IBM Integration Bus

A REST API processing a .jpeg image with the image provided as a MIME attachment to the JSON message

Featuring:

- REST API using MIME domain
- Message parsing using multiple domains
- Testing using the Chrome Postman utility

November 2016

Hands-on lab built at product Version 10.0.0.6

1. Introduction

This lab guide shows you how to construct a REST API in IBM Integration Bus that can process messages that contain data of different types. Specifically, the scenario has a REST API that expects a message that has a JSON part and a binary part, as follows:

Part 1 - employeeData

```
{
"EMPLOYEE": {
"EMPNO": "000003",
"FIRSTNME": "Albert",
"MIDINIT": "J",
"LASTNAME": "Einstein",
"WORKDEPT": "A00",
"PHONENO": "0101",
"HIREDATE": "1912-07-27",
"JOB": "Manager",
"EDLEVEL": 9,
"SEX": "M",
"BIRTHDATE": "1879-03-14",
"SALARY": 9990,
"BONUS": 4440,
"COMM":6660 }
}
```

Part 2 - employeeImage

Image of employee in binary format

1.1 Lab preparation

This lab is based on the solution of the REST API HR_Service. This lab uses an IIB node called TESTNODE_iibuser. You should have it already created. If not, refer to the REST API HR Service Lab Guide document.

This lab is based on a set of REST scenarios that are described in other lab guides in this series. Specifically, it may be useful to review the lab “Lab 16L02, Developing a REST API using a Swagger JSON document”.

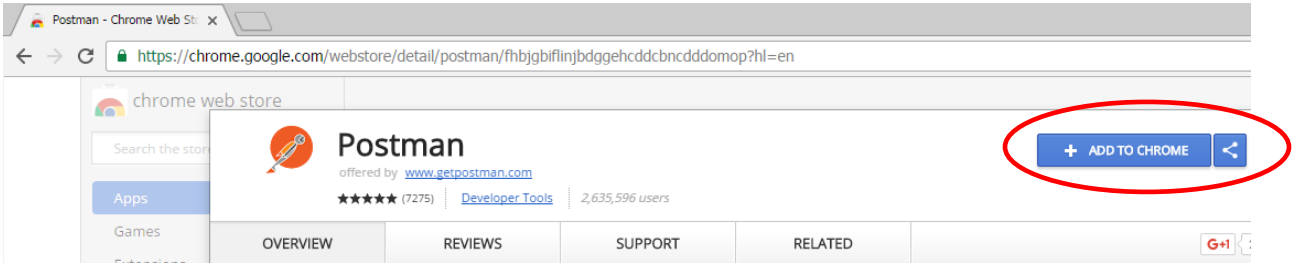
This lab does not ask you to build the solution from scratch. A complete solution is provided, and you will investigate various aspects of the solution, and perform a test of the provided solution.

1.1.1 The Chrome Postman plugin

Because this scenario needs to send a REST request with a mixed format message payload, you will need to use a test tool that is capable of generating such a request. In the development of this lab, we have used Chrome Postman for this. To obtain this tool, from a Chrome browser, search for “get Postman”. Select the “Postman Chrome Web Store” at Google, and click “Add to Chrome” to install.

Note that when it executes, this app does not actually run under Chrome; it executes as a stand-alone application.

If you are using the supplied VMWare workshop image, Postman has already been installed.



1.1.2 MQ Configuration

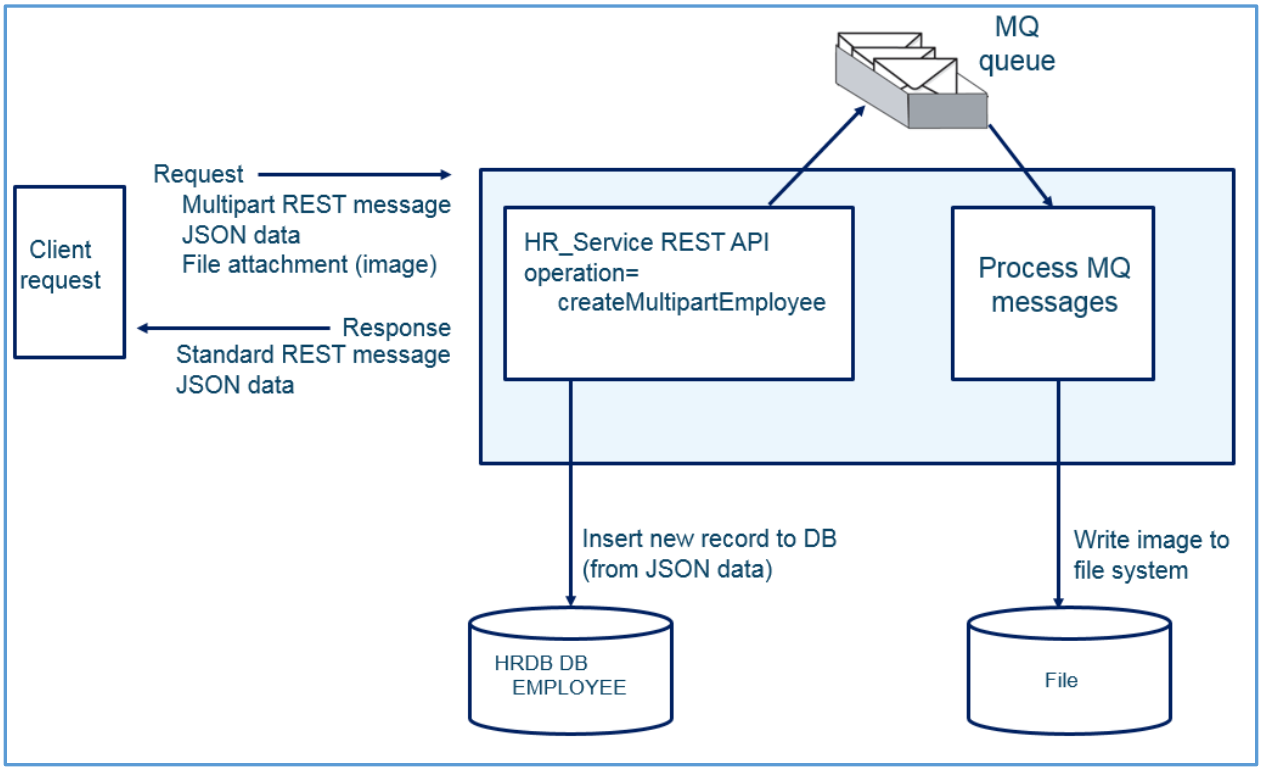
This lab uses an MQ queue to hold incoming images from the REST request prior to writing to a file.

In your queue manager (IB10QMGR on the workshop VMWare system), create an MQ queue called IMAGES.MULTIPART.OUT.

1.2 Outline of Lab

This lab provides a technique for an IIB REST API to receive a REST request that contains a multipart message payload in MIME format. The scenario was constructed to address the requirement to process a REST message that contained an image, as well as the standard JSON payload.

In this case, the REST message contains a standard JSON-formatted message, but also includes an attachment. The REST message contains “new employee” data, and the IIB REST API will take this informnation and add a new row to the EMPLOYEE table. It will also take the attached file (an image of the new employee) and write this to a local MQ queue for subsequent processing.



1.3 Configure TESTNODE_iibuser for REST APIs

The instructions in this lab guide are based on a Windows implementation, with a user named "iibuser".
The Windows VMWare image on which this lab is based is not available outside IBM, so you will need to provide your own software product installations where necessary.

Login to Windows as the user "iibuser", password = "passw0rd". (You may already be logged in).

Start the IIB Toolkit from the Start menu.

The IIB support for the REST API requires some special configuration for the IIB node and server.

If you have already done a previous lab involving the REST API function in this series of lab guides, you can skip to the next heading.

1. Ensure that TESTNODE_iibuser is started.

2. Enable Cross-Origin Resource Scripting for REST. This is required when testing with the SwaggerUI test tool. See http://www.w3.org/TR/cors/?cm_mc_uid=09173639950214518562833&cm_mc_sid_50200000=1452177651 for further information.

   (Helpful hint - the VM keyboard is set to UK English. If you cannot find the "\" with your keyboard, use "cd .." to move the a higher-level folder in a DOS window), or change the keyboard settings to reflect your locale.)

   In an IIB Integration Console (shortcut on the Start menu), run the following command.

   Note, the text should be typed on a single line - the parameters are shown on different lines here for readability; the same approach is taken throughout this and other lab documents.

```
mqsichangeproperties TESTNODE_iibuser
-e default
-o HTTPConnector
-n corsEnabled -v true
```

1.4 Configure Integration Bus node to work with DB2

If you have already done a previous lab involving the HRDB database in this series of lab guides, you can skip to the next heading.

To run this lab, the Integration Bus node must be enabled to allow a JDBC connection to the HRDB database.

1. Open an IIB Command Console (from the Start menu), and navigate to

   `c:\student10\Create_HR_database`

2. Run the command

   `3_Create_JDBC_for_HRDB`

   Accept the defaults presented in the script. This will create the required JDBC configurable service for the HRDB database.

3. Run the command

   `4_Create_HRDB_SecurityID`

4. Stop and restart the node to enable the above definitions to be activated

   `mqsistop TESTNODE_iibuser`

   `mqsistart TESTNODE_iibuser`

This will create the necessary security credentials enabling TESTNODE_iibuser to connect to the database.

Recreating the HRDB database and tables
The HRDB database, and the EMPLOYEE and DEPARTMENT tables have already been created on the supplied VMWare image. If you wish to recreate your own instance of this database, the command **`1_Create_HRDB_database.cmd`** and **`2_Create_HRDB_Tables.cmd`** are provided for this. If used in conjunction with the VM image, these commands must be run under the user "iibadmin". Appropriate database permissions are included in the scripts to GRANT access to the user iibuser.

2. Import the REST API

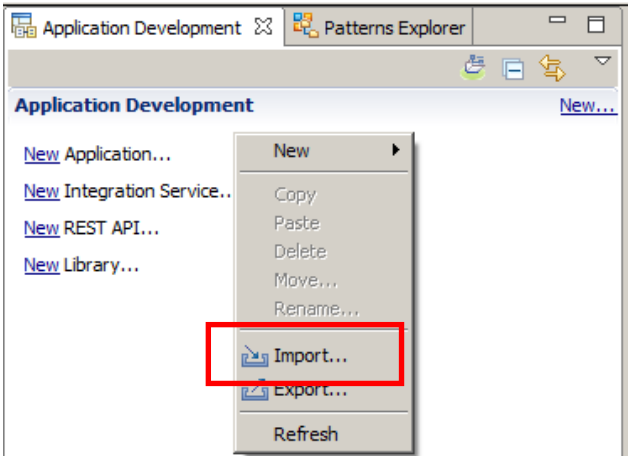
In this part of the lab exercise, you will import and deploy the provided IIB shared library and REST API.

2.1 Import the IIB REST API

1. To avoid naming clashes with earlier labs, this lab will be developed using a new workspace.

   In the Integration Toolkit, click File, Switch Workspace. Give the new workspace the name "ImageMultipart", or similar.

2. Right-click in the Application Development pane and click 'Import':



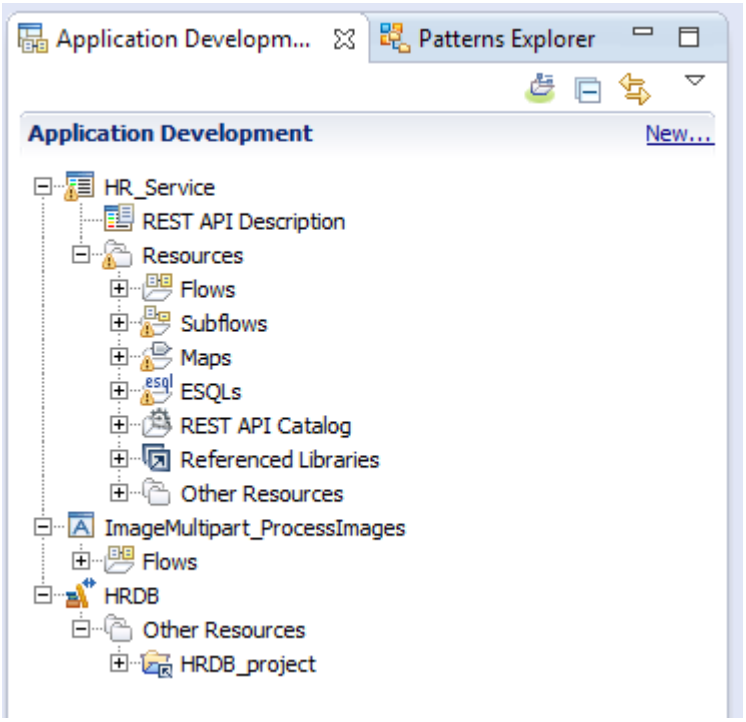
   Import the following Project Interchange (PI) file:

   C:\student10\REST_withImages\REST_Multipart\solution\HR_Service_multipart.10.0.0.6.zip

   Note: Make sure that all four projects in this PI file are selected for import. The PI includes the HRDB shared library and database project.

3. When imported, you should have in your workspace the **HR_Service** REST API and the HRDB shared library that is referenced by the REST API.

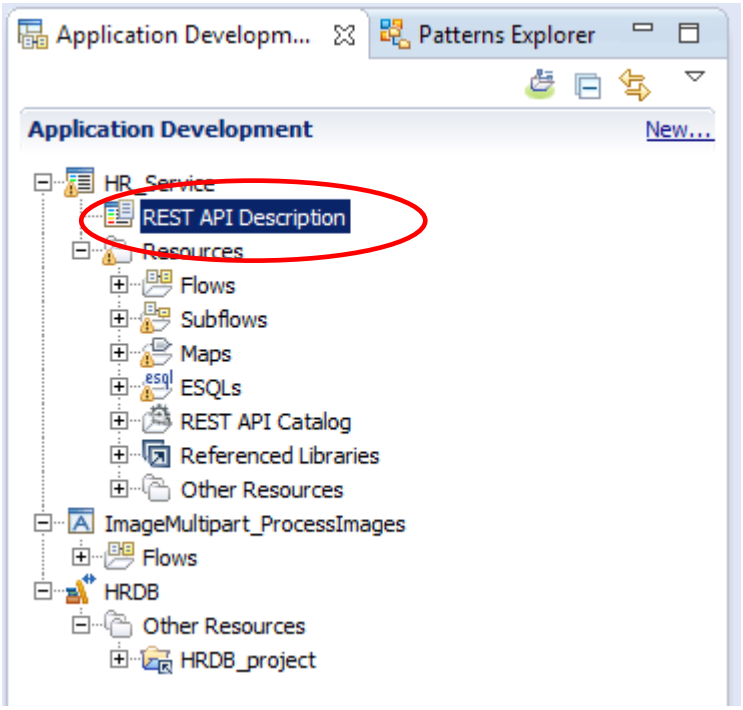
   The application **ImageMultipart_ProcessImages** is also provided to perform the subsequent processing of images using MQ. This will be described towards the end of this lab.



2.2 Explore the HR_Service REST API

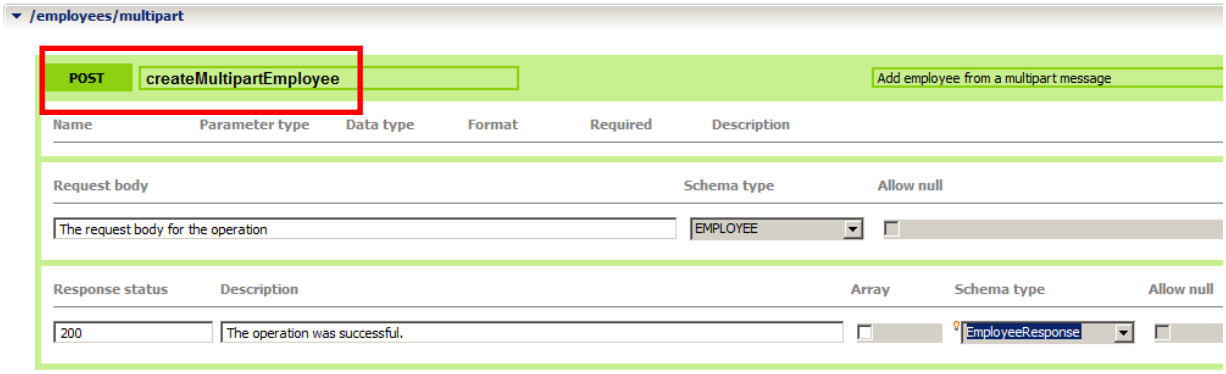
In this section, you will explore the imported REST API.

1. Expand the REST API Service and double-click “REST API Description”.

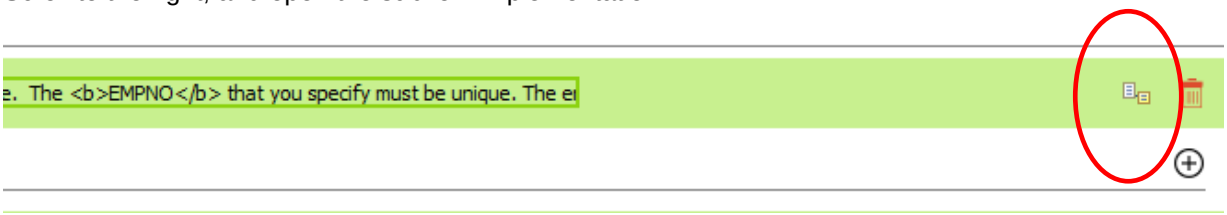


2. You will be familiar with the REST API Description, Header and Resources from earlier lab scenarios, so proceed to the Resources section, and focus on the **createMultipartEmployee** POST operation in the /**employees/multipart** resource.

   Note that the Request body has a schema type of EMPLOYEE, and the Response body has a schema type of EmployeeResponse.



3. Scroll to the right, and open the subflow implementation.



4. This opens the implementation subflow. Each node will be investigated in detail over the next few pages.

   At a high level, the subflow performs the following actions:
   1. Because the application is a REST API, the default parser has been set to JSON. Therefore, initially, the subflow will not be able to properly parse the incoming message (which is a MIME multipart message). To handle this, the first processing node in the subflow is a “Reset Content Descriptor” node, which will reparse the message using the MIME domain. When this happens, the multipart message is parsed into its two constituent parts, Part1 (the JSON data, but still held in binary format at this stage) and Part2 (the binary image).

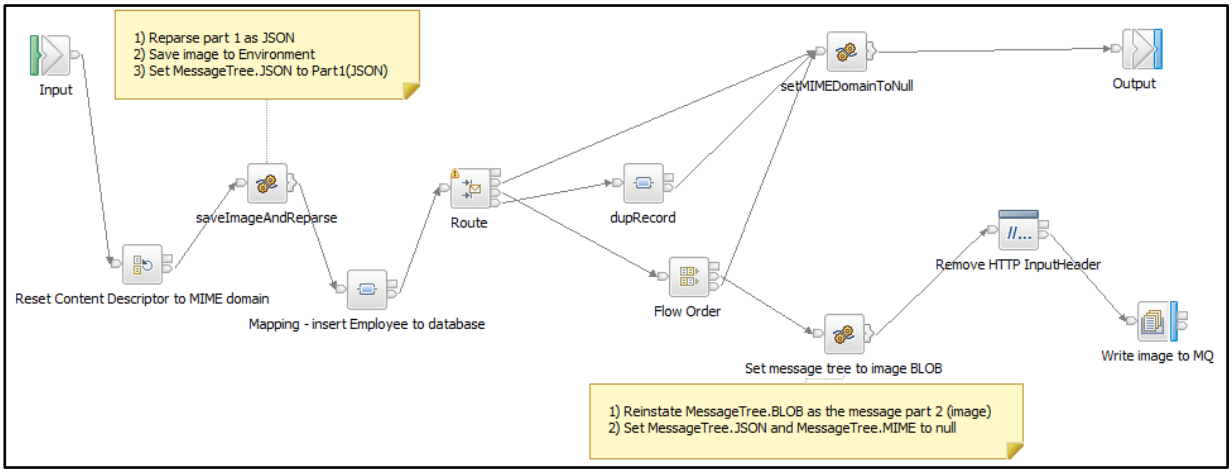
   2. The saveImageAndReparse node is an ESQL Compute node which further processes these two parts.
      - Part 2 (the image) is stored in the IIB Environment tree.
      - Part 1 is reparsed using the JSON parser, and the resulting parsed data is stored in the message tree, now in JSON format.

   3. The first mapping node inserts the new employee row into the HRDB database, using the JSON data from the message tree as input.

   4. The Route node tests whether the insert was successful.
      - If the insert was successful (rows inserted not = 0), then control passes to the FlowOrder node.
      - If the insert resulted in a duplicate key response, control passes to the dupRecord map node.
      - Otherwise, control passes directly to the setMIMEDomainToNull Compute node.

   5. The FlowOrder node will:
      1. Send the message tree to the “Set message tree to image BLOB” compute node for processing of the binary image part.
      2. Send a response to the client (userReturnCode=0 will indicate success to the client).

   6. The “Set message tree to image BLOB” compute node retrieves the binary image from the Environment tree, and reinstates it in the main message tree, from where it is subsequently written to a holding MQ queue, for later processing. This approach was taken because it allows the employee image to be stored transactionally (ie. to MQ), before being written to a file.



2.3 Explore the REST API in detail

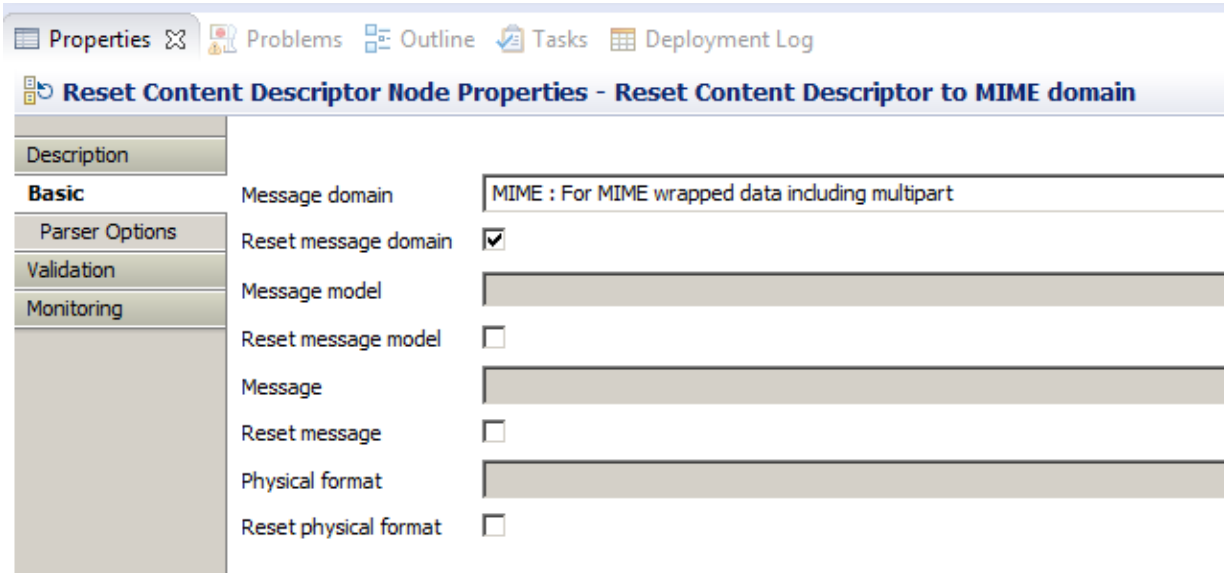
1. Select (click on) the Reset Content Descriptor node, and highlight the node Properties.

   On the Basic tab, the following properties have been set:

   - Message domain = MIME: for MIME wrapped data including multipart
   - Reset message domain = ticked

   This means that the incoming message, which is initially in MIME format, will be reparsed as a MIME multipart message (the default parser for a REST API is JSON). When this is done, the message tree will consist of several message parts, as many parts as exist in the incoming message. In this example, we will see two message parts.

   Note that subsequent nodes in the flow can address different parts of the message only by using the MIME parser, unless the message is again reparsed.



2. Open the ESQL Compute node **saveImageAndReparse**.

   At this point in the flow, the message tree has been parsed by the MIME parser and split into MIME "Parts". Each part is referenced by the element name "Part[n]", so is referenced like this:

   OutputRoot.MIME.Parts.Part[n]

   The raw data (BLOB) is referenced by the value
   OutputRoot.MIME.Data.BLOB.BLOB

   So, the following statement adds a new element called "Data", under the "Part[1]" element, and populates the contents of "Data" by using the PARSE option with the input as shown here. Note that 546 represents the Encoding of the data and 1208 represents the CCSID.

```
CREATE LASTCHILD OF OutputRoot.MIME.Parts.Part[1].Data
     DOMAIN('JSON')
     PARSE(OutputRoot.MIME.Parts.Part[1].Data.BLOB.BLOB, 546, 1208);
```

   The next statement sets the "Data.BLOB" portion of the Part[1] output message to Null. This is required because we no longer need the BLOB form of the message.

```
SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;
```

   This statement saves the entire contents of Part[2] to the Environment tree (in a folder called Variables).

```
set Environment.Variables.Image = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;
```

   And finally this statement constructs the new contents of the main message tree, in JSON format, setting it to the contents of the parsed Part[1] data.

```
set OutputRoot.JSON.Data = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.*;
```

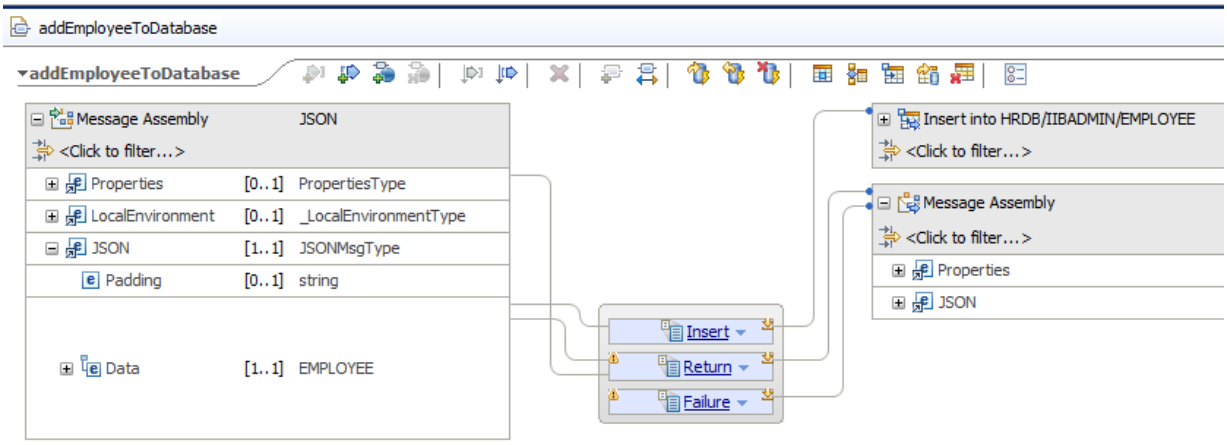
   Close the Compute node.

3. Open the map “Mapping – insert Employee to database”.

   This map has an input assembly in JSON format, which contains a full Employee record. The map inserts the EMPLOYEE data into the HRDB/EMPLOYEE table.

   Failure scenarios such as “duplicate record” are handled by saving database returned data such as SQLSTATE in the output message.

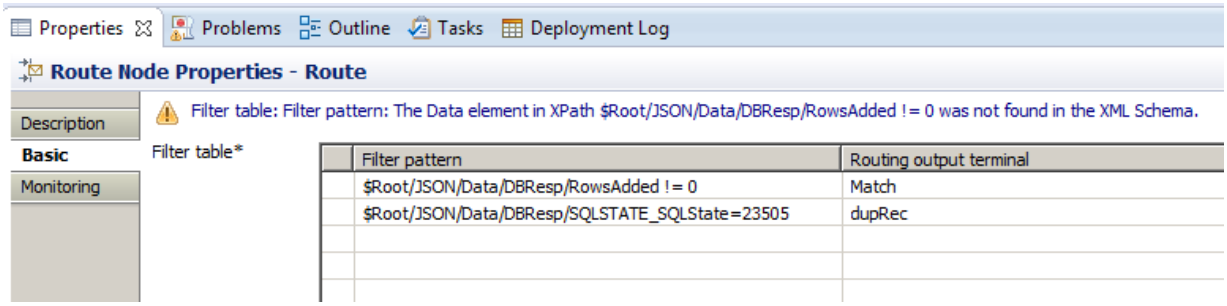
   Close the map.



4. Highlight the Route node and review the node properties.

   - The Match terminal is used when the database insert was successful (when the number of rows added was not zero).
   - The dupRec terminal is used when the SQLSTATE value is 23505 (SQL duplicate row). You can extend the flow yourself if you want to check for other specific returns.

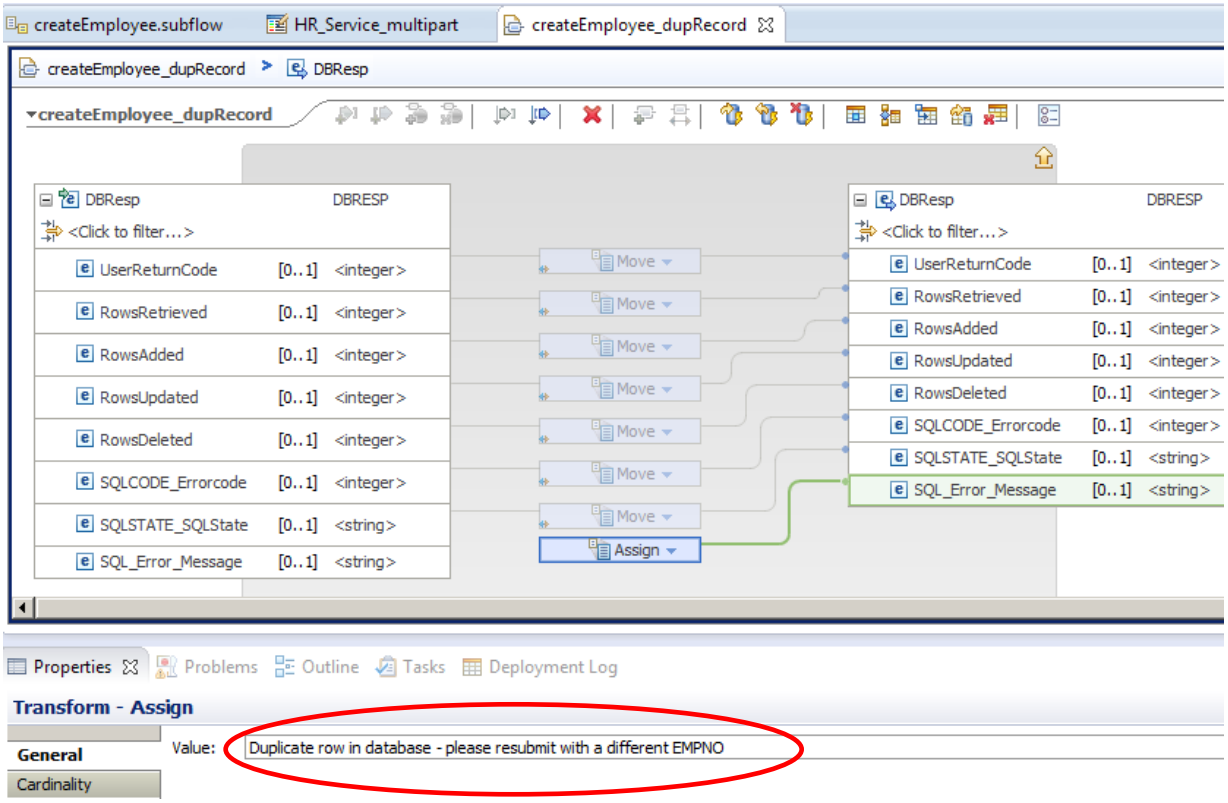
   You will see a message suggesting that the Data element in the Filter pattern was not found in the XML schema. This is because the XPath Expression builder does not support the JSON form of messages, so XPath evaluation expressions (including filter patterns) must be manually built.



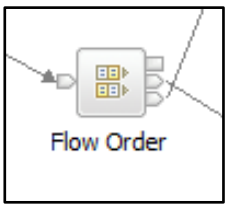
5. Open the dupRecord record map node. Click “Local Map” in the main map editor.

   Review the various mappings that are provided. In particular, the SQL_Error_Message is set with an Assign transform, setting the message to a more readable form of the SQL error message.

   Close the map.



6. The Flow Order node first invokes the processing of the image contained in the original request, and then sends a response back to the originating client.



7. The **setMIMEDomainToNull** Compute node simply sets the MIME part of the output message to null, prior to returning a response to the invoking client.

   Note that if the database insert is successful, the response message is simply the output of the map that inserts the data, ie. the EmployeeResponse message.

```
SET OutputRoot = InputRoot;
set OutputRoot.MIME = NULL;
```

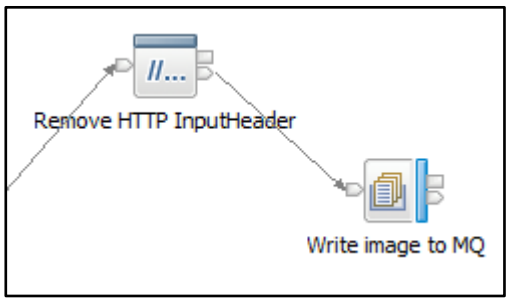
8. Open the **setMessageTreeToImageBLOB** compute node.

   This node constructs a new output message, OutputRoot.BLOB.BLOB, which contains just the image part of the main input message.

   It then sets both the JSON and MIME parts of the output message to null.

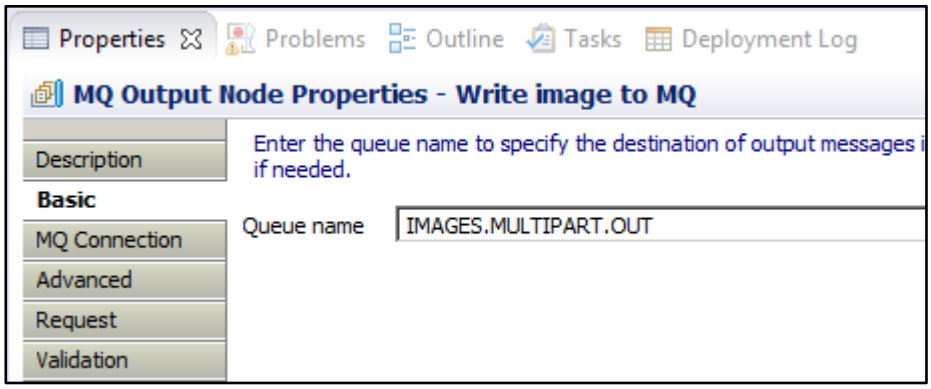
```
CALL CopyEntireMessage();
--  Note - the following line can be used if required, because the FlowOrder
--  node propagates the OutputRoot unchanged.
--     set OutputRoot.BLOB.BLOB = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;

--  However, for consistency with the earlier part of this subflow, we will
--  retrieve the binary image from the Environment tree.
set OutputRoot.BLOB.BLOB = Environment.Variables.Image;

--  Now set both the JSON and MIME parts of OutputRoot to null, so that the
--  OutputRoot.BLOB message can be written to MQ
set OutputRoot.JSON = NULL;
set OutputRoot.MIME = NULL;
RETURN TRUE;
```

9. The remainder of the flow constructs the output MQ message. The HTTP InputHeader is removed from the message, and an MQ Output node is provided to write the message to a local queue.



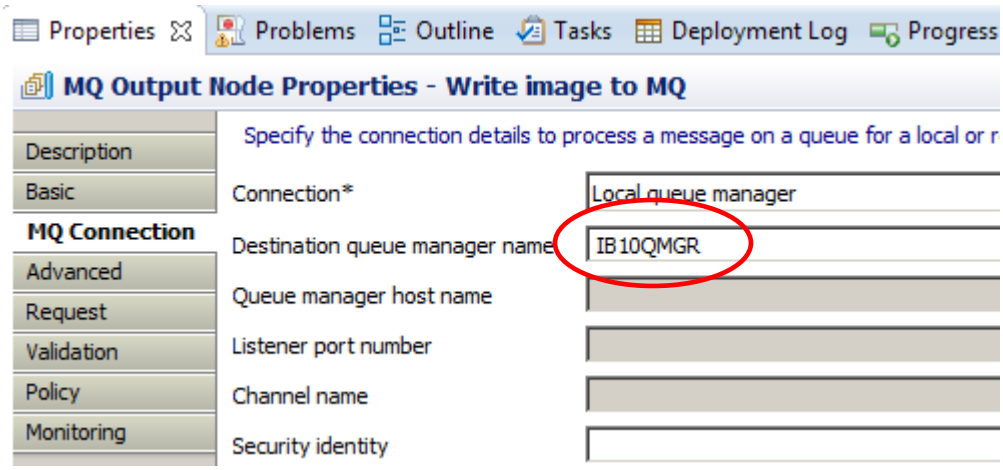
10. Click the MQOutput node to highlight its properties.

    On the Basic tab, note that the Queue Name has been set to **IMAGES.MULTIPART.OUT**. If you are running this lab on your own system, you will need to define this queue on your nominated MQ queue manager.



11. On the MQ Connection tab, note that the Destination queue manager name has been set to IB10QMGR. As in the previous step, if you are running this lab on your own system, you will need to change this property to the name of your own queue manager.

    If required on your own system, you may also need to change the Connection type to a value other than "Local queue manager".

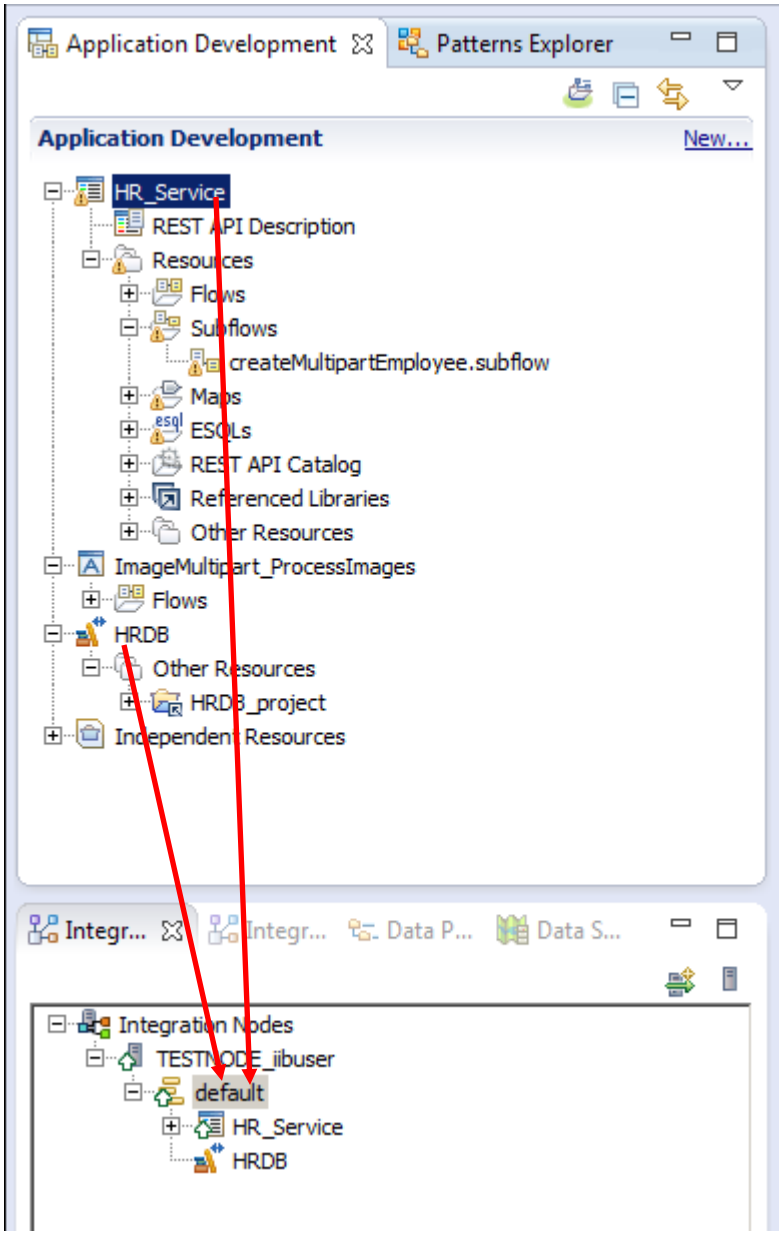


3. Deploy and Test the REST API

3.1 Deploy

1. Deploy the following resources:
   - HRDB shared library
   - HR_Service

   Do not deploy the ImageMultipart_ProcessImages application at this time.



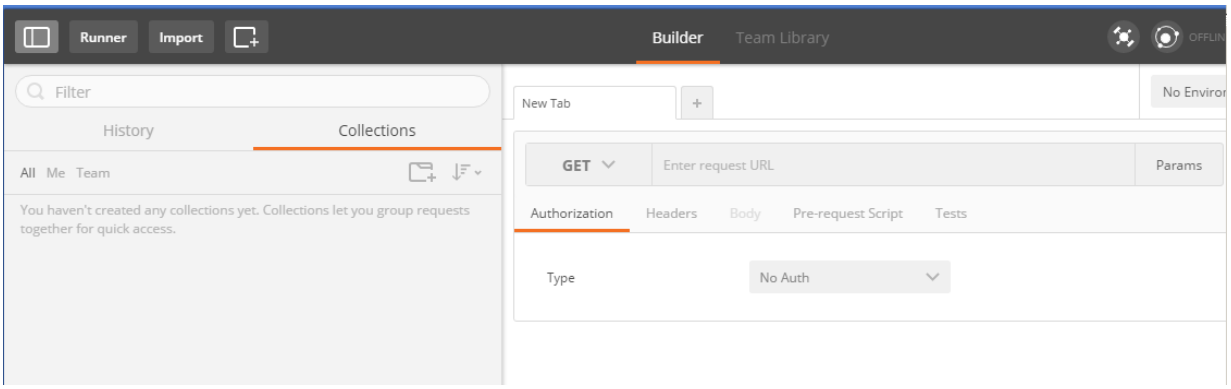
3.2 Test

3.2.1 Start Postman test tool

1. From the Start menu, or from your installation folder, start the Postman tool (in the workshop VM system, type Postman into the Start Search menu).

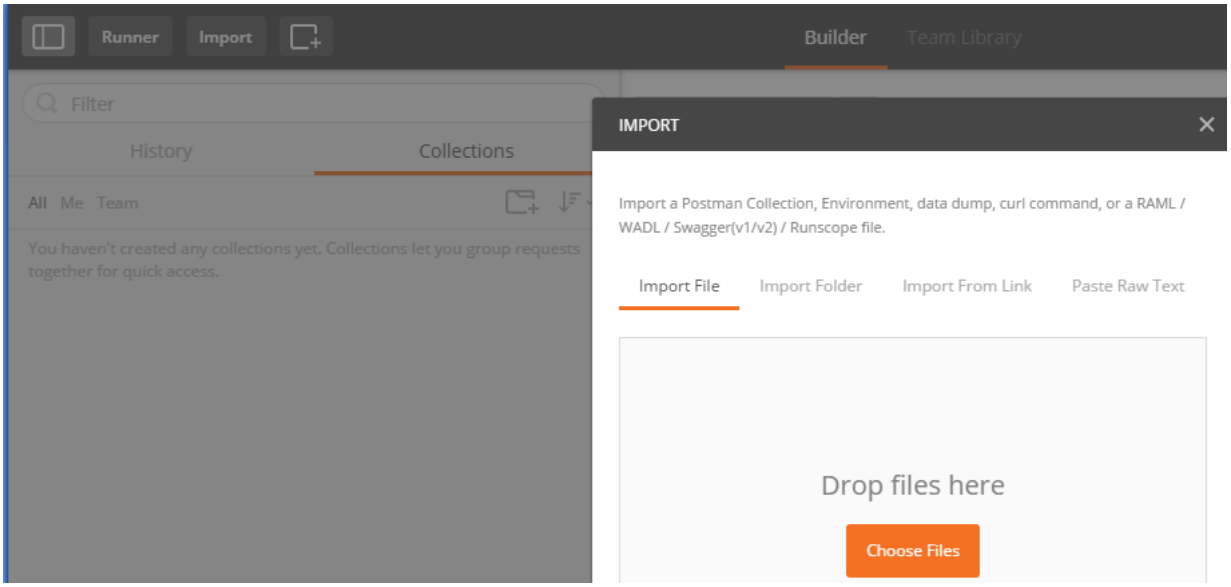
   After the progress message…

   … you will see the Postman main menu.

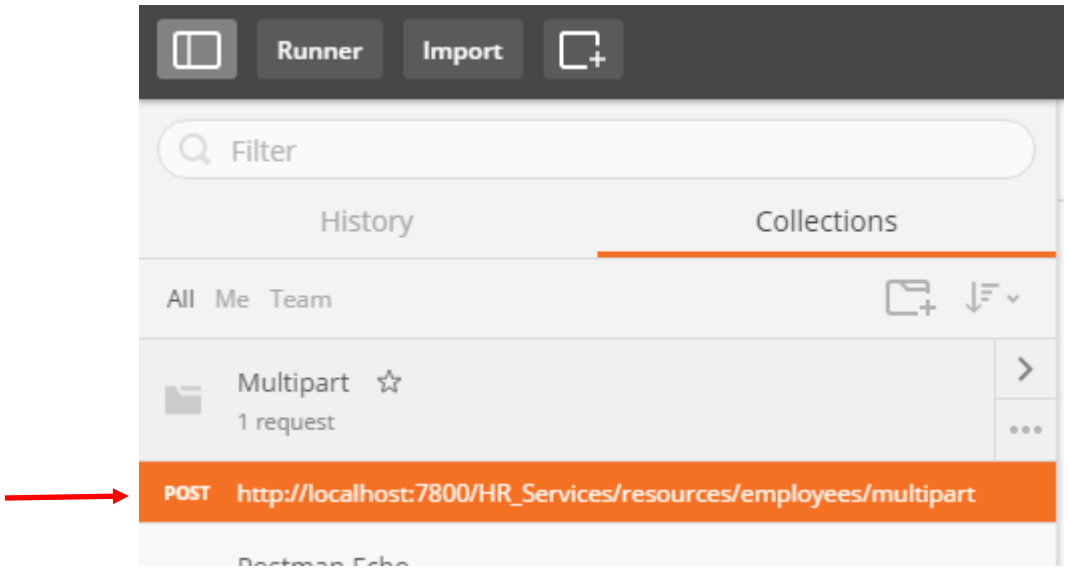


2. Import the Postman project for the multipart lab. Click Import and import the file :

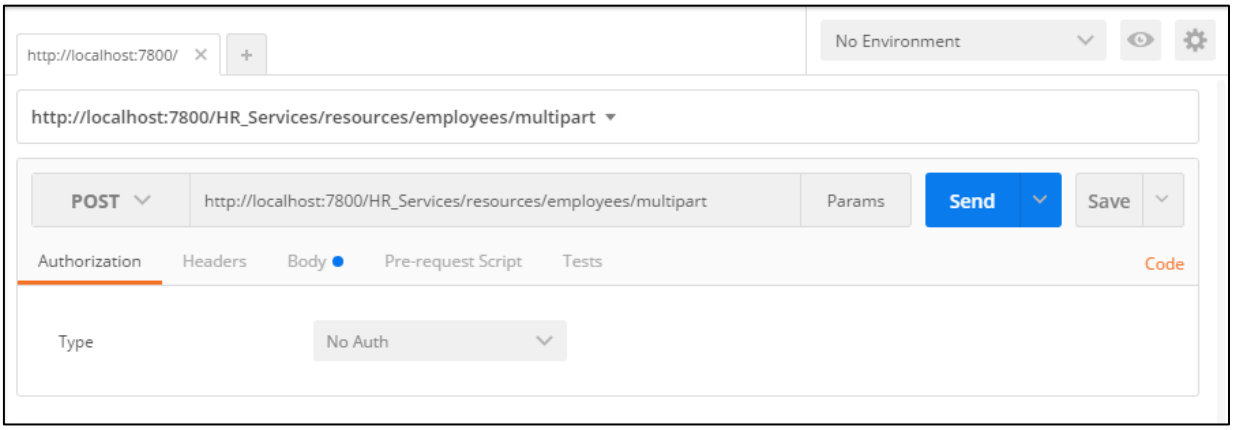
```
c:\student10\REST_withImages\REST_Multipart\postman\
Multipart.postman_collection.json
```



3. Highlight the line with the POST request on the Collections tab.



4. On the right pane, note that the URL is set to the required URL for the imported REST API. If your hostname or port are different, you will need to make appropriate changes to these values.



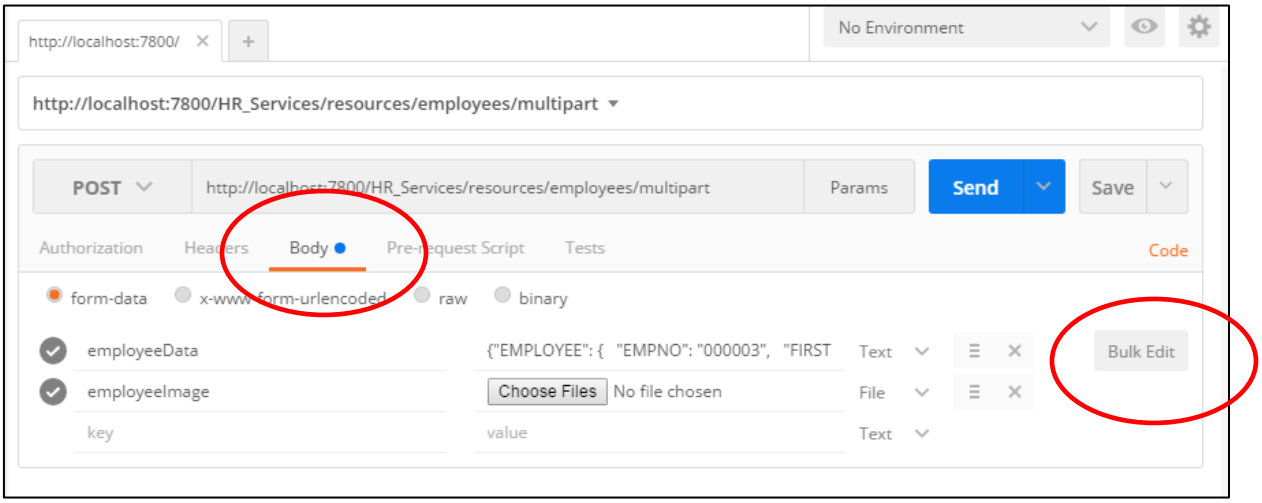
5. Click the “Body” tab.

   Note the format of the message is “form-data”. Selecting this option means that you can construct the payload of the message using a multipart format.

   Note that the message has two parts:

   - employeeData – the JSON part of the multipart message, with a message payload representing a new EMPLOYEE.
   - employeeImage – the binary part of the multipart message. In this example, you will attach a jpg image of the new employee.

   Note that the names employeeData and employeeImage do not need to match any part of the message elements sent to the REST API.



6. Click “Bulk Edit” (above) to see the input message JSON data in its entirety.

   Click Key-Value-Edit to return to the earlier display.

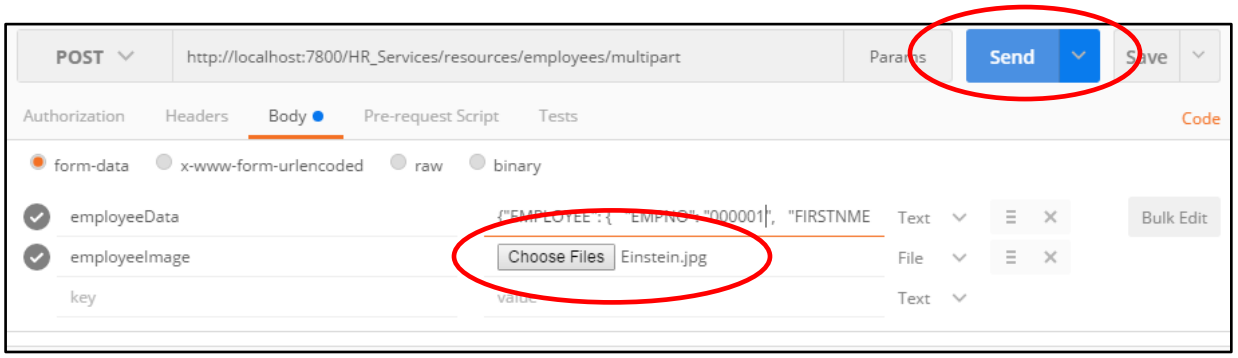
3.2.2 Test with Postman

1. Specify the name of the employeeImage file. Using the Choose Files button, set this to

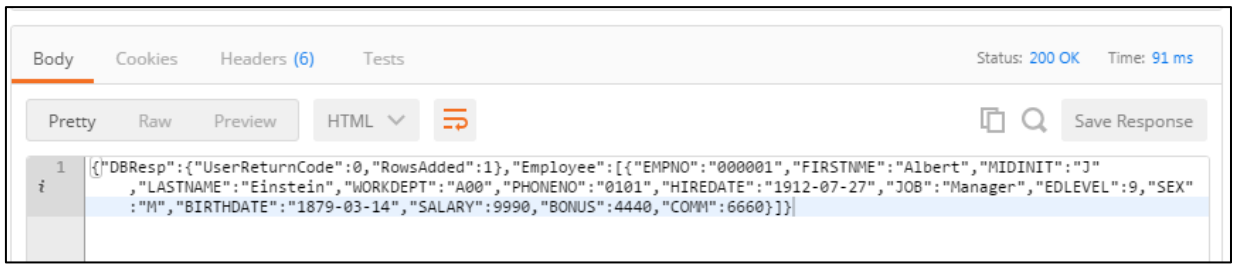
   `c:\student10\REST_withImages\REST_Multipart\data\Einstein.jpg`

   Set the employee number (EMPNO) to one that is known not to already exist (eg. 000001).

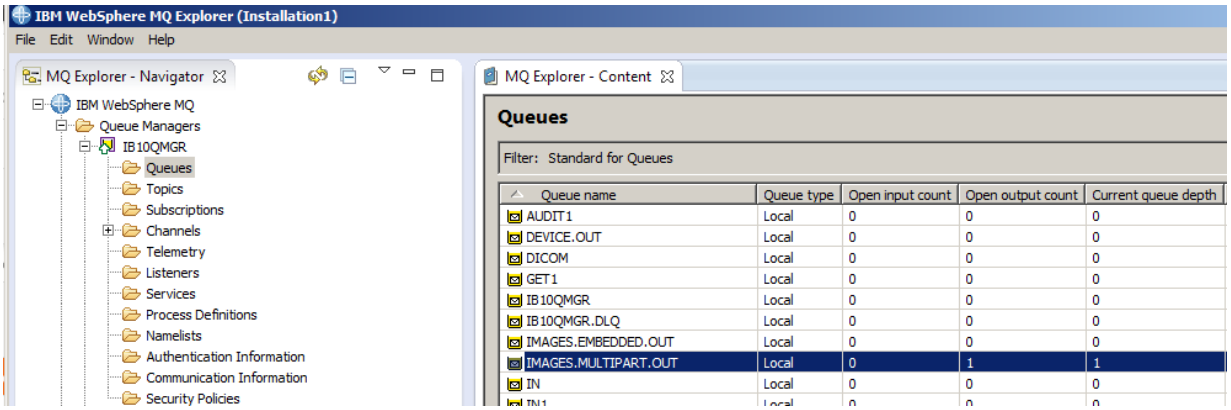
   Click Send.



2. The EmployeeResponse message will be returned. This contains the Employee data sent with the original request, along with userReturnCode=0, indicating that the data was successfully added to the database.



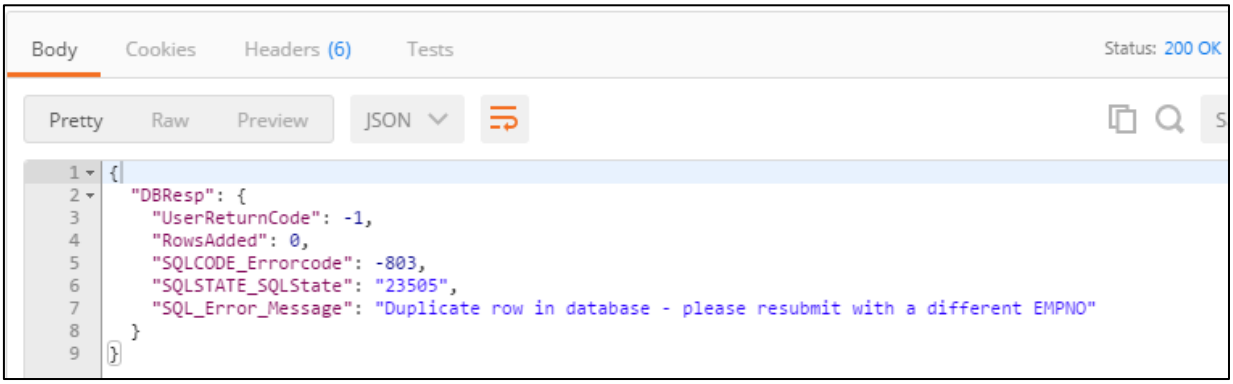
3. Using MQ Explorer, or your MQ tool of choice, observe that the queue IMAGES.MULTIPART.OUT has one message.



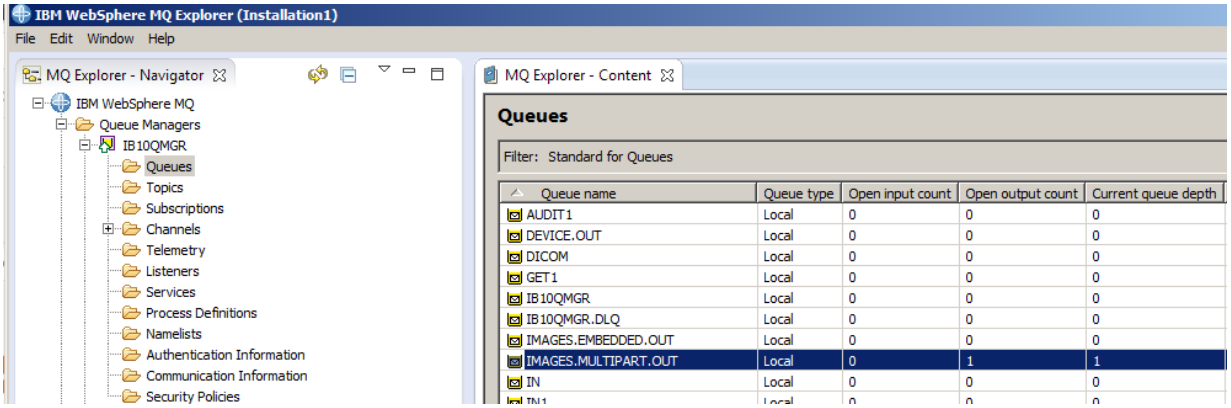
4. Return to the Postman tool, and click Send again, without changing the value of EMPNO.

   Note that this time, the response shows that you are trying to add a duplicate row to the database, and this has been rejected.

   Note – this response is only received if the database EMPLOYEE table definition has defined the EMPNO column as UNIQUE, or has defined EMPNO as a primary key. The HRDB table definition DDL (in c:student10\Create_HR_database\Create_HRDB_Tables1.ddl) has been defined with a primary key in this way.



5. Back in MQ Explorer, double-check that the number of MQ messages waiting to be processed is still 1. This means that the incoming message (attached image) has not been written to the queue manager.

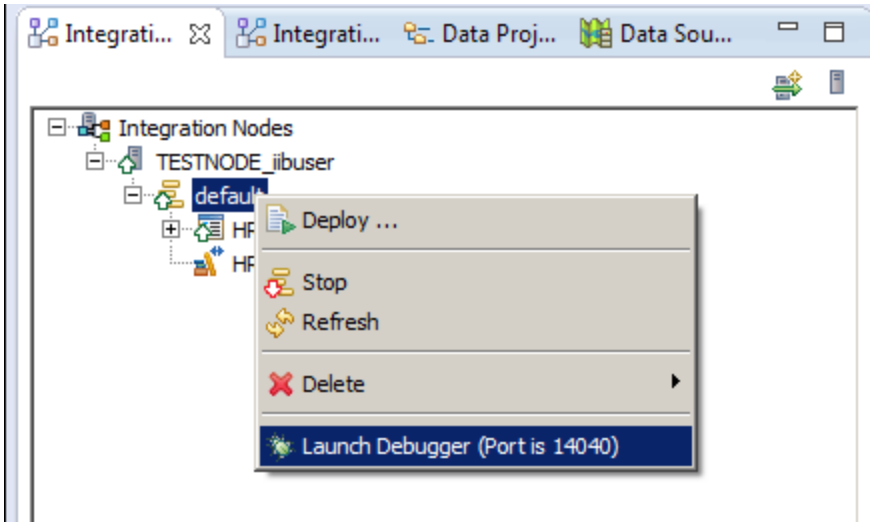


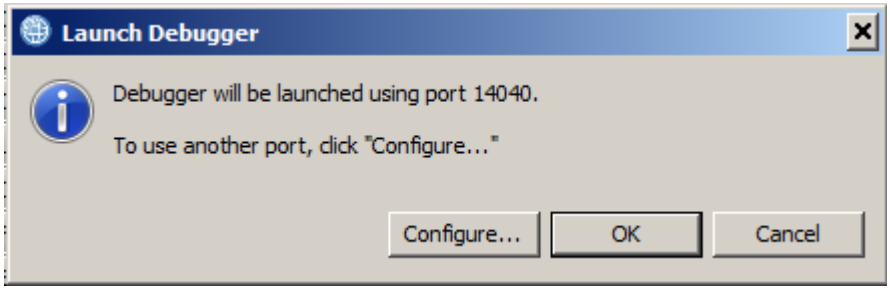
3.3 Investigate in more detail using debug mode

In this section, you will perform the test again, whilst having the IIB REST API in debug mode. Using this tool, you will see the message tree at various stages in the flow, and see the multipart, MIME and JSON messages as they are manipulated by the IIB flow.

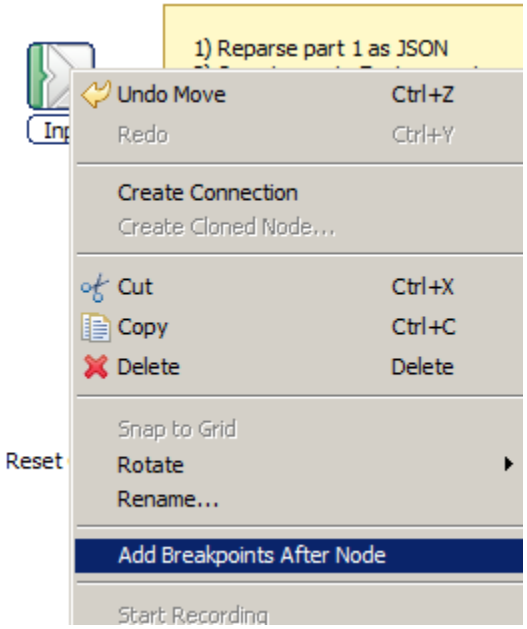
1. First, activate the debugger on your IIB node/server.

   Right-click the IIB server, and select Launch Debugger. If you have not already configured a port for the debugger to use, use the configure button to specify a suitable port.

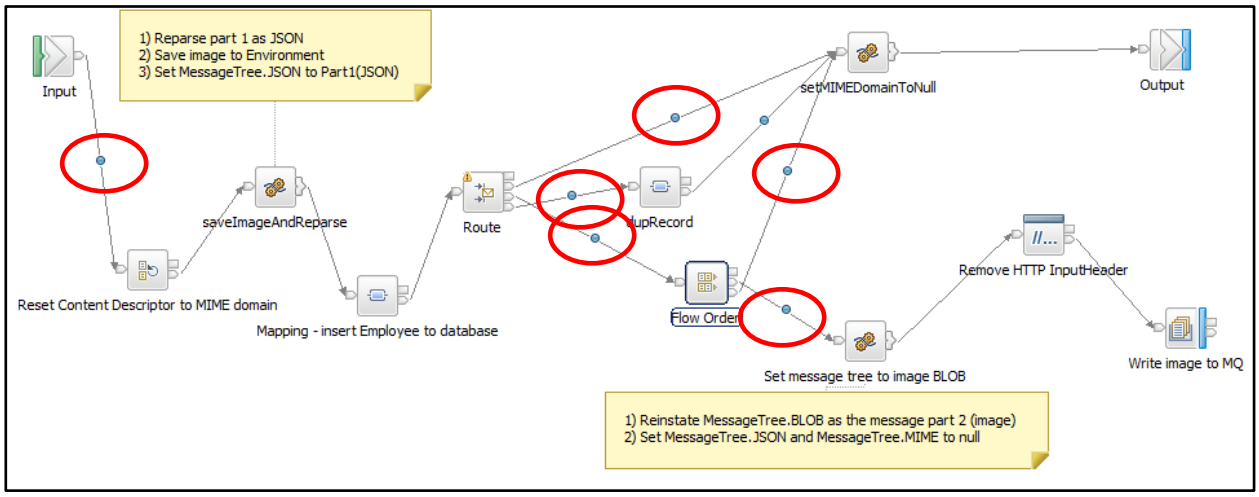




2. In the createMultipartEmployee subflow (should still be open from above), add a breakpoint after the Input node (right-click, Add Breakpoints….".

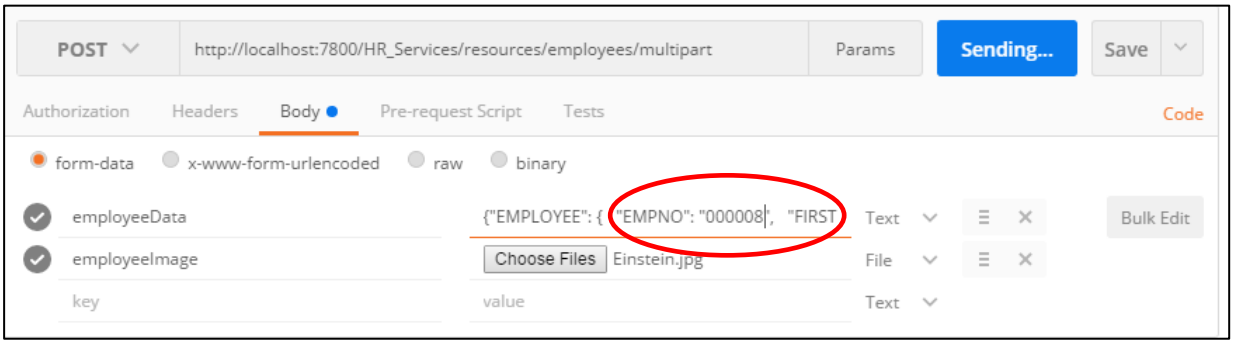


3. To ensure that all flow execution paths are included in the debugger, in the same way, set breakpoints as shown here. Note that if a node has more than one output terminal connected, then each connector should have an explicit breakpoint.

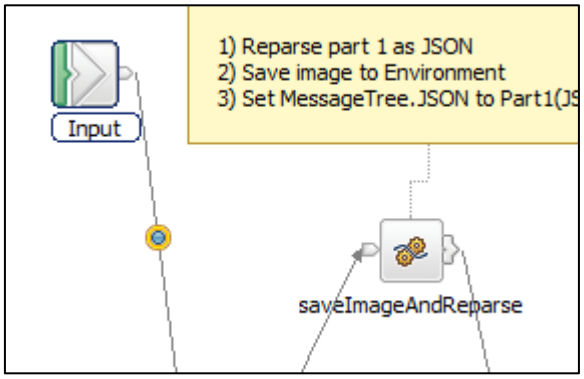


4. Invoke the test again. In Postman, provide a new value for EMPNO (eg. 000008), and click Send.

   You may need to use the slide bar to move to the top of the Postman window to see the input data.

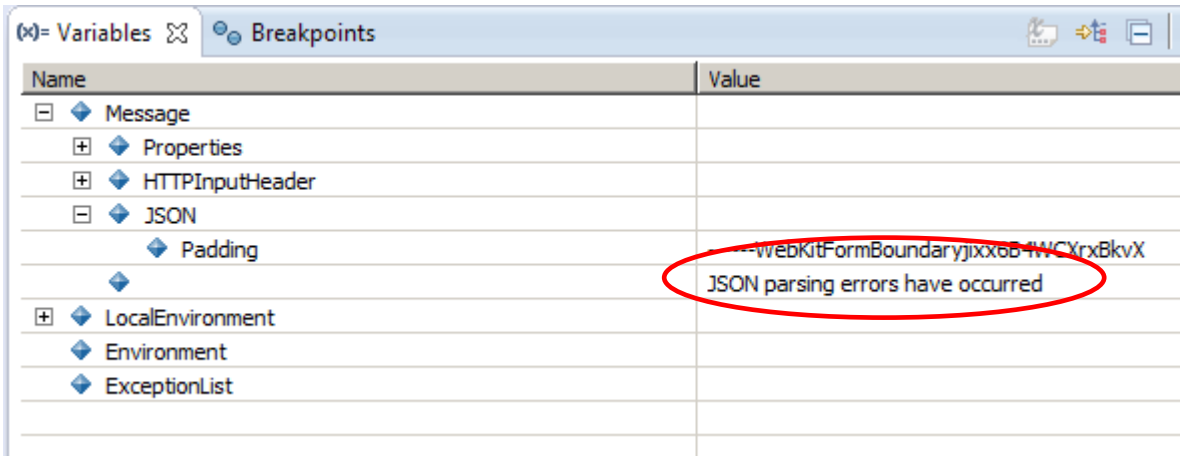


5. The flow will start, and execution will stop at the first breakpoint. (Respond Yes to switch to the Debug perspective).

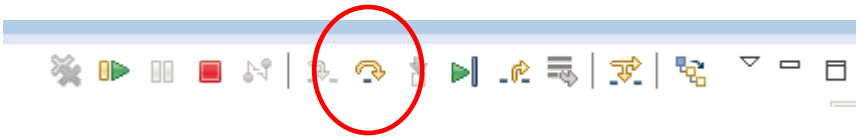


6. Highlight the debugger Variables view, and expand the incoming message.

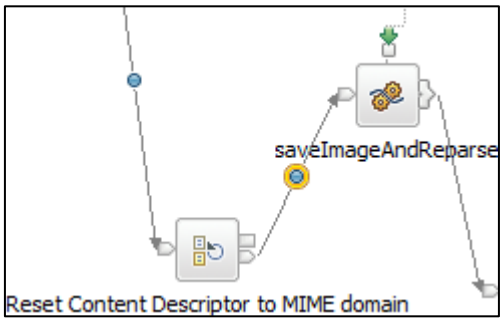
   Note that no user data is visible, and JSON parsing errors have occurred. This is because the REST API is configured to expect JSON data, but the message payload is not JSON. It is a multipart message with a JSON component, and a binary component, so the message has failed to be parsed.



7. Click the Step Over icon in the debugger control view.

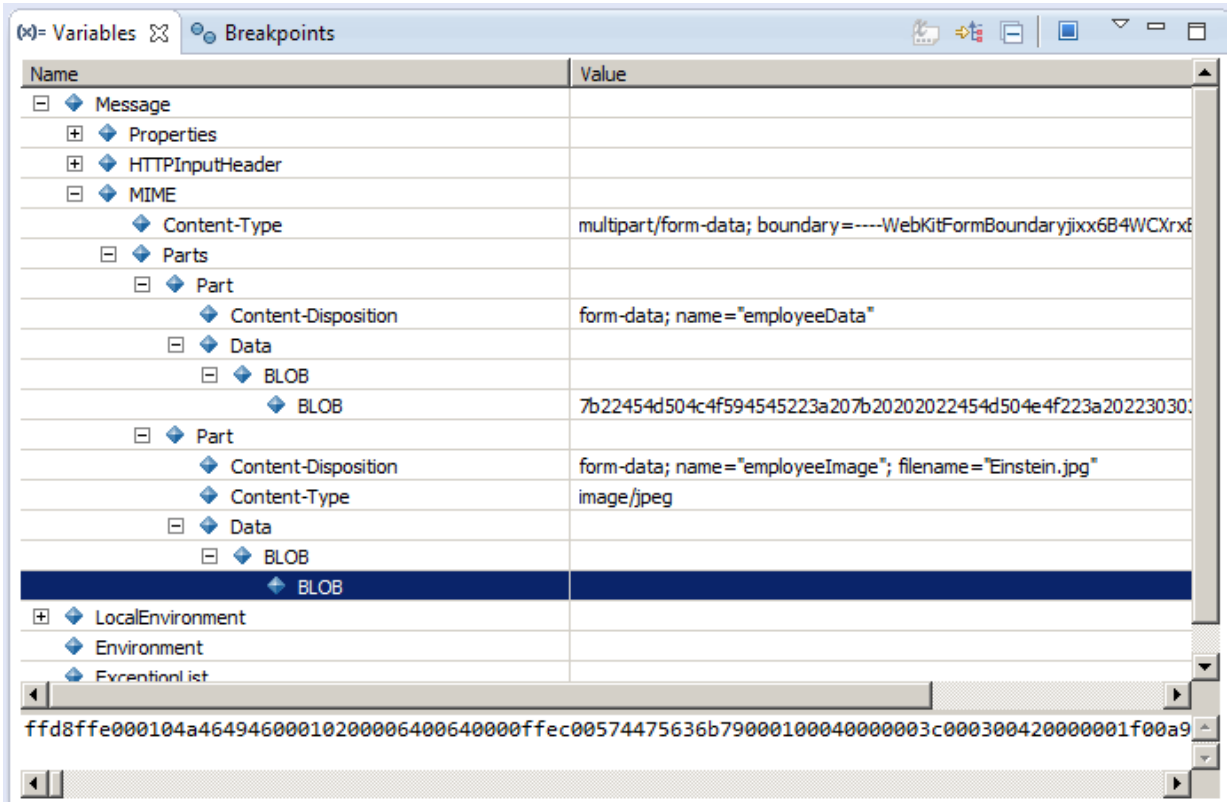


8. The debugger will pause after the Rest Content Descriptor node.

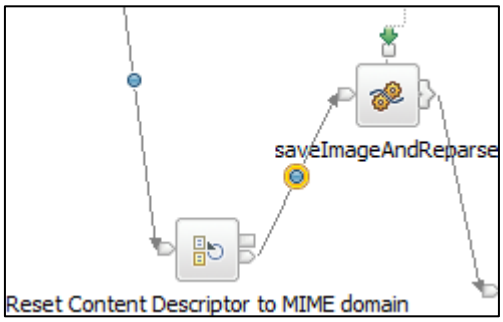


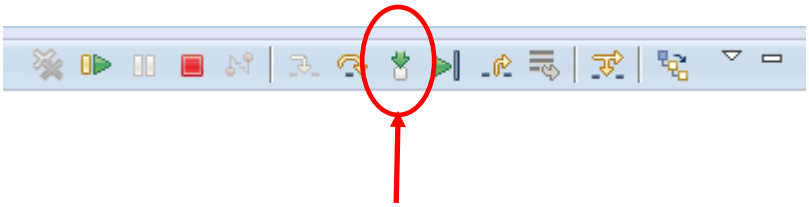
In the Variables view, expand the Message. Note that the message has been parsed by the MIME parser. Fully expand the MIME message, and note that there are two parts to the message.

- The JSON part of the message. (The data has not yet been parsed by the JSON parser, so the data is a BLOB and is not yet readable. However, the Debug perspective in the Toolkit renders the data in a readable hexadecimal format, as shown below).
- Part 2 is the binary data, containing the attached jpg image. Note the Content-Disposition contains the filename, and the Content-Type contains the type of data. Selecting the BLOB item shows the raw data of the binary part of the message.



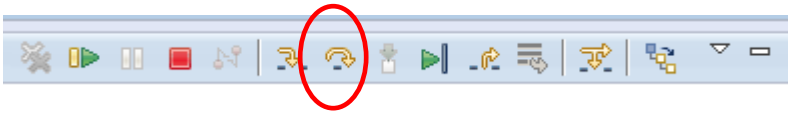
9. The debugger is currently paused just before the saveImageAndReparse compute node. It is instructive to observe the ESQL, and how the message tree is manipulated, so click the green down-arrow, as shown on the debug control line, to enter the ESQL node in debug mode.



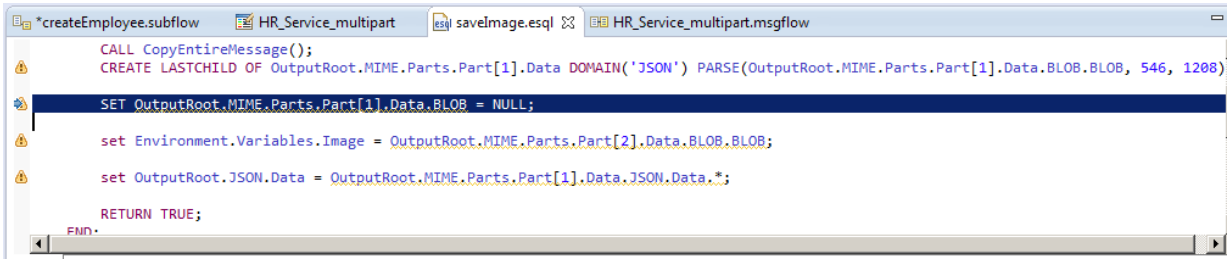


10 Click the Step Over icon a few times, until the **next** line to be executed (the highlighted line) is the line starting

```
SET OutputRoot.MIME.Parts …
```



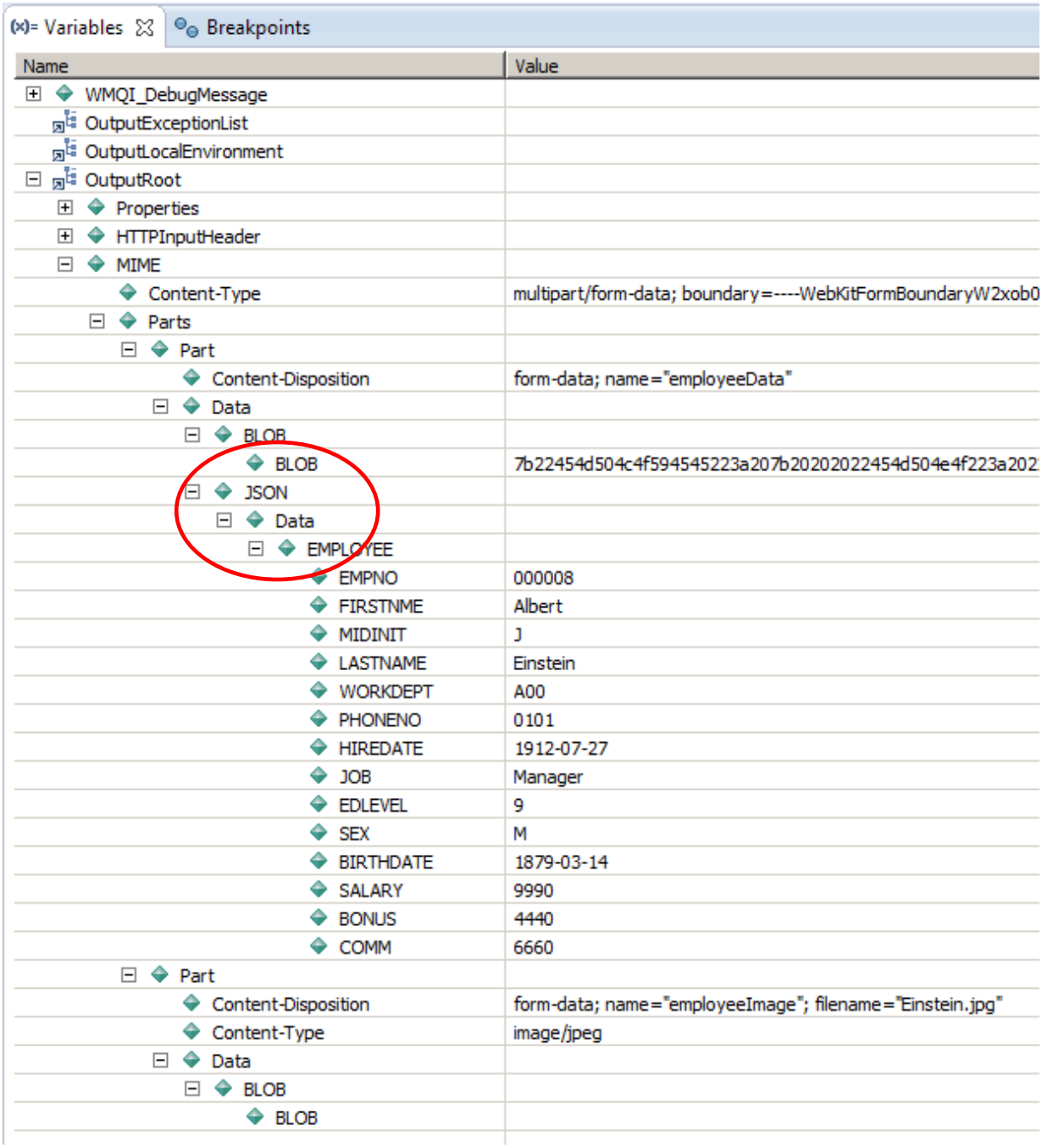
This means that the line starting **`CREATE LASTCHILD . .`** has just been executed.



This statement does two things:

1) Creates a new element called “JSON” in the array OutputRoot.MIME.Parts.Part[1].Data
2) Parses the BLOB part of the message and uses this to populate the new element Parts.Part[1].Data.JSON.
   Because the BLOB data was a JSON message, this results in the EMPLOYEE message being recreated in the new output element.

11 In the Variables view, expand OutputRoot. You will see that MIME section now has a new Data element under Parts.Part[1]. This Data element has been created in the JSON domain, so you are now able to see the Employee data in its fully parsed state (even though it is currently held under the MIME part of the message).

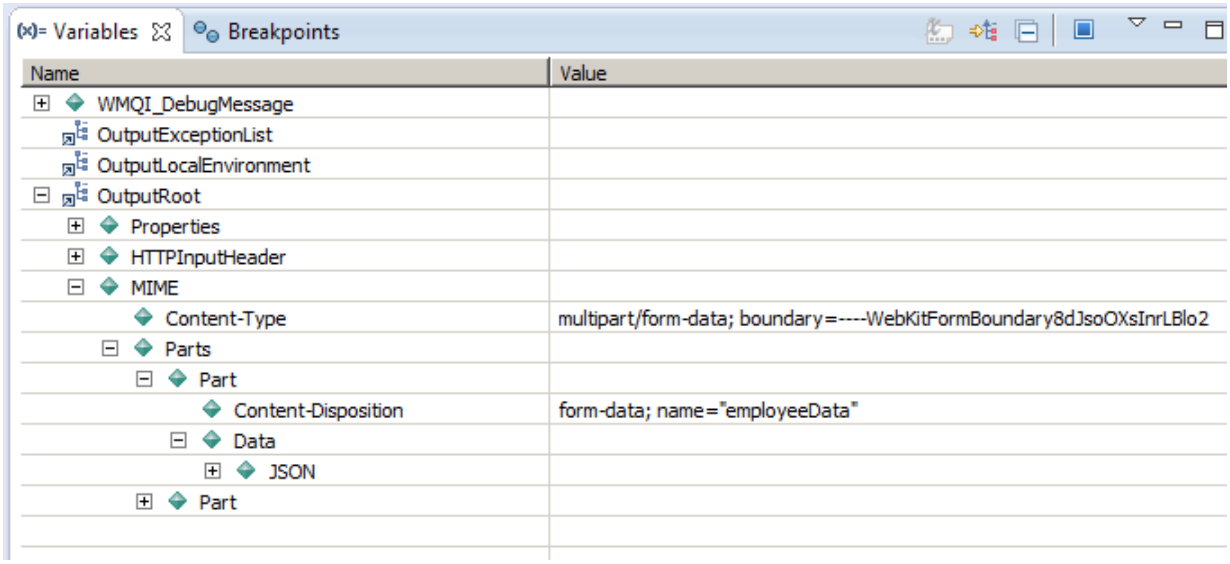


12 In the debugger, step over once more.

The line above the highlighted line shown below will have been executed.

```
CALL CopyEntireMessage();
CREATE LASTCHILD OF OutputRoot.MIME.Parts.Part[1].Data DOMAIN('JSON') PARSE(OutputRoot.M
SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;
set Environment.Variables.Image = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;
set OutputRoot.JSON.Data = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.*;
RETURN TRUE;
```

13 In Variables, note that the OutputRoot, under the MIME section, does not now contain a BLOB folder (the last line just set it to Null).

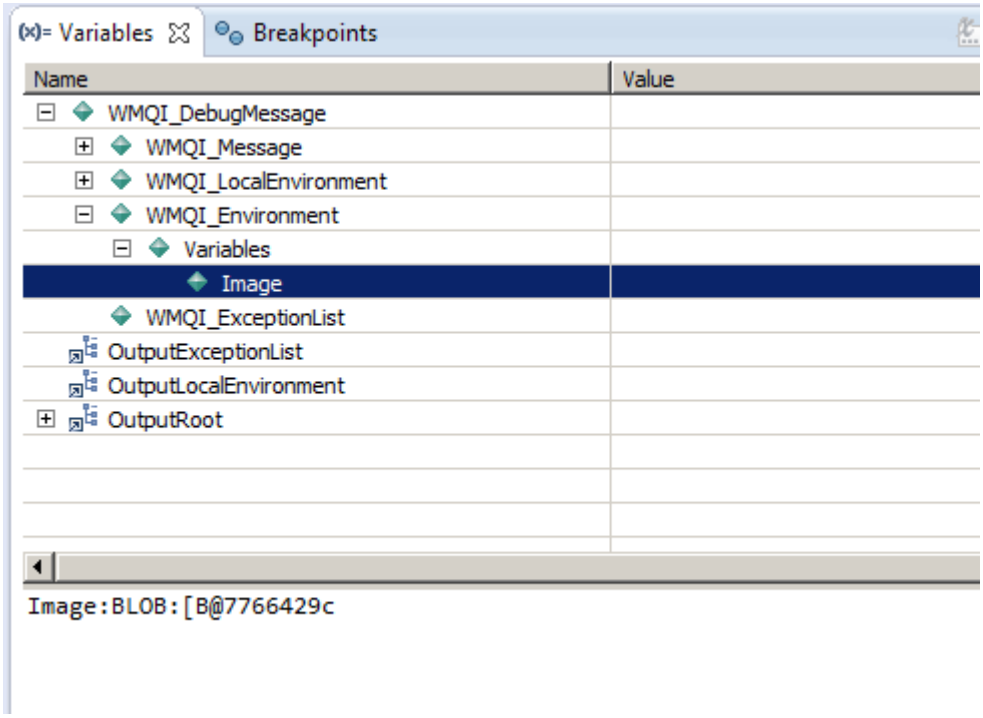


14 In the debugger, step over once more.

```
        CALL CopyEntireMessage();
        CREATE LASTCHILD OF OutputRoot.MIME.Parts.Part[1].Data DOMAIN('JSON') PARSE(OutputRoot
        SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;
        set Environment.Variables.Image = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;
        set OutputRoot.JSON.Data = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.*;
        RETURN TRUE;
    END;
```

15 In variables, expand WMQI_DebugMessage (the Environment tree appears in here, not under the OutputRoot).

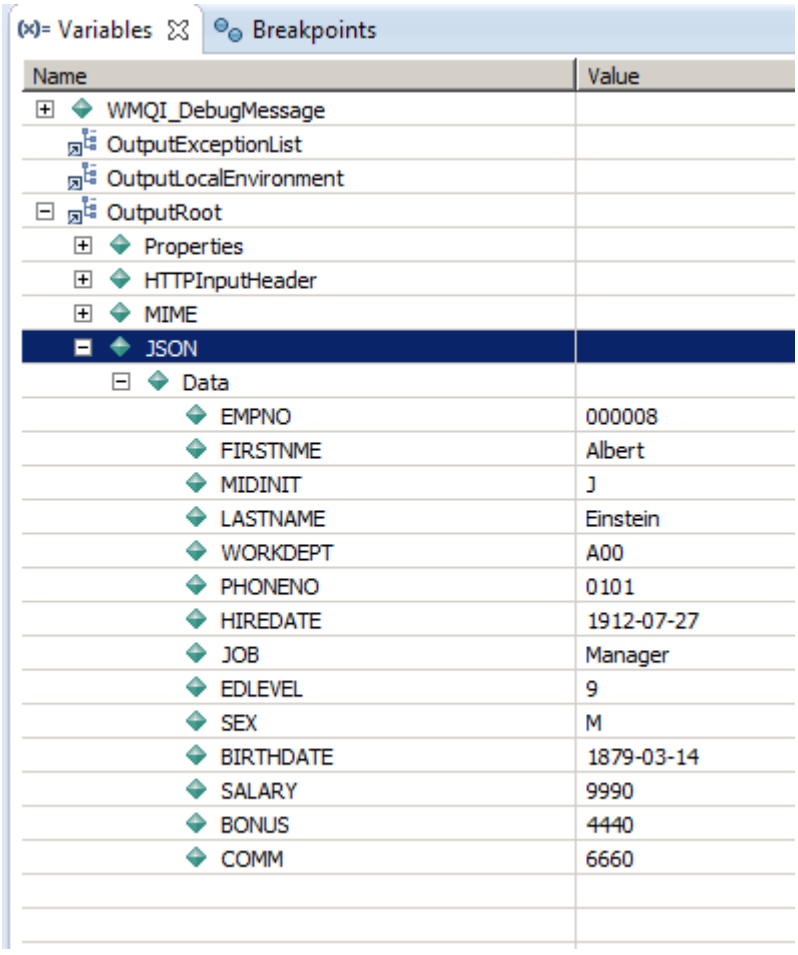
Note that the WMQI_Environment folder now has a folder called Variables, with an element called Image.



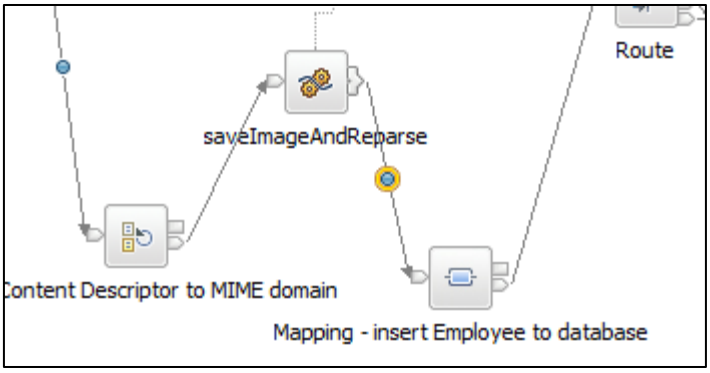
16 Step over once more.

```
        CALL CopyEntireMessage();
        CREATE LASTCHILD OF OutputRoot.MIME.Parts.Part[1].Data DOMAIN('JSON') PARSE(OutputRo
        SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;
        set Environment.Variables.Image = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;
        set OutputRoot.JSON.Data = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.*;
        RETURN TRUE;
    END;
```

17 The OutputRoot message now has a JSON folder as a primary folder in the JSON domain. It contains the full data of the EMPLOYEE input, although the EMPLOYEE folder name has not been propagated in this example.



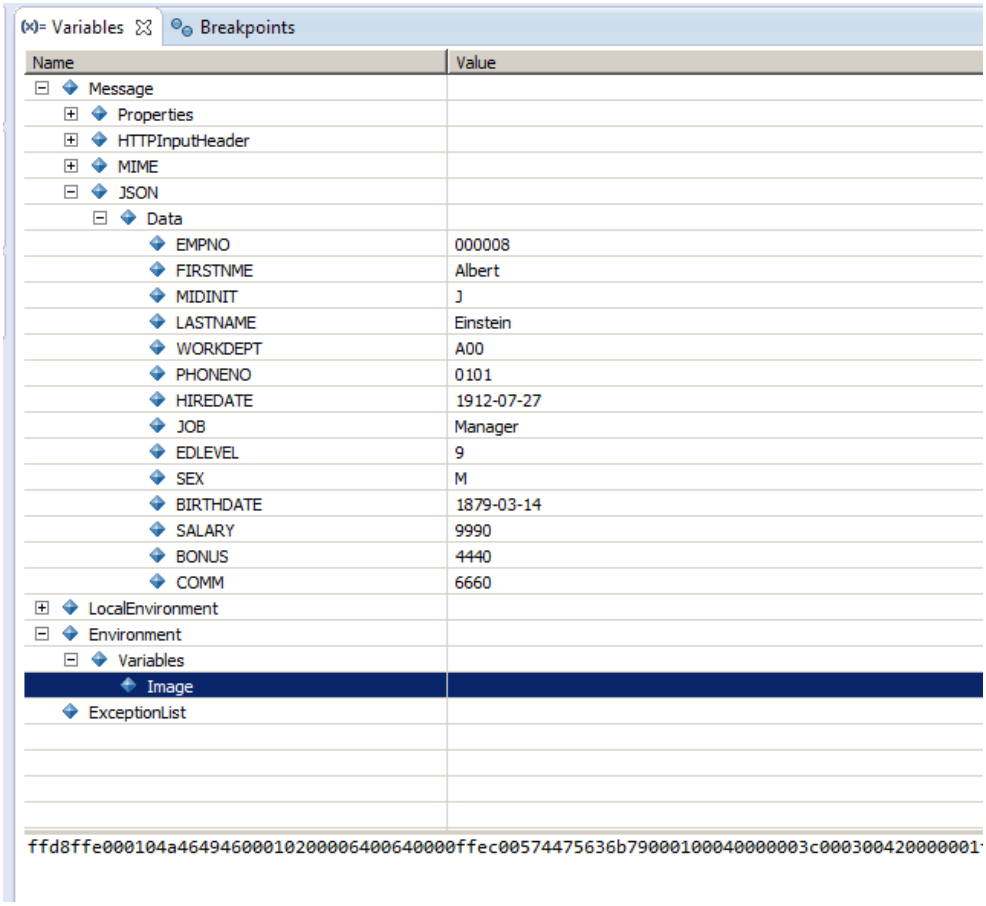
18 Step over once more. The ESQL compute node will complete, and flow execution will resume. The flow will stop at the next node breakpoint.



19 In the Variables view, you will see that the flow has now extracted the JSON part of the message, and this is now held in the message tree, directly under the JSON folder.

Additionally, the attached JPG image has been extracted, and is located in the Environment tree, under Variables/Image.

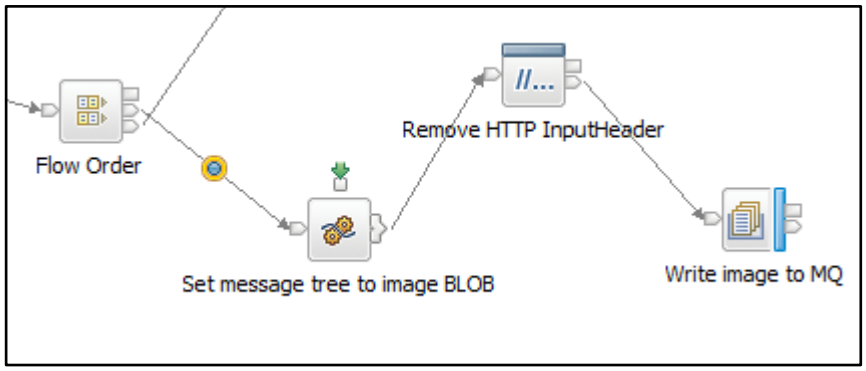
So, we now have the incoming message split into its two parts. The JSON part now represents the main message tree, and the image is in the Environment tree. Highlighting “Image” will show the raw contents of the image file.



3.4 Execute the remainder of the flow

The remainder of this lab guide will not explicitly show the execution of every ESQL statement, although you are welcome to do so in your own testing.

1. Step over the flow after the Flow Order node.

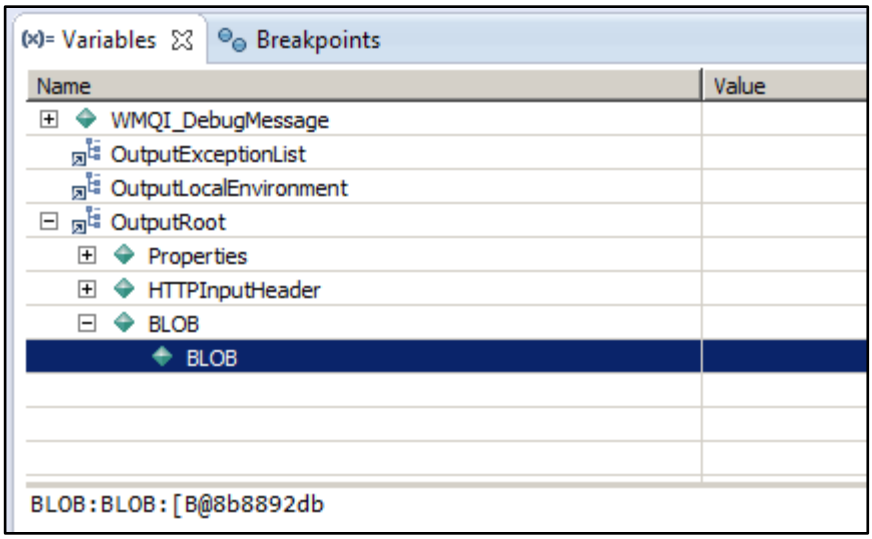


2. Step into the ESQL source to observe updates to the message tree.

   Step over each line of ESQL until the **RETURN TRUE** line is highlighted.

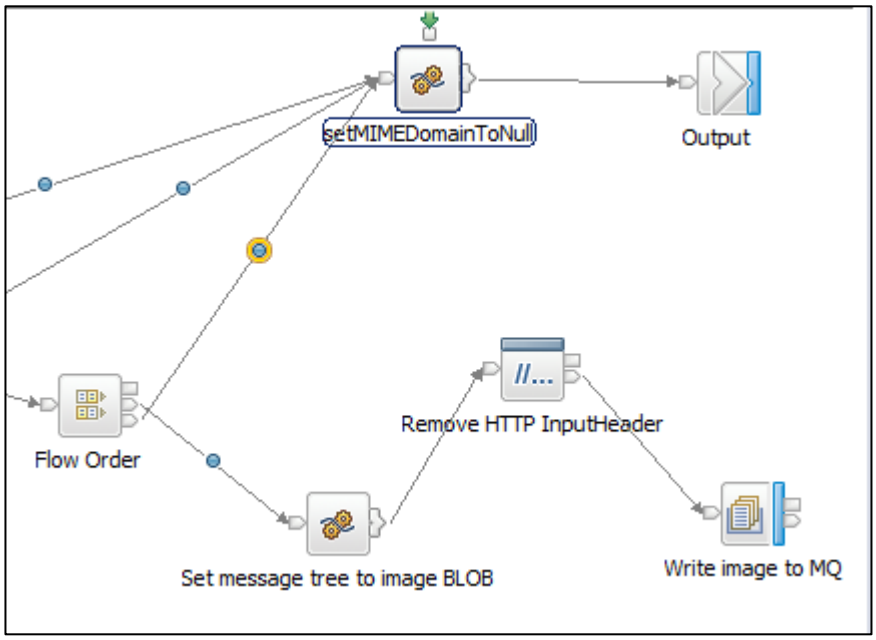
```
CREATE COMPUTE MODULE setMessageTreeToImageBLOB
    CREATE FUNCTION Main() RETURNS BOOLEAN
    BEGIN
        -- CALL CopyMessageHeaders();

        CALL CopyEntireMessage();
    --  Note - the following line can be used if required, because the FlowOrder
    --  node propagates the OutputRoot unchanged.
    --  set OutputRoot.BLOB.BLOB = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;

    --  However, for consistency with the earlier part of this subflow, we will
    --  retrieve the binary image from the Environment tree.
        set OutputRoot.BLOB.BLOB = Environment.Variables.Image;

    --  Now set both the JSON and MIME parts of OutputRoot to null, so that the
    --  OutputRoot.BLOB message can be written to MQ
        set OutputRoot.JSON = NULL;
        set OutputRoot.MIME = NULL;
        RETURN TRUE;
    END;
```

3. The Variables now show that the only part of the OutputRoot is BLOB.



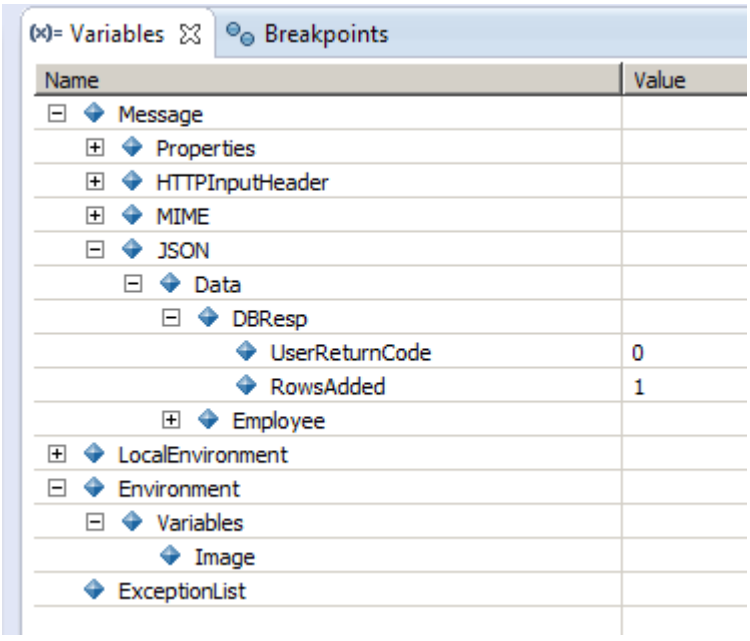
4. Using the debugger, step over several more nodes. After the MQOutput node is executed, flow will switch back to the FlowOrder node, and the Second terminal will be propagated.

   Pause the debugger when the breakpoint after the FlowOrder (second terminal) is encountered.

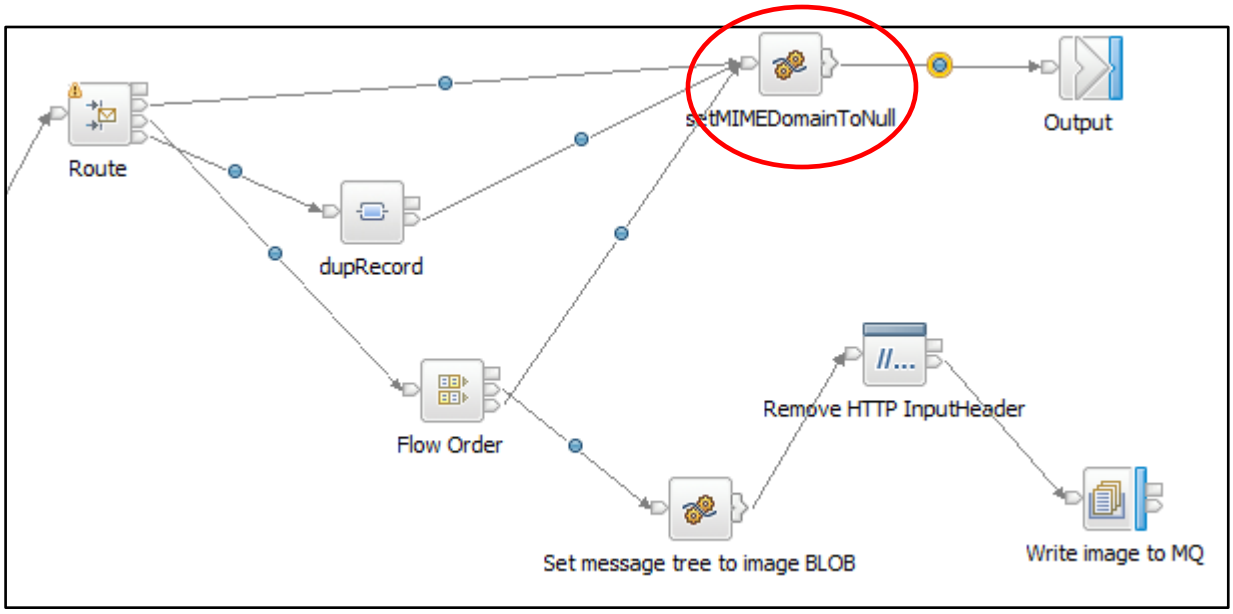


   Note that the message tree has been reinstated to the contents that were available when the FlowOrder node was first executed. The MIME and JSON folders are now available again.

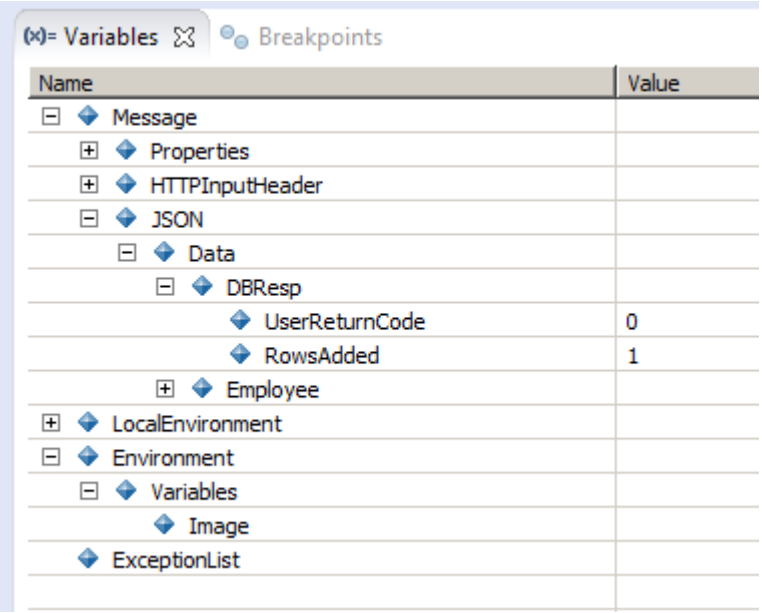
   The Variables view will show that the Message now contains MIME and JSON folders. The JSON folder will show that the database insert was successful, and the number of rows added was 1.



5. Step over the setMIMEDomainToNull node, and pause just before the Output node.



Note that the output message has had the MIME folder removed, and now contains only the JSON folder. It can therefore be sent back to the originating client as a REST response in correct JSON format.



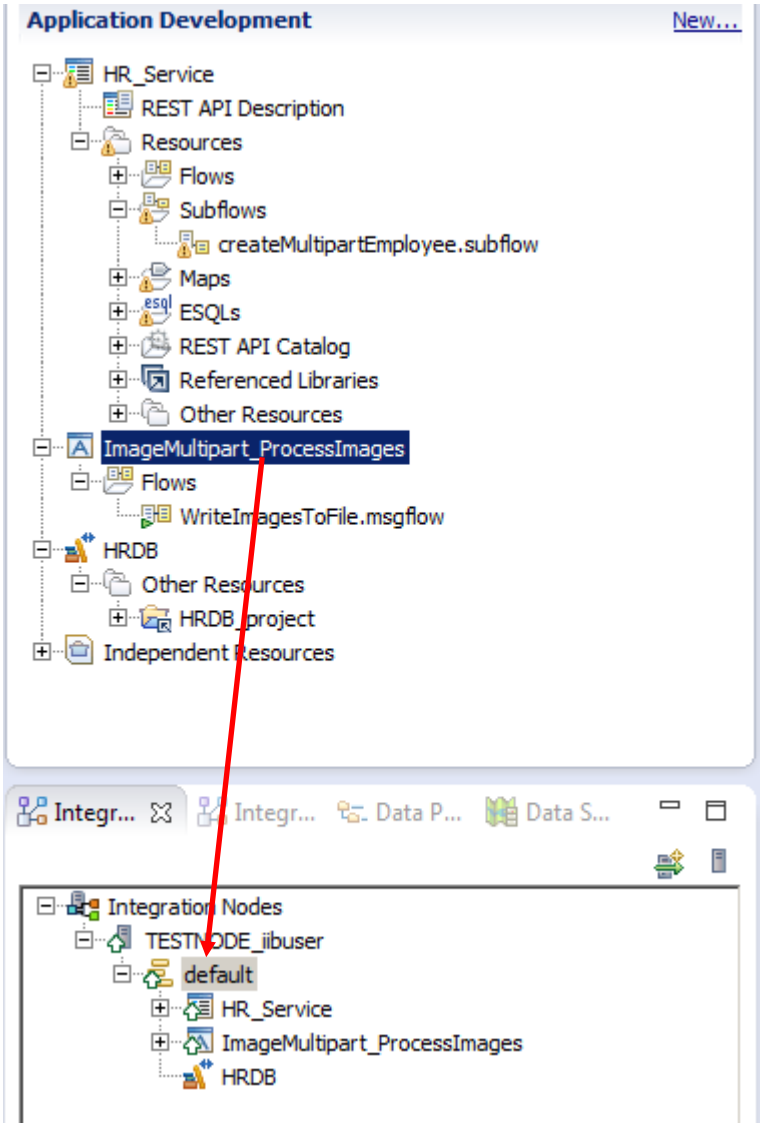
6. Click Step Over to complete execution of the flow. If you have taken more than 180 seconds to execute this flow, the debugger will probably terminate. Click the debugger terminate buttons.

3.5 Deploy and execute the MQ Application

To complete the full processing of the scenario, use the ImageMultipart_ProcessImages application that is provided.

1. From the tests throughout this lab, there should be at least 1 message waiting on the MQ queue IMAGES.MULTIPART.OUT. These will be processed immediately when this application is deployed. Check that you have at least one MQ message on this queue.

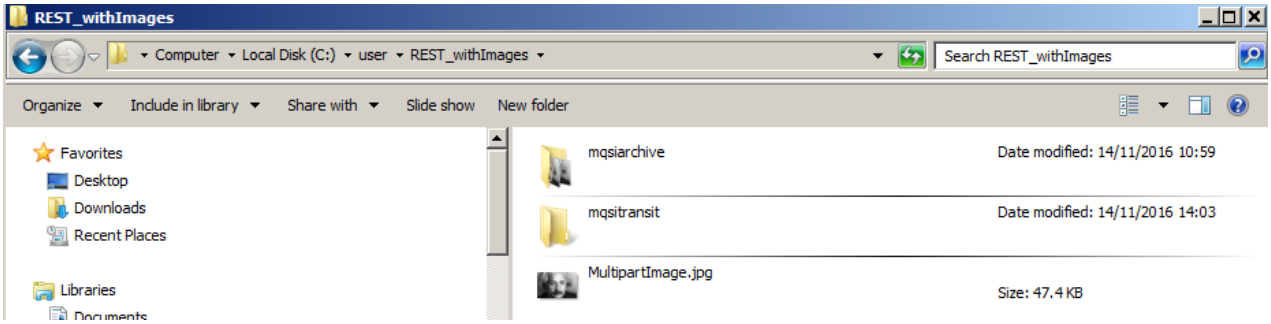
2. Deploy the ImageMultipart_ProcessImages application.



3. The messages, containing the original image attachment, will be written to file.

   In Windows Explorer, navigate to c:\user\REST_withImages. You will see a newly-written file, MultipartImage.jpg, which contains a copy of the original data that was attached to the REST request.

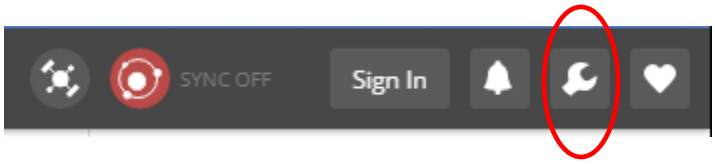
   Also, note that the mqsiarchive folder contains several older copies of the same file.These have filename components based on the current date and time, and have been created as a result of the properties specified on the FileOutput node in the WriteImagesToFile message flow.

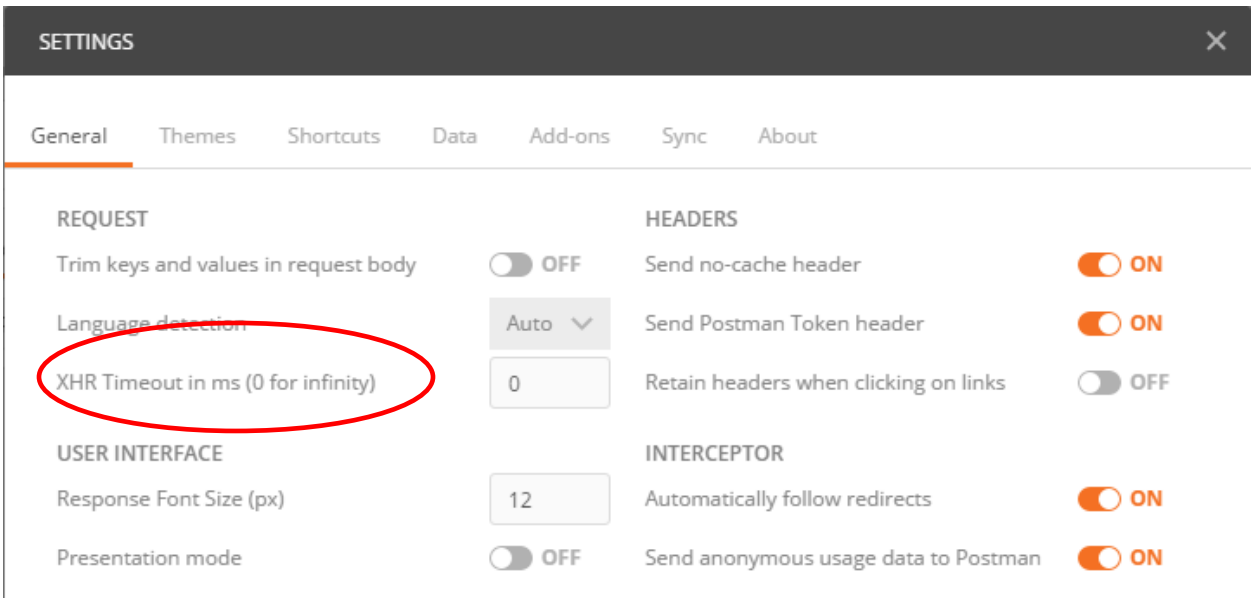


4. Appendix – HTTP Timeout options

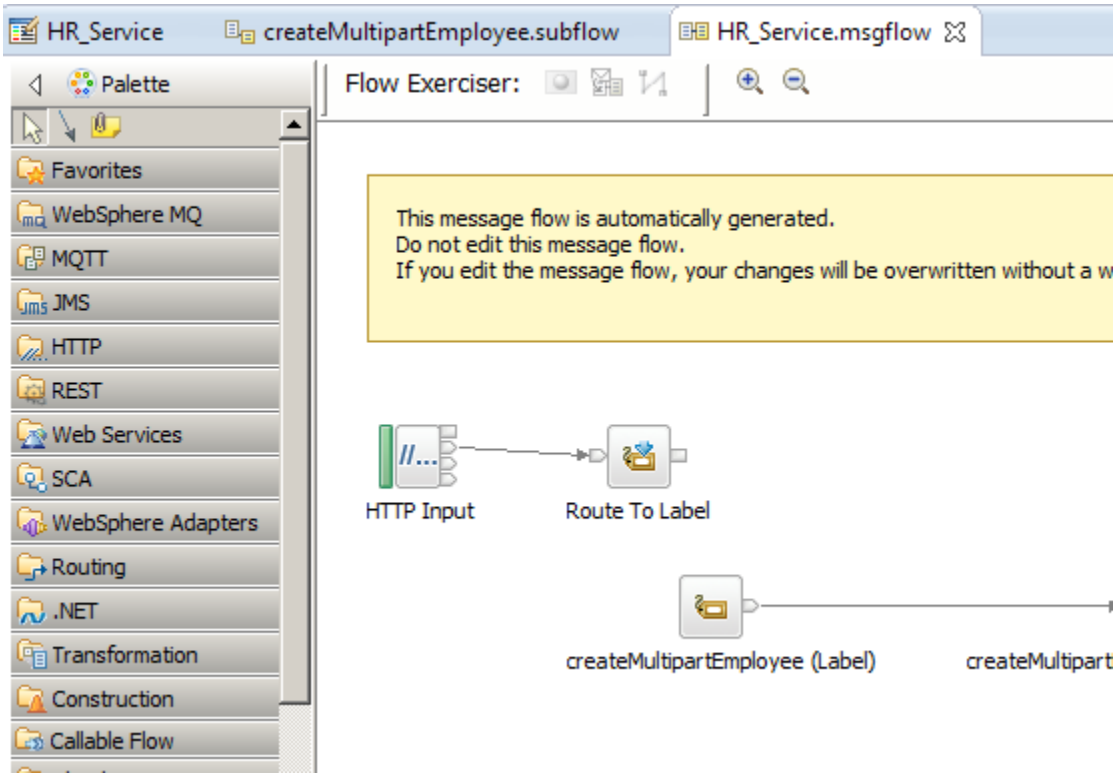
In HTTP scenarios, there are various points at which an HTTP timeout may occur. For example, in this lab, Postman acts as an HTTP client, and sends a request to the HR_Service REST API. Postman will wait until an HTTP reply is received, but may wish to terminate if a reply is not received.

The default timeout value used by Postman is 0, which means that it will wait indefinitely for a reply from the REST API. This default setting in Postman can be changed in Settings, under the General tab. Set the property **XHR Timeout**.

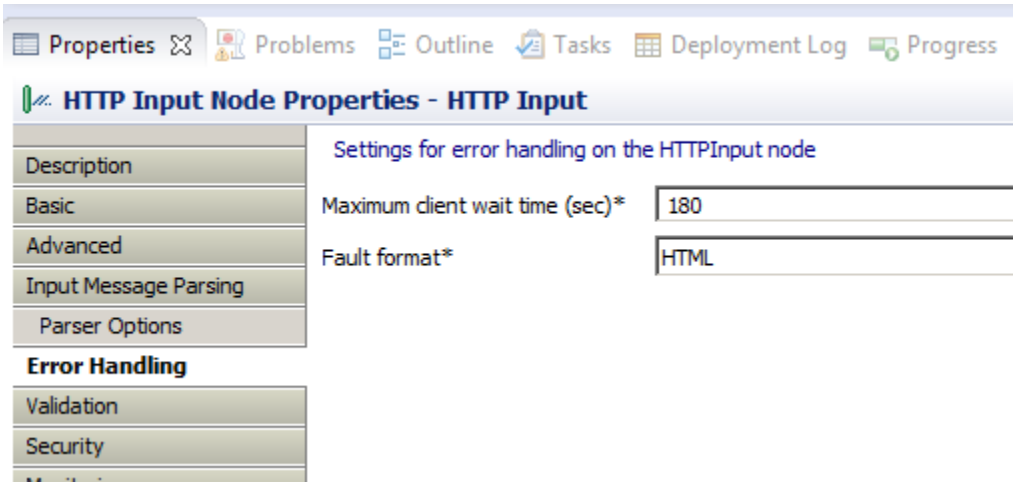




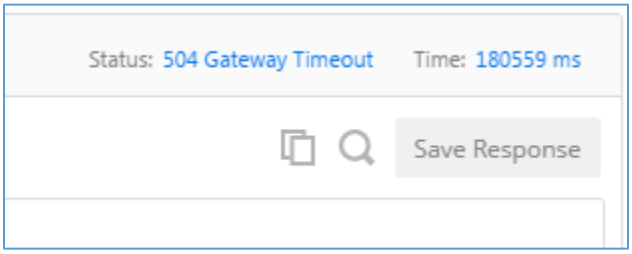
In the IIB REST API, the primary message flow HR_Service.msgflow contains an HTTPInput node (you can open this message flow by right-clicking on the flow in the navigator and selecting “Open with Message Flow Editor”).



Click the HTTP Input node and select the node properties, Error Handling tab. You will see that the default value for the Maximum Client Wait Time is 180 seconds. This means that if the IIB message flow fails to provide a response to the client (eg. Postman) within this time, then a default response will be sent.



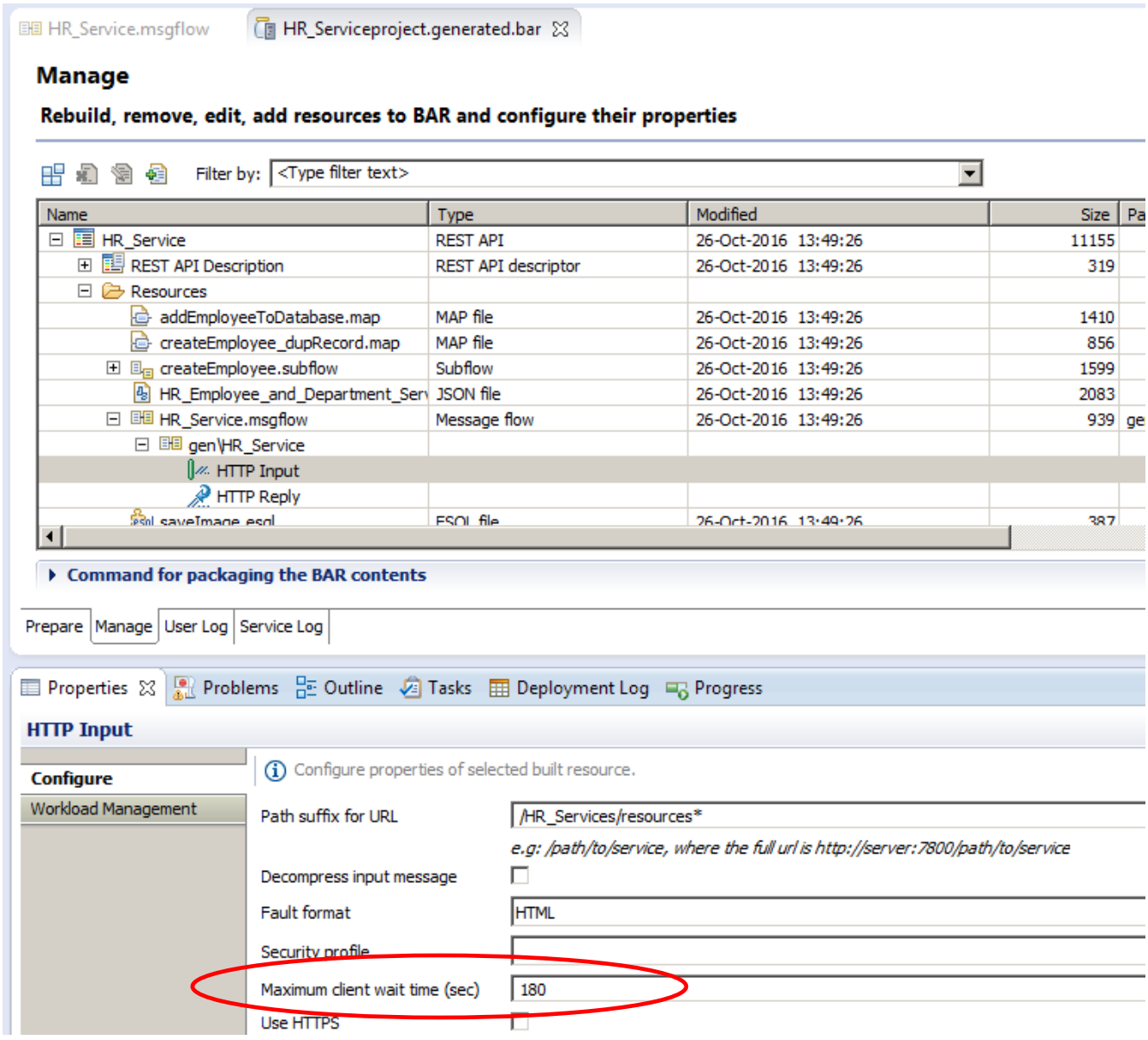
If this happens, then Postman would report such an event like this. Note that a Timeout was reported, and the Time was 180559 msecs (ie. this time was a result of the Timeout property on the IIB HTTP Input node).



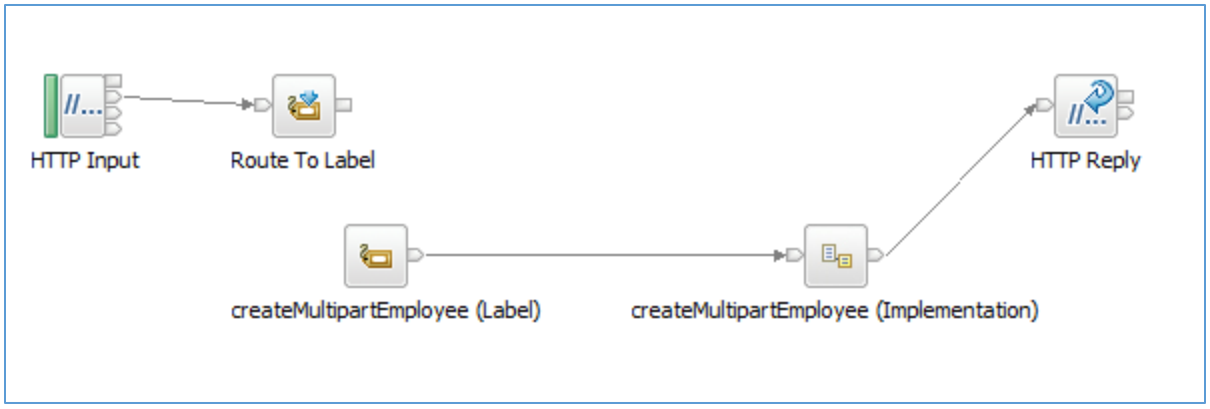
If the IIB Maximum Client Wait Time was set to 0, then no timeout would be reported by Postman (unless the XHR Timeout property had been changed, as described above).

If you wish to change the Maximum Client Wait Time property on the IIB HTTP Input node, you should not do this by editing the message flow. This is because any changes may be overwritten by the Toolkit when new operations are added.
Such changes should be made by editing the barfile that is deployed to the IIB server. For example, generated barfile for this lab looks like this. Select the HTTP Input node. You can see that you can change the value of the Maximum Client Wait Time (default is 180 seconds).

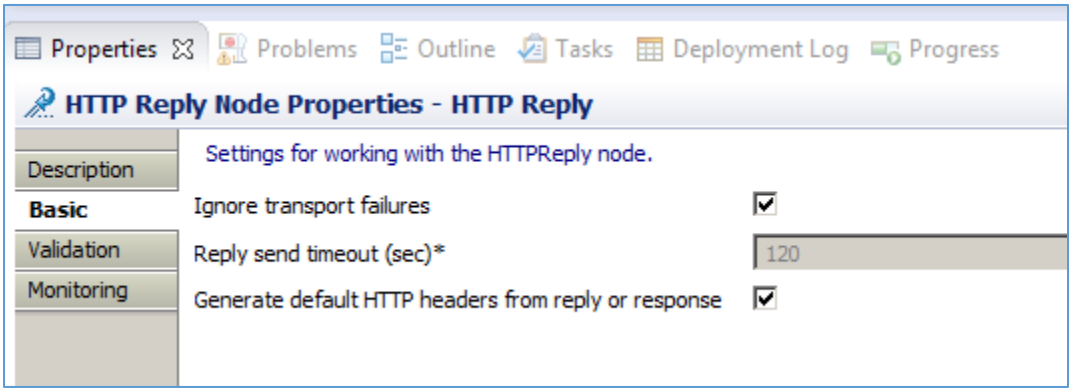
However, note that generated barfiles will be regenerated when you drag/drop an application from the navigator to the server. If you are going to make changes to your barfile, it is normally best practice to create and build your own barfile.



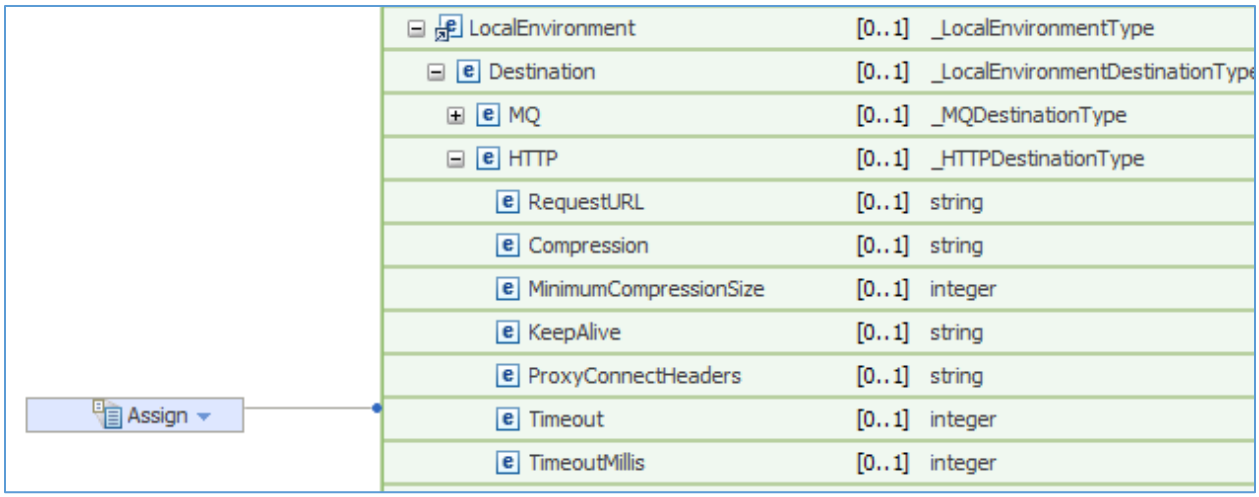
Finally, again on the generated main message flow, the HTTP Reply node also has a timeout property. Click the HTTP Reply node in the main message flow.



The Reply Send Timeout is set to 120. This value represents the time that the HTTP reply node will wait for a final acknowledgement from the originating client. If this client (eg. Postman) has terminated during the processing of the IIB message flow, then the sending of the final HTTP reply will fail. This property specified how long IIB will wait before terminating.



To set this property for REST API applications (and SOAP-based Integration Services), you should not make edits to the node properties in the editor. Instead, the value of this property can be set in the LocalEnvironment, in the Destination/HTTP/Timeout element. This can be set in a map (shown below, or programmatically with an ESQL or Java Compute node).



END OF LAB GUIDE

.