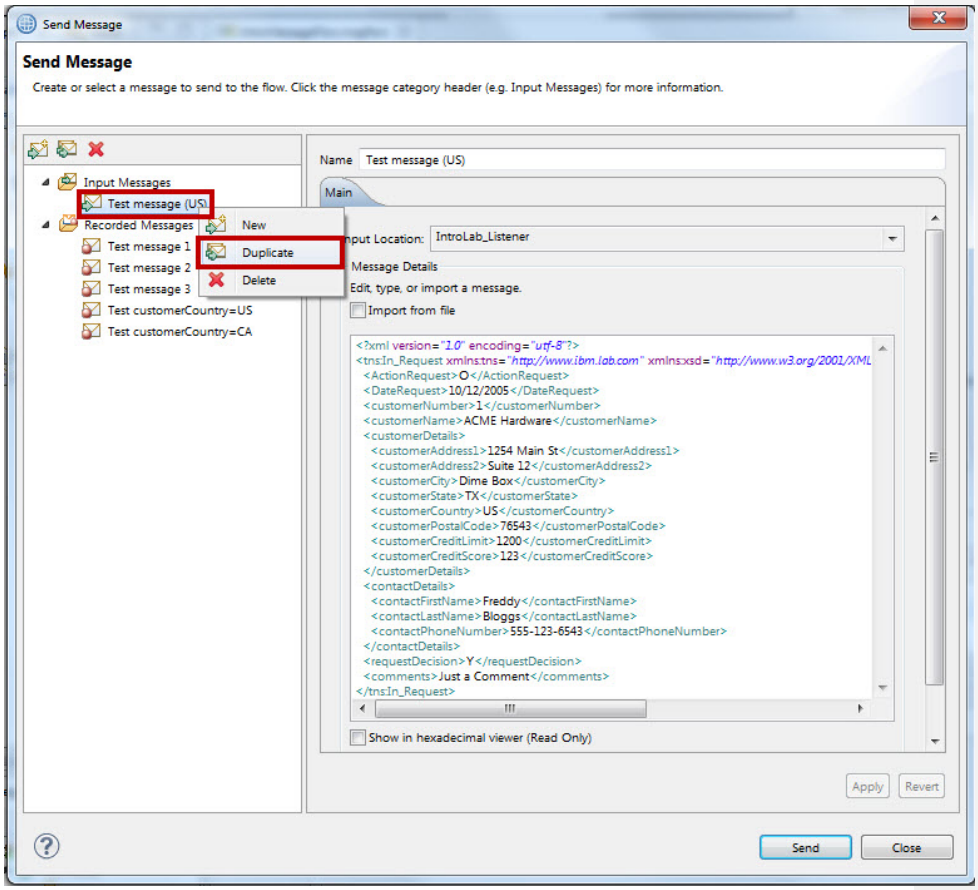


__34. Repeat these steps to test the Canada path in the message flow.

Click the **Send** message icon to configure a test message.



__35. Right-click **Test message (US)**, and select **Duplicate** from the menu.



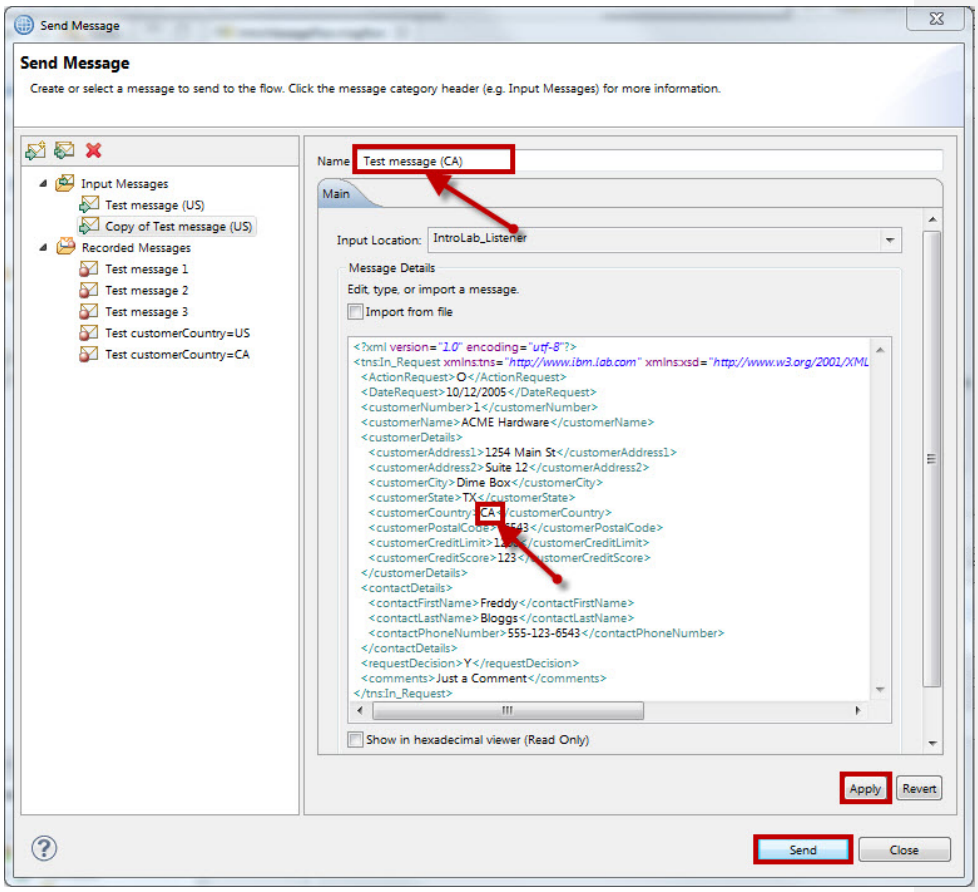
__36. Change the name of the new test message to “Test message (CA)”.

Change **customerCountry** to “**CA”**.

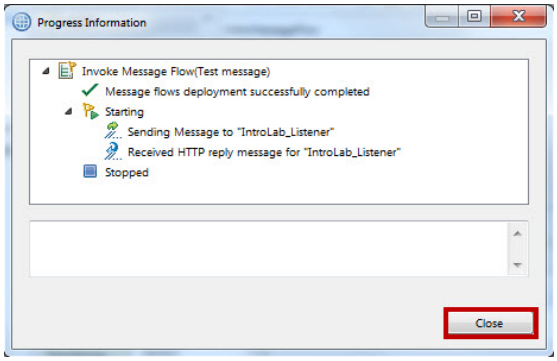
Click **Apply** to 

the change.

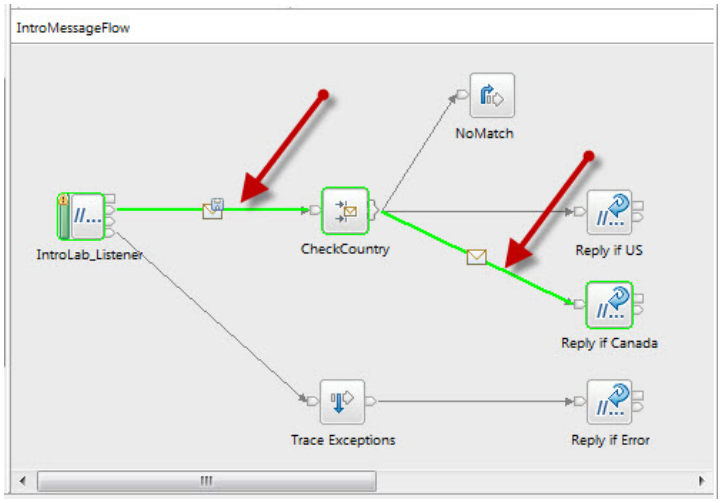
Click **Send** to send the message to the flow.



__37. The Flow Exerciser will run. Click **Close** after it stops.

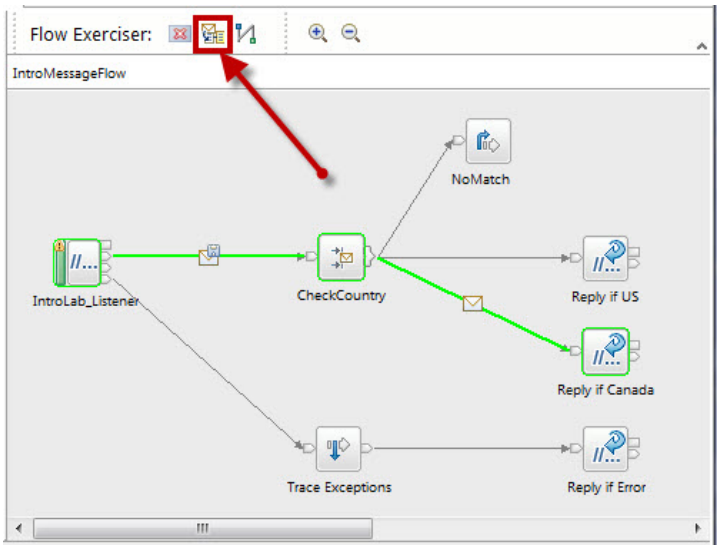


__38. The Flow Exerciser shows the path the CA message took through the flow.

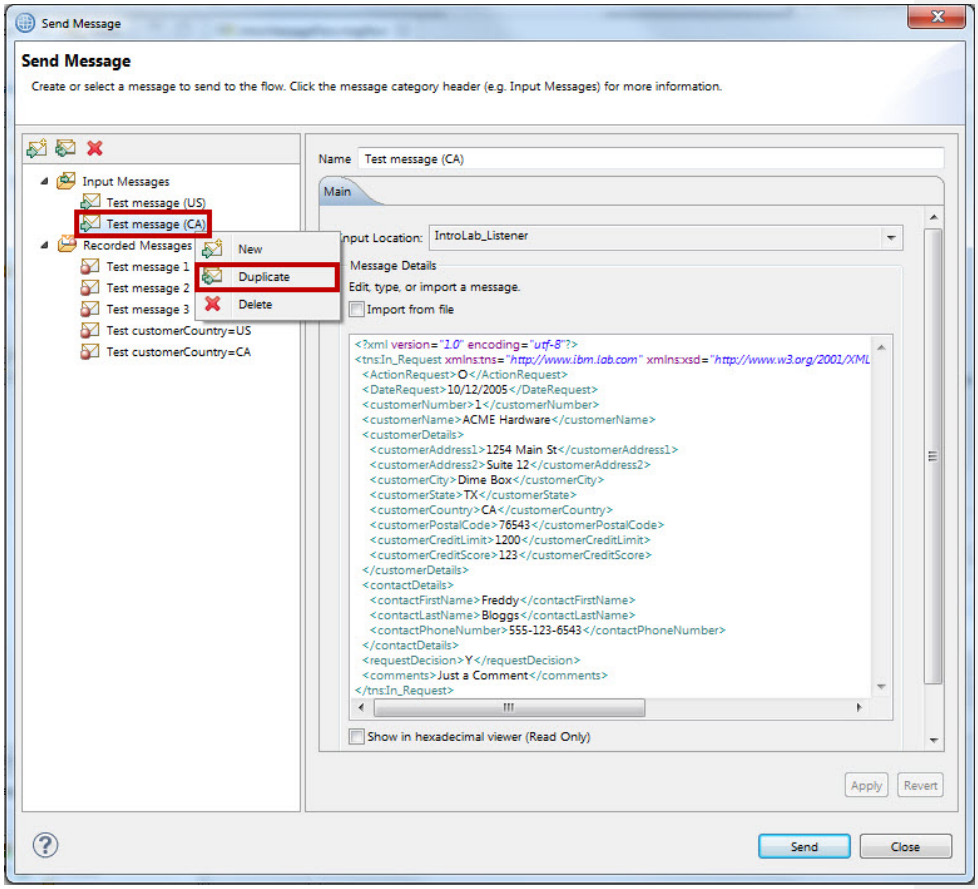


__39. Repeat these steps again, this time creating an **MX** test message.

Click **Send** to configure another message.



__40. Right-click **Test message (CA)**, and select **Duplicate** from the menu.



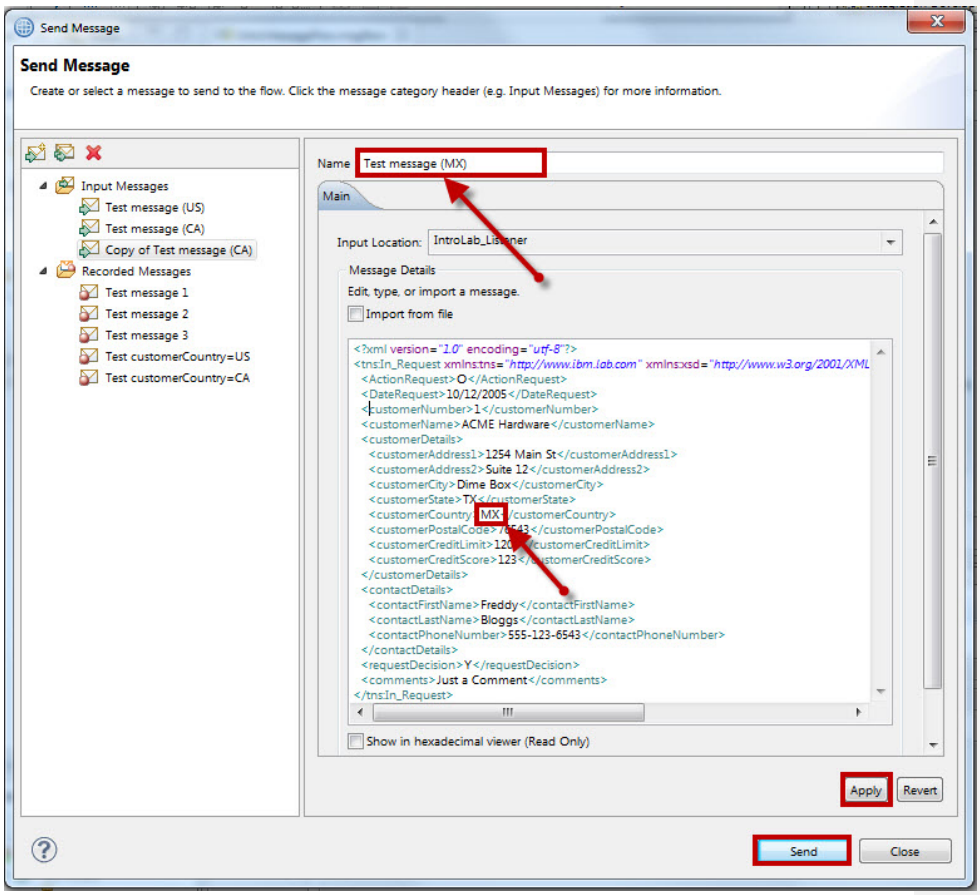
__41. Change the name of the new test message to “Test message (MX)”.

Change **customerCountry** to “MX”.

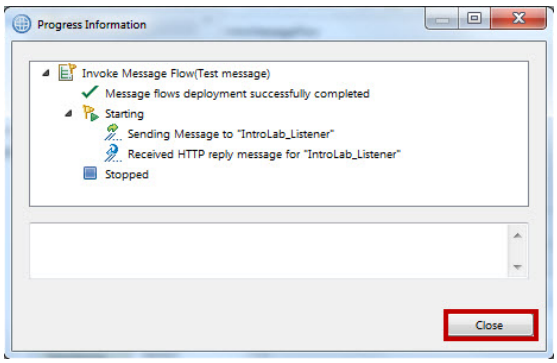
Click **Apply** to 

the change.

Click **Send** to send the message to the flow.



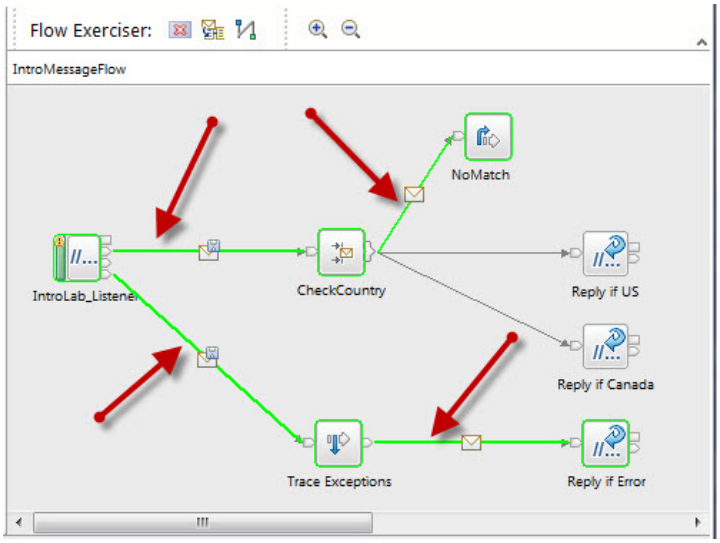
__42. The Flow Exerciser will run. Click **Close** after it stops.



__43. The Flow Exerciser shows the path the **MX** message took through the flow.

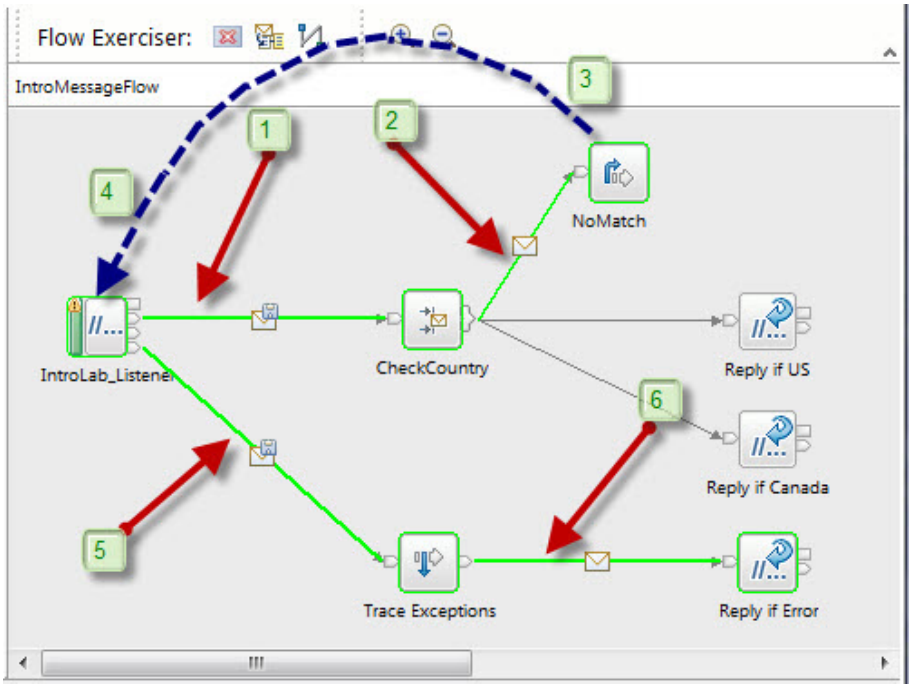
This path is rather interesting.

Examine what happened here.



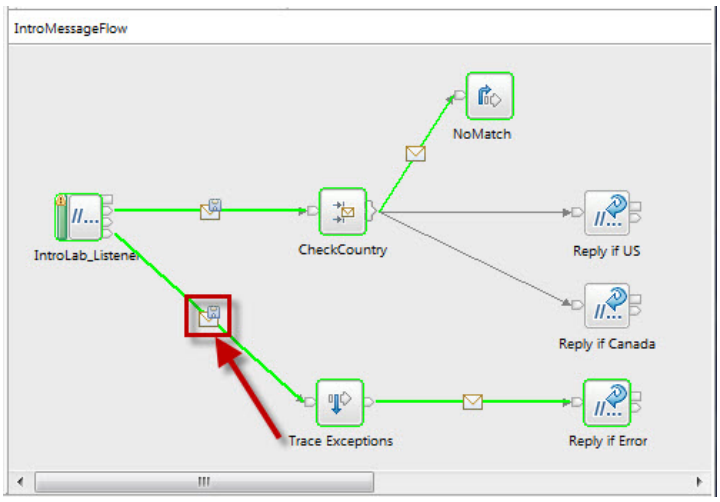
__44. Looking at these steps in order:

1. The input message was processed by the IntroLab_Listener node and passed to the CheckCountry node.
2. The CheckCountry node determined that MX did not have a matching entry in its routing table, and so passed the message via the default terminal to the NoMatch throw node.
3. The NoMatch throw node threw an exception.
4. The thrown exception was caught by the IntroLab_Listener node
5. The IntroLab_Listener node passed the input message, along with exception data, to the Trace Exceptions node via the Catch terminal.
6. After logging the message along with exception data to a file, the Trace Exceptions node passed control to the Reply if Error node, and the flow ended.



__45. Look at the message tree that was passed via the Catch terminal from the IntroLab_Listener node to the Trace Exceptions node.

Do this by clicking the recorded message to view it.

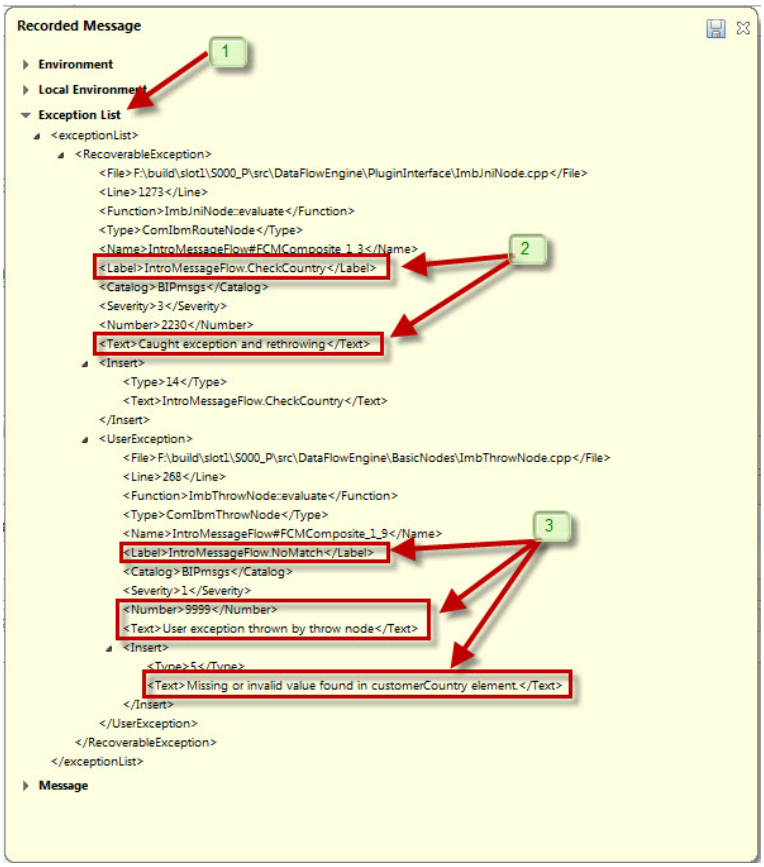


__46. Focus on the Exception List structure in the tree (1).

The Exception List structure provides a great deal of detail about what led up to, and what caused, an exception.

The items labeled (2) indicate that the CheckCountry node caught and rethrew an exception generated from a downstream node.

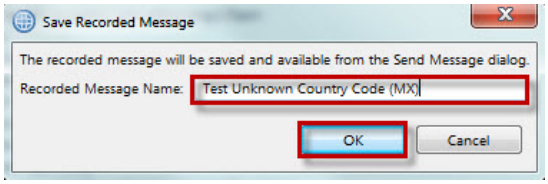
The items labeled (3) indicate that the downstream node that threw the exception was the NoMatch node, that the exception thrown was a User exception with an error number of 9999, and that the cause was a Missing or invalid value found in customerCountry element.



__47. Save this recorded message by clicking the **save icon** in the upper right corner.

Note**:** If the recorded message is closed, you can double-click the **icon** to reopen it.

__48. Call this recorded message “Test Unknown Country Code (MX)” and click **OK**.

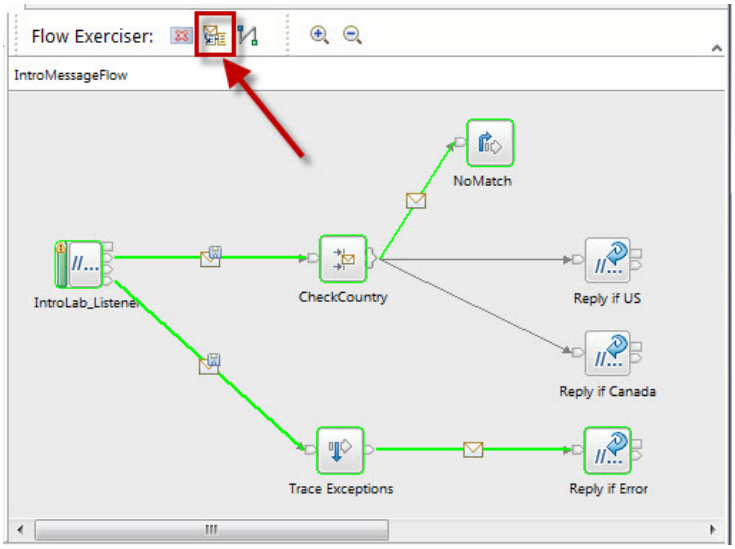


__49. Close the **Recorded Message** window.

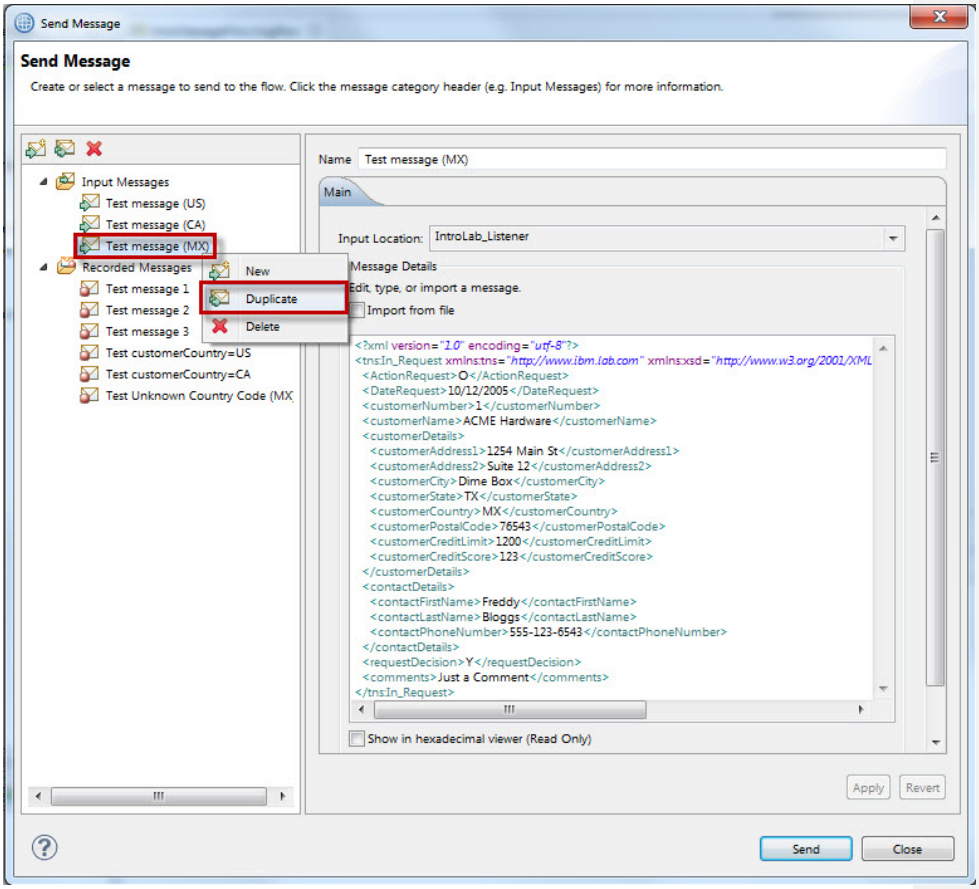
The approach of having an exception handler for the entire flow means that exceptions thrown for any reason, from any point in the flow, can be handled using a common exception handler.

The following steps will demonstrate this, by sending a malformed XML message into the flow, and observing what happens.

__50. Click **Send** to configure another message.



__51. Right-click **Test message (MX)**, and select **Duplicate** from the menu.



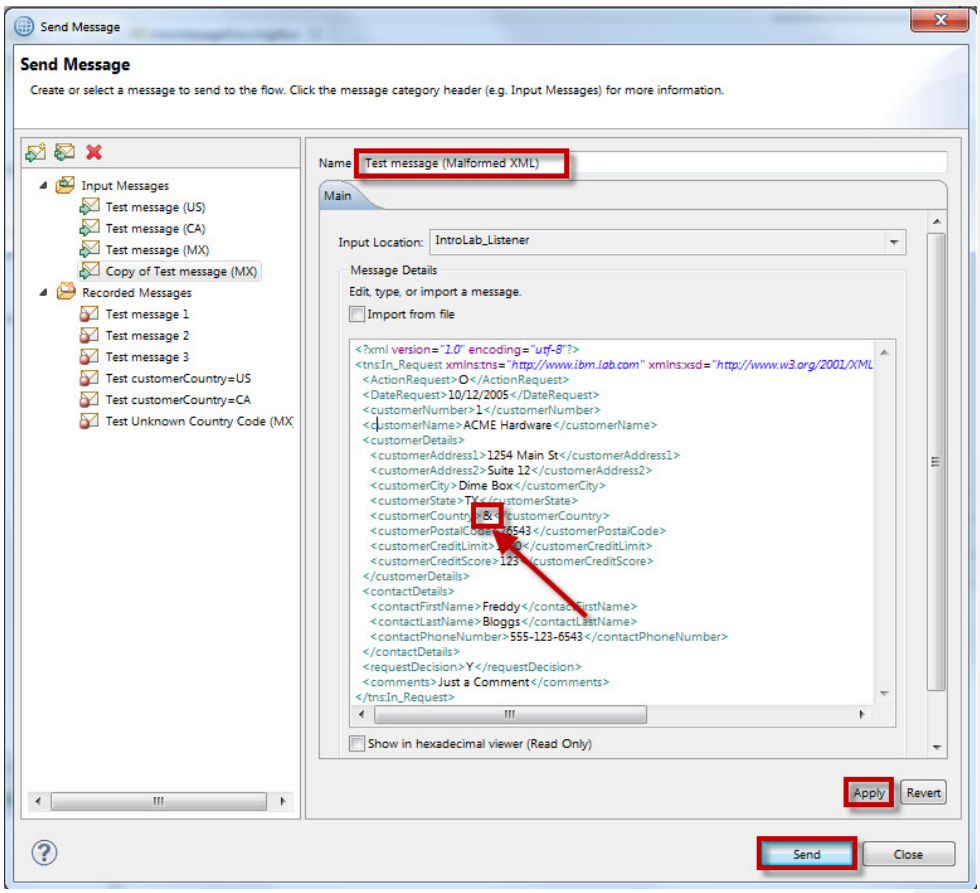
__52. Change the name of the new test message to “Test message (Malformed XML)”.

Change **customerCountry** to “&” (which is illegal in XML).

Click **Apply** to 

the change.

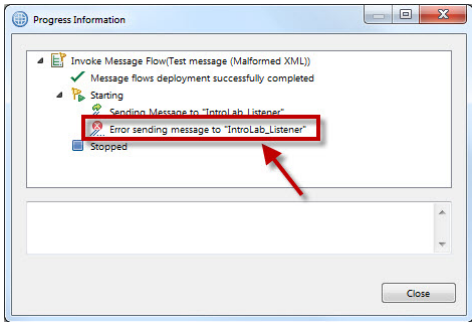
Click **Send** to send the message to the flow.



__53. The Flow Exerciser will run.

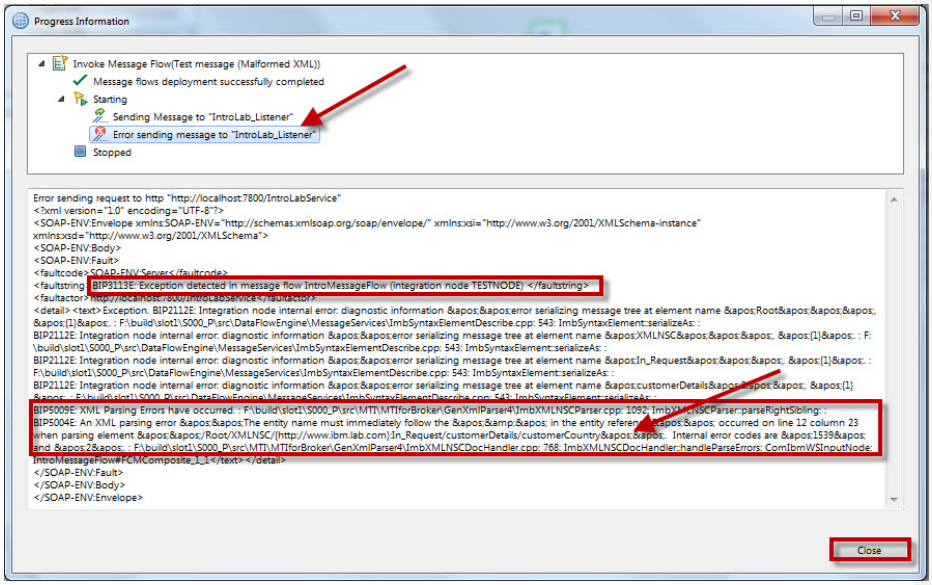
This time you get an error back from the flow.

Click the **error message** to see its contents.



__54. Review the highlighted text. Briefly, it explains that an exception occurred, and that the cause was an XML parsing error at element “customerCountry”.

Click **Close**.



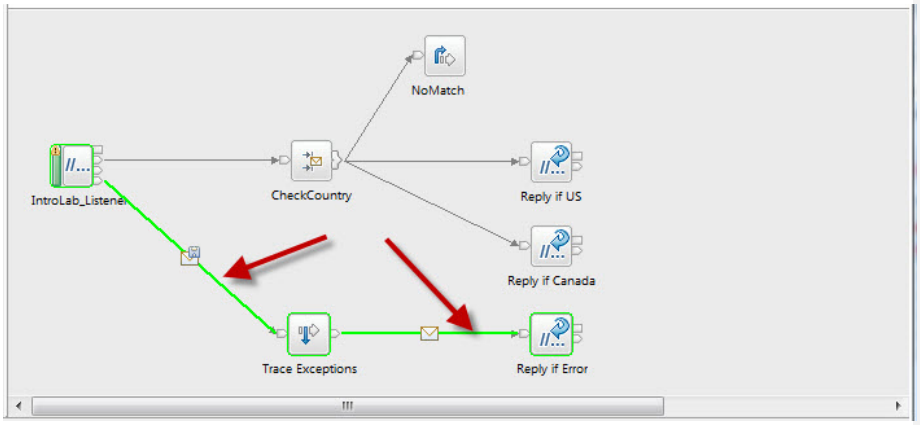
__55. The Flow Exerciser shows the path the malformed XML message took through the flow.

Notice here that the message never made it to the CheckCountry node.

The reason was that ACE tried to parse the XML message before passing it to the CheckCountry node, but because it could not be successfully parsed, the message went straight to the Catch terminal on the IntroLab_Listener node.

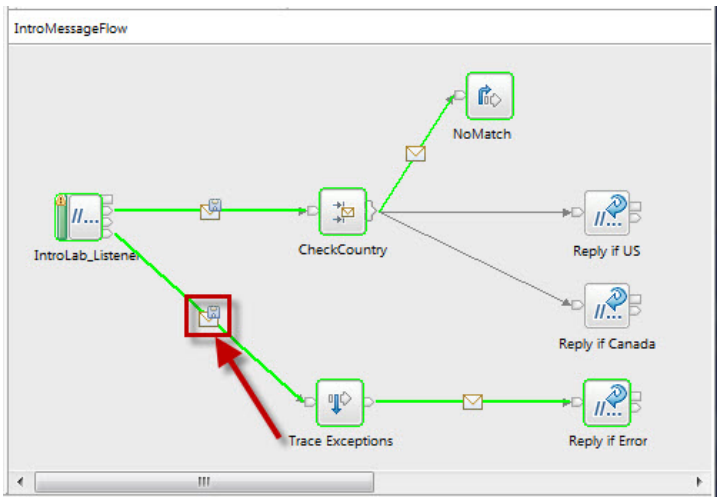
Why was the Failure terminal not used?

Because On-Demand parsing is in effect, no attempt was made to parse the message *inside* the IntroLab_Listener node. The first attempt to parse the message was *after* the IntroLab_Listener node but before the CheckCountry node. Had the error been detected *inside* the IntroLab_Listener node, the Failure terminal would have been used.



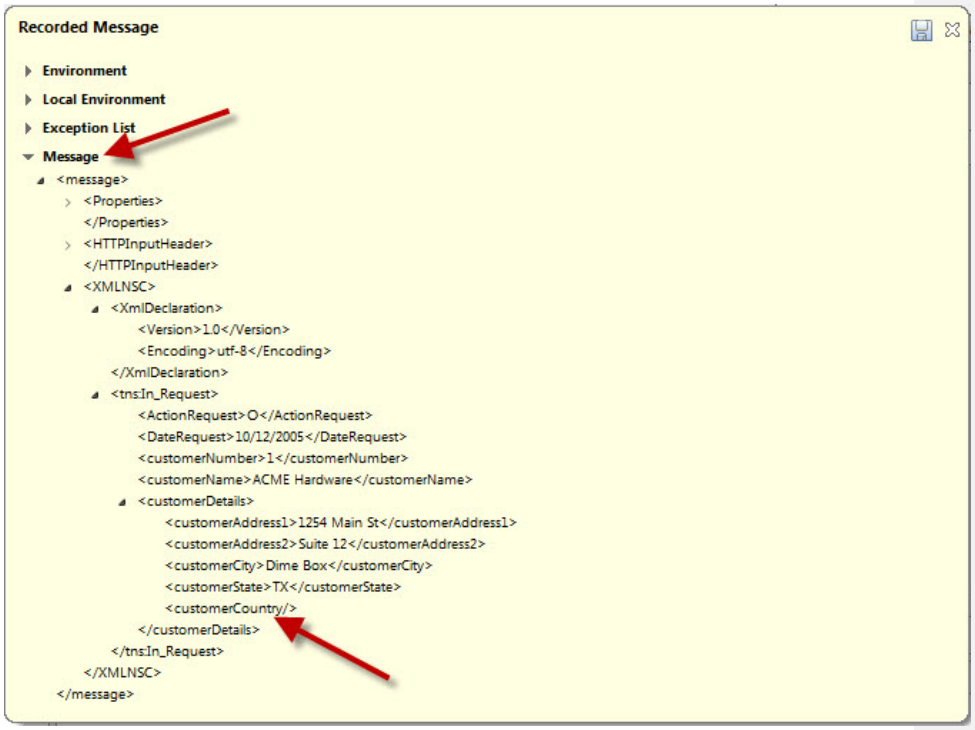
__56. Look at the message tree that was passed via the Catch terminal from the IntroLab_Listener node to the Trace Exceptions node.

Do this by clicking the **recorded message** to view it.



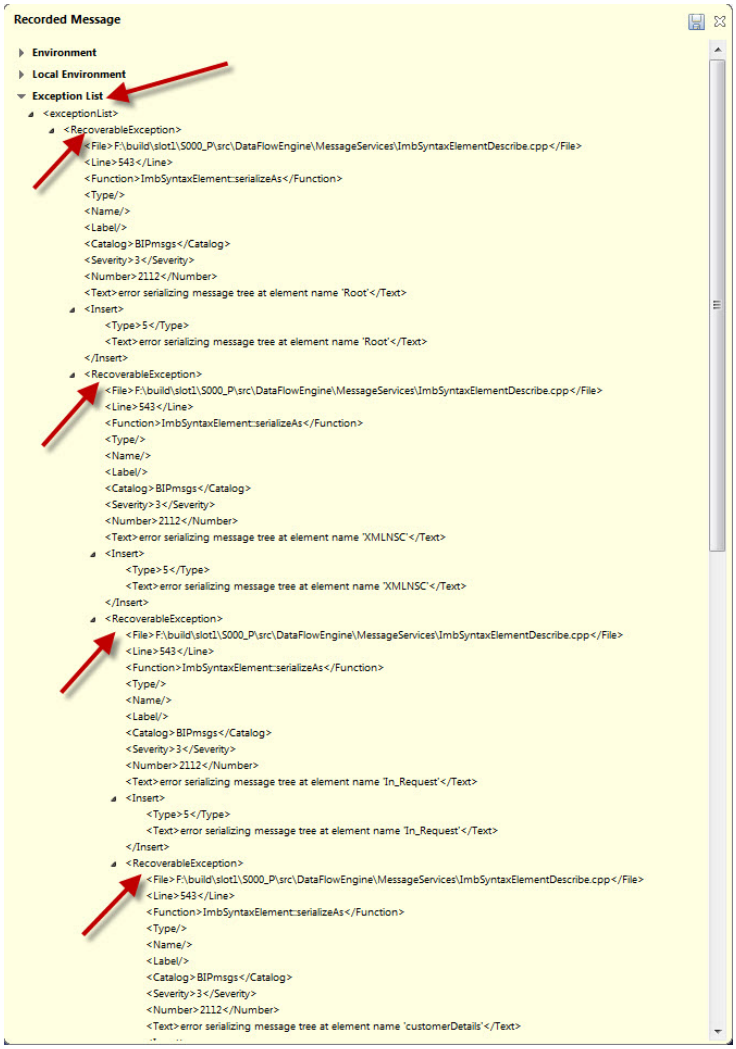
__57. First, focus on the message body.

The Flow Exerciser will display as much of the message tree as possible. You see here that the message up to customerCountry is displayed, but customerCountry itself, and any elements below it, are not displayed. The reason is that the XMLNSC parser was only able to successfully parse the message up to that point, but no further, due to the malformed data (“&”) in the message.



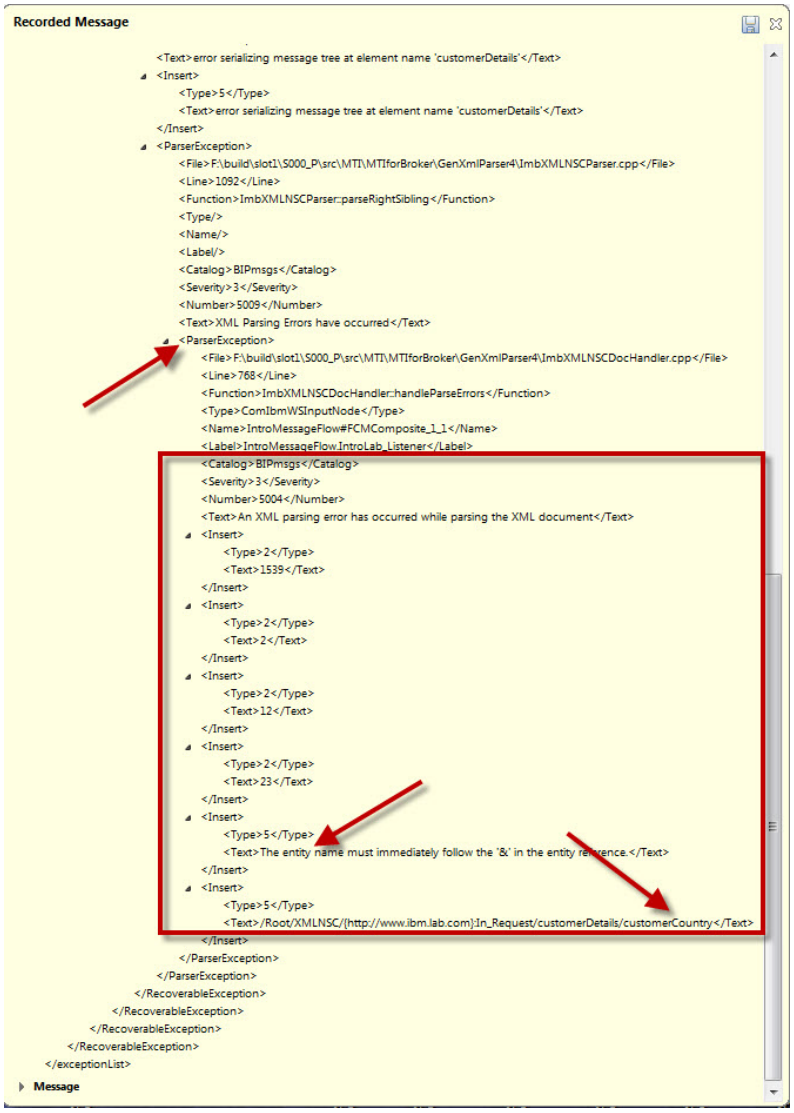
__58. Now collapse the **Message** section, and expand the **Exception List** section.

There are multiple RecoverableException sections, showing the XMLNSC parser attempting to find a way to successfully parse the data.



__59. Scroll to the bottom of the **Exception List** structure – the original exception will be found there.

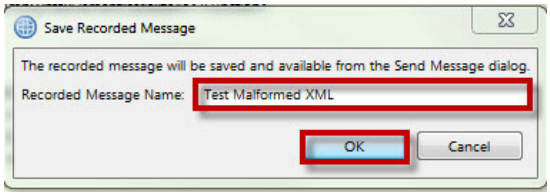
The exception type (“ParserException”) as well as the cause of the parser error and the element involved (“customerCountry”), can be found here.



__60. Save this recorded message by clicking the **save icon** in the upper right corner.

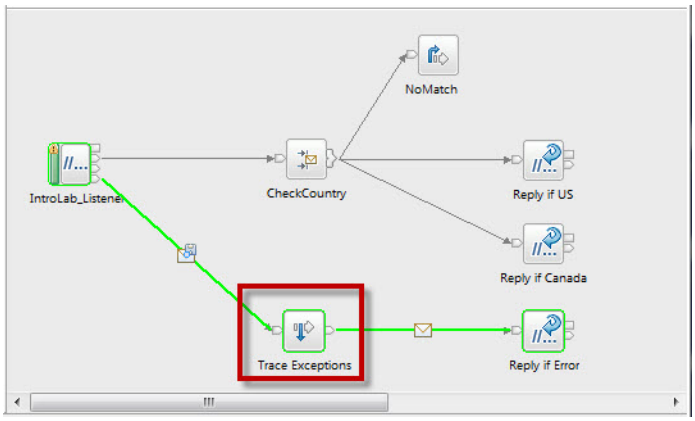
Note: If the recorded message is closed, you can double-click the **icon** to reopen it.

__61. Call this recorded message “Test Malformed XML” and click **OK**.



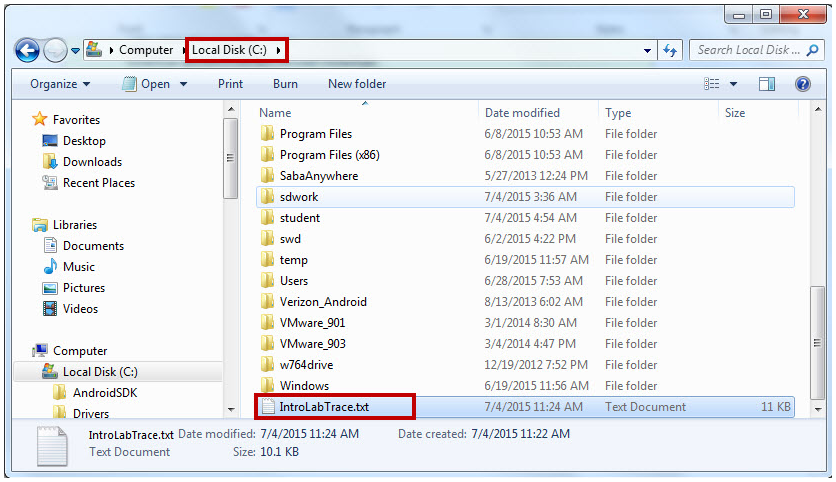
__62. Close the **Recorded Message** window.

__63. Remember that the Trace Exceptions node was also logging the message tree to a file.



__64. Use the Windows Explorer to locate the file you configured the Trace Exceptions node to use.

Navigate to **C:\IntroLabTrace.txt**, **(_or location of your choice)**, and double-click the file to open it.



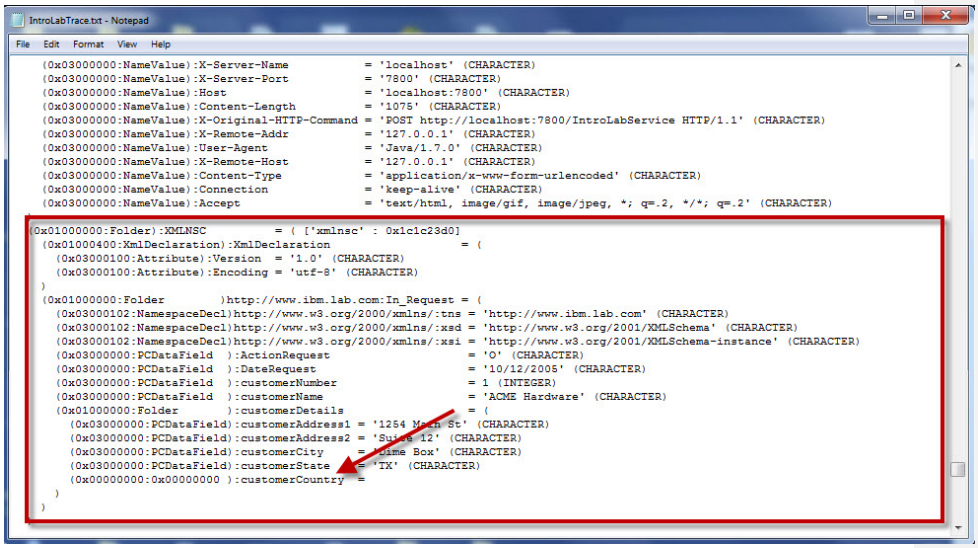
__65. Recall that in Lab 2 you configured the Trace node to capture the “ACE-internal” view of the message tree.

By this time there will be quite a bit of data in the trace file – you will need to scroll to near the bottom of the file to find the most recently logged data.

Scroll down to locate the most recent **Message** data.

Note the correspondence with what you saw in the Recorded Message from the Flow Exerciser – the XMLNSC parser was able to parse to the customerCountry element, but failed at that point.

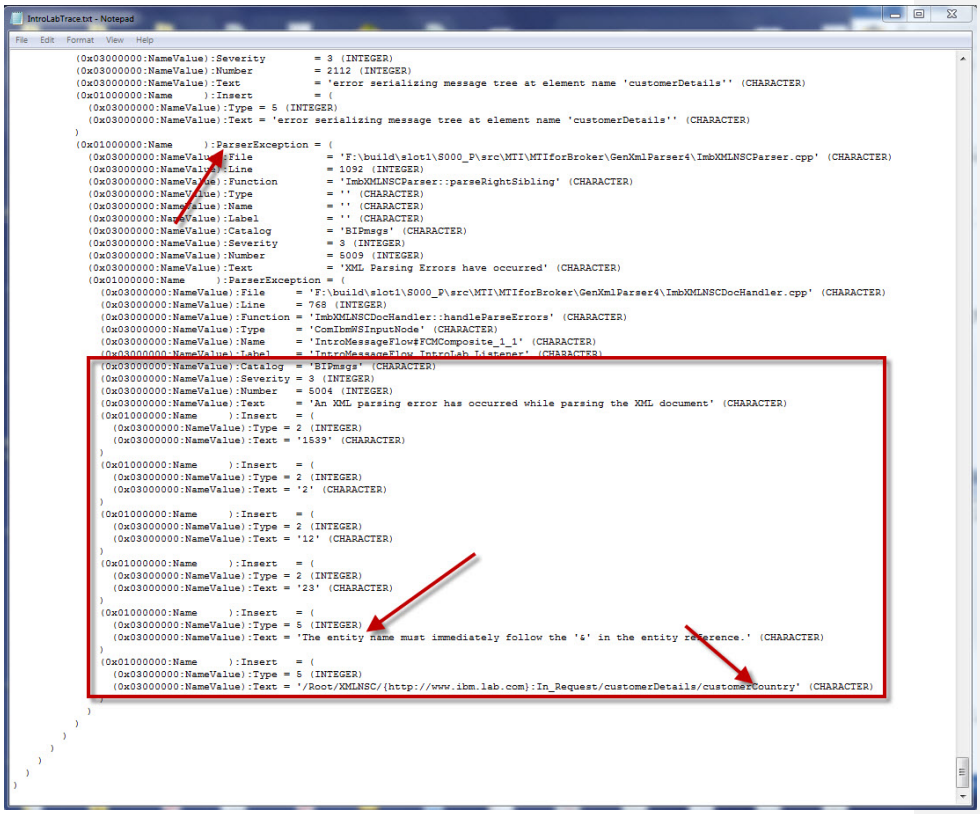
Note that the “ACE-internal” view of the message tree contains more detail about the message tree than does the XML-rendered view – for example, showing the different data types (such as INTEGER and CHARACTER).



__66. Now scroll down further to find the **Exception List** structure.

Scroll all the way to the bottom of that section, so you can see the original exception.

Again, note the correspondence with what you saw in the Recorded Message from the Flow Exerciser—the exception type is “ParserException” and the element involved is “customerCountry”.

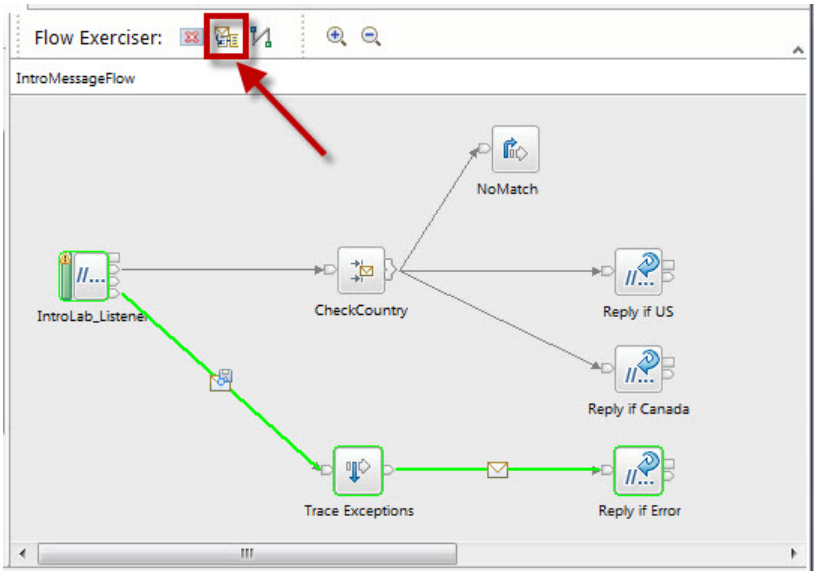


The detailed exception information provided by App Connect Enterprise makes it possible for flow-level exception handlers to be written that can log detailed information about failures, as well as to take action in some if not many cases to recover from those exceptions programmatically.

__67. Close the Notepad window.

Think also about what you’ve configured in terms of testing assets by using the Flow Exerciser as you have in this lab, as well as the first two labs.

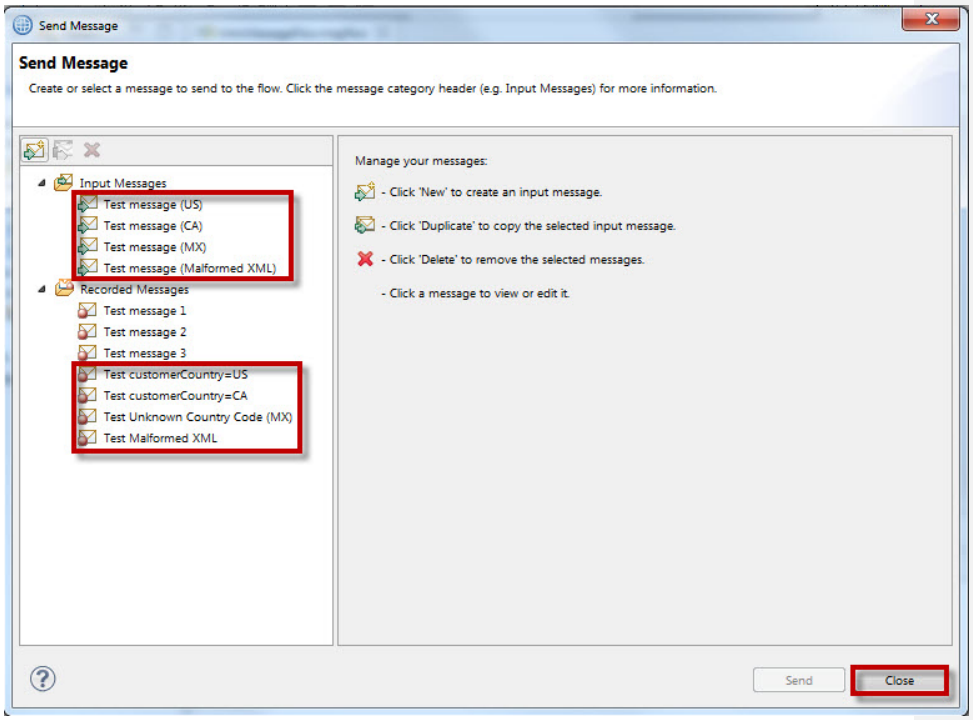
__68. Return to the Flow Exerciser, and click **Send** to view the configured test messages.



__69. You configured test messages for four scenarios: two “happy paths” (for US and CA) and two error paths.

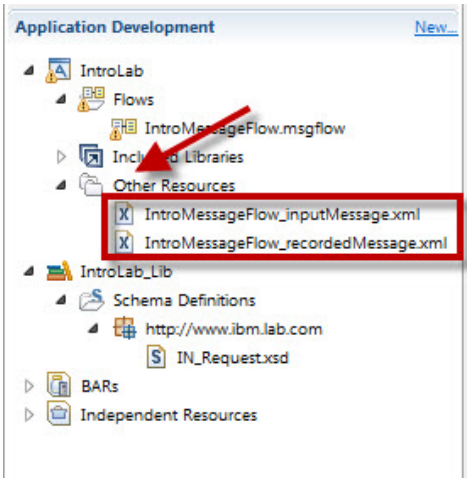
In addition, you saved four Recorded Messages that captured the results of those tests.

Click **Close** to close the window.

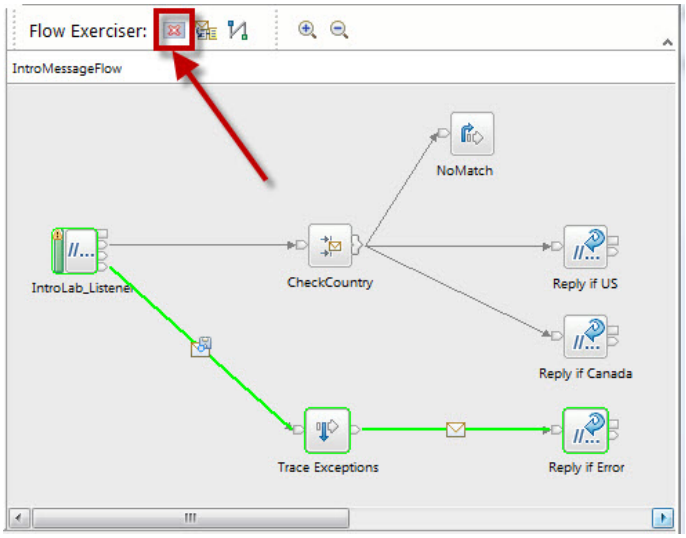


__70. In the Application Development pane, the IntroMessageFlow_inputMessage.xml file and the IntroMessageFlow_recordedMessage.xml file can be exported and used by other testers, or by external test tools, to perform regressing testing on this flow in the future.

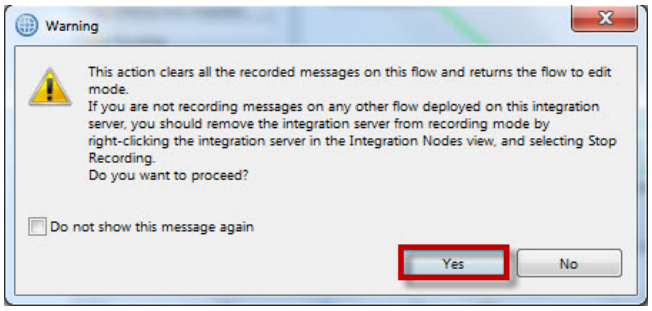
Commented [EL6]: Should this be "regression testing"? Please confirm



__71. Stop the **Flow Exerciser**.



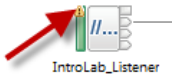
__72. Click **Yes** to dismiss the pop-up.



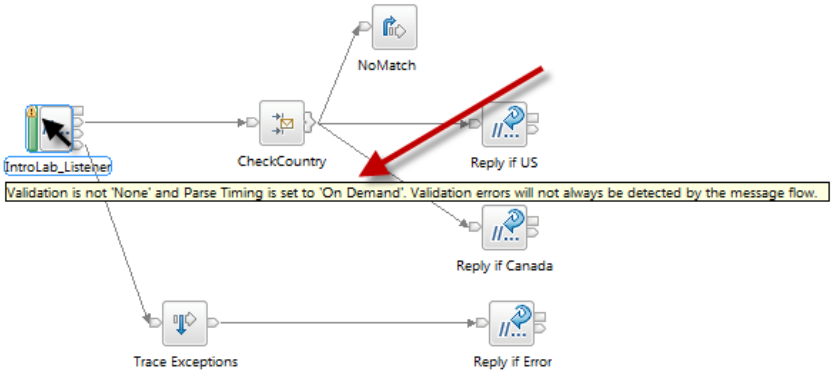
6.4 Diagnosing failures

The Flow Exerciser is a very useful tool for testing a new or modified message flow. But there are times when the Flow Exerciser may not provide sufficient information for diagnosing a problem. In this section you will see an example of such a situation.

__73. You may recall from Lab1 that a change you made resulted in a warning appearing on the IntroLab_Listener node.

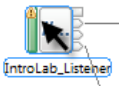


__74. Hover the mouse pointer over the node to see the warning.

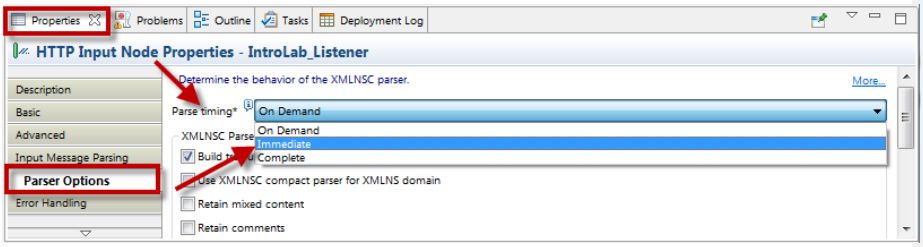


__75. In Lab1 you were told to ignore this warning. You are now going to address it.

Click the **IntroLab_Listener** node.



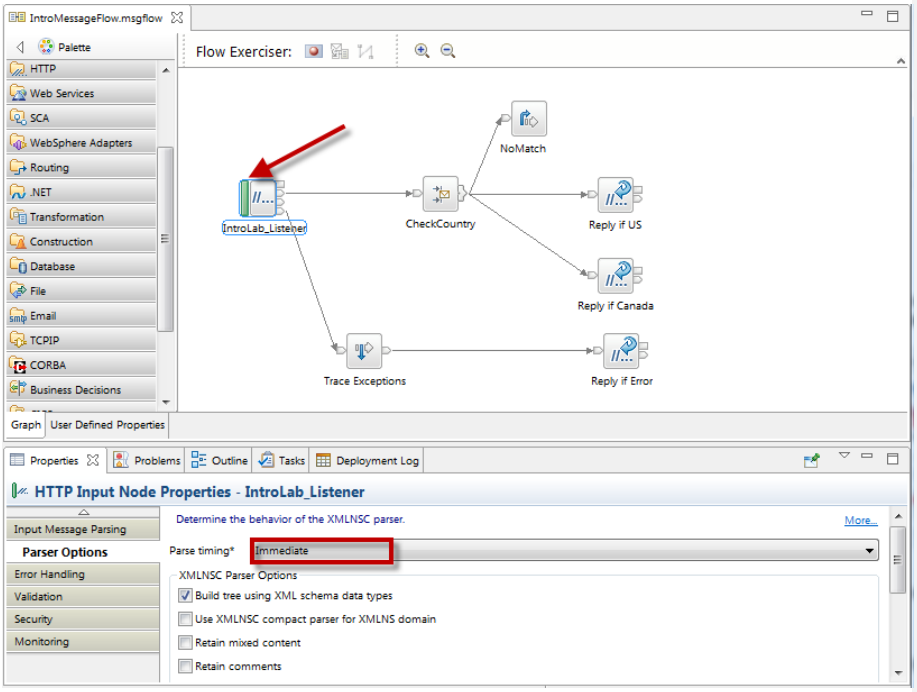
__76. In the **Properties** sheet for the node, under **Parser Options**, change **Parse timing** from On Demand to **Immediate**.



__77. 

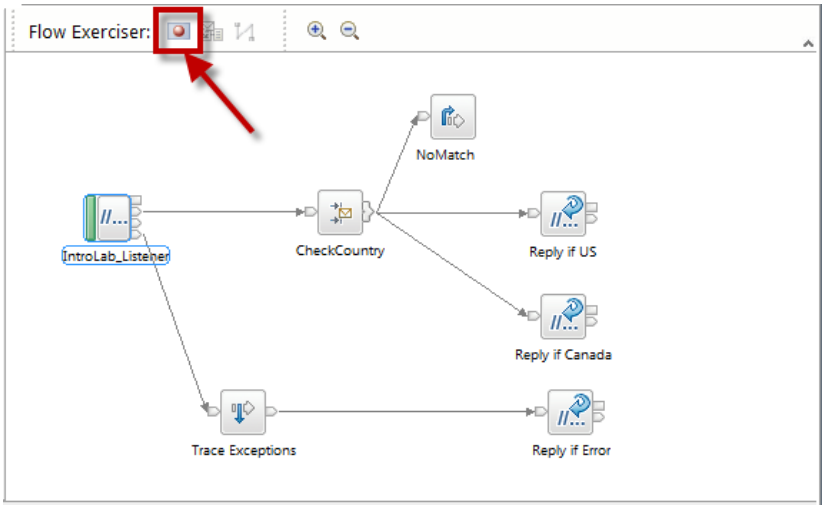
Save the message flow (Press **Ctrl+S**).

__78. With Parse Timing set to Immediate, the warning no longer appears.

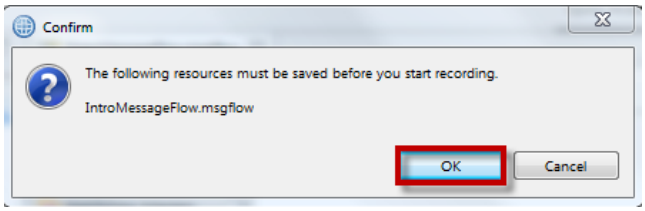


__79. You will now repeat the test using the malformed XML document.

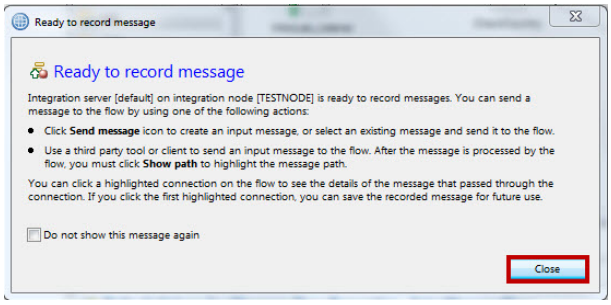
Start the **Flow Exerciser**.



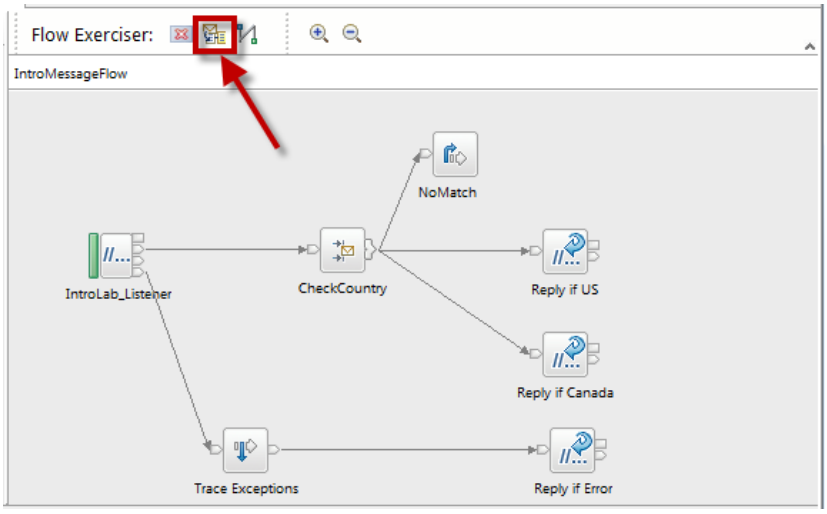
__80. If prompted to save the message flow, click **OK**.



__81. Click **Close** to start recording.



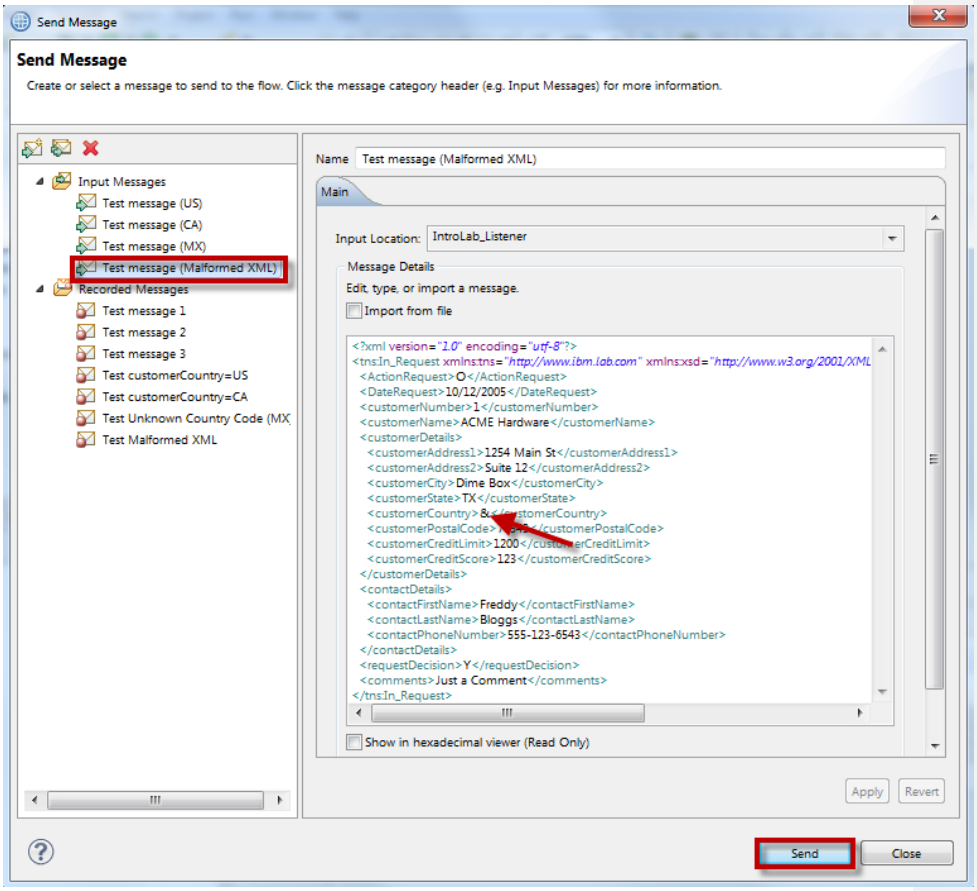
__82. Click the **Send** message icon to configure a test message.



__83. Select **Test message (Malformed XML)**

Verify that the “&” is in the customerCountry field.

Click **Send**.

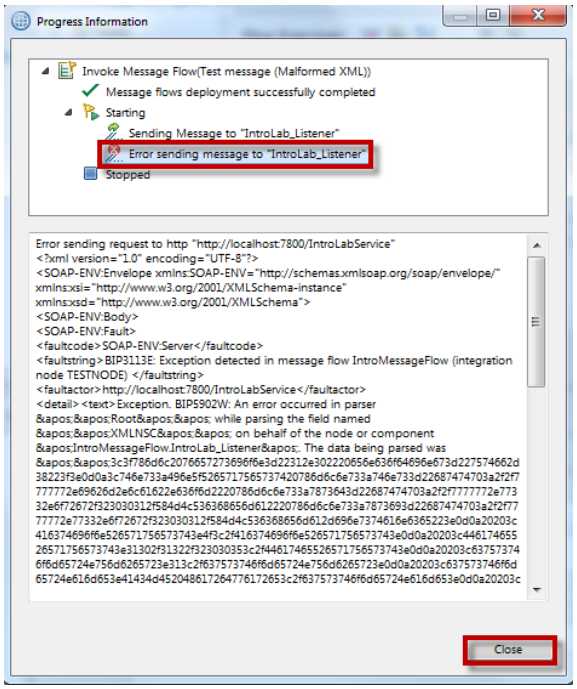


__84. Because the message was malformed, you expected this error.

Click the **error** and examine the returned error explanation.

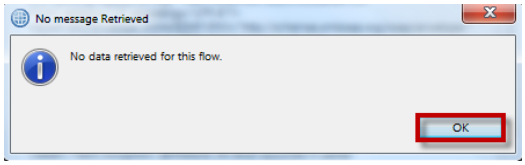
You will notice some differences between this exception data and the data that was returned in your earlier test.

Click **Close**.

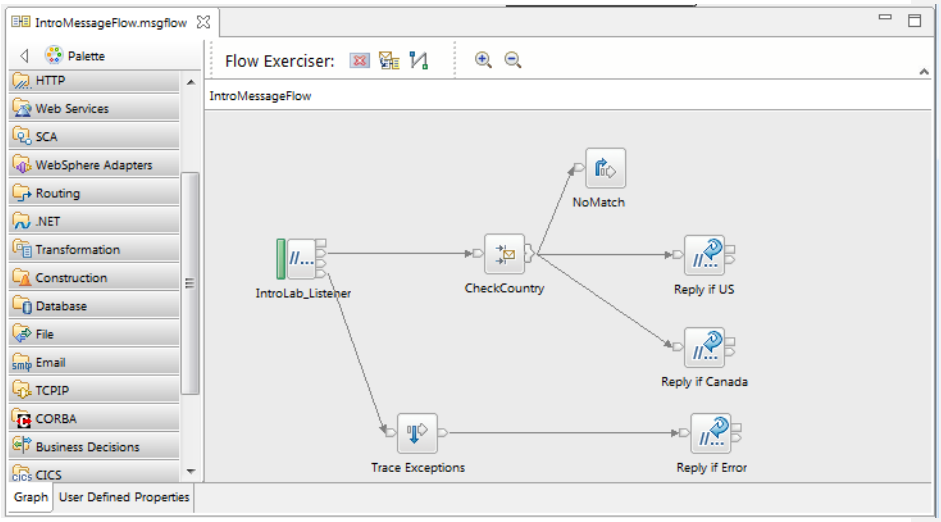


__85. What is this? You did not get this on your previous test.

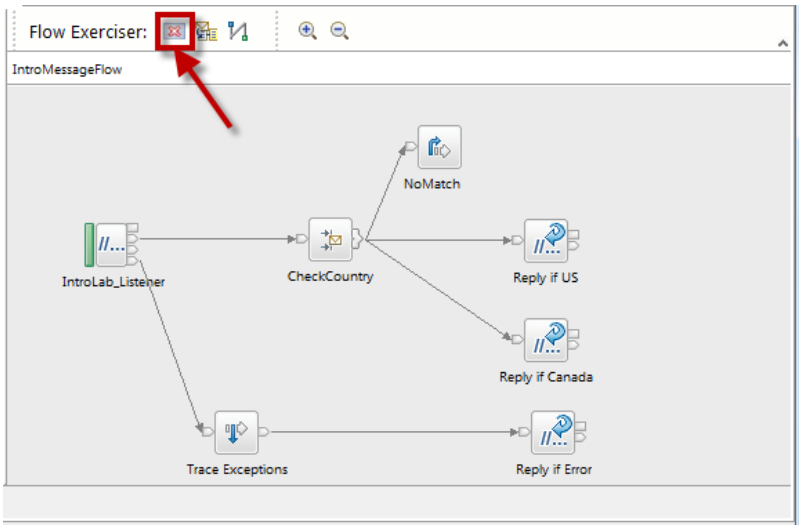
Click **OK** to dismiss.



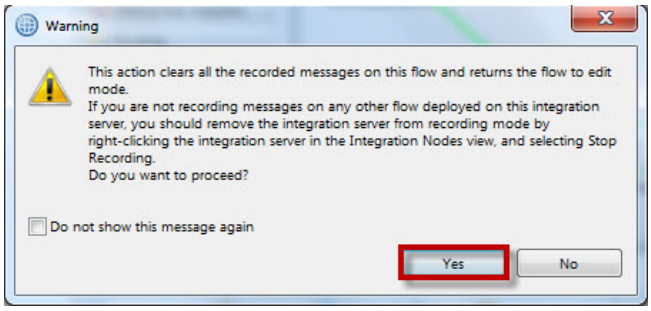
__86. Here's another surprise – Because no data was retrieved from the Flow Exerciser, you cannot see what path the message took through the flow.



__87. Stop the **Flow Exerciser**.



__88. Click **Yes** to dismiss the pop-up.



Previously, when you tested the malformed XML message, you were able to see the exception path being taken when testing with the Flow Exerciser. Something about the change you made to the Parse Timing resulted in the Flow Exerciser behaving differently.

Because you want to be able to handle parsing errors in your flow, you need to determine why the change you made to the Parse Timing is preventing your flow from detecting the error and handling it.

In the next section you will use the Flow Debugger to better understand what is happening in this case and how it is different from your previous test.

6.5 Using the integrated Flow Debugger

The Flow Exerciser is a very useful tool for testing a new or modified message flow. But there are times when the Flow Exerciser may not provide sufficient information for diagnosing a problem. In situations like that, the integrated Flow Debugger can be a useful alternative.

6.5.1 Key Idea: The Flow Debugger

Key Idea: Flow Debugging.
Read the section below for an overview of the integrated Flow Debugger.

The Debug perspective in the IBM ACE Toolkit provides a Flow Debugger which enables more detailed message flow debugging than is possible using the Flow Exerciser.

Using the Flow Debugger, you can set breakpoints in a message flow, and then step through some or all of the nodes in a flow. While you are stepping through, you can examine and change elements in the message assembly, as well as variables used by ESQL code and Java code. This is in contrast to the Flow Exerciser, which shows you the path a given message takes through the flow, but provides no ability to set breakpoints or to change variables other than at the outset. Using the Flow Debugger, the ability to set breakpoints at strategic points in the flow, to see and modify the message assembly between nodes, and to "step into" nodes such as Java Compute nodes for line-by-line debugging, provide for much more detailed analysis of flow processing than is possible with the Flow Exerciser.

You can debug a wide variety of error conditions in flows, including the following:

- Errors occurring at flow initiation
- Nodes that are wired incorrectly (such as outputs that are connected to the wrong inputs)
- Errors in transformation or logic within your code or maps
- Incorrect conditional branching in transition conditions
- Unintended infinite loops in flow

From a single IBM ACE Toolkit, you can attach the debugger to one or more Integration Servers, and debug multiple flows in different Integration Servers (and therefore multiple messages) at the same time.

The Flow Debugger does have some restrictions that must be kept in mind. For example, flows in an Integration Server can be debugged by only one user at a time. Therefore, if you attach your debugger to an Integration Server, another user cannot attach a debugger to that same Integration Server until you have ended your debugging session. This is true even if you are debugging different message flows.

Another important restriction on the Flow Debugger is that it cannot be used in conjunction with the Flow Exerciser. So message flows that are to be debugged must be driven either by external tools (such as SoapUI) or using the Test Client that is integrated with the ACE Toolkit. You will be using the latter throughout the remainder of this lab.

__a. By default, the debugger is not enabled. So, you need to set the JVM debug port by modifying the **server.conf.yaml** configuration file for your integration server. Navigate to **c:\myaceworkdir** (or different if you choose a different location) and open the **server.conf.yaml** configuration file by using a text editor.

Search for the „**jvmDebugPort**„ in the Resource Managers section of the .yaml file, uncomment it (remove # character) and set a value for the **jvmDebugPort** property to **4449**, which sets the JVM debug port to be used for debugging flows in the Toolkit.

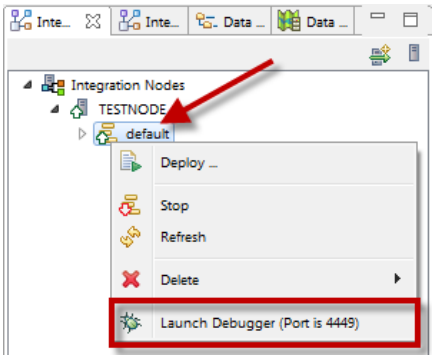
Save the configuration file.

Restart the integration server (just use ctrl+c in the integration console in which you have started it, and start it again using the „IntegrationServer --work-dir c:\myaceworkdir" command.

__b. In the Integration Nodes view of the Toolkit, right-click on the **myaceworkdir** integration server.

You should see Launch Debugger (Port is 4449) appear, indicating that the debugger is ready for use.

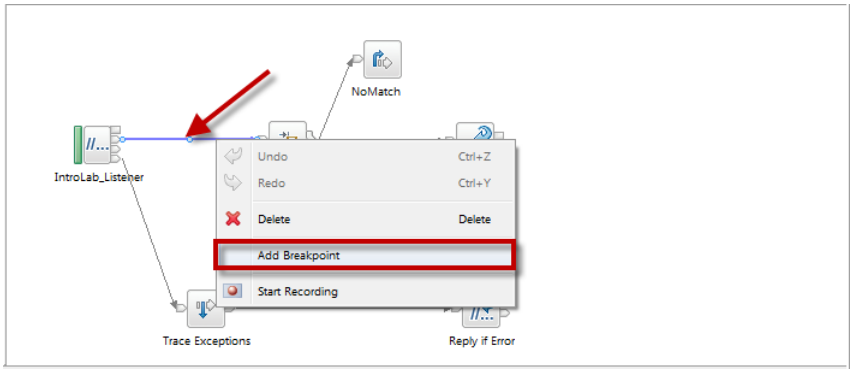
Hopefully, your ZABA laptop is allowing you to connect to the local port through your local firewall…



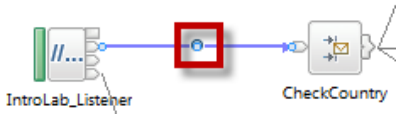
__89. One very nice feature of the Flow Debugger is that you can set breakpoints at specific points in your message flow.

To do this, right-click the **connector** between the IntroLab_Listener node and the CheckCountry node.

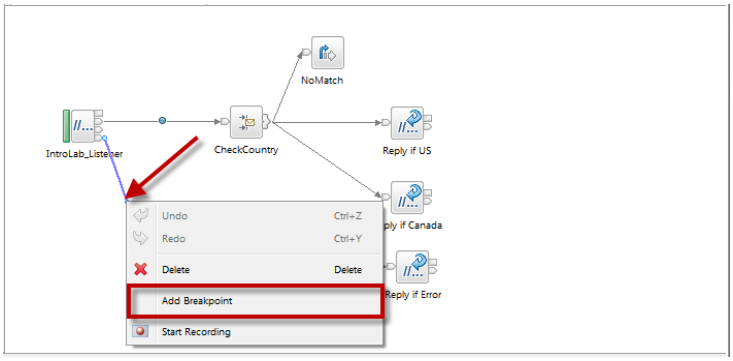
Select **Add Breakpoint**.



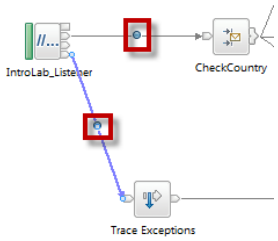
__90. A small blue circle will appear on the connector, indicating at a breakpoint has been set.



__91. Do the same thing on the connector between the IntroLab_Listener node and the TraceExceptions node – right-click the **connector** and select **Add Breakpoint**.



__92. A small blue circle will appear on the connector, indicating at a breakpoint has been set.



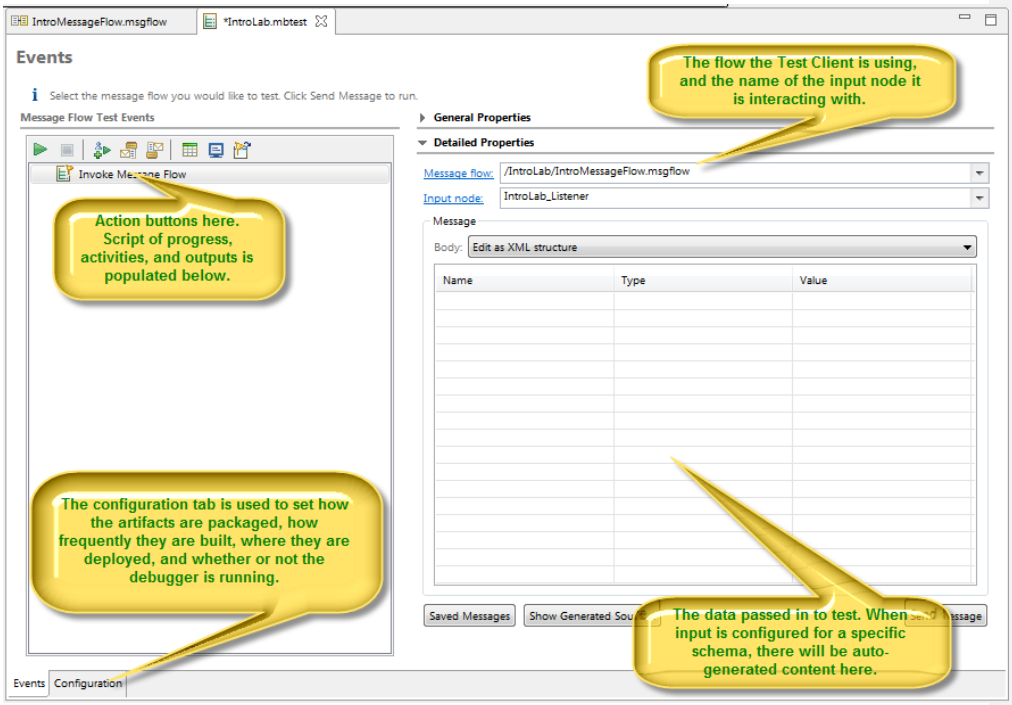
Because the Flow Analyzer has been disabled, message flows that are to be debugged must be driven either by external tools (such as SoapUI) or using the Integrated Test Client that is part of the ACE Toolkit.

6.5.2 Key Concept: The integrated Test Client

Key Concept: The integrated Test Client
Read the section below for information on the integrated Test Client and how to use it in conjunction with the Flow Debugger.

With App Connect Enterprise there are two integrated tools for testing integration solutions directly, from inside the Toolkit, without the need to use external tools. These are the Flow Analyzer (which you have seen) and the Test Client.

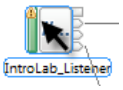
Take a moment to familiarize yourself with the Test Client.



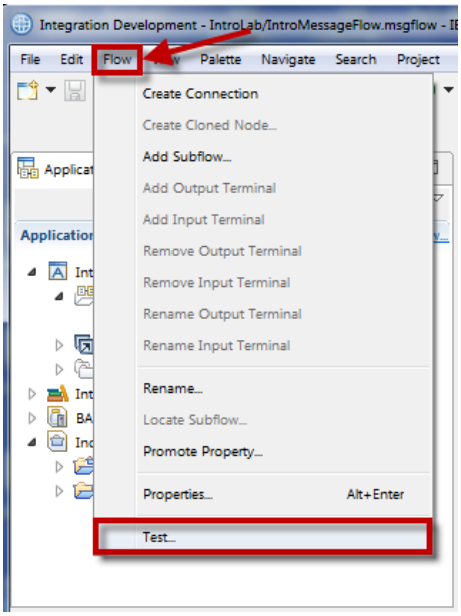
Test Client instances can be created for MQ, JMS, HTTP, SOAP and SCA input nodes. They exist as a single file in the workspace with a .mbtest file extension. They can be embedded into the applications or libraries and thus passed from developer to developer with a project.

Like the Flow Analyzer, the Test Client has support for features such as monitoring all supported output paths through a flow, and storing sample messages for replay purposes. It also encapsulates the build and deploy process, and can be used to launch the debugger.

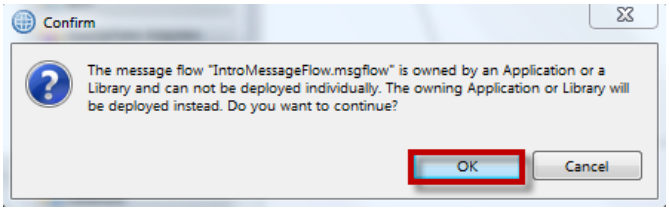
__93. To launch the Integrated Test Client, click the **IntroLab_Listener** node to give it focus.



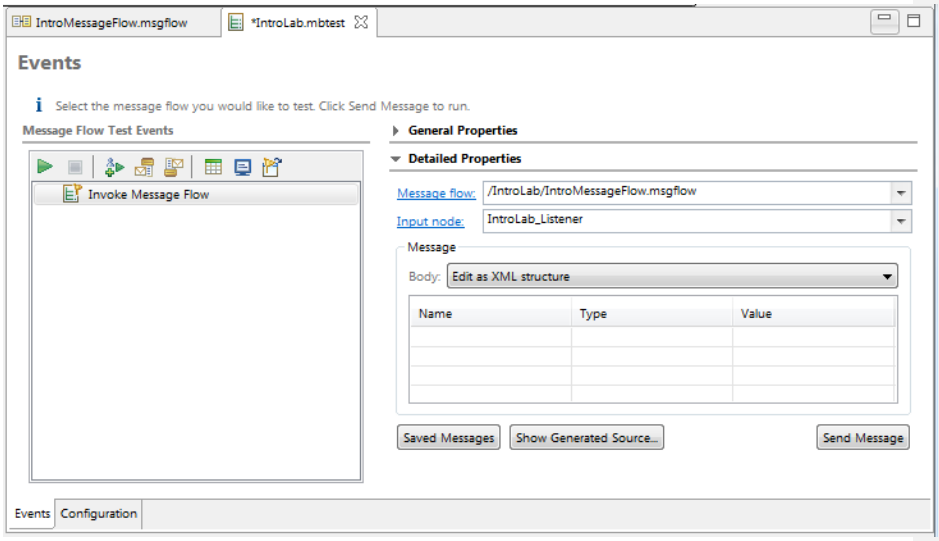
__94. In the Toolkit menu bar, select **Flow > Test.**



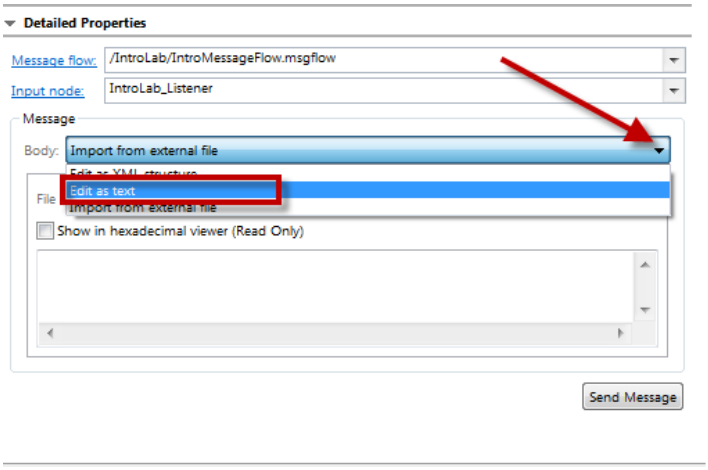
__95. Click **OK** on the confirmation popup.



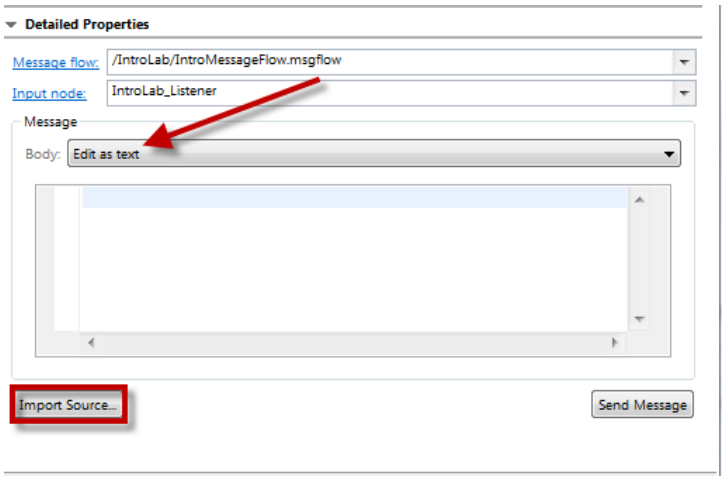
__96. The Test Client will be started.



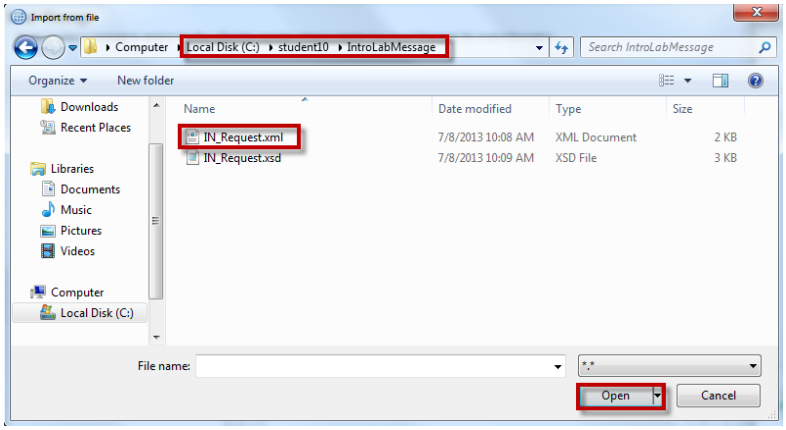
__97. Under **Detailed Properties**, click the drop-down for **Body**, and select **Edit as text**.



__98. With “Edit as text” set, click **Import Source.**



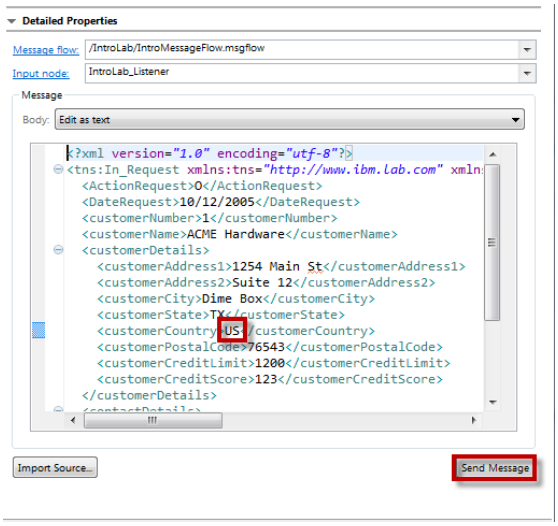
__99. In the Open window, navigate to **C:\student10\IntroLabMessage**, select the **IN_Request.xml** file, and click **Open**.



__100. First you will run a good message through.

In the test message, locate customerCountry, and change it from “USA” to “US”.

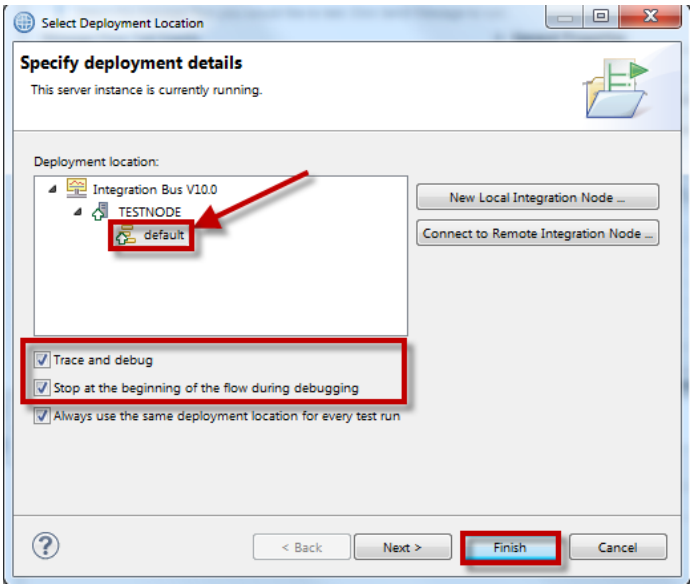
Click **Send Message**.



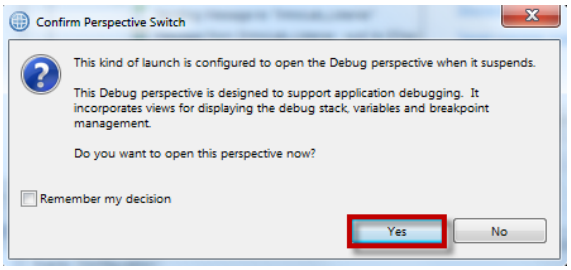
__101. Select the **myaceworkdir** integration server as the Deployment location.

Also make sure that *both* the **Trace and debug** as well as the **Stop at the beginning of the flow during debugging** checkboxes are selected.

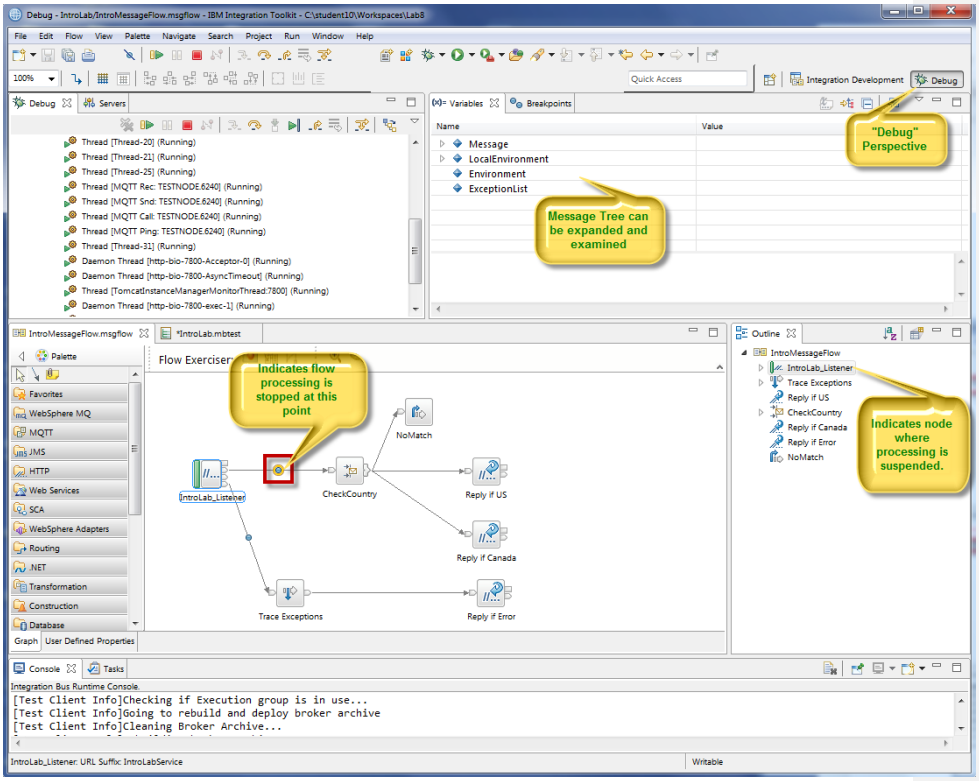
Click **Finish**.



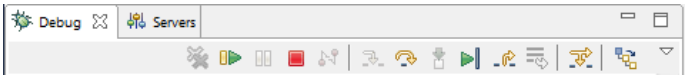
__102. Click **Yes** on the confirmation to launch the Debug perspective.



__103. Review the Debug perspective, and the various featured that are presented.



__104. Note the toolbar and the controls it contains.

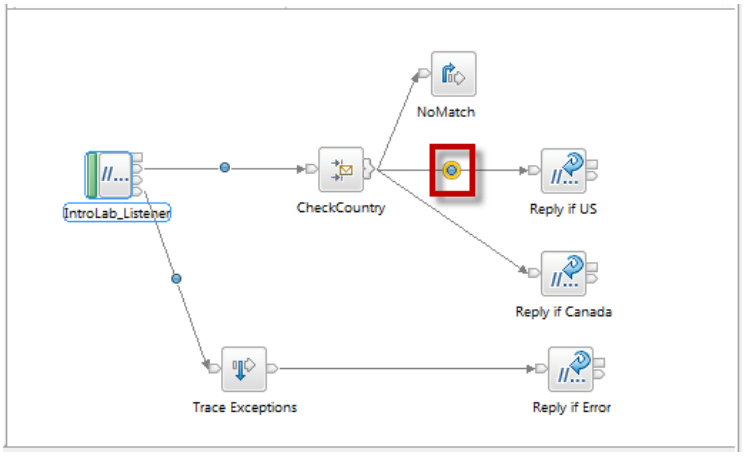


__105. Hover the mouse pointer for a brief description of each. Significant controls include:

- Resume execution
- Terminate
- Step over a node
- Run to completion

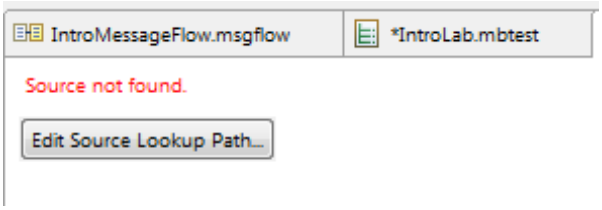
__106. Click to step over the next node in the flow.

Because customerCountry was set to “US”, you should see the following:



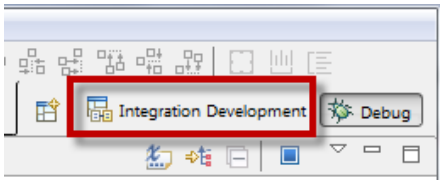
__107. Click  to run to completion.

__108. You may see the following:

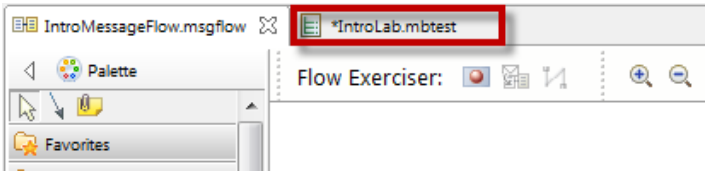


If so, click  to resume execution.

__109. In the upper right corner of the Toolkit, click the **Integration Development** tab to return to the Integration Development perspective.



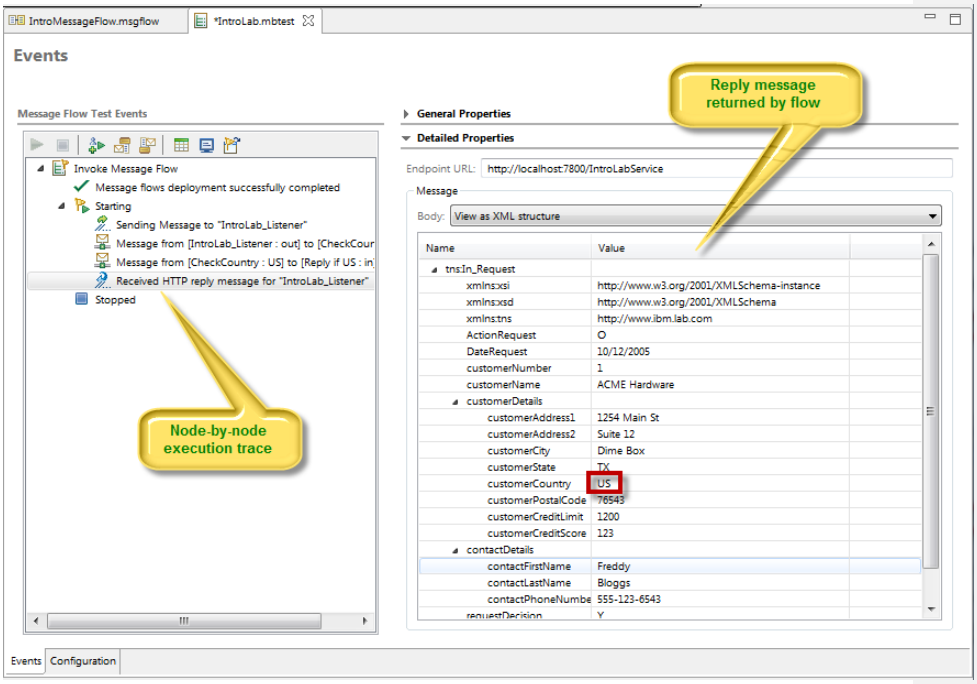
__110. Click the **IntroLab.mbtest** tab to bring the Test Client into focus.



__111. In the Test Client, you see on the left a node-by-node execution trace of the message flow.

On the right you see the response message that was returned by the flow. Because this flow does no transformation there is no difference from the input message.

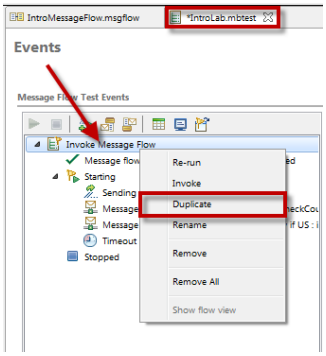
And you see the modification you made to the customerCountry, which you changed from “USA” to “US”.



__112. Next you will run the same test, but this time make a change while the debugger is running.

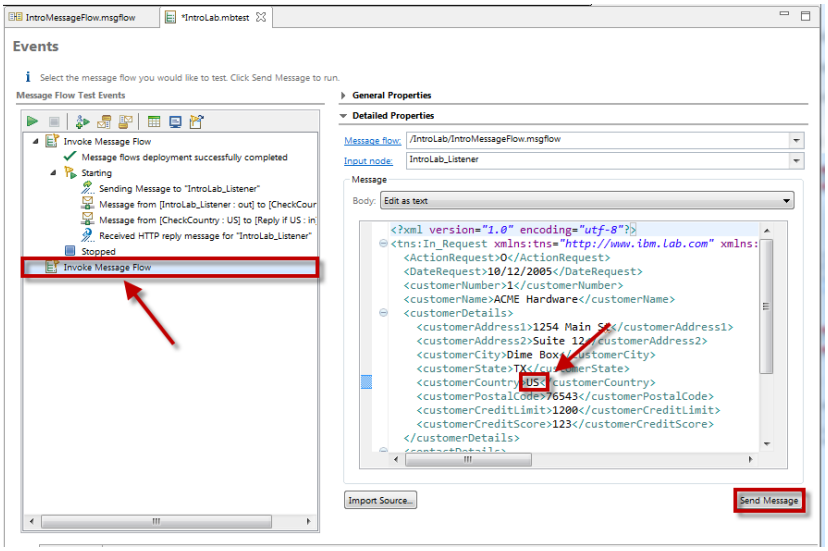
You will use this modified message (with customerCountry set to “US”) as the starting point.

In the Test Client, right-click the **Invoke Message Flow** event at the top, and select **Duplicate**.

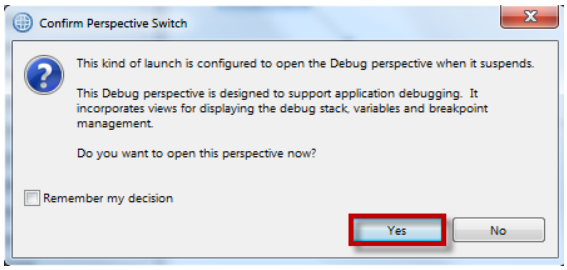


__113. A duplicate of the flow test will be created. You can see customerCountry is set to ”US”.

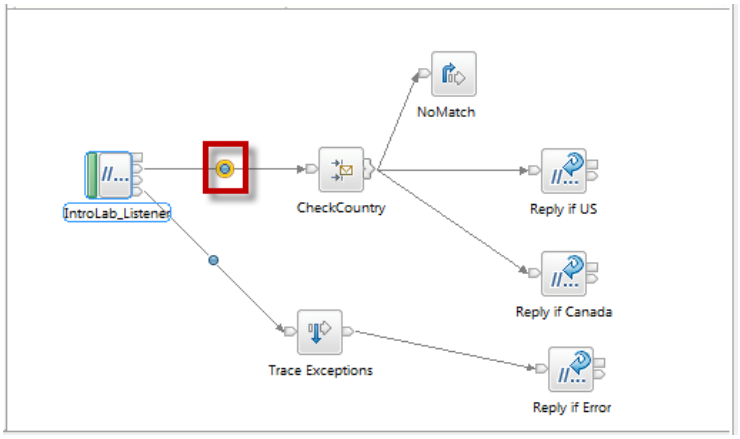
Click **Send Message**.



__114. Click **Yes** on the confirmation to launch the Debug perspective.

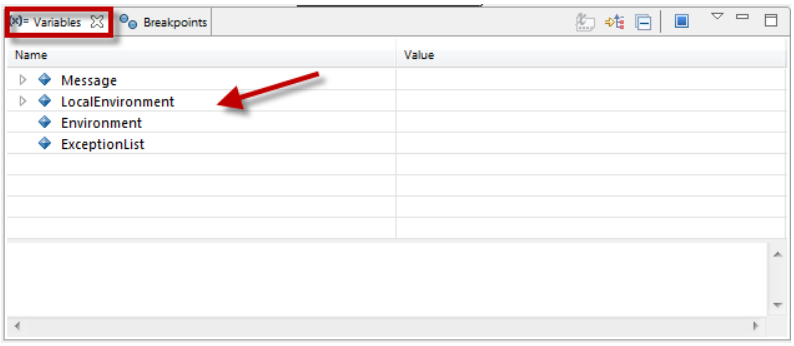


__115. As before, the debug perspective will launch and the flow will pause at the first breakpoint.



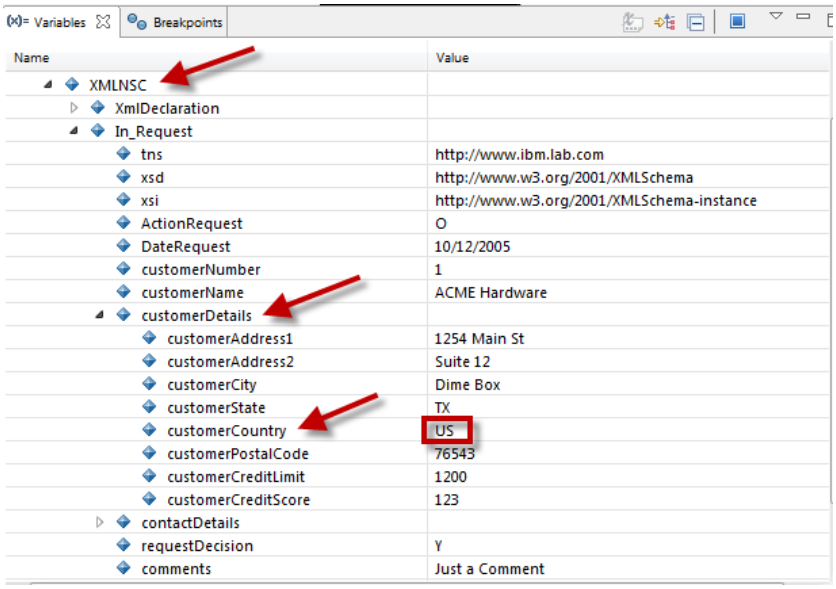
__116. Look at the upper right of the Debug perspective. Locate the **Variables** view.

Here you can see all four trees that make up the message assembly. At present two of the trees are populated: The Message and LocalEnvironment trees.



__117. Expand the message tree, and locate the customerCountry element.

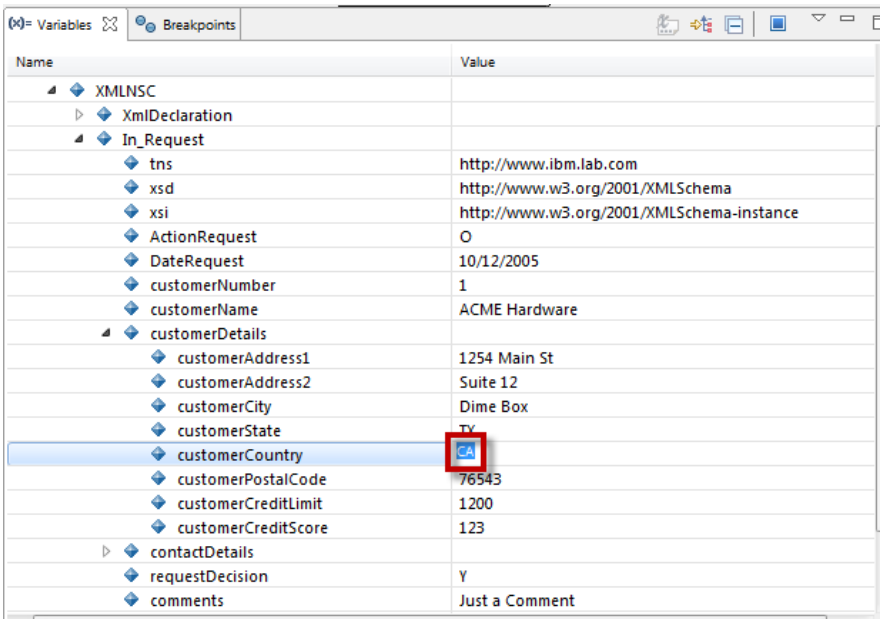
It should be set to US, as shown below.



__118. You have already tested the US path in the debugger. So you will change the customerCountry variable to “CA” and test the Canada path.

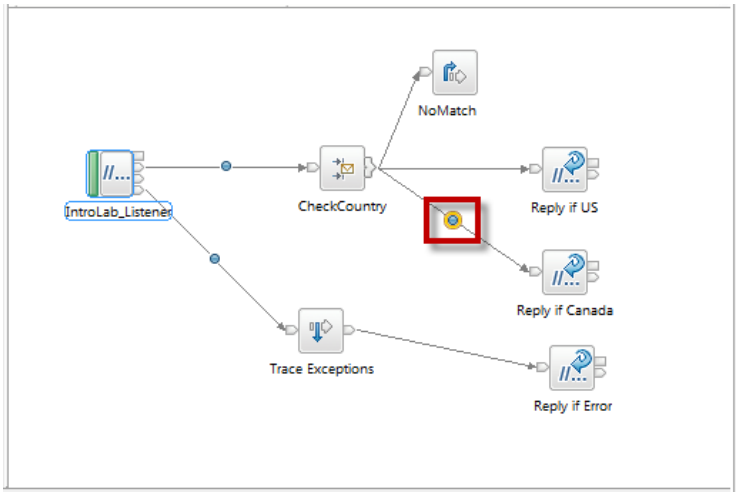
Click **US** in customerCountry, and change the value to **CA**.

Press **Enter** to commit the change.



__119. In the debug toolbar, click to step over the next node in the flow.

Because customerCountry was set to “CA”, you should see the following:

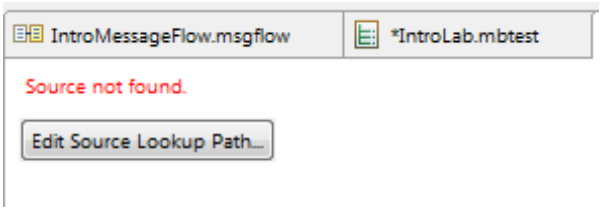


The ability to change variables in the message assembly, while the flow is executing, makes the Flow Debugger a very useful tool for exercising different behaviors the flow might take.

As an optional exercise, you might rerun these steps again, testing the flow with a customerCountry of “MX”, and observe the behavior in that case.

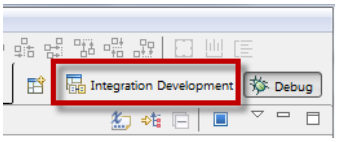
__120. Click  to run to completion.

You may see the following:

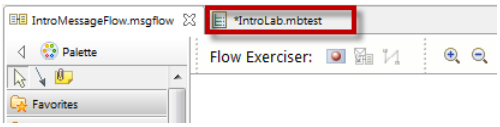


If so, Click  to resume execution.

__121. In the upper right corner of the Toolkit, click the **Integration Development** tab to return to the Integration Development perspective.

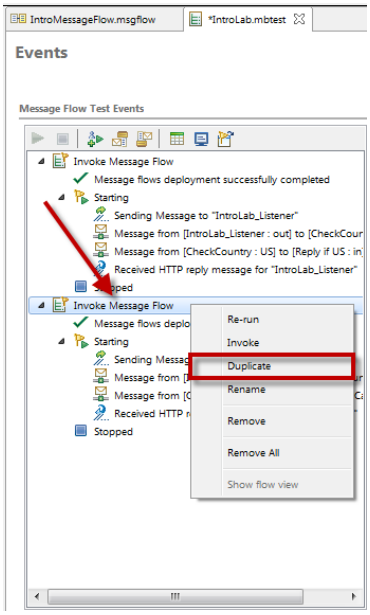


__122. Click the **IntroLab.mbtest** tab to bring the Test Client into focus.



__123. Next you will run the test that was inconclusive when using the Flow Exerciser: the test with malformed XML.

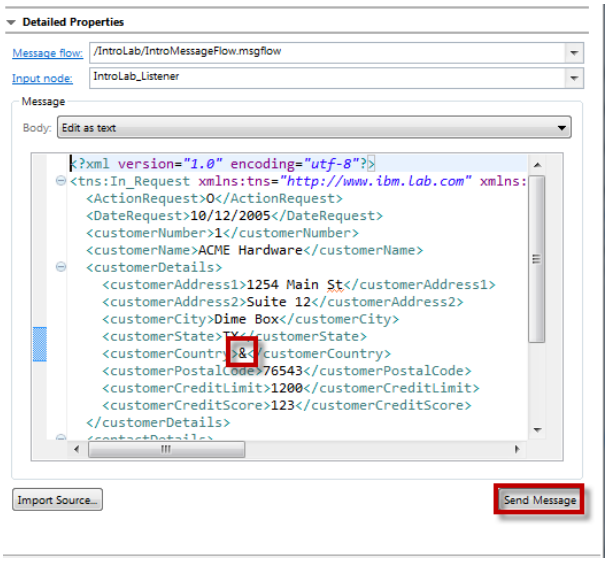
In the Test Client, right-click the second **Invoke Message Flow** event, and select **Duplicate**.



__124. A duplicate of the flow test will be created.

Modify customerCountry by replacing “US with ”&”.

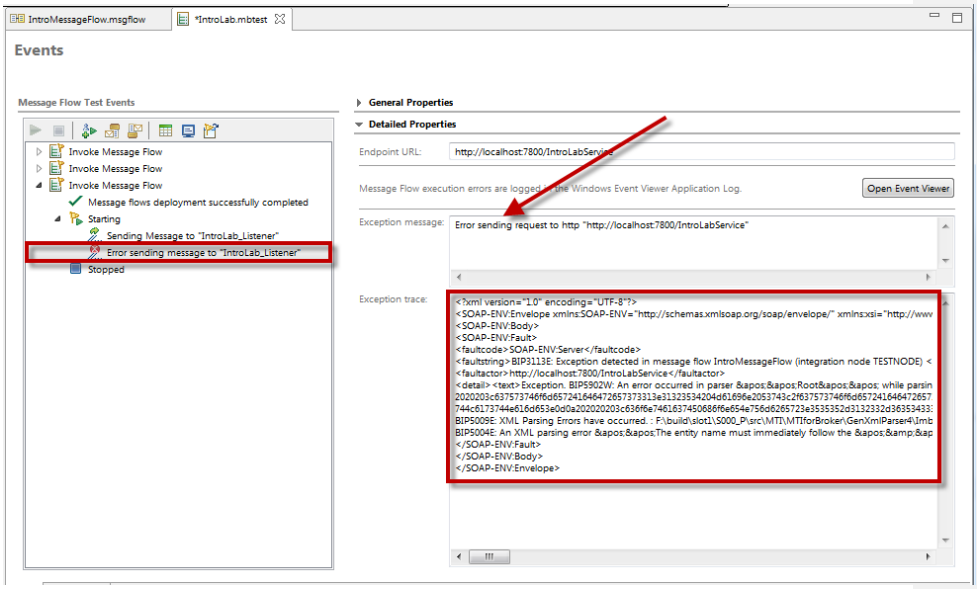
Click **Send Message**.



__125. What happened? The Debugger did not start. Remember that this is also how the Flow Exerciser behaved.

But in this case, you have an additional clue. Note the exception message below:

Error sending request to http "http://localhost:7800/IntroLabService"



Why is this clue useful?

Remember how both the flow exception handling and how the Flow Debugger work:

__a. Catching exceptions in a message flow require that the exception be thrown *downstream* from the node with the Catch terminal (either an Input node or a TryCatch node).

__b. When the Flow Debugger pauses, it is either at a breakpoint or after doing a StepOver – either way, it pauses *after* node execution completes.

What does that tell you?

That the exception must have occurred *before control left the Input node* in the message flow.

Question: Do you recall how exceptions occurring *inside* a node need to be handled?

Answer: By wiring the Failure terminal of the flow where the exception is occurring.

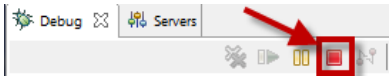
6.6 Modify flow and rerun using the Flow Exerciser

Because the exception is occurring *before control left the Input node*, rather than in a downstream node, it cannot be handled by wiring the Catch terminal of the Input node. Instead, you need to wire the Failure terminal of the node in order for your flow to handle the exception

In this section, you will return to the Flow Exerciser to see if wiring the Failure terminal of the IntroLab_Listener node will catch this exception.

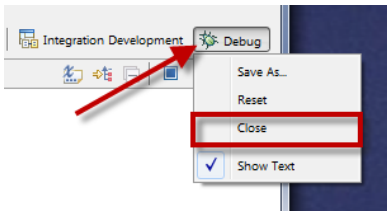
__128. Return to the Debug perspective by clicking on the Debug tab in the upper right corner.

__129. In the Debugger toolbar, click the terminate control to end the debugger

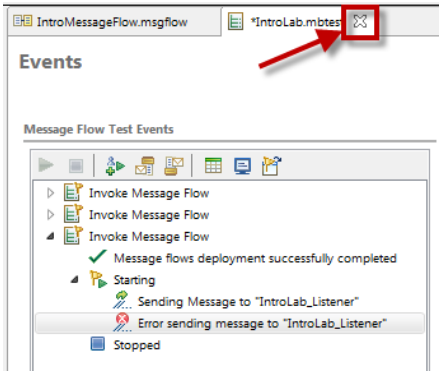


__130. In the upper right corner, right-click the Debug tab.

Select **Close** to close the Debug perspective.

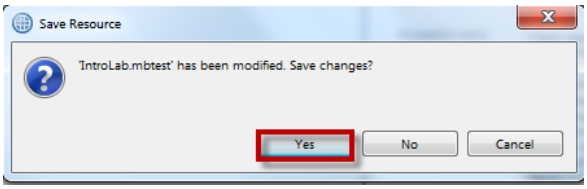


__131. Close the Test Client by clicking the ⊠ in the upper right corner of the **IntroLab.mbtest** window.



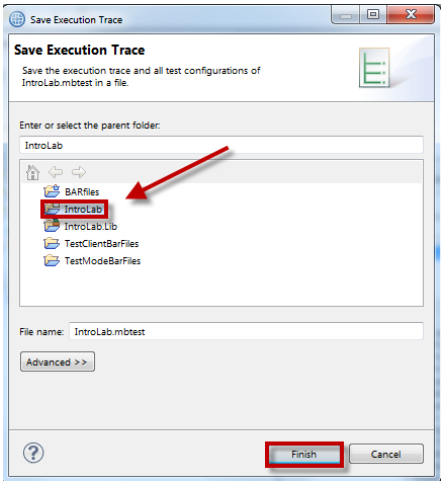
__132. Click **Yes** to save the Test Client script.

Commented [EL7]: In the original version it was unclear whether this is one click or two - please confirm this rewrite

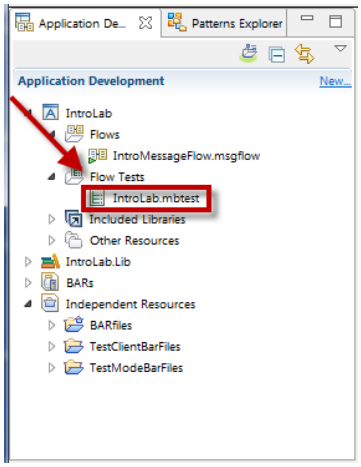


__133. Clicking Yes to the above allows you to save, and then reuse later, the execution script you created using the Test Client.

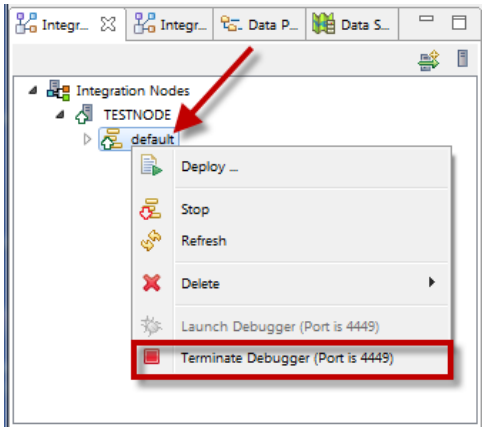
In the Save Execution Trace dialog, select **IntroLab** as the project folder to save the script, and click **Finish**.



__134. Note that the IntroLab.mbtest script has been saved in a folder called “Flow Tests” in the IntroLab application folder. This preserves the script for use again at a later time.

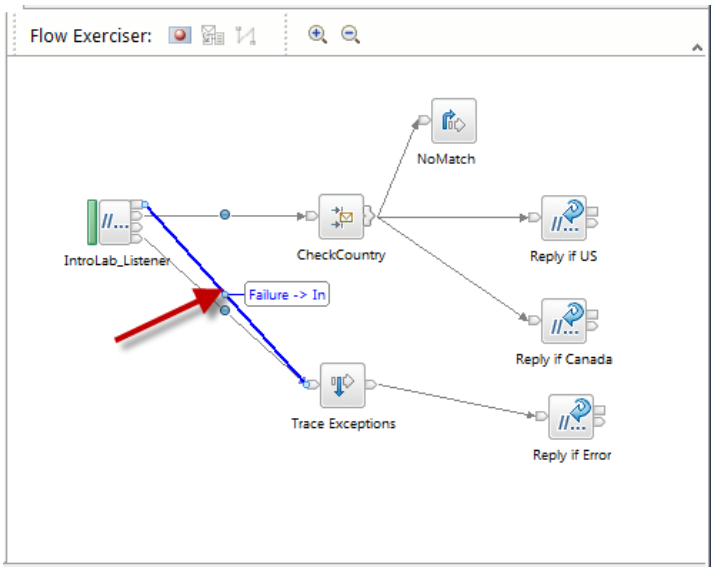


__135. In the Integration Nodes view at lower left, right-click the **myaceworkdir integration server**, and if **Terminate Debugger** is not greyed out, click it to stop the debugger.



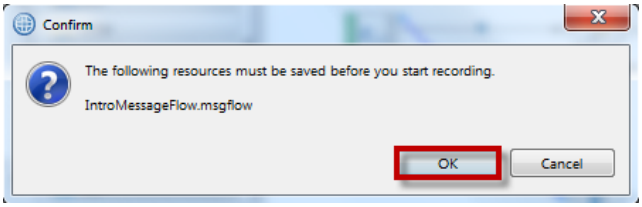
You will now return to the message flow and make the needed changes so that your flow can handle this exception.

__136. Change the message flow by wiring a connector from the Failure terminal of the **IntroLab_Listener** node to the In terminal of the **Trace Exception** node.

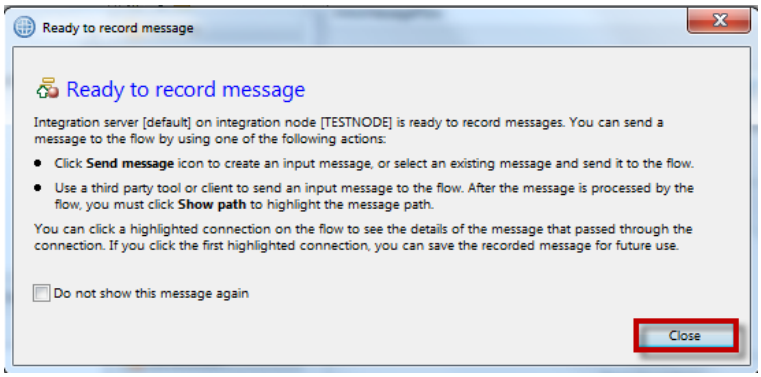


__137. On the message flow toolbar, start the Flow Exerciser.

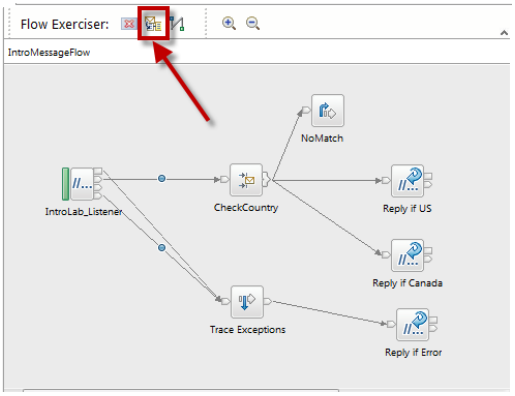
__138. If prompted to save the message flow, click **OK**.



__139. Click **Close** to dismiss the Ready to Record message popup.

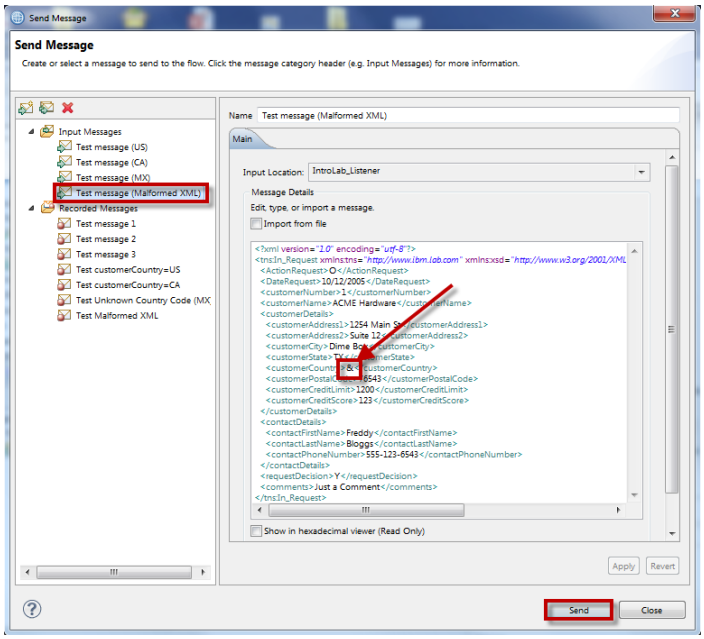


__140. On the message flow toolbar, Click **Send**.



__141. Select **Test Message (Malformed XML)**. Verify an “&” is specified for customerCountry.

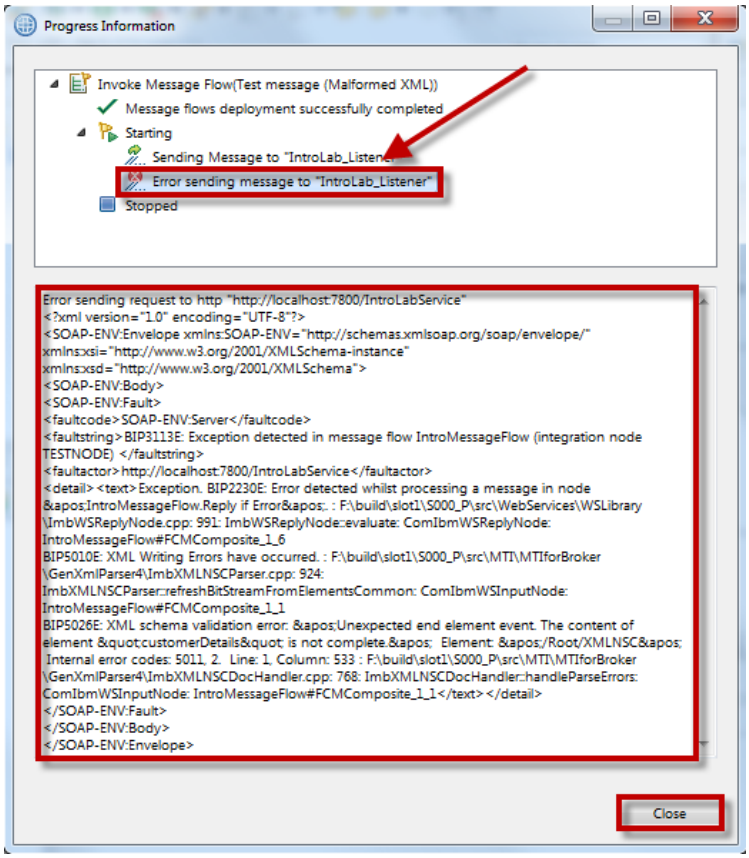
Click **Send**.



__142. Click the error.

This is the same error you encountered when testing this message previously using the Flow Exerciser.

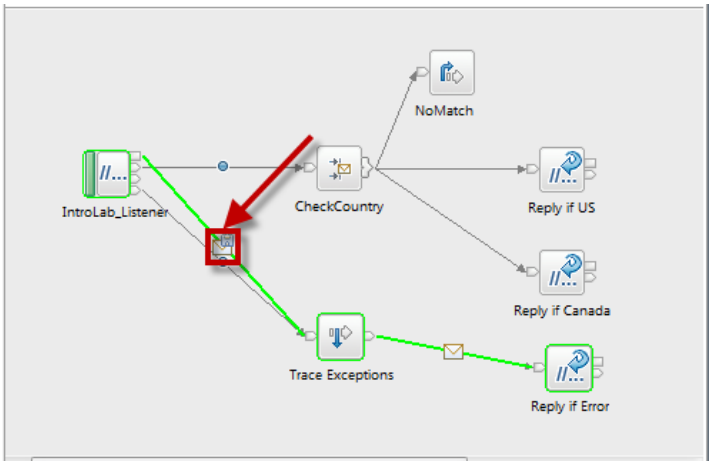
Click **Close** to dismiss.



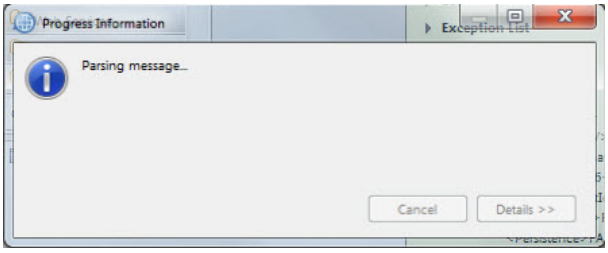
__143. You will see that, unlike your earlier test with the Flow Exerciser, this time the message flow was able to get control after the exception occurred.

Control was passed to the Failure terminal of the IntroLab_Listener node, which you wired to the exception path of the flow.

Click the **icon** to see the message tree that was passed out the Failure terminal.



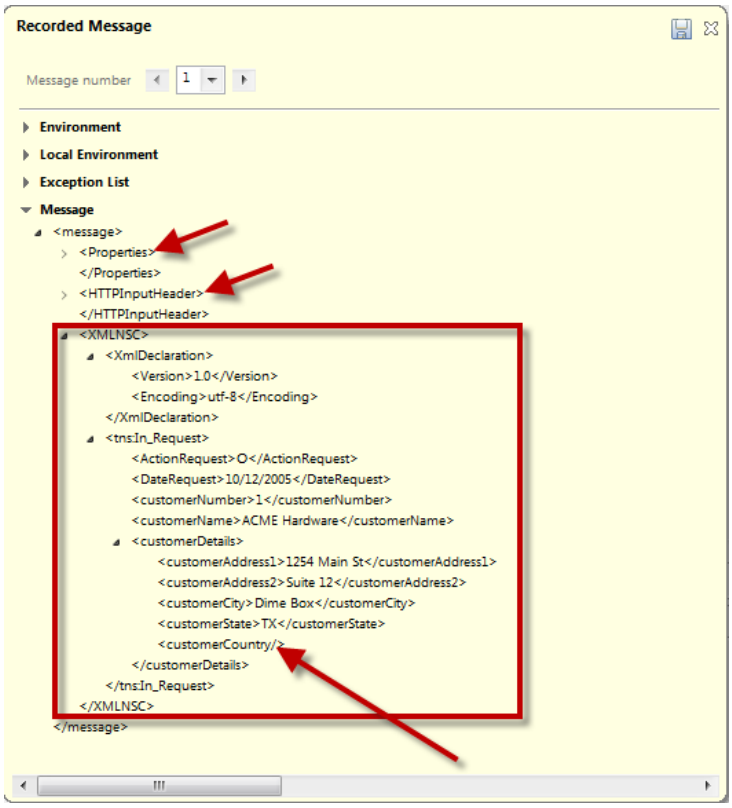
__126. The Flow Exerciser will retrieve the recorded message tree.



__144. When complete, the message tree will be displayed, with the Message portion of the tree expanded.

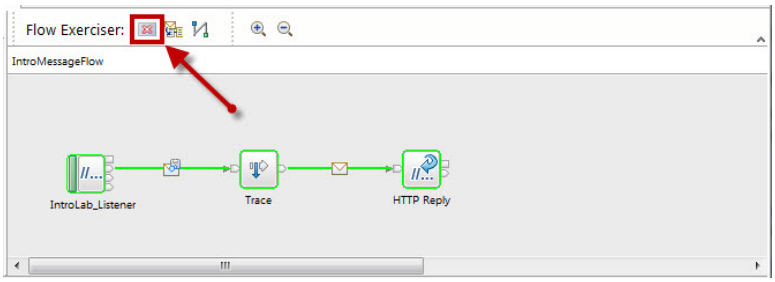
Collapse the **Properties** and **HTTPInputHeader** folders so that you can just see the payload.

You can see the payload could be parsed as far as the customerCountry element, but parsing halted at that point.

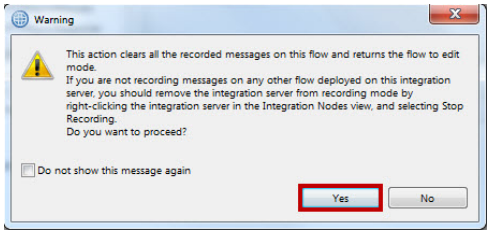


__145. Close the **Recorded Message** window.

__146. Stop the **Flow Exerciser.**



__147. Click **Yes** on the pop-up.



Why did changing Parse timing from “On-Demand” to “Immediate” alter the behavior of your message flow in this way?

Despite the fact that Parse timing is an Input node option, in many cases the input node does not actually need to know the structure of the message. So when Parse timing is set to On Demand, the message will not be parsed until it reaches a downstream node that does require parsing of the message. In the case of your flow, that would be the Route node.

Changing Parse timing to Immediate does two things:

1) Forces the message to be fully parsed, as well as validated if that option is also selected.

2) This is done immediately, *before* the message leaves the input node.

Previously, the malformed element in the message was not detected until the message reached the Route node. An exception was thrown, caught by the Input node, and routed down your exception path.

With Parse timing set to Immediate, a full parse of the message was attempted before the message was passed to the next node in the flow. The parse failed, and control was passed to the Failure terminal.

Close all the open editor tabs but leave the Toolkit running.

END OF LAB 8

IBM Software