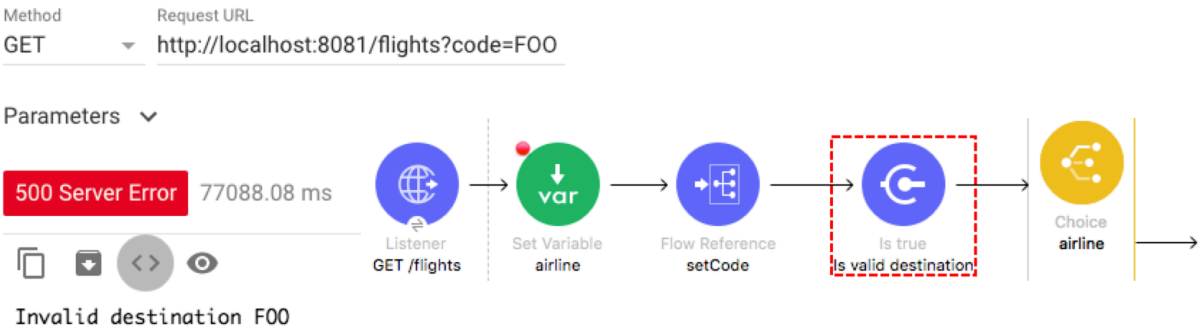
Walkthrough 9-3: Validate events

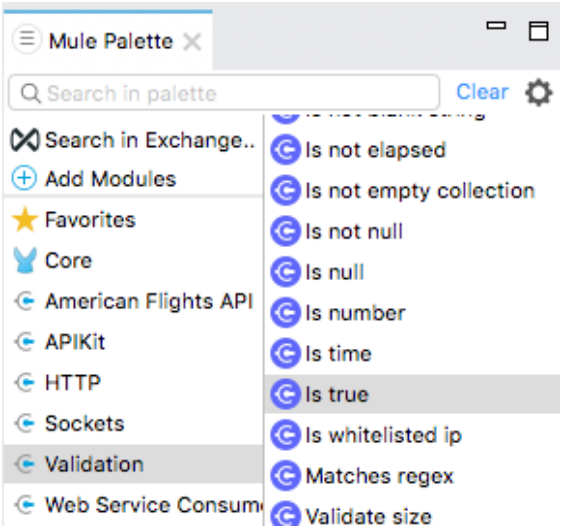
In this walkthrough, you use a validator to check if a query parameter called code with a value of SFO, LAX, CLE, PDX, or PDF is sent with a request and to throw an error if it is not. You will:

- Add the Validation module to a project.
- Use an Is true validator to check if a query parameter called code with a value of SFO, LAX, CLE, PDX, or PDF is sent with a request.
- Return a custom error message is the condition is not met.



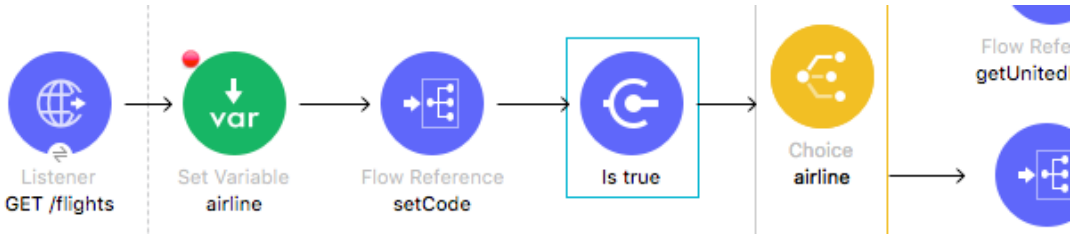
Add the Validation module to the project

1. Return to implementation.xml.
2. In the Mule Palette, select Add Modules.
3. Select the Validation module in the right side of the Mule Palette and drag and drop it into the left side.
4. If you get a Select module version dialog box, select the latest version and click Add.



Use the Is true validator to check for a valid destination code

5. Locate the Is true validator in the right side of the Mule Palette.
6. Drag and drop the Is true validator after the setCode Flow Reference in the getFlights flow.



7. In the Is true properties view, set the display name to: Is valid destination.
8. Change the expression from False (Default) to Expression.
9. Add a DataWeave expression to check if the code variable is one of the five destination codes.

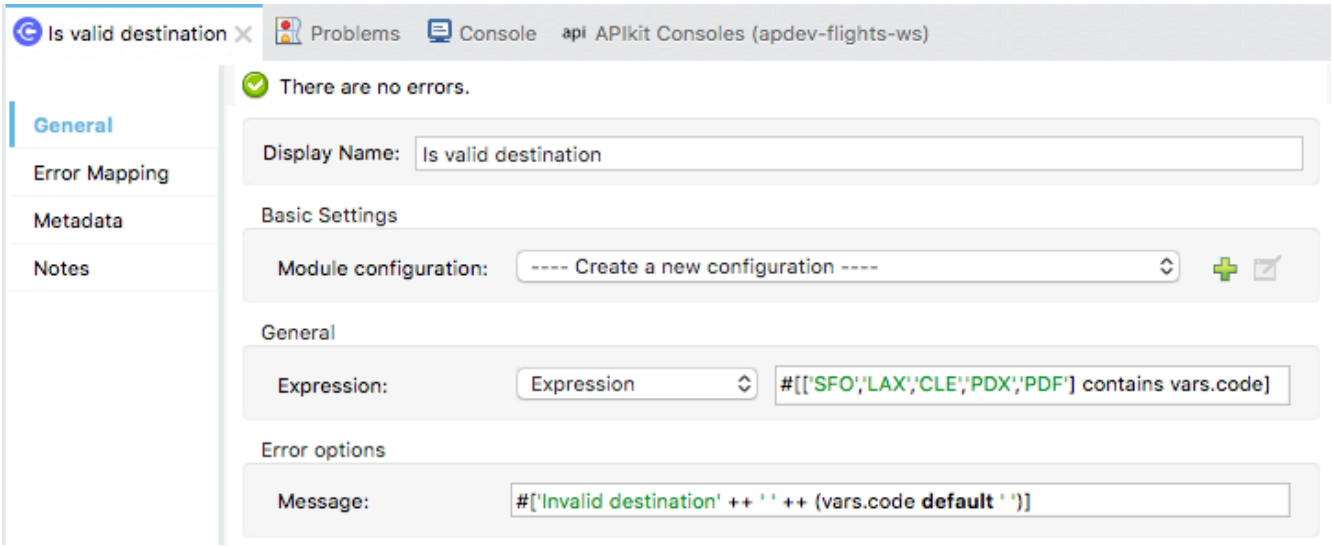
```
#[['SFO','LAX','CLE','PDX','PDF'] contains vars.code]
```

Note: You can copy this expression from the course snippets.txt file.

10. Set the error message to Invalid destination followed by the provided code.

```
#['Invalid destination' ++ ' ' ++ (vars.code default ' ')]
```

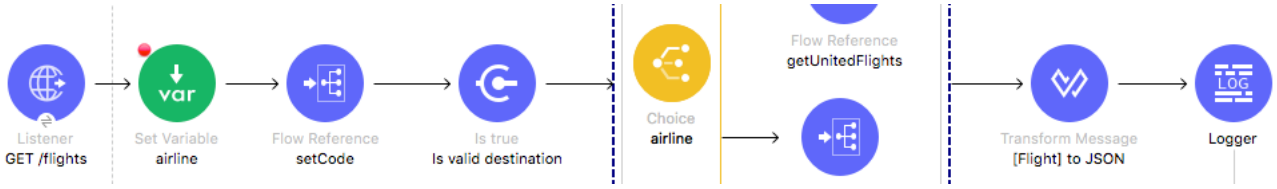
Note: You can copy this expression from the course snippets.txt file.



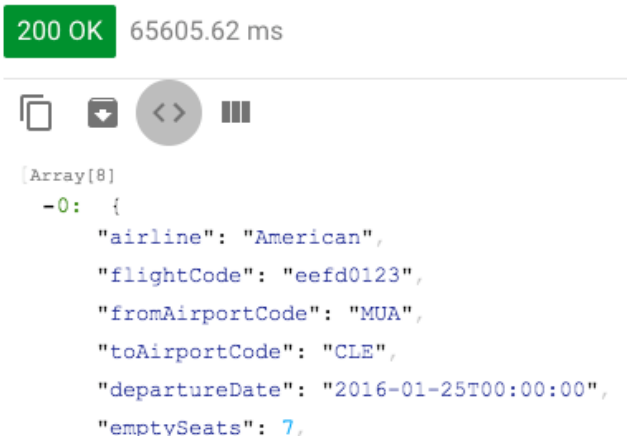
Debug the application

11. Save the file to redeploy the project in debug mode.
12. In Advanced REST Client, change the code to make a request to http://localhost:8081/flights?code=CLE.

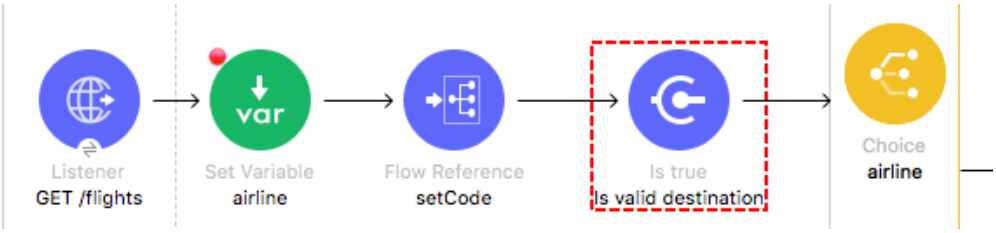
13. In the Mule Debugger, step past the validator; you should see the code is valid and application execution steps to the Choice router.



14. Resume through the rest of the application.
15. In Advanced REST Client, you should see flights.



16. Change the code and make a request to http://localhost:8081/flights?code=FOO.
17. In the Mule Debugger, step to the validator; you should see an error.

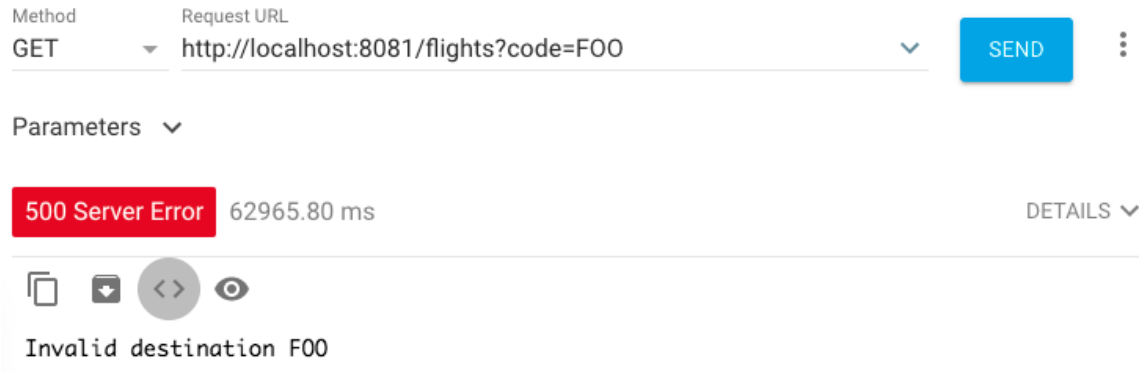


18. Step again; you should see your Invalid destination message in the console.

```
ERROR 2018-04-20 11:55:44,836 [[MuleRuntime].cpuLight.15: [apdev-flights-ws].getFlights.CPU_LITE @65ffdff1] [event: 0-48
bd68f0-44cc-11e8-a62a-8c85900da7e5] org.mule.runtime.core.internal.exception.OnErrorPropagateHandler:
********************************************************************************
Message               : Invalid destination FOO.
Error type            : VALIDATION:INVALID_BOOLEAN
Element               : getFlights/processors/2 @ apdev-flights-ws:implementation.xml:15 (Is valid destination)
Element XML           : <validation:is-true doc:name="Is valid destination" doc:id="e6d96ea1-71ee-45f1-9661-9048ce5382c9
" expression="#[['SFO','LAX','CLE','PDX','PDF'] contains vars.code]" message="#['Invalid destination' ++ ' ' ++ vars.cod
e]"></validation:is-true>
```

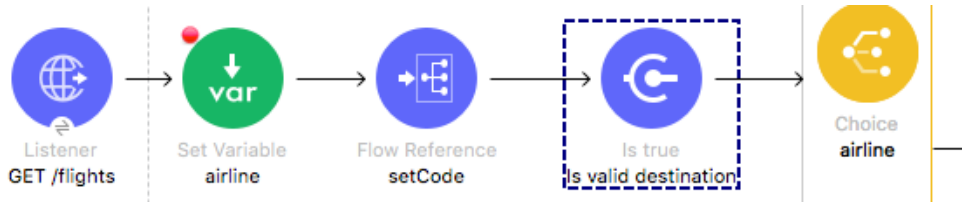
19. Resume through the rest of the application.

20. Return to Advanced REST Client; you should get a 500 Server Error with your Invalid destination message.

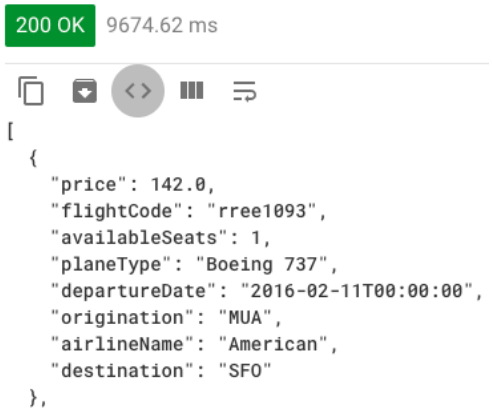


Note: You will catch this error and send a JSON response with a different status code in the next module.

21. Remove the code and make a request to http://localhost:8081/flights.
22. In the Mule Debugger, step through the application; you should not get any errors.



23. Resume through the rest of the application.
24. Return to Advanced REST Client; you should get a 200 response with all airline flights to SFO.

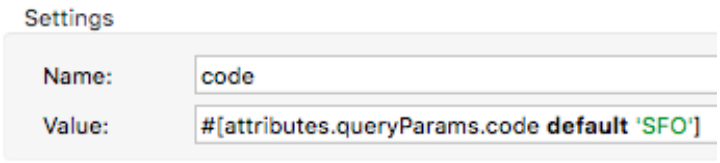


25. Return to Anypoint Studio and switch to the Mule Design perspective.

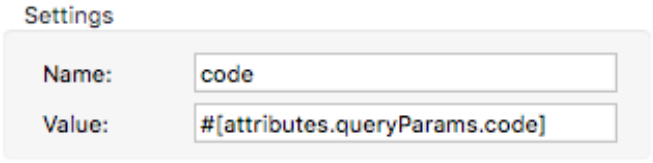
Remove the default destination

26. Locate the setCode subflow.

27. In the Set Variable properties view, review the default value.

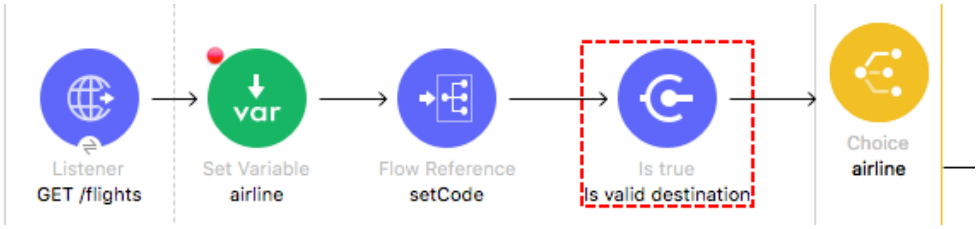


28. Remove the default value from the value.

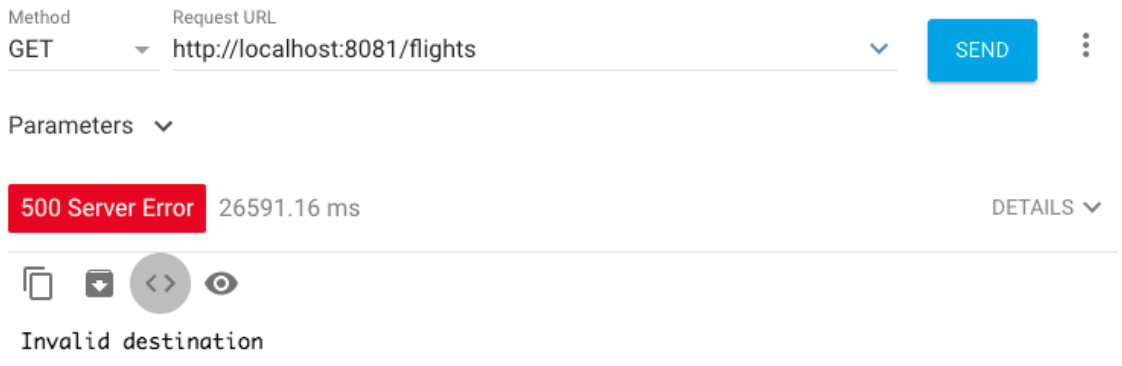


Debug the application

29. Save the file to redeploy the project.
30. In Advanced REST Client, make another request to http://localhost:8081/flights.
31. In the Mule Debugger, step to the validator; you should now get an error.



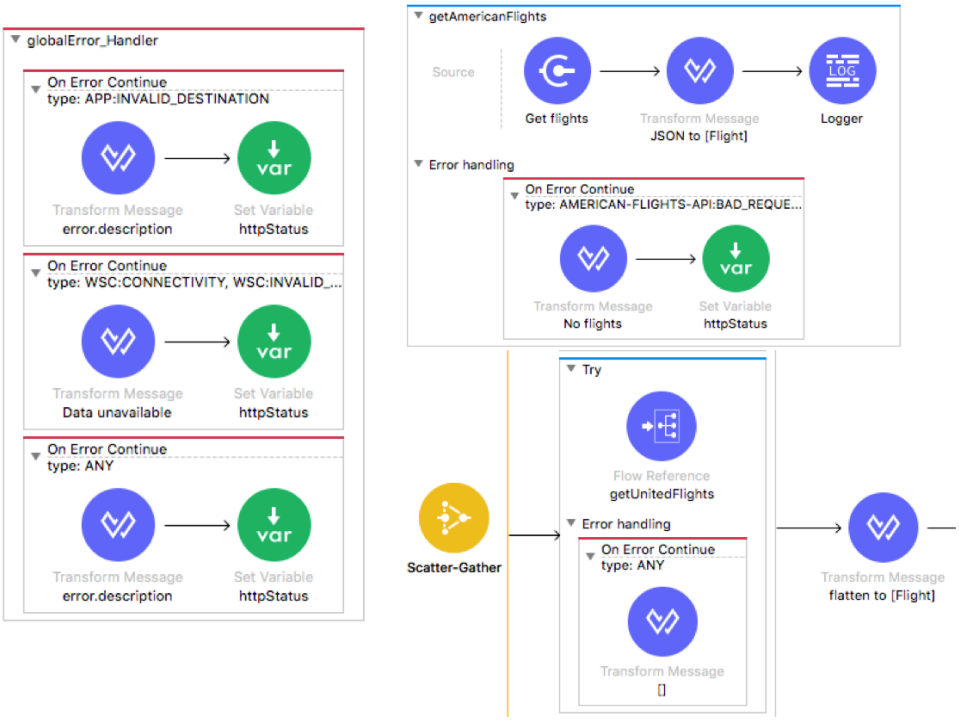
32. Resume through the rest of the application.
33. Return to Advanced REST Client; you should get a 500 Server Error with your Invalid destination message.



Note: You will catch this error and send a JSON response with a different status code in the next module.

34. Return to Anypoint Studio and stop the project.

Module 10: Handling Errors



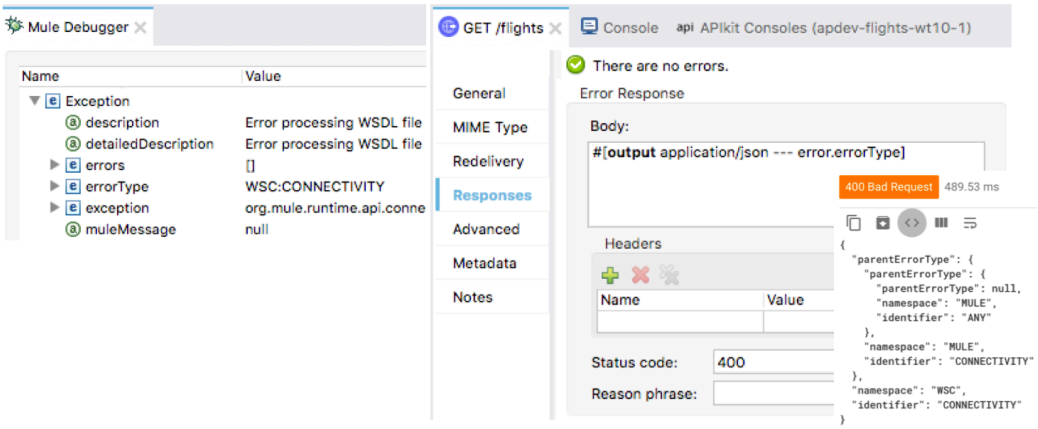
At the end of this module, you should be able to:

- Handle messaging errors at the application, flow, and processor level.
- Handle different types of errors, including custom errors.
- Use different error scopes to either handle an error and continue execution of the parent flow or propagate an error to the parent flow.
- Set the success and error response settings for an HTTP Listener.
- Set reconnection strategies for system errors.

Walkthrough 10-1: Explore default error handling

In this walkthrough, you get familiar with the three errors you will learn to handle this module. You will:

- Explore information about different types of errors in the Mule Debugger and the console.
- Review the default error handling behavior.
- Review and modify the error response settings for an HTTP Listener.



Create a connectivity error

1. Return to apdev-flights-ws in Anypoint Studio.
2. Open config.yaml.
3. Change the delta.wsdl property from /delta?wsdl to /deltas?wsdl.

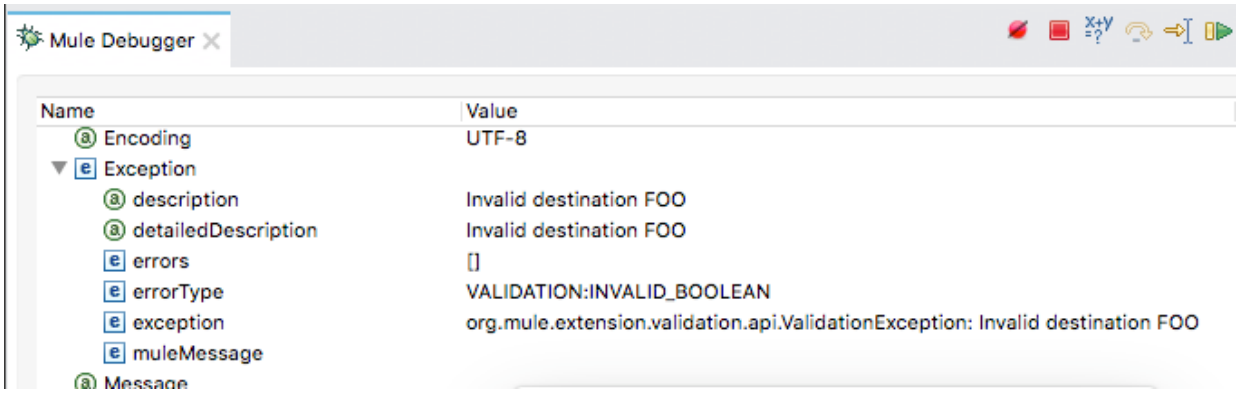
```
delta:
  wsdl: "http://mu.learn.mulesoft.com/deltas?wsdl"
  service: "TicketServiceService"
  port: "TicketServicePort"
```

4. Save the file.
5. Return to implementation.xml and debug the project.

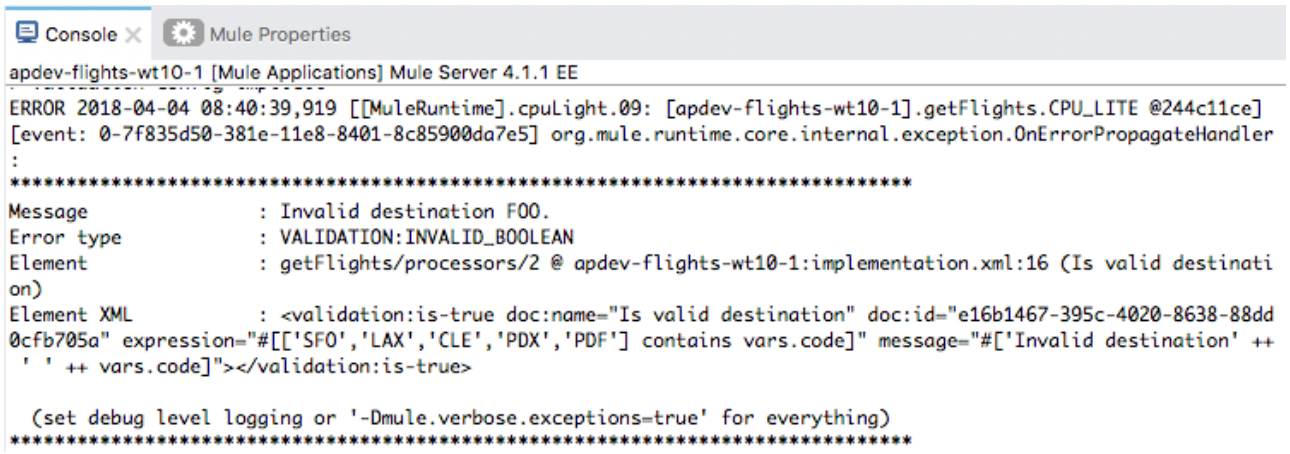
Review a validation error in the Mule Debugger and console

6. In Advanced REST Client, add a code and make a request to http://localhost:8081/flights?code=FOO.

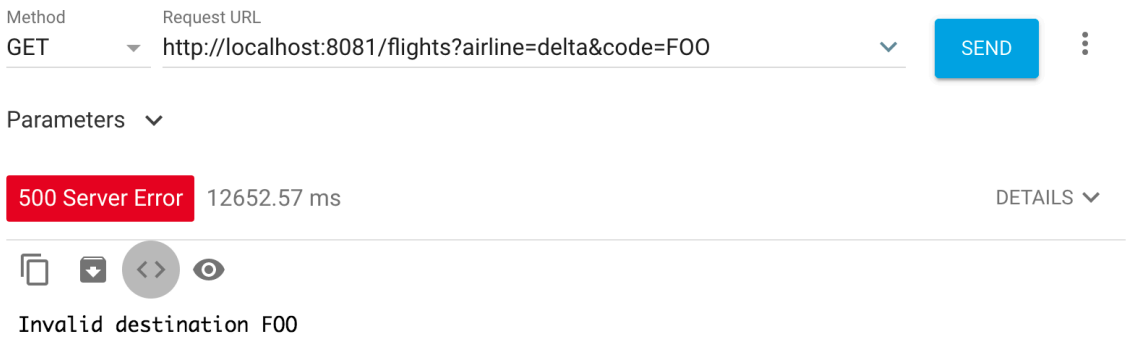
7. In the Mule Debugger, step to where the error is thrown and expand the Exception object; you should see an error description, errorType, and other properties.



8. Step again and review the error information logged in the console.

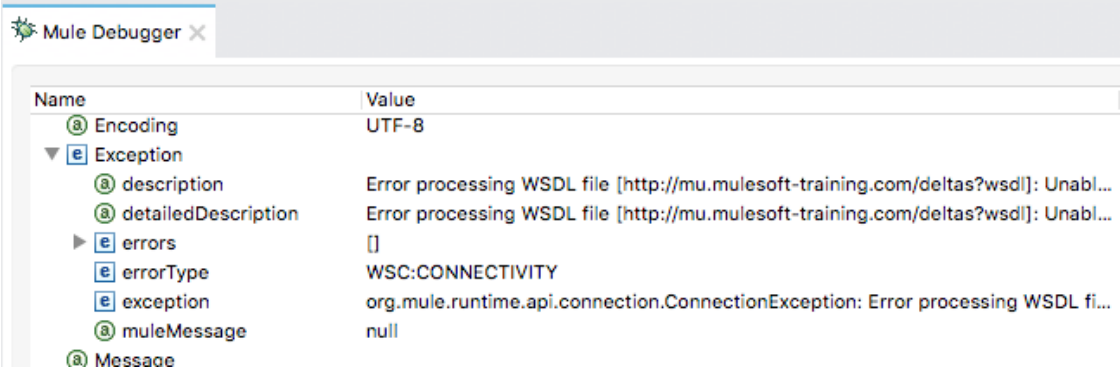


9. Step through the rest of the application.
10. In Advanced REST Client, locate the response status code, status reason, and body; you should see a 500 status code, a status reason of Server Error, and a response body equal to the error description.

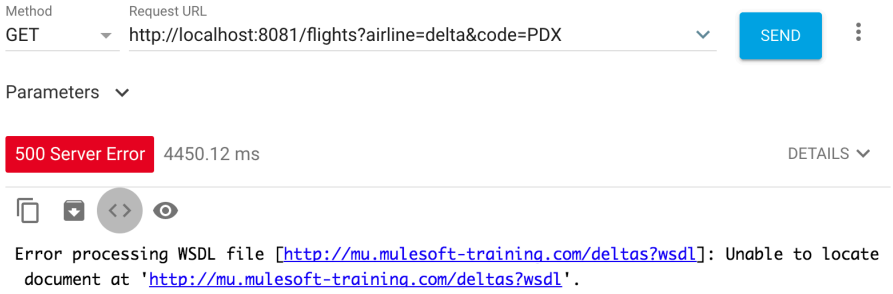


Review a connectivity error in the Mule Debugger and console

11. Add an airline and change the code to make a request to http://localhost:8081/flights?airline=delta&code=PDX.
12. In the Mule Debugger, step to where the error is thrown and review the Exception object.

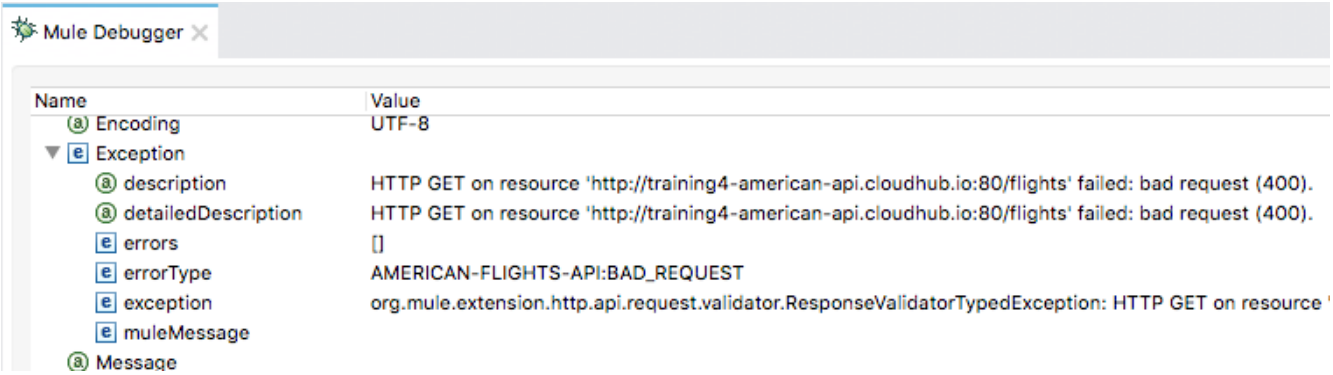


13. Review the error information logged in the console.
14. Step through the rest of the application.
15. Return to Advanced REST Client; you should see a 500 Server Error with the error description.

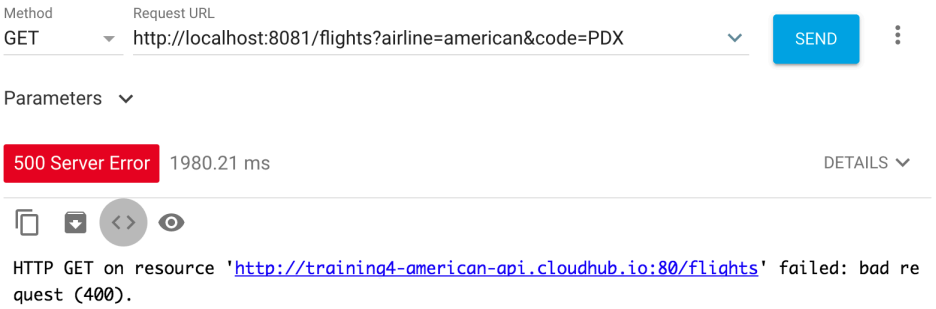


Review a bad request error in the Mule Debugger and console

16. Change the airline to make a request to http://localhost:8081/flights?airline=american&code=PDX.
17. In the Mule Debugger, step to where the error is thrown and review the Exception object.



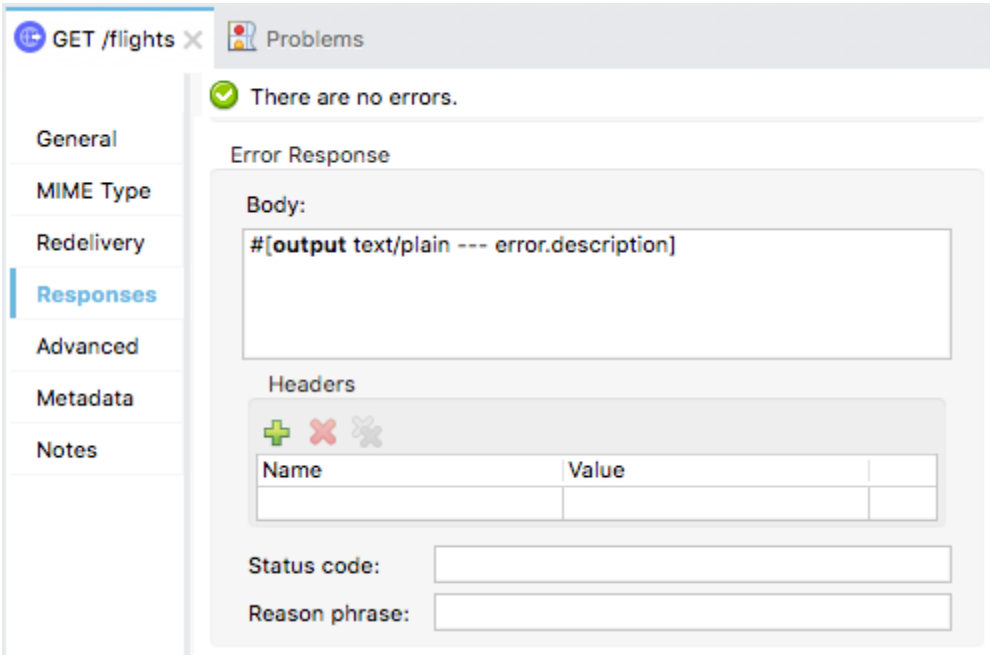
18. Step through the rest of the application.
19. Review the error information logged in the console.
20. Return to Advanced REST Client; you should see a 500 Server Error with the error description.



21. Return to Anypoint Studio and switch to the Mule Design perspective.
22. Stop the project.

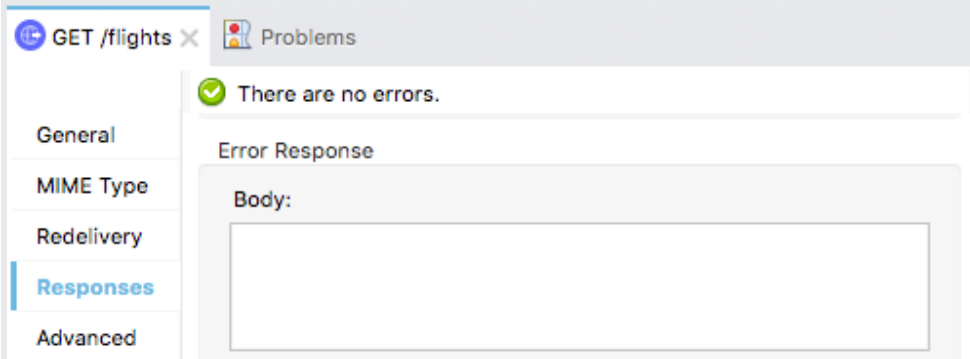
Review the default error response settings for an HTTP Listener

23. Navigate to the properties view for the GET /flights Listener in getFlights.
24. Select the Responses tab.
25. Locate the Error Response section and review the default settings for the body, status code, and reason phrase.



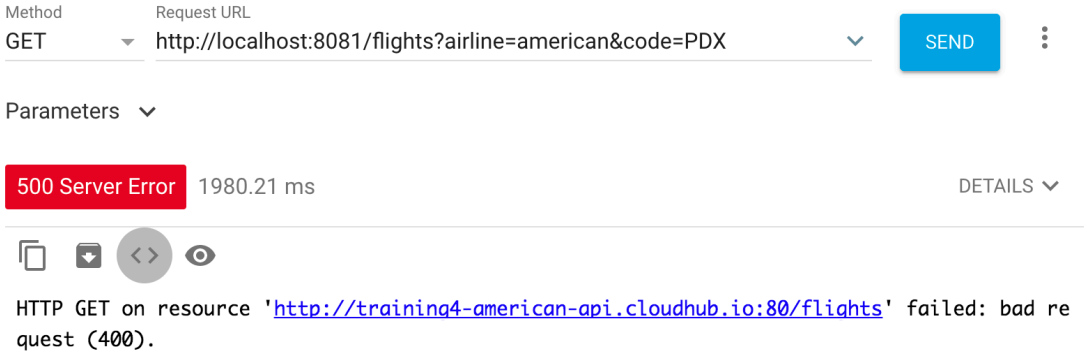
Remove the default error response settings for the HTTP Listener

26. Select and cut the body expression (so it has no value, but you can paste it back later).



Test the application

27. Save the file and run the project.
28. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=PDX; you should get the same response.



Modify the default error response settings for the HTTP Listener

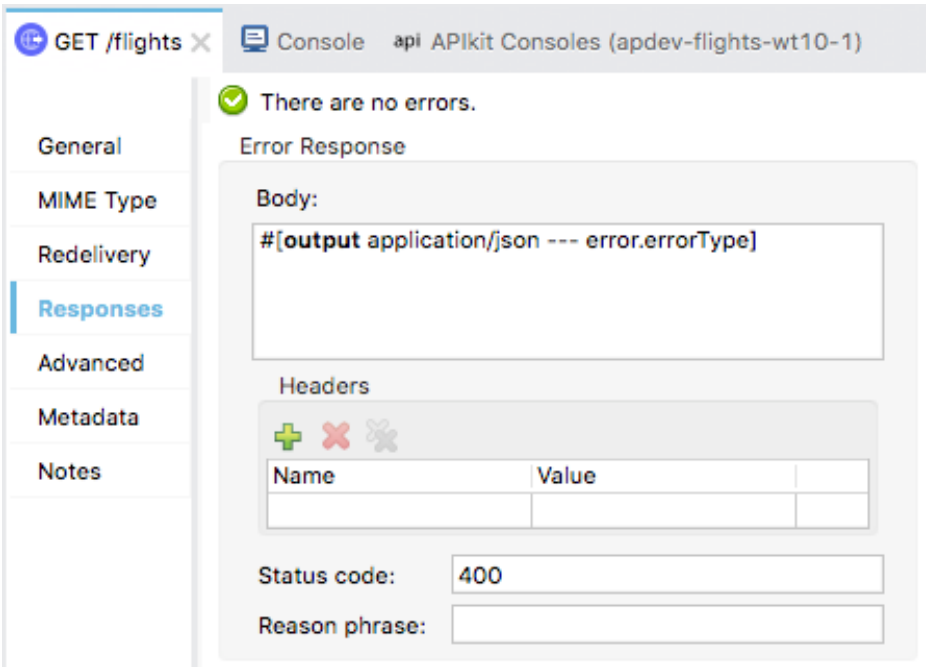
29. Return to Anypoint Studio.
30. Return to the Responses tab in the properties view for the GET /flights Listener.
31. In the error response body, paste back the original value.

```
#[output text/plain --- error.description]
```

32. Change the output type of the error response to application/json and display the error.errorType.

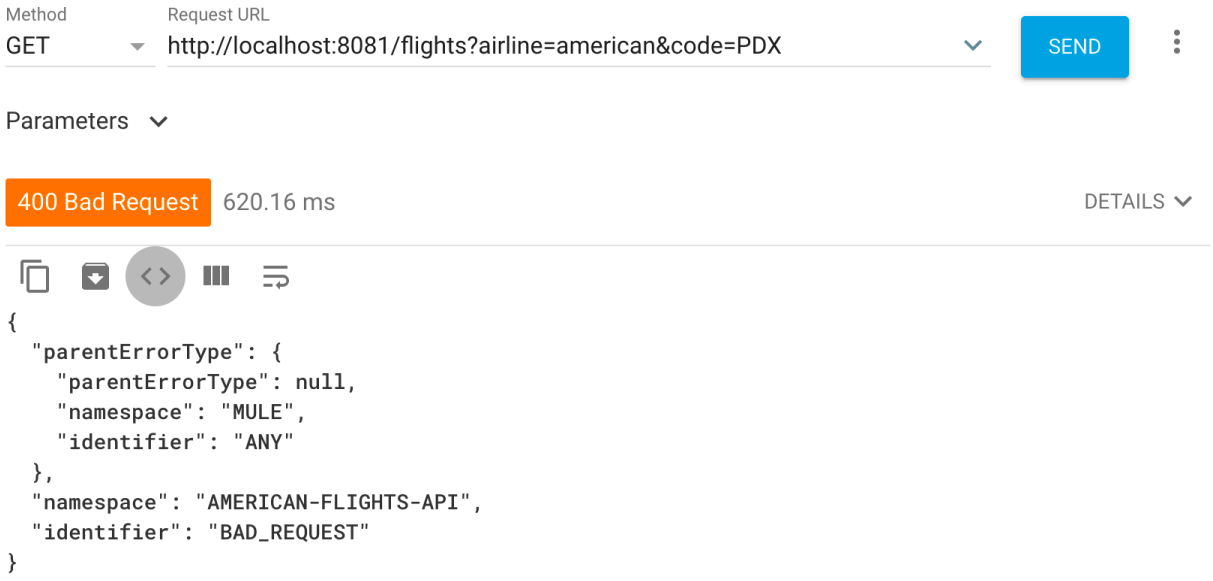
```
#[output application/json --- error.errorType]
```

33. Set the error response status code to 400.



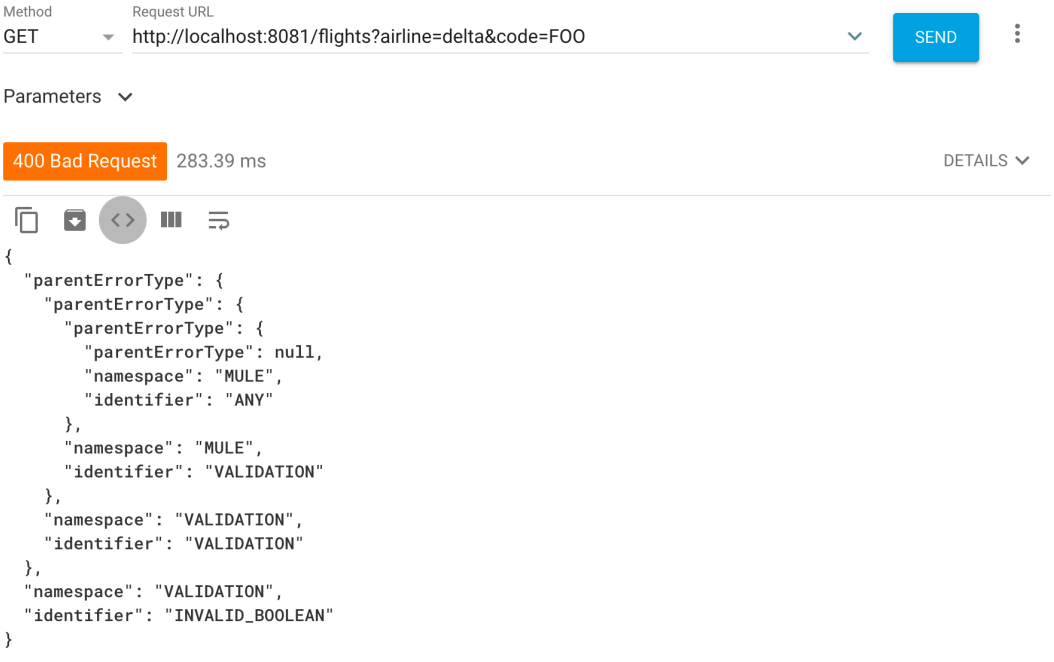
Test the application

34. Save the file to redeploy the application.
35. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=PDX; you should get the new status code and reason and the response body should be the errorType object.
36. Look at the error namespace and identifier.



37. Change the airline to make a request to http://localhost:8081/flights?airline=delta&code=PDX.
38. Look at the error namespace and identifier.

39. Change the code to make a request to http://localhost:8081/flights?airline=delta&code=FOO.
40. Look at the namespace and identifier.



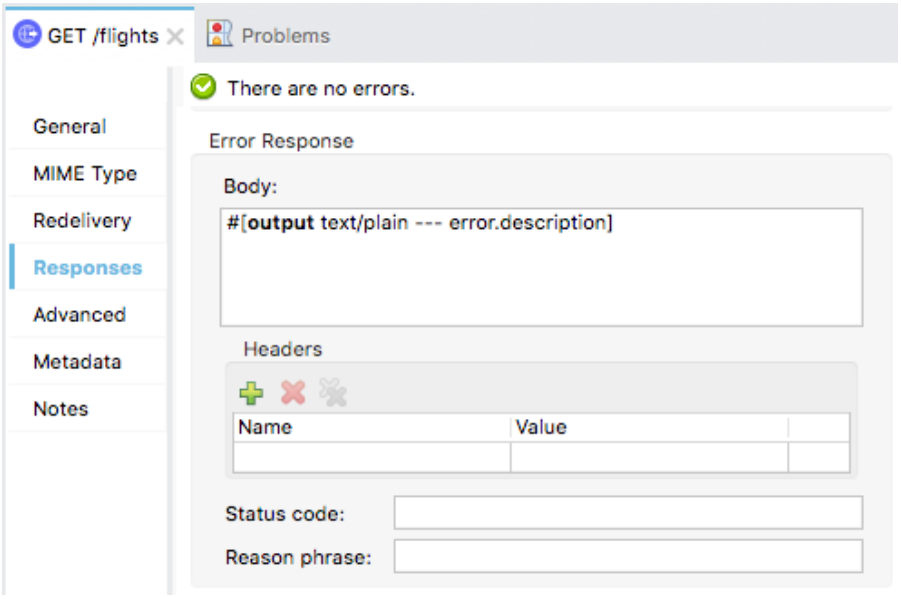
Return the default error response settings for the HTTP Listener

41. Return to Anypoint Studio.
42. Return to the Responses tab in the properties view for the GET /flights Listener.

43. Change the error response output type back to text/plain and display the error.description.

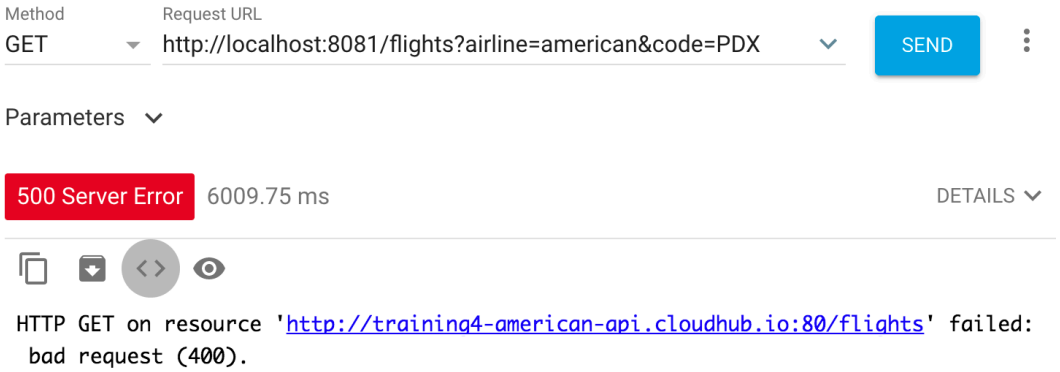
```
#[output text/plain --- error.description]
```

44. Remove the status code.



Test the application

45. Save the file to redeploy the application.
46. In Advanced REST Client, change the code to make a request to http://localhost:8081/flights?airline=american&code=PDX; you should get the 500 Server Error again with the plain text error description.

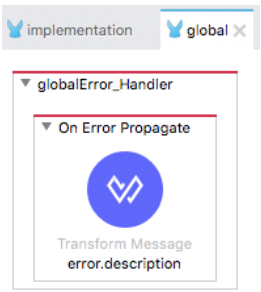


47. Return to Anypoint Studio.
48. Stop the project.

Walkthrough 10-2: Handle errors at the application level

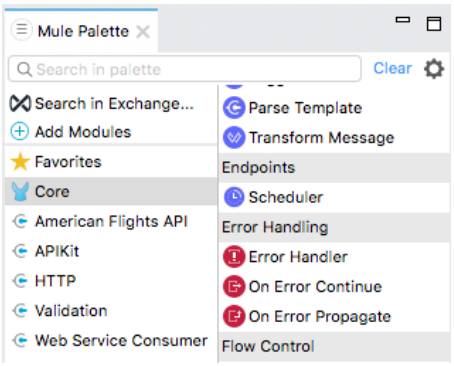
In this walkthrough, you create a default error handler for the application. You will:

- Create a global error handler in an application.
- Configure an application to use a global default error handler.
- Explore the differences between the On Error Continue and On Error Propagate scopes.
- Modify the default error response settings for an HTTP Listener.



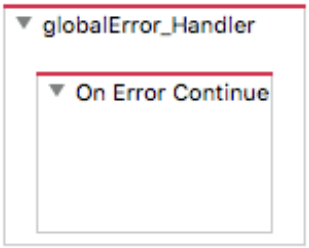
Browse the error handling elements in the Mule Palette

1. Return to global.xml and switch to the Message Flow view.
2. In the Core section of the Mule Palette, locate the Error Handling elements.

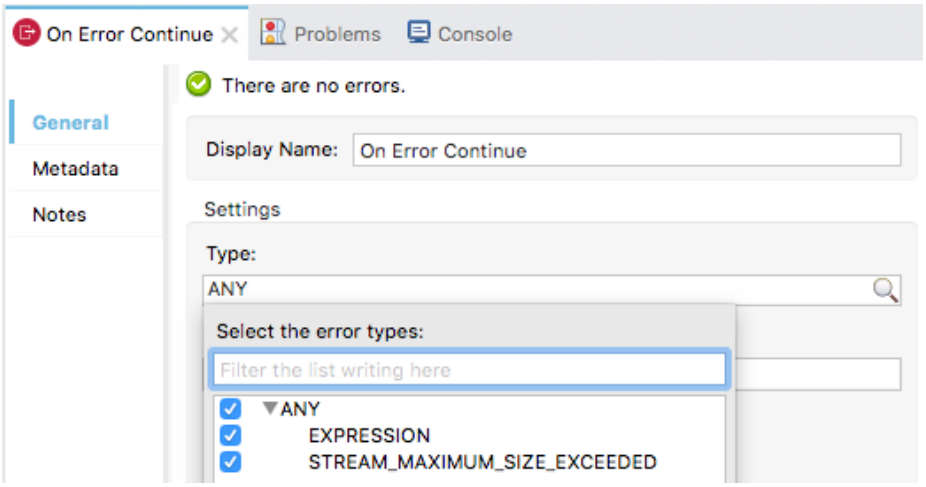


Create a global error handler with an On Error Continue scope

3. Drag out an Error Handler element and drop it in the canvas of global.xml.
4. Drag out an On Error Continue element and drop it in globalError_Handler.

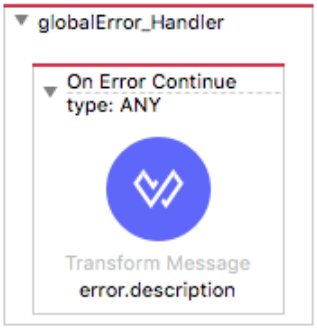


5. In the On Error Continue properties view, click the Search button for type.
6. In the drop-down menu that appears, select ANY.

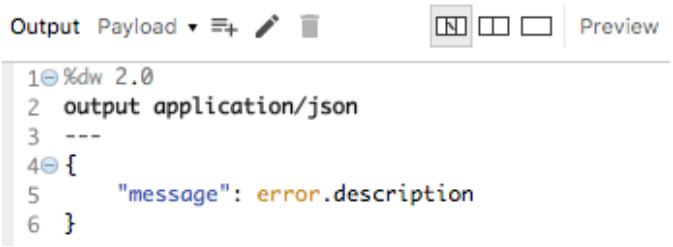


Set the payload in the error handler to a JSON message

7. Add a Transform Message component to the On Error Continue.
8. Set the Transform Message display name to error.description.



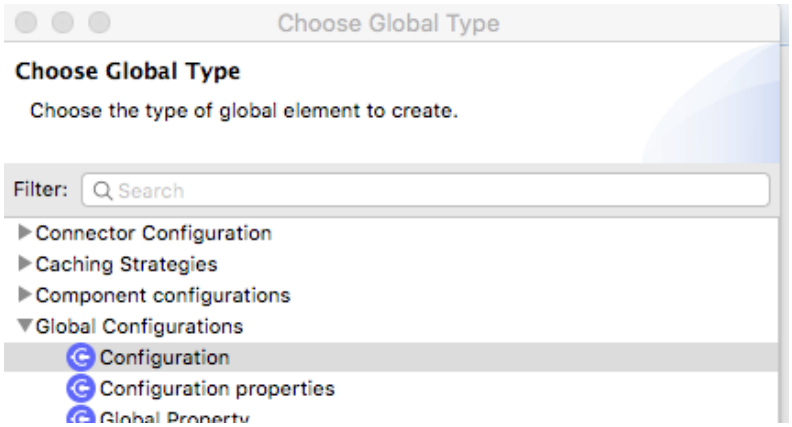
9. In the Transform Message properties view, change the output type to application/json.
10. Add a message property to the output JSON and give it a value of the error.description.



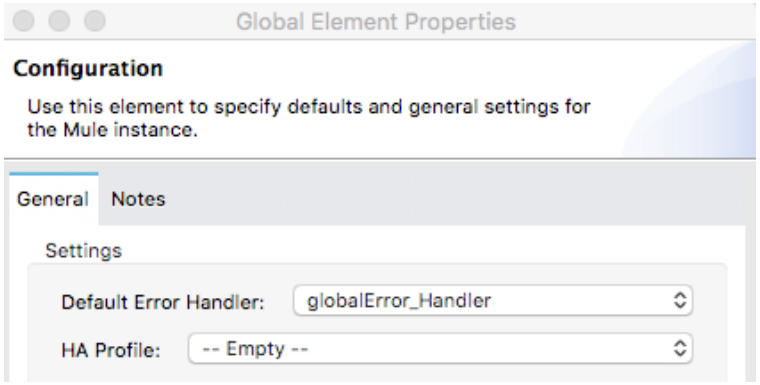
Set a default error handler for the application

11. Switch to the Global Elements view of global.xml.
12. Click Create.

13. In the Choose Global Type dialog box, select Global Configurations > Configuration and click OK.



14. In the Global Element Properties dialog box, set the default error handler to globalError_Handler and click OK.

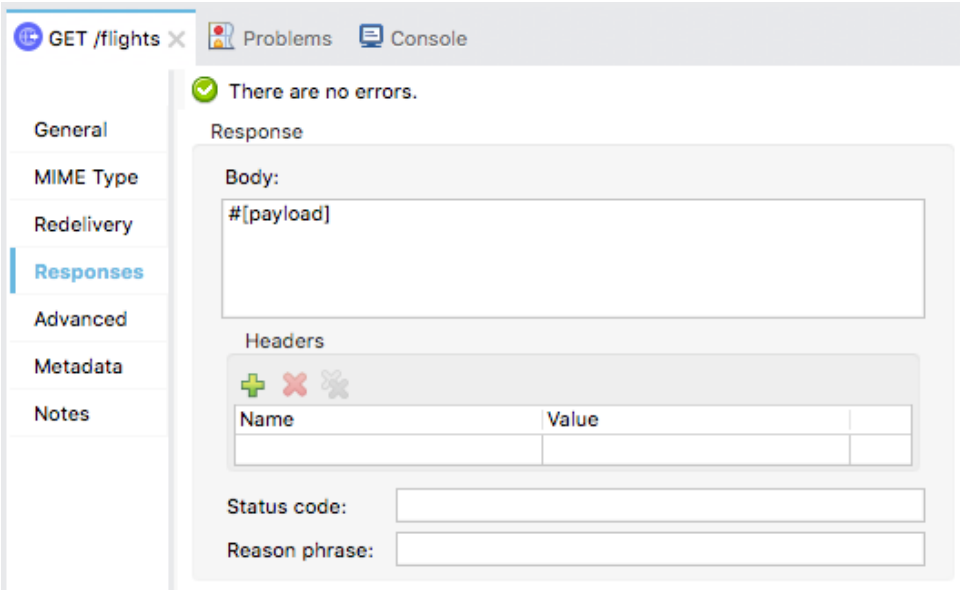


15. Switch to the Message Flow view.

Review the default response settings for an HTTP Listener

16. Return to implementation.xml.
17. Navigate to the properties view for the GET /flights Listener in getFlights.
18. Select the Responses tab.

19. Locate the Response section (not the Error Response section) and review the default settings for the body, status code, and reason phrase.



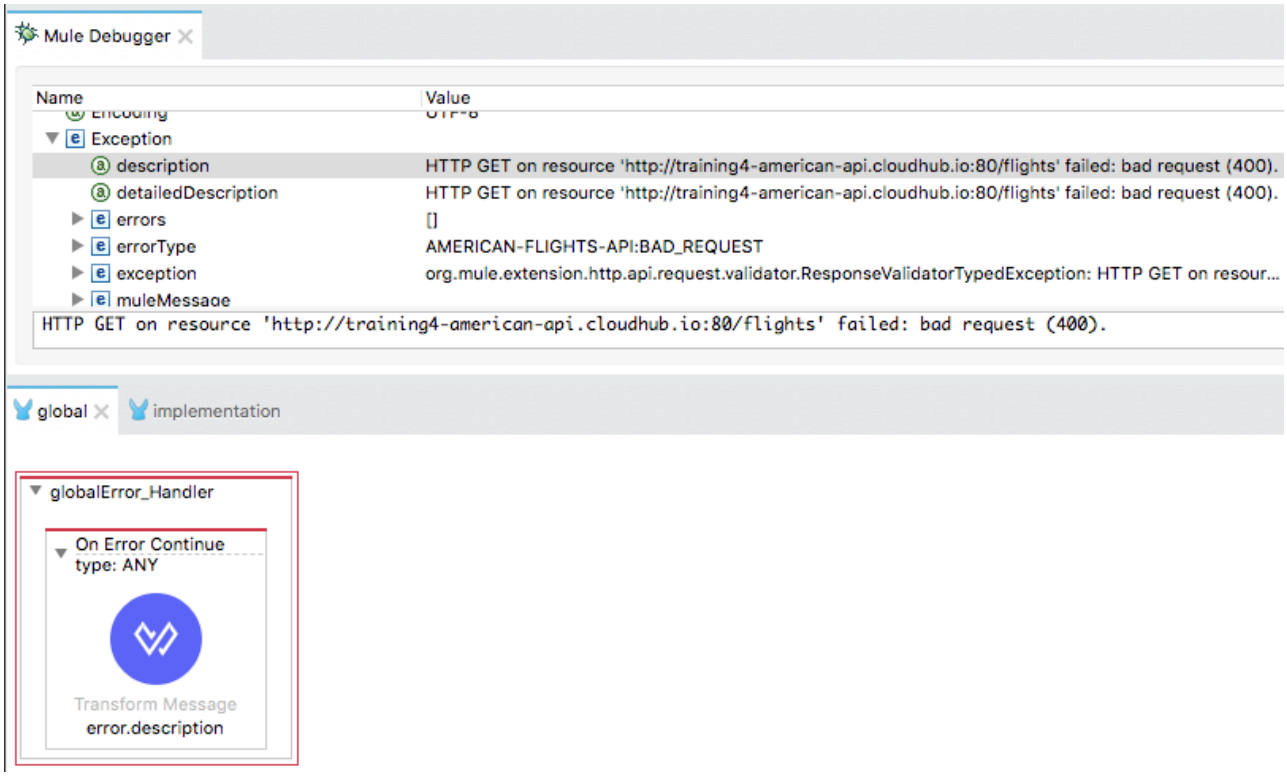
20. Locate the Error Response section and review the default settings for the body, status code, and reason phrase.

Test the On Error Continue behavior

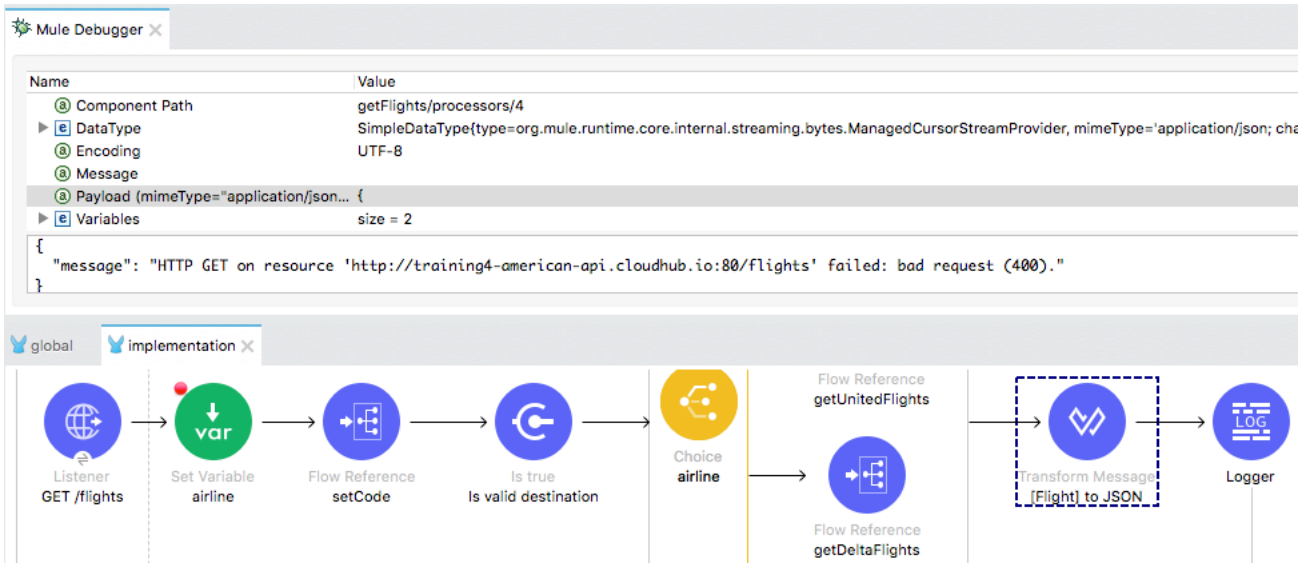
21. Save all files and debug the project.
22. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=PDX.
23. In the Mule Debugger, step through the application until the event is passed to globalError_Handler.

    Note: In Anypoint Studio 7.1.X, focus will switch to global.xml but you will not see the application execution step through an individual error handler.

24. In the Mule Debugger, locate and expand Exception.



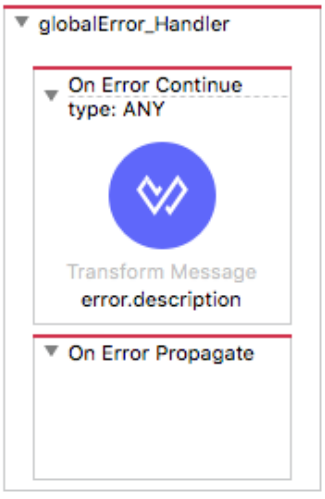
25. Step again; the event should be passed back to getFlights, which continues to execute after the error occurs.
26. In the Mule Debugger, look at the payload and the absence of an exception object in the Mule Debugger.



27. Step through the rest of the application.
28. Return to Advanced REST Client; you should see get a 200 OK response with the response body equal to the payload value set in the error handler.

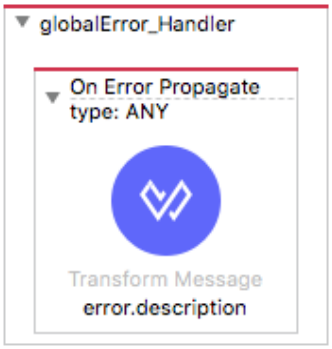
Change the default error handler to use an On Error Propagate scope

29. Return to Anypoint Studio and switch to the Mule Design perspective.
30. Return to global.xml.
31. Drag an On Error Propagate element from the Mule Palette and drop it to the right or left of the On Error Continue to add it to globalError_Handler.



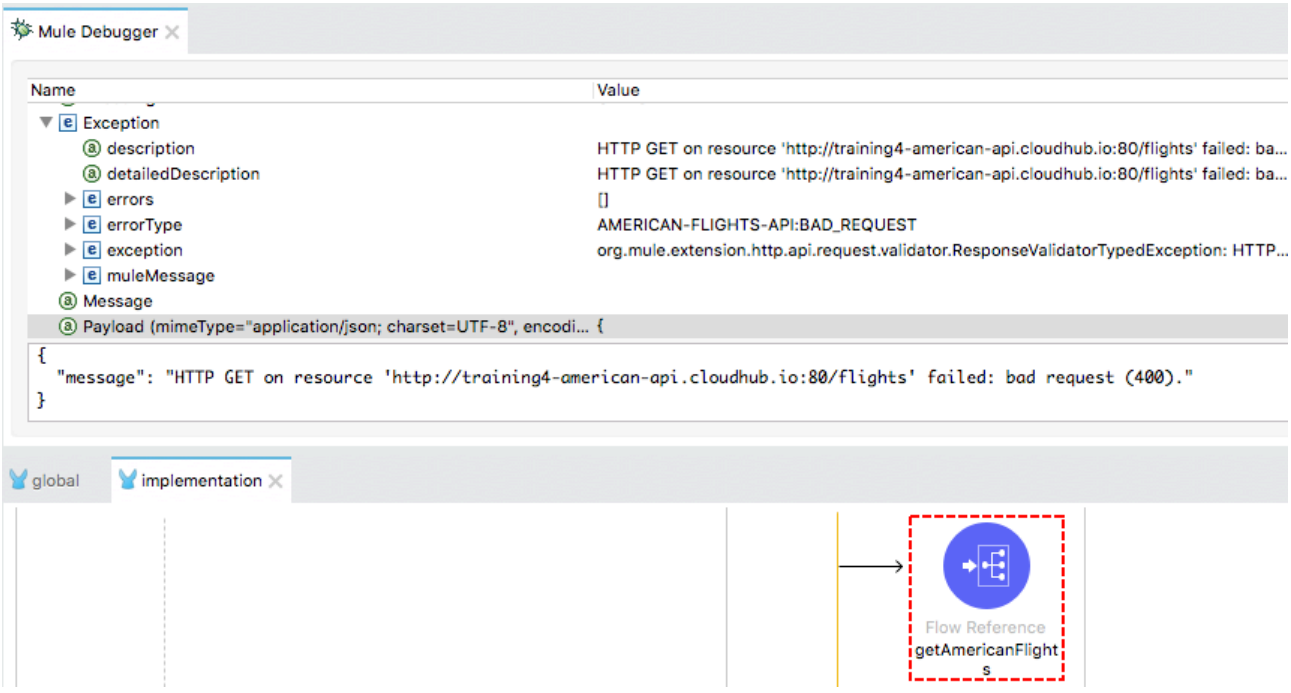
32. In the On Error Propagate properties view, set the type to ANY.
33. Move the Transform Message component from the On Error Continue to the On Error Propagate.

34. Delete the On Error Continue scope.



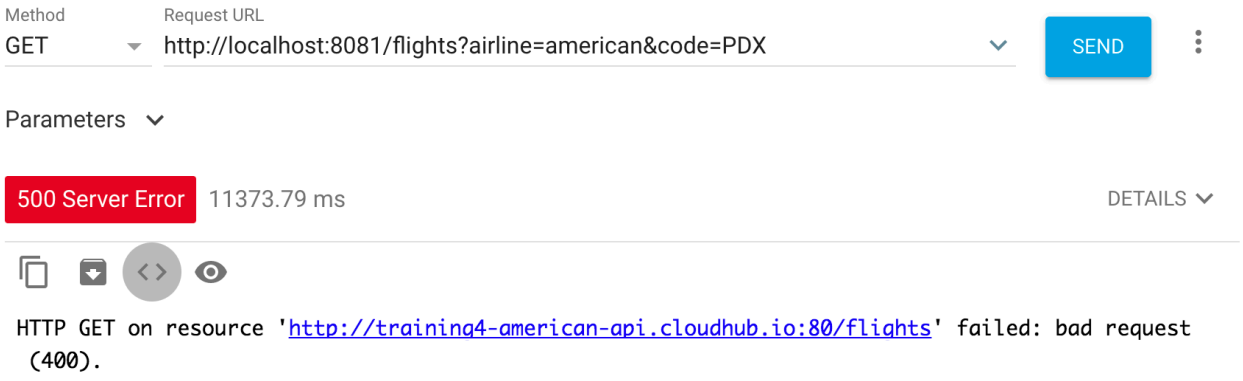
Test the On Error Propagate behavior

35. Save the files to redeploy the project.
36. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=PDX.
37. Step through the application until the event is passed to globalError_Handler.
38. Look at the payload and the exception in the Mule Debugger; they should be the same as before.
39. Step again; the error is propagated up to the getFlights parent flow.
40. Look at the payload and the exception in the Mule Debugger.



41. Step again; the error is handled by the application's default error handler again.

42. Look at the payload and the exception in the Mule Debugger.
43. Step through the rest of the application.
44. Return to Advanced REST Client; you should see get a 500 Server Error response with the response body equal to the plain text error description – not the payload.

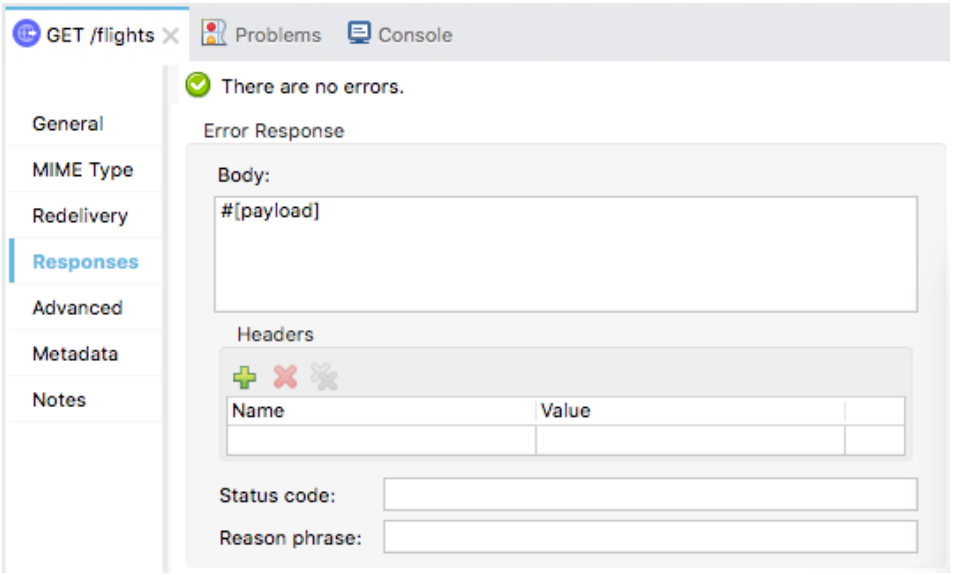


45. Return to Anypoint Studio and switch to the Mule Design perspective.
46. Stop the project.

Modify the default error response settings for the HTTP Listener

47. Return to implementation.xml.
48. Navigate to the Responses tab in the properties view for the GET /flights Listener.
49. Change the error response body to return the payload instead of error.description.

```
#[payload]
```



Test the application

50. Save the file and run the project.
51. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=PDX; you should now get the message that you set in the error handler.

Note: You will handle this error differently in a later walkthrough, so it does not return a server error. It is a valid request; there are just no American flights to PDX.

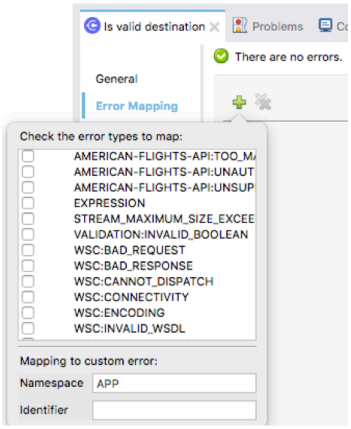
52. Change the airline to make a request to http://localhost:8081/flights?airline=delta&code=PDX; the error is handled by the same default handler.

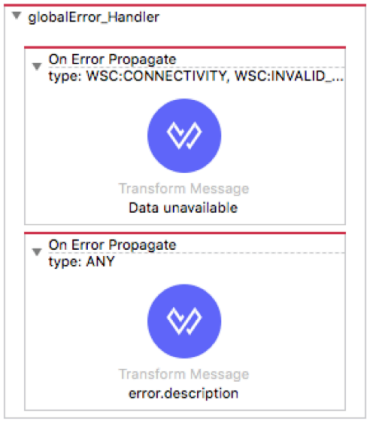
53. Return to Anypoint Studio.

Walkthrough 10-3: Handle specific types of errors

In this walkthrough, you continue to work with the application's default error handler. You will:

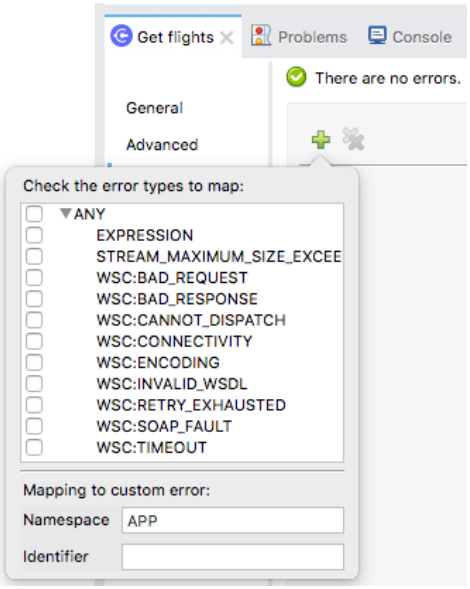
- Review the possible types of errors thrown by different processors.
- Create error handler scopes to handle different error types.





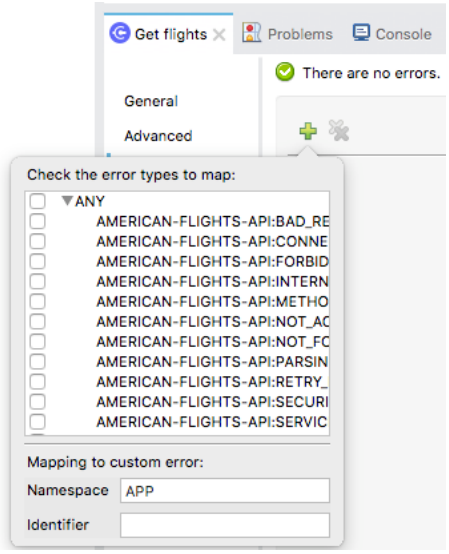
Review the possible types of errors thrown by a Web Service Consumer operation

1. Return to implementation.xml.
2. Navigate to the properties view for the Get flights Consume operation in getDeltaFlights.
3. Select the Error Mapping tab.
4. Click the Add new mapping button and review (but don't select!) the WSC error types.



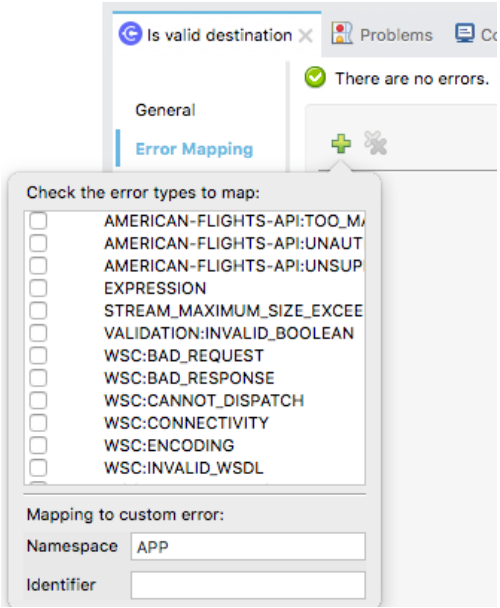
Review the possible types of errors thrown by a REST Connector operator

5. Navigate to the properties view for the Get flights operation in getAmericanFlights.
6. Select the Error Mapping tab.
7. Click the Add new mapping button, and review (but don't select!) the AMERICAN-FLIGHTS-API error types.



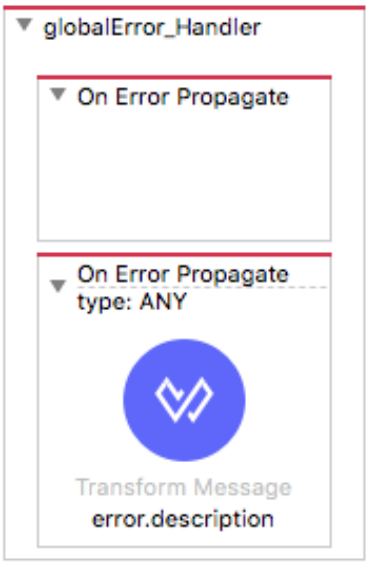
Review the possible types of errors thrown by a Validator operation

8. Navigate to the properties view for the Is true validator in getFlights.
9. Select the Error Mapping tab.
10. Click the Add new mapping button and locate (but don't select!) the VALIDATION error type.

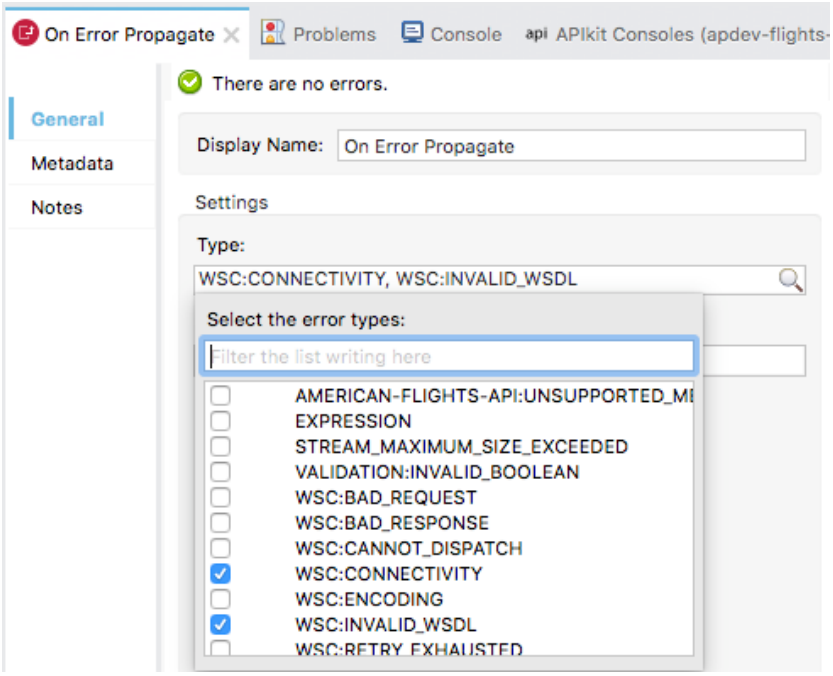


Add a second error handler to catch Web Service Consumer connectivity errors

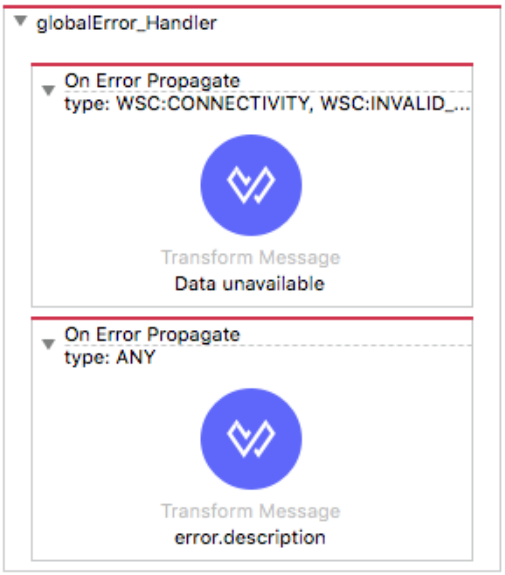
11. Return to global.xml.
12. Add a second On Error Propagate to globalError_Handler.
13. If necessary, drag and drop it so it is first scope inside the error handler.



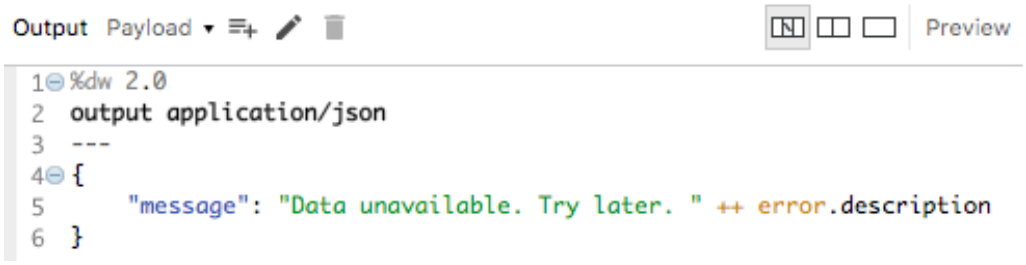
14. In the properties view for the new On Error Propagate, click the Search button.
15. Select WSC:CONNECTIVITY and WSC:INVALID_WSDL.



16. Add a Transform Message component to the new On Error Propagate and set its display name to Data unavailable.



17. In the Transform Message properties view, change the output type to application/json.
18. Add a message property to the output JSON and set give it a value of “Data unavailable. Try later.”.
19. For development purposes, concatenate the error.description to the message; be sure to use the auto-completion menu.



Test the application

20. Save the file to redeploy the project.

21. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=delta&code=PDX; you should now get a 500 Server Error response with your new Data unavailable message.

22. Change the code to make a request to http://localhost:8081/flights?airline=delta&code=FOO; you should still get a 500 Server Error response with the Invalid destination JSON message.

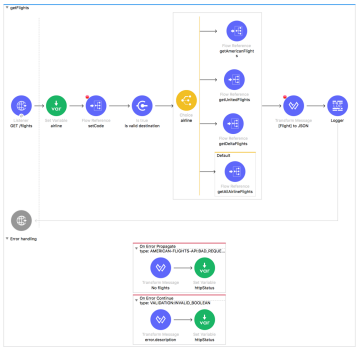
    Note: This should also not be a server error, but either a 4XX bad request or a 200 OK with an appropriate message. You will handle this error differently in the next walkthrough.

23. Return to Anypoint Studio.
24. Stop the project.

Walkthrough 10-4: Handle errors at the flow level

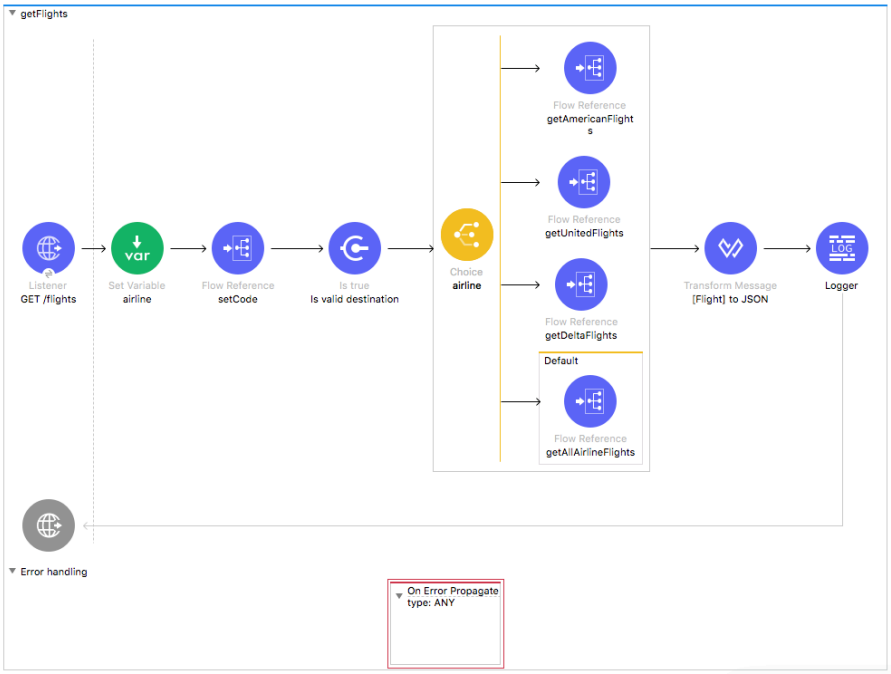
In this walkthrough, you add multiple error handlers to a flow. You will:

- Add error handlers to a flow.
- Test the behavior of errors thrown in a flow and by a child flow.
- Compare On Error Propagate and On Error Continue scopes in a flow.
- Set an HTTP status code in an error handler and modify an HTTP Listener to return it.



Add an On Error Propagate error handler to a flow

1. Return to implementation.xml.
2. Expand the Error handling section of the getFlights flow.
3. Add an On Error Propagate scope.
4. Set the error type to ANY so it will initially catch both the validation and American flights errors.



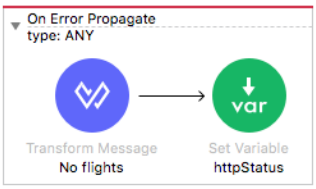
Set the payload in the error handler to the error description

5. Add a Transform Message component to the scope and set the display name to No flights.
6. In the Transform Message properties view, set the payload to a JSON message property equal to the string No flights to and concatenate the code variable.
7. Coerce the variable to a String.

```
%dw 2.0
output application/json
---
{
    "message": "No flights to " ++ vars.code as String
}
```

Set the HTTP status code in the error handler to 200

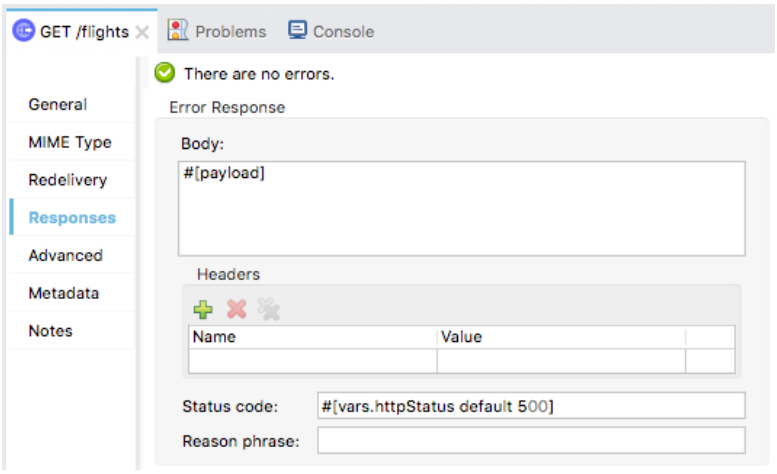
8. Add a Set Variable transformer to the On Error Propagate.
9. In the Set Variable properties view, set the display name and name to httpStatus.
10. Set the value to 200.



Set the HTTP Listener error response status code to use a variable

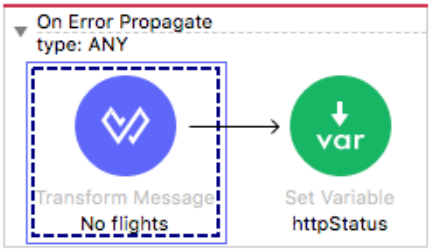
11. Navigate to the Responses tab in the properties view for the GET /flights Listener.
12. Change the error response status code (not the response status code!) to an expression that sets it equal to the value of an httpStatus variable with a default value of 500.

```
#[vars.httpStatus default 500]
```

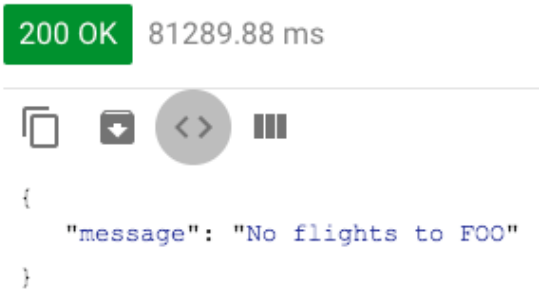


Test the flow's On Error Propagate with an error in the flow

13. Save the file and debug the project.
14. In Advanced REST Client, change the airline to make a request to http://localhost:8081/flights?airline=american&code=FOO.
15. In the Mule Debugger, step until the event is passed to the flow's error handler.



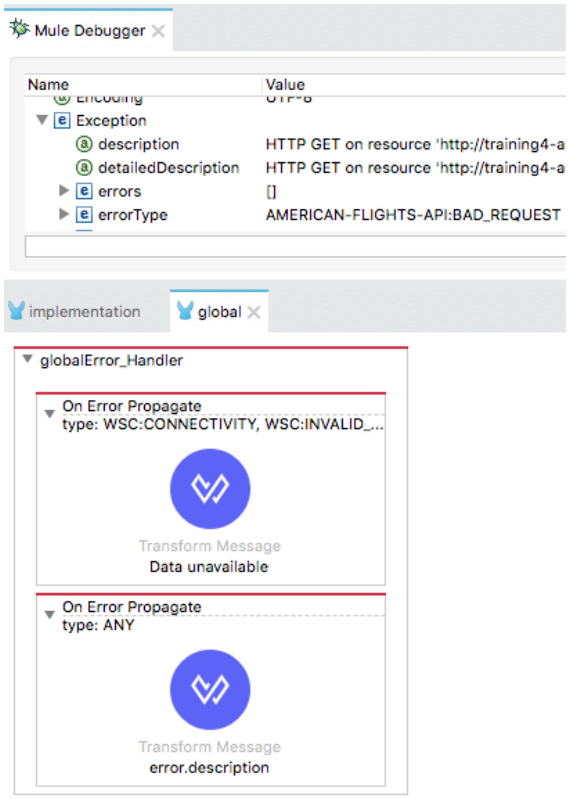
16. Step through the rest of the application.
17. Return to Advanced REST Client; you should get the 200 status code and the JSON message.



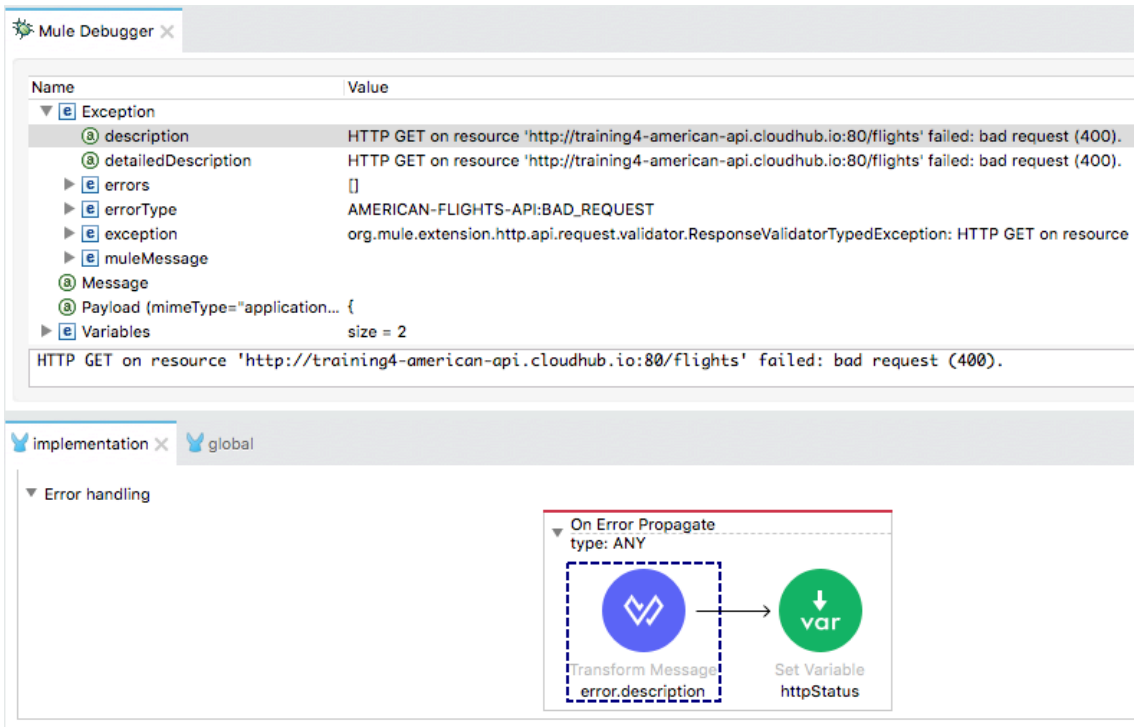
Test the flow's On Error Propagate with an error in a child flow

18. In Advanced REST Client, change the code to make a request to http://localhost:8081/flights?airline=american&code=PDX.

19. Step until the event is passed to the globalError_Handler.

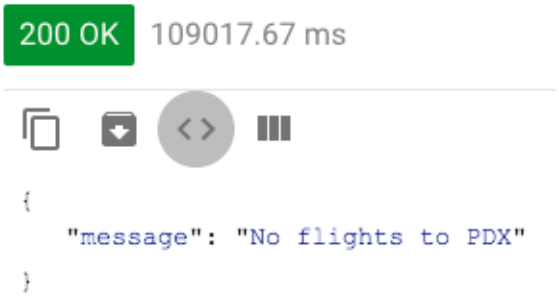


20. Step until the event is passed to the parent flow's error handler.



21. Step through the rest of the application.

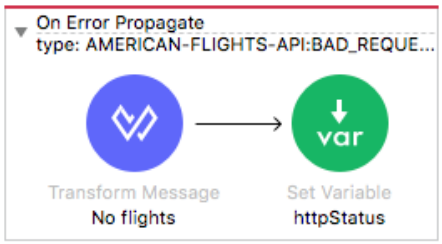
22. Return to Advanced REST Client; you should get the 200 status code and the JSON error message.



23. Return to Anypoint Studio and switch to the Mule design perspective.

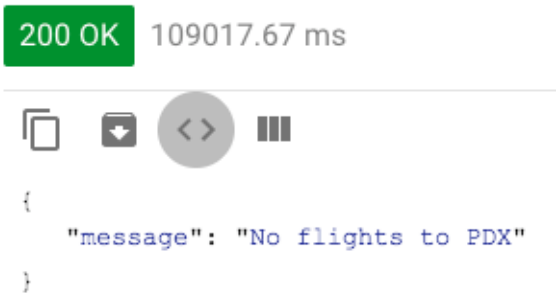
Specify the type of error to be handled by the flow's error handler

24. Navigate to the properties view for the On Error Propagate in getFlights.
25. Change the type from ANY to AMERICAN-FLIGHTS-API:BAD_REQUEST.

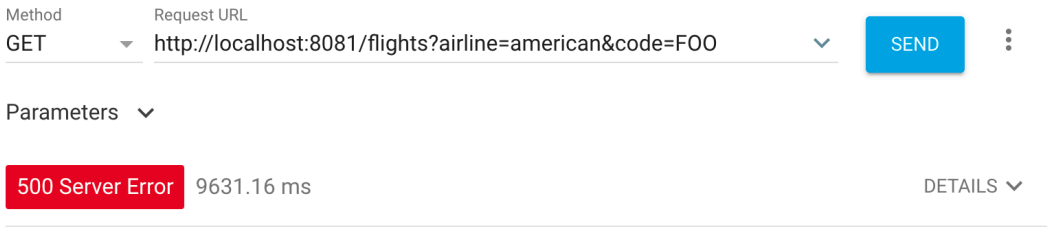


Test the application

26. Save the file to redeploy the project.
27. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=PDX.
28. In the Mule Debugger, step through the application; the error should still be handled by globalError_Handler and then the getFlights flow error handler.



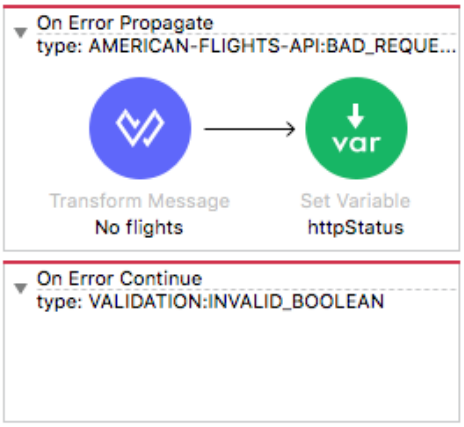
29. In Advanced REST Client, change the code to make a request to http://localhost:8081/flights?airline=american&code=FOO.
30. In the Mule Debugger, step through the application; you should get a 500 Server Error and no message because the error is not handled by the getFlights flow error handler or by globalError_handler.



31. Return to Anypoint Studio and switch to the Mule Design perspective.
32. Stop the project.
33. Navigate to the Responses tab in the properties view for the GET /flights Listener.
34. Review the error response body (not the response status code!) and ensure you know why you got the last response for http://localhost:8081/flights?airline=american&code=FOO.

Use an On Error Continue error handler in a flow

35. Add an On Error Continue scope to getFlights.
36. In the On Error Continue properties view, set the type to VALIDATION:INVALID_BOOLEAN.

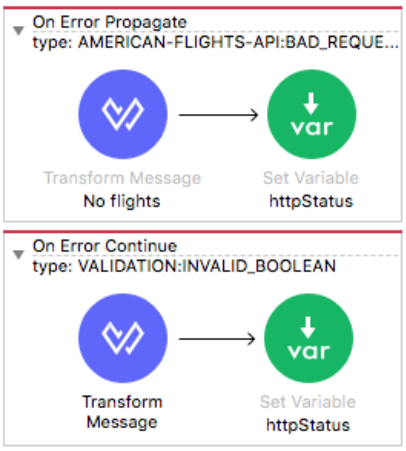


37. Add a Transform Message component to the scope and set the display name to error.description.

38. In the Transform Message properties view, set the payload to a JSON message property equal to the error.description.

```
%dw 2.0
output application/json
---
{
    "message": error.description
}
```

39. Add a Set Variable transformer to the On Error Continue scope.
40. Set the display name and name to httpStatus and set the value to 400.



Test the application

41. Save the file and run the project.
42. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=FOO; you should get a 200 response and the invalid destination message again – not the 400 response.

Set the HTTP Listener response status code

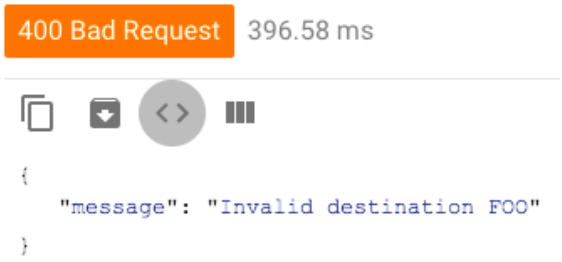
43. Return to Anypoint Studio.
44. Navigate to the Responses tab in the properties view for the GET /flights Listener.

45. Set the response status code (not the error response status code!) to an expression that sets it equal to the value of an httpStatus variable with a default value of 200.

```
#[vars.httpStatus default 200]
```

Test the application

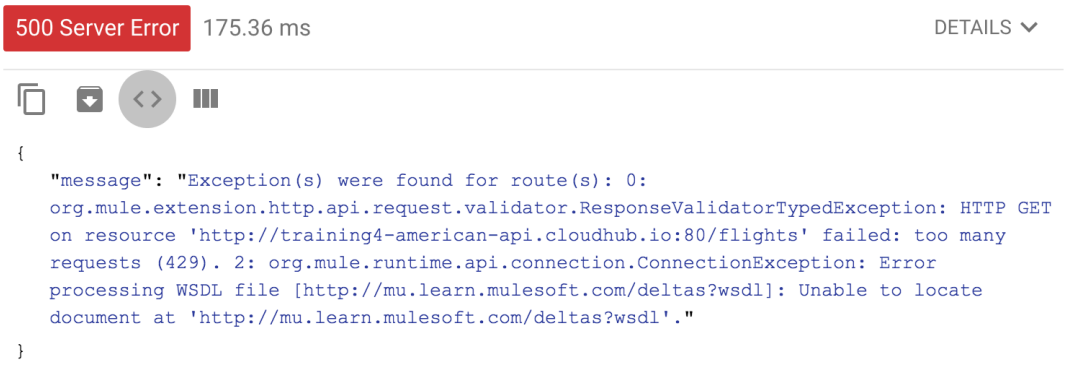
46. Save the file to redeploy the project.
47. In Advanced REST Client, make another request to http://localhost:8081/flights?airline=american&code=FOO; you should now get the 400 response.



Test the application to see what happens with the Scatter-Gather

48. In Advanced REST Client, change the code and make a request to http://localhost:8081/flights?airline=american&code=PDX; you should get a 200 response and no flights to PDX.
49. Change the airline to make a request to http://localhost:8081/flights?airline=united&code=PDX; you should get flights to PDX.

50. Remove the airline to make a request to http://localhost:8081/flights?code=PDX; you should get flights to PDX because United has flights, but you get none.

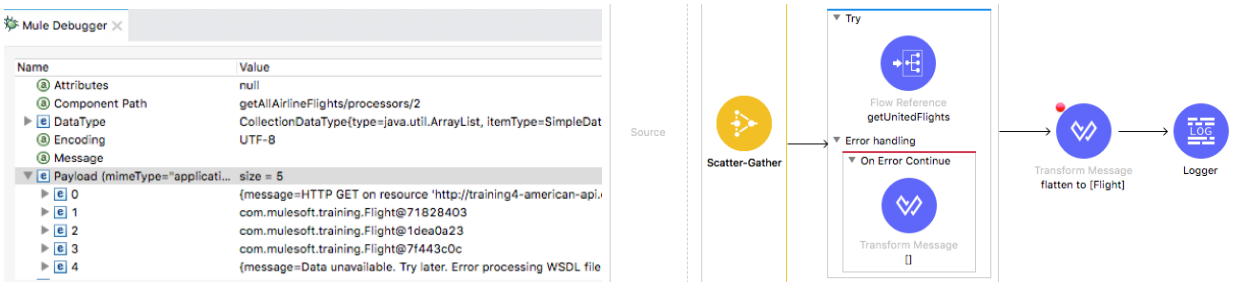


51. Return to Anypoint Studio.
52. Stop the project.

Walkthrough 10-5: Handle errors at the processor level

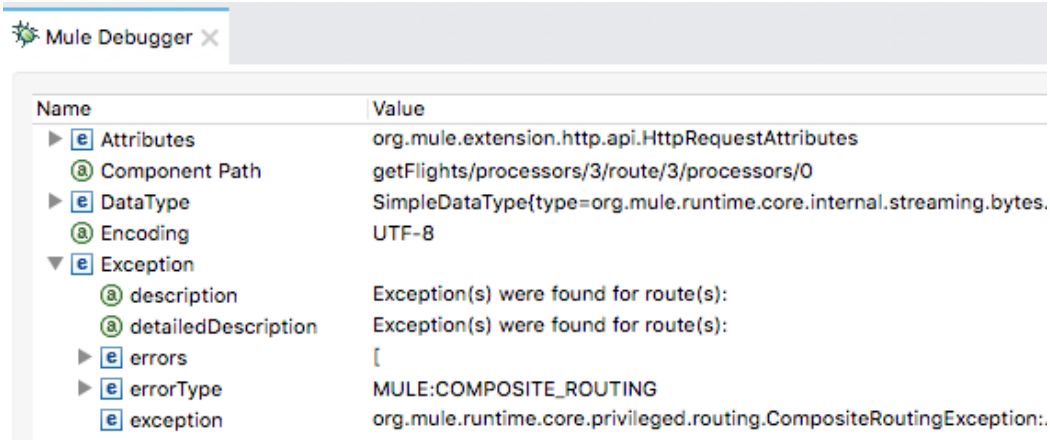
In this walkthrough, you work with the Scatter-Gather in getAllAirlineFlights so that it correctly returns results when one but not all airlines have flights. You will:

- Wrap the Flow Reference in each branch of a Scatter-Gather in a Try scope.
- Use an On Error Continue scope in each Try scope error handler to provide a valid response so flow execution can continue.



Debug the application when a Scatter-Gather branch has an error

1. Return to implementation.xml in Anypoint Studio.
2. Debug the project.
3. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX.
4. Return to the Mule Debugger and step through the application; you should see a routing exception.

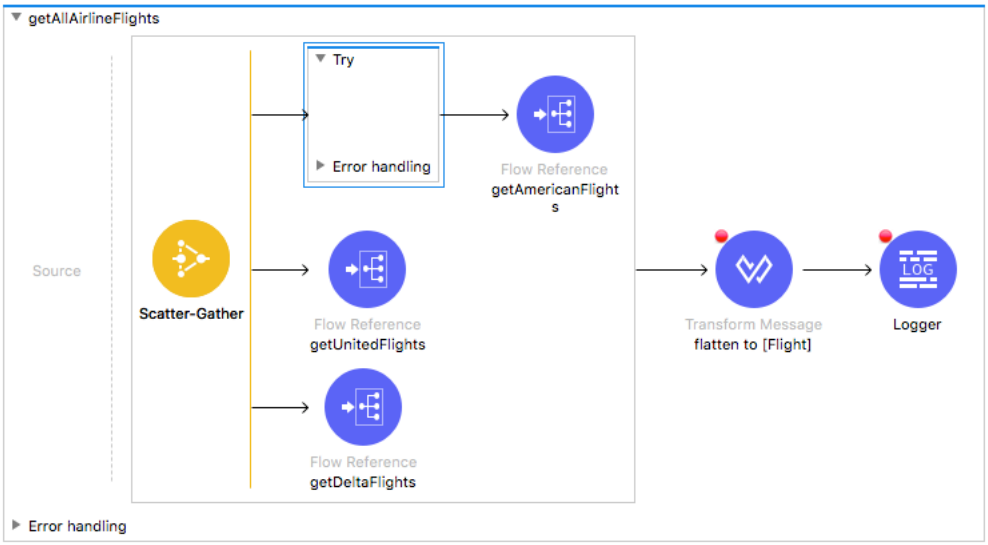


5. Return to Advanced REST client, you should get the 500 Server Error that there were exceptions in routes 0 and 2.

6. Return to Anypoint Studio and switch to the Mule Design perspective.

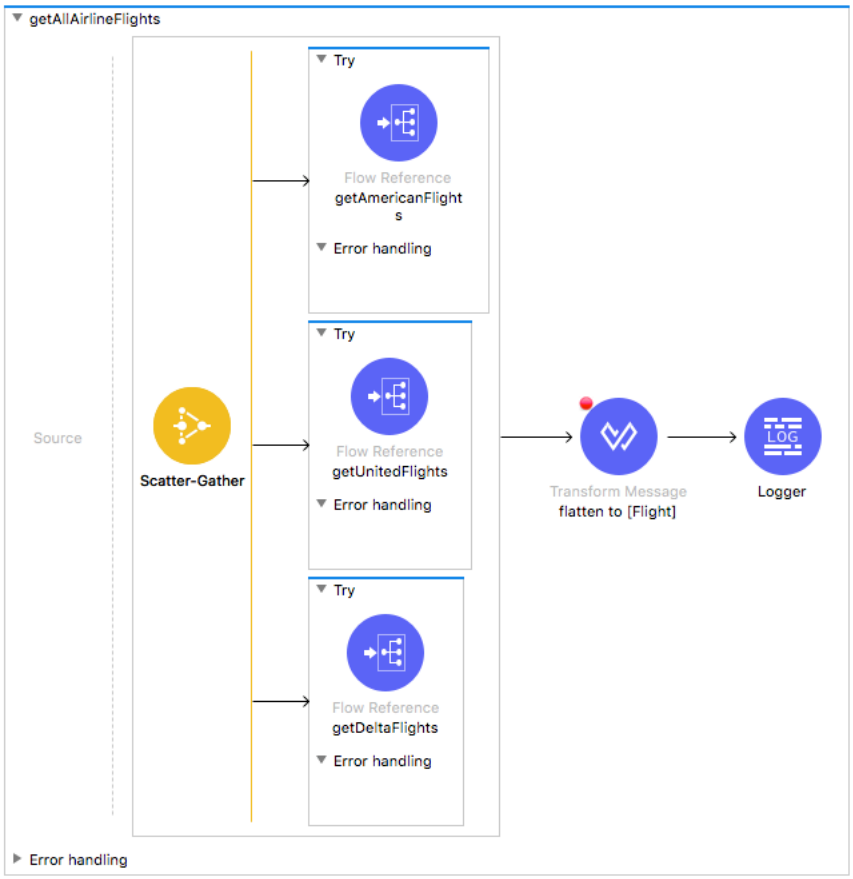
Place each Flow Reference in the Scatter-Gather in a Try scope

7. Locate getAllAirlineFlights.
8. Drag a Try scope from the Mule Palette and drop it in the Scatter-Gather.



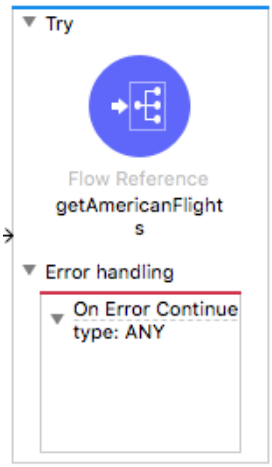
9. Drag the getAmericanFlights Flow Reference into the Try scope.
10. Add two more Try scopes to the Scatter-Gather.

11. Move the getUnitedFlights Flow Reference into one of the Try scopes and the getDeltaFlights Flow Reference into the other.



Add an error handler to each branch that passes through the error message

12. Expand the error handling section of the first Try scope.
13. Add an On Error Continue scope and set the types of errors it handles to ANY.
14. Repeat these steps to add the same error handler to the two other Try scopes.

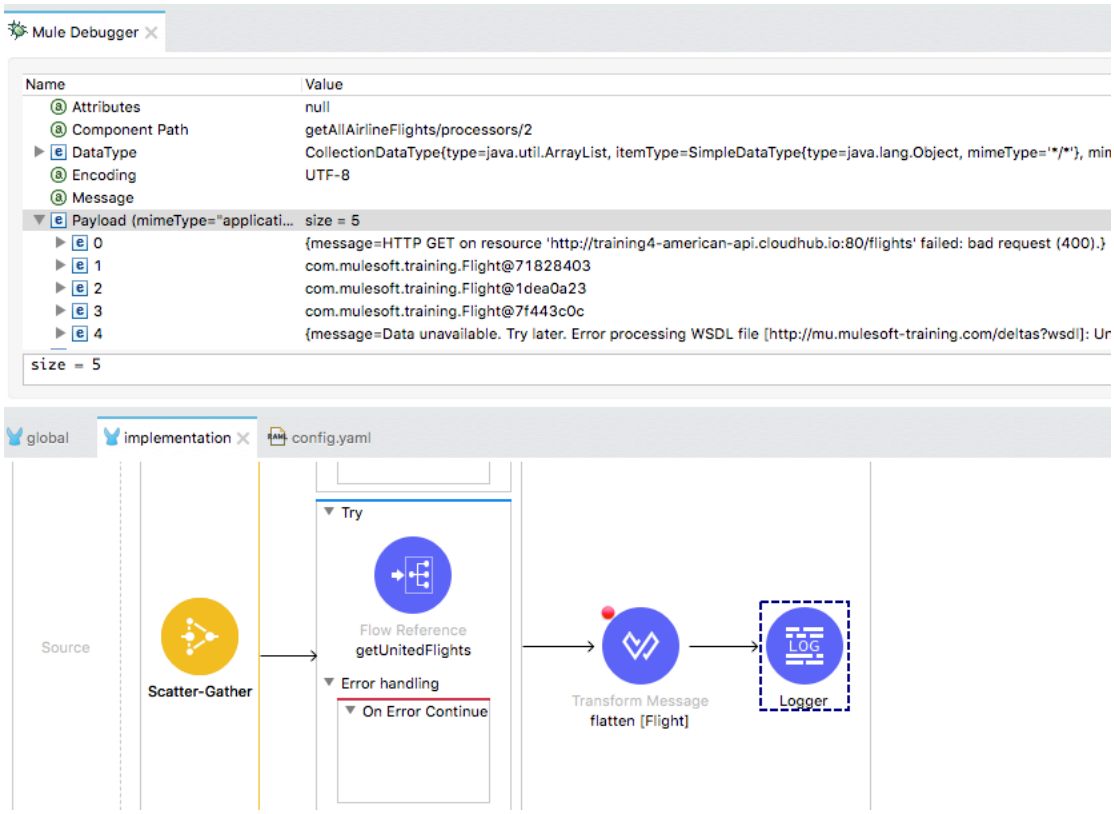


Debug the application

15. Save the file to redeploy the project.

    Note: If you get an error when deploying the project about a duplicate flow-ref name attribute, review the error and then switch to the Configuration XML view to fix it.

16. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX.
17. In the Mule Debugger, step through the application until you are at the Logger after the Scatter-Gather in getAllAirlineFlights.
18. Expand Payload; you should see three flights (from United) and two error messages.

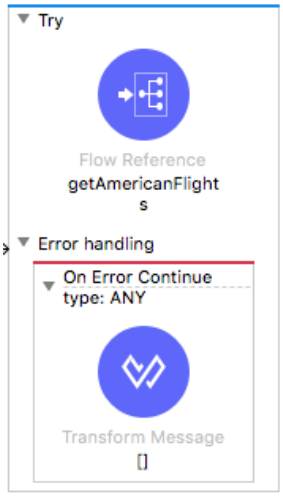


19. Step through the rest of the application.

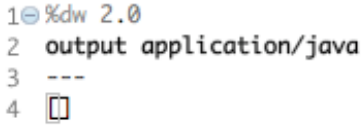
20. Return to Advanced REST Client; you should see both United flights and error messages.

Modify each error handler to set the payload to an empty array

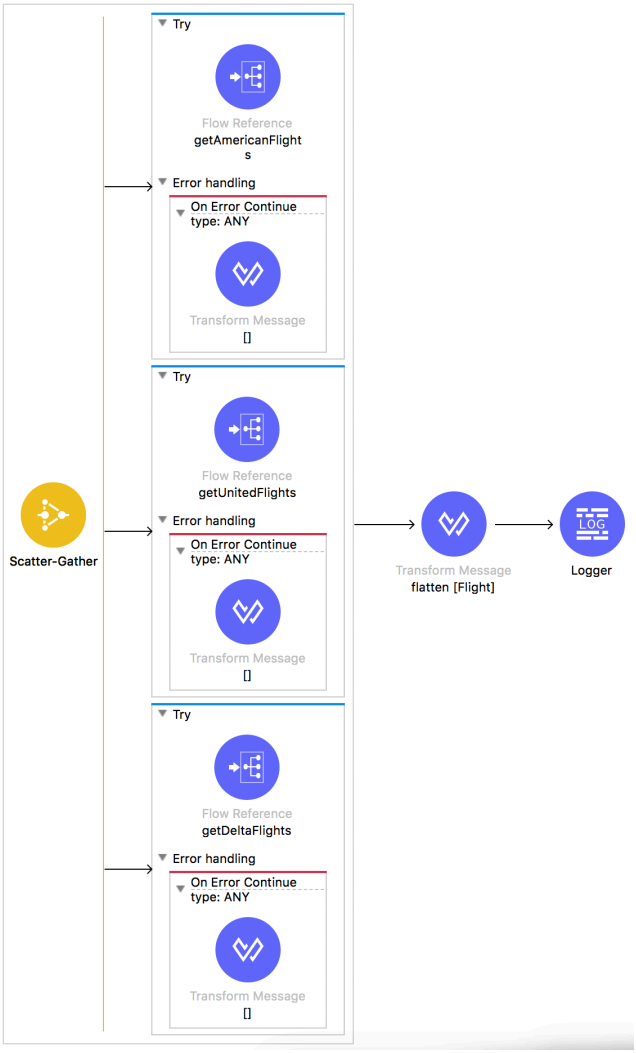
21. Return to Anypoint Studio and switch to the Mule Design perspective.
22. In getAllAirlineFlights, add a Transform Message component to one of the On Error Continue scopes.
23. Set the display name to [].



24. In the Transform Message properties view, set the payload to an empty array.



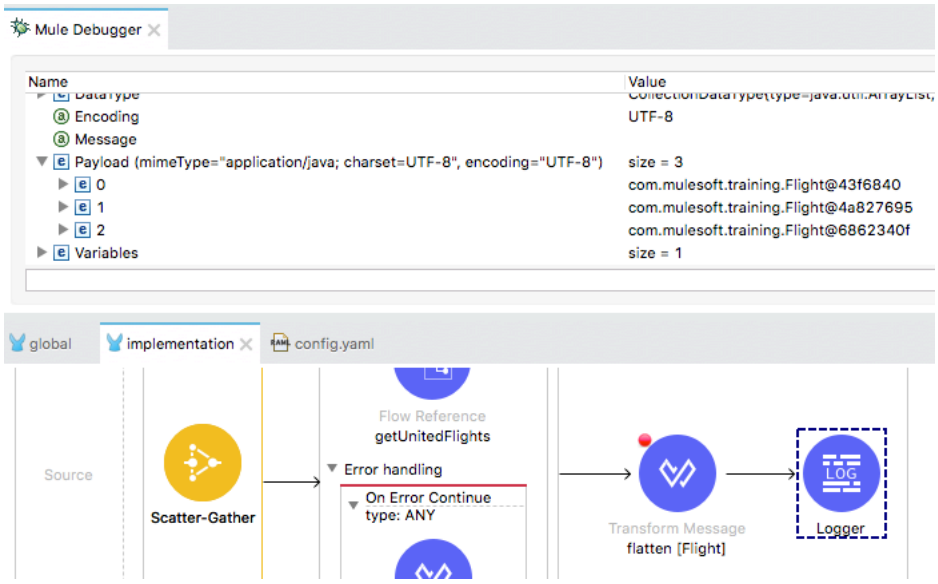
25. Repeat the steps to add the same Transform Message component to the two other error handlers.



Debug the application

26. Save the file to redeploy the project.
27. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX.
28. In the Mule Debugger, step through the application until you are at the Logger after the Scatter-Gather in getAllAirlineFlights.

29. Expand Payload; you should now see the three flights and no error messages.



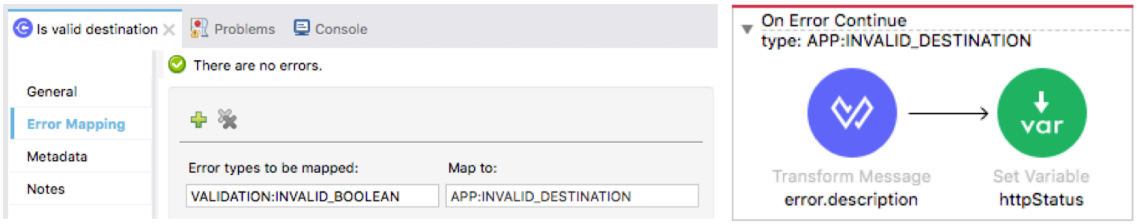
30. Step through the rest of the application.
31. Return to Advanced REST Client; you should now only see the three flights.

32. Return to Anypoint Studio and switch to the Mule Design perspective.
33. Stop the project.

Walkthrough 10-6: Map an error to a custom error type

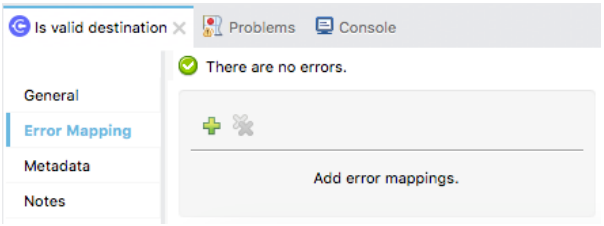
In this walkthrough, you map the Validation error to a custom error type for the application. You will:

- Map a module error to a custom error type for an application.
- Create an event handler for the custom error type.

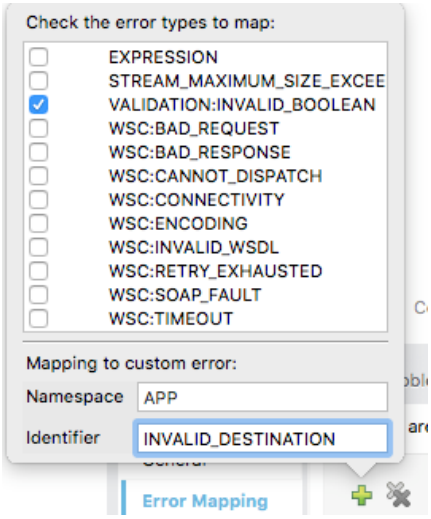


Map a validation module error to a custom error type

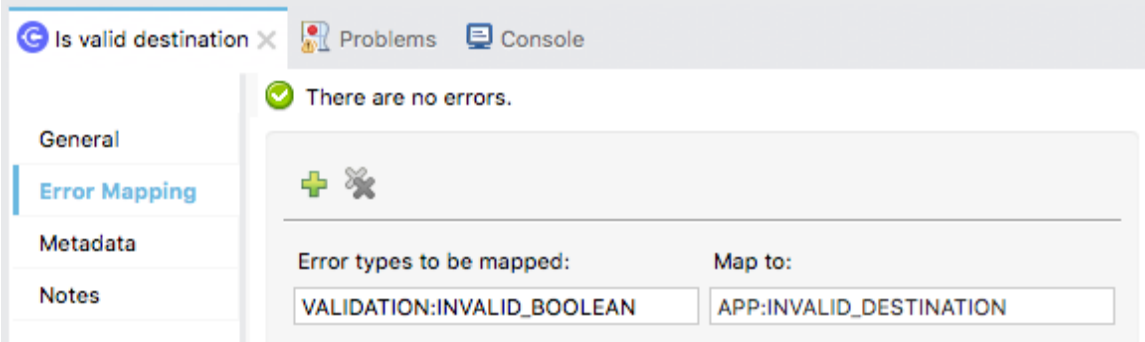
1. Return to implementation.xml.
2. Navigate to the properties view for the validator in getFlights.
3. Select the Error Mapping tab.
4. Click the Add new mapping button.



5. Select the VALIDATION:INVALID_BOOLEAN error type.
6. Leave the namespace of the custom error to map to set to APP.
7. Set the identifier to INVALID_DESTINATION.

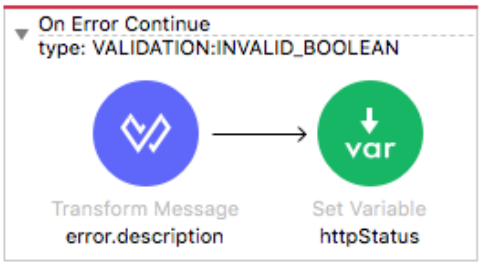


8. Press Enter; you should see your new error mapping.

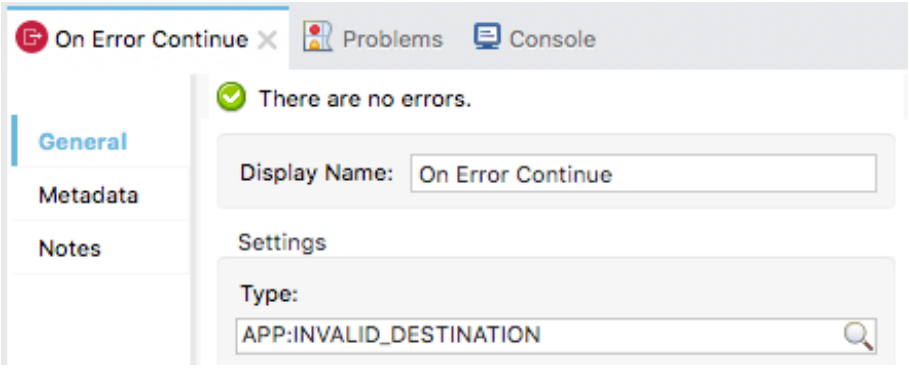


Change the existing validation error handler to catch the new custom type

9. Navigate to the properties view for the validation getFlights On Error Continue error handler.



10. Change the type from VALIDATION:INVALID_BOOLEAN to the new APP:INVALID_DESTINATION that should now appear in the type drop-down menu.



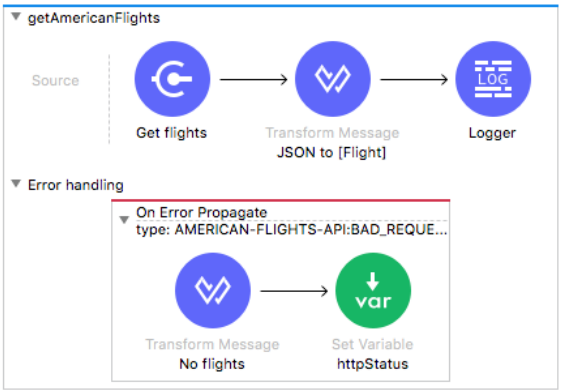
Test the application

11. Save the file and run the project.

12. In Advanced REST Client, change the code and make a request to http://localhost:8081/flights?code=FOO; you should still get a 400 response with the invalid destination error.

Move the American error handler into the American flow

13. Collapse the getAllAirlineFlights and setCode flows.
14. Move the AMERICAN-FLIGHTS-API error scope from getFlights into getAmericanFlights.



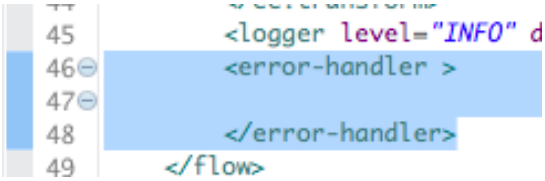
Move the error handler to the global error handler

Note: This description has instructions to make changes in the XML. If you prefer, you can copy and paste the error handler to move it between files and then delete the original.

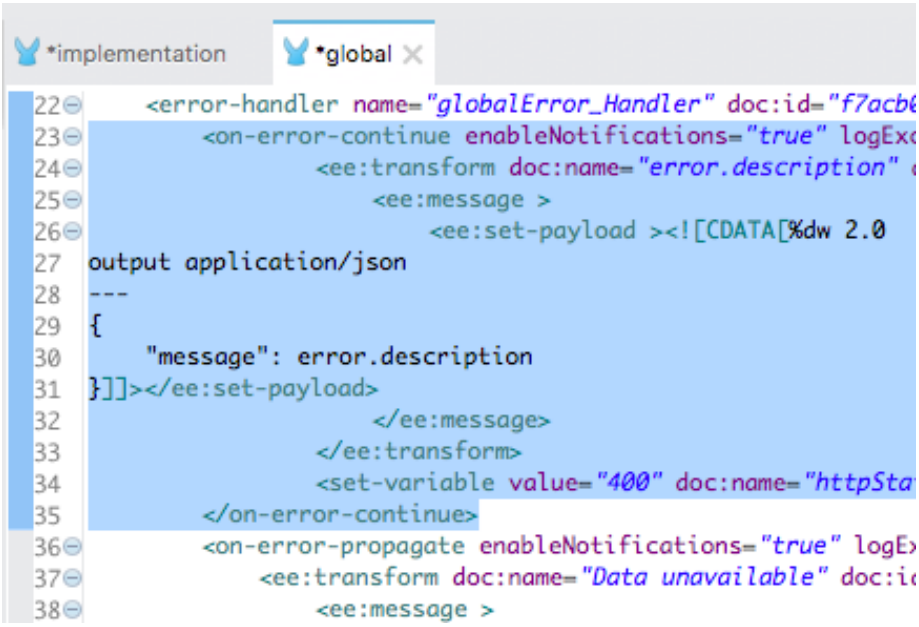
15. Return to implementation.xml.
16. Right-click the validation error scope in getFlights and select Go to XML.

17. Select and cut the APP:INVALID_DESTINATION on-error-continue.

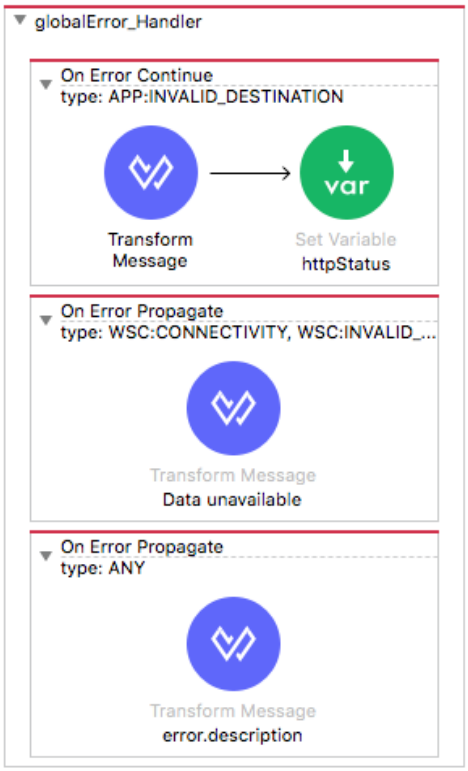
18. Delete the remaining, empty error-handler tag set.



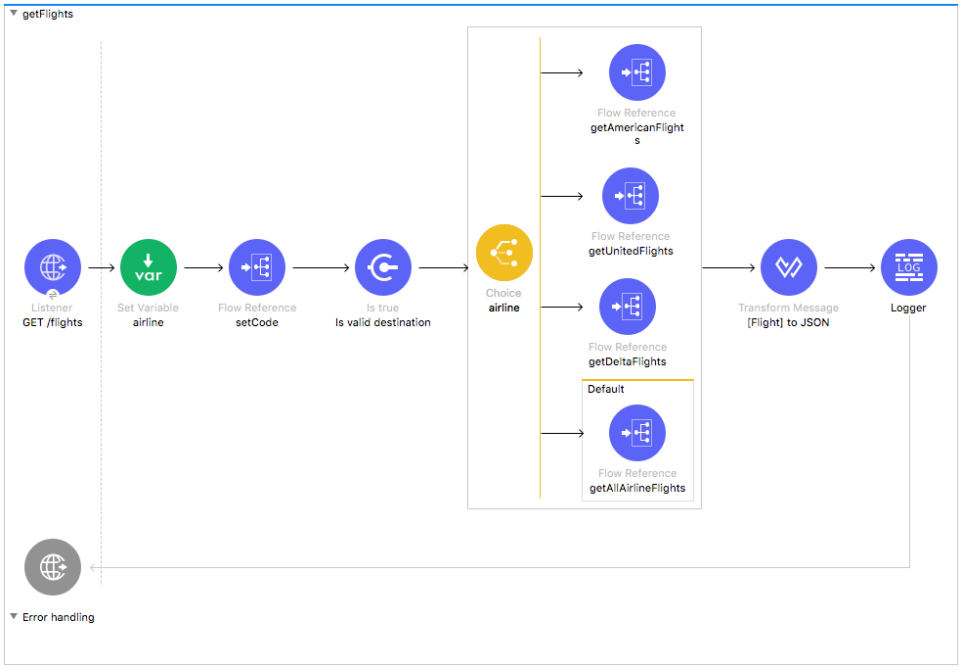
19. Return to global.xml.
20. Right-click the globalError_Handler and select Go to XML.
21. Place the cursor on a new line inside and at the top of the error-handler.



22. Switch back to the Message Flow view; you should see the INVALID_DESTINATION handler.

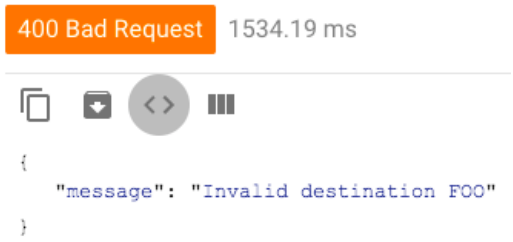


23. Return to implementation.xml and switch to the Message Flow view; you should no longer see an error handler in getFlights.

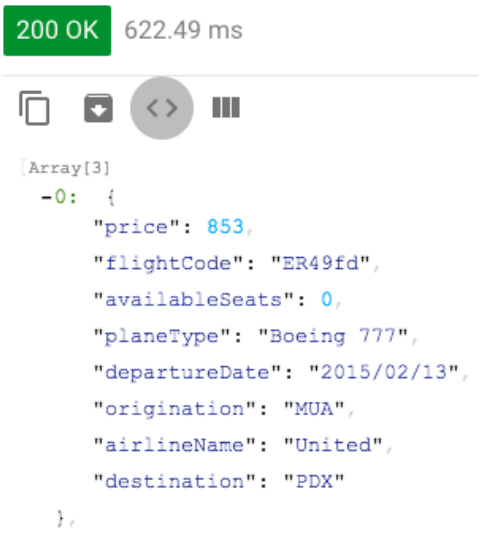


Test the application

24. Save the files to redeploy the project.
25. In Advanced REST Client, make another request to http://localhost:8081/flights?code=FOO; you should still get a 400 response with the invalid destination error.



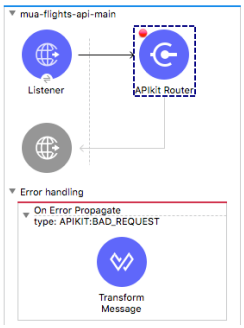
26. Change the code and make a request to http://localhost:8081/flights?code=PDX; you should still get only United flights.



Walkthrough 10-7: Review and integrate with APIkit error handlers

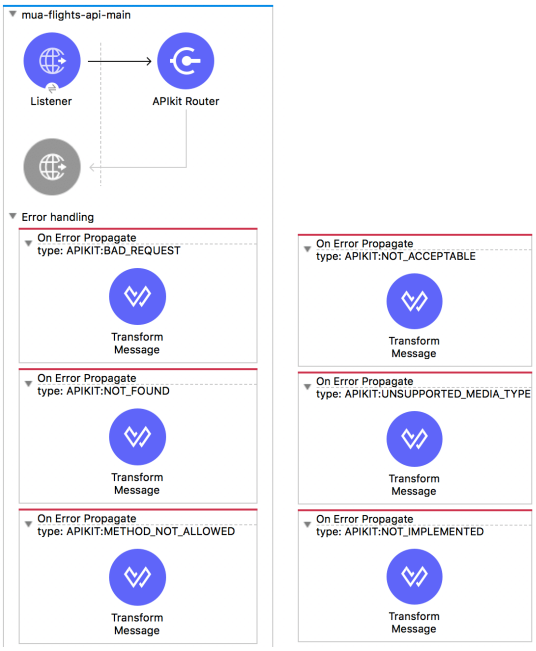
In this walkthrough, you connect the application's implementation to the interface and ensure all the error handling works. You will:

- Review the error handlers generated by APIkit.
- Review settings for the APIkit Router and HTTP Listener in the APIkit generated interface.
- Connect the implementation to the interface and test the error handling behavior.
- Modify implementation error scopes so they work with the APIkit generated interface.

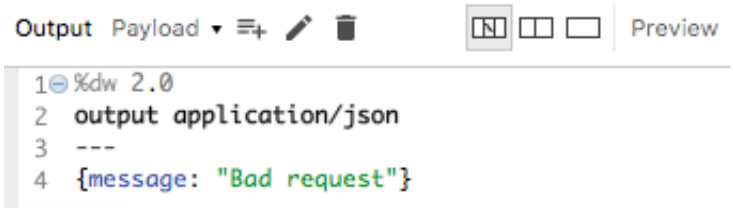


Review APIkit generated error handlers

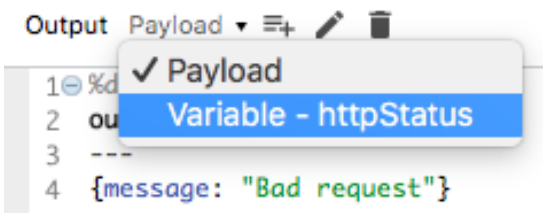
1. Return to apdev-flights-ws in Anypoint Studio.
2. Open interface.xml.
3. Review the error handling section in mua-flights-api-main.



4. Review the types of errors handled by each error handler scope.
5. Navigate to the Transform Message properties view for the first error handling scope.
6. Review the expression that sets the payload.



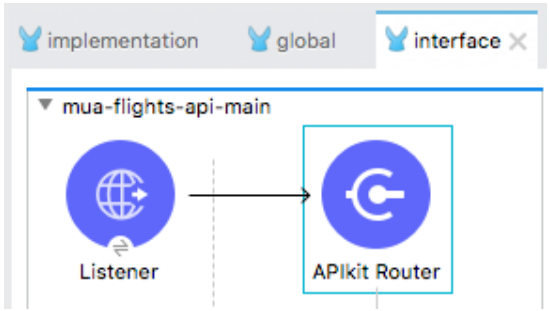
7. Use the output drop-down menu to change the output to Variable – httpStatus.



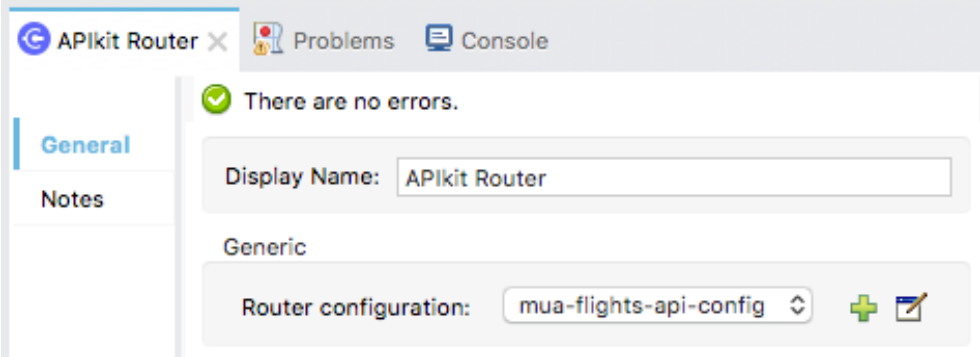
8. Review the expression that sets an httpStatus variable; you should see for a bad request that the httpStatus variable is set to 400.

Review settings for the APIkit Router

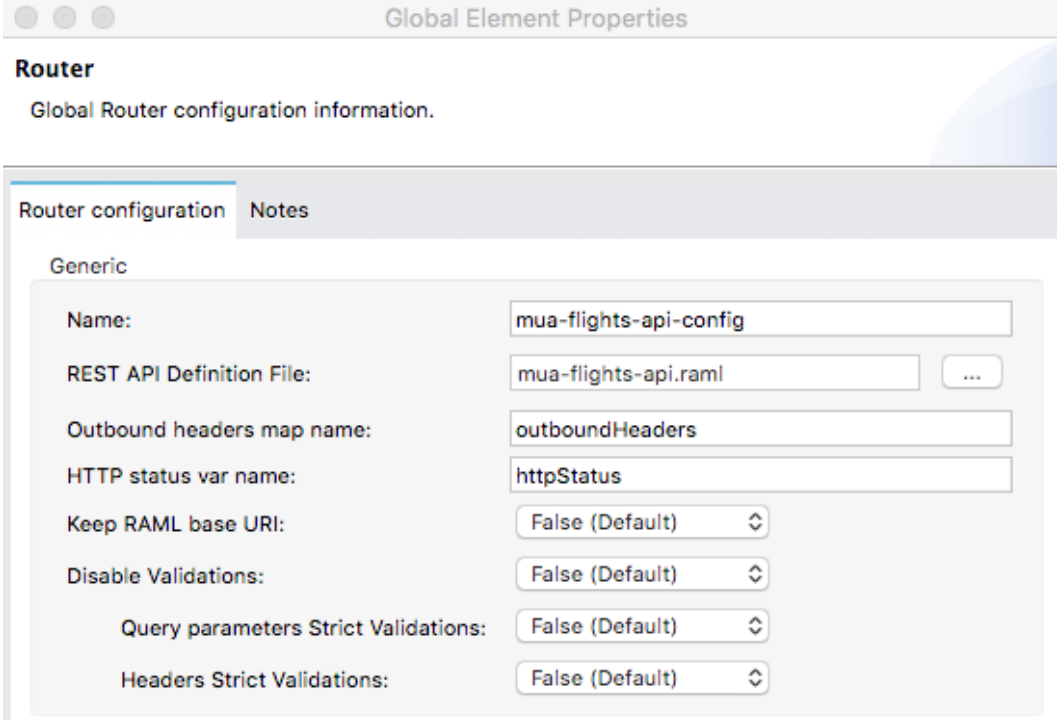
9. Navigate to the properties view for the APIkit Router in mua-flights-api-main.



10. Click the Edit button next to router configuration.



11. In the Global Element Properties dialog box, locate the HTTP status var name setting; you should see httpStatus.

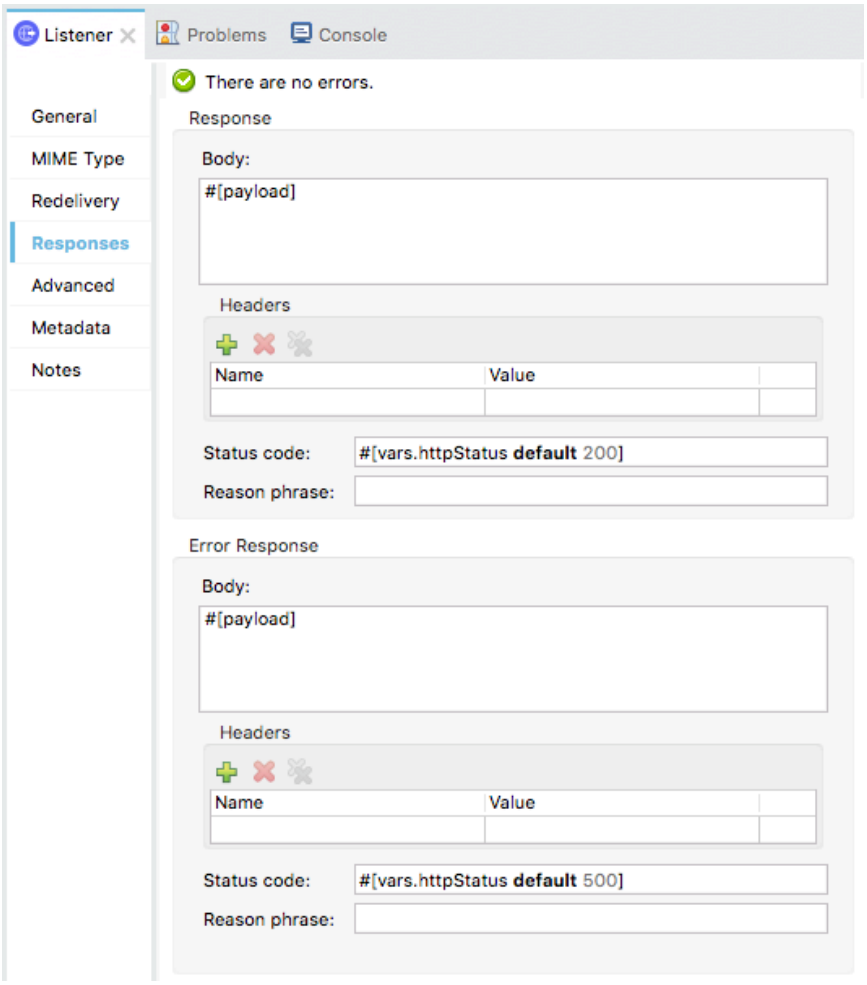


12. Click OK.

Review settings for the HTTP Listener

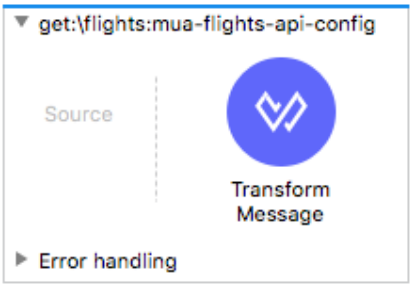
13. Navigate to the Responses tab in the properties view for the HTTP Listener in mua-flights-api-main.

14. Review the response body and status code.
15. Review the error response body and status code.



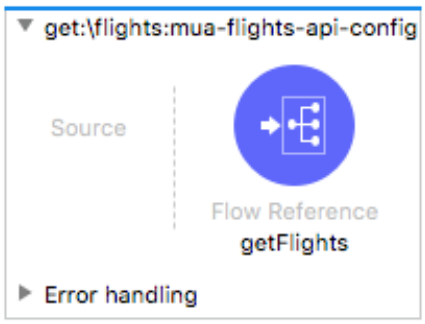
Connect the interface to the implementation

16. In interface.xml, locate the get:\flights flow.

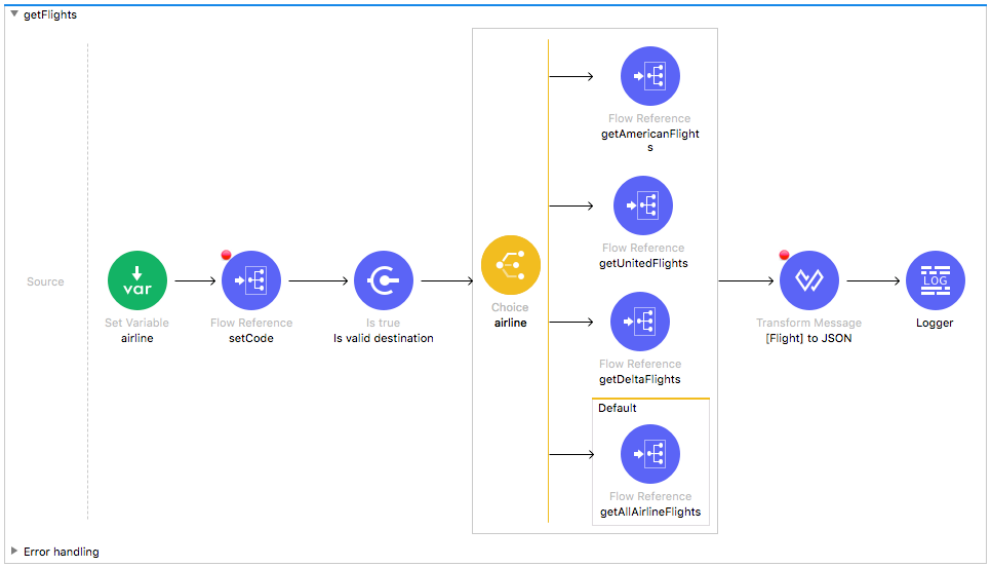


17. Review the default transformation in this flow.
18. Delete the Transform Message component.

19. Add a Flow Reference component to the flow.
20. In the Flow Reference properties view, set the flow name to getFlights.



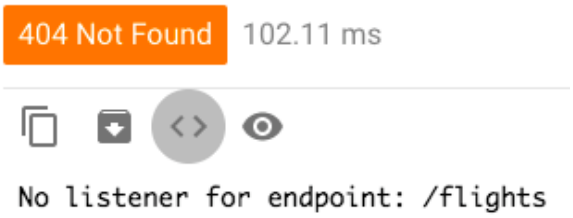
21. Return to implementation.xml.
22. Delete the GET /flights Listener in getFlights.



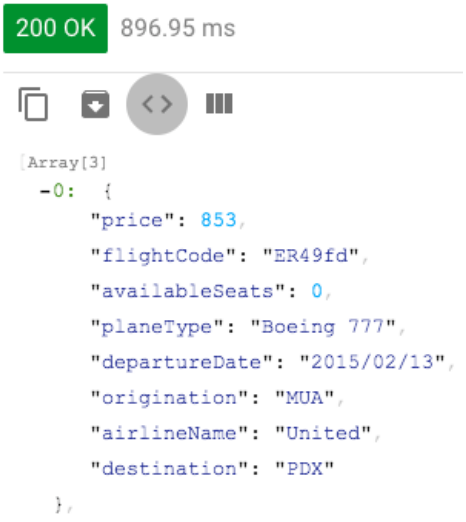
23. Save all the files to redeploy the application.

Test the application

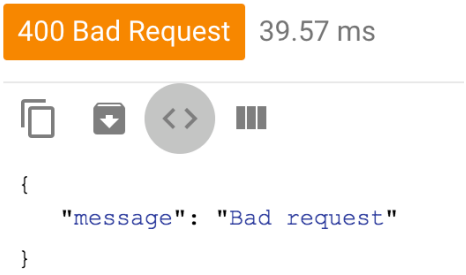
24. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX; you should get a 404 response with a no listener message.



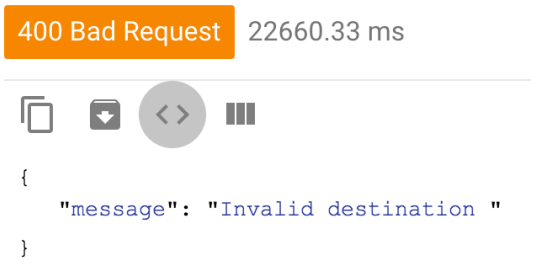
25. Add /api to the URL and make a request to http://localhost:8081/api/flights?code=PDX; you should get flights as before.



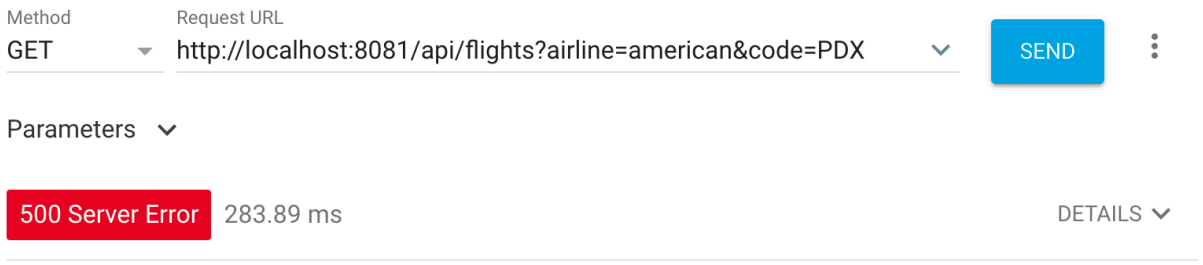
26. Change the code to make a request to http://localhost:8081/api/flights?code=FOO; you should now get a 400 Bad Request response instead of your custom message.



27. Remove the code to make a request to http://localhost:8081/api/flights; you should now get your custom error message.



28. Add the airline and code to make a request to http://localhost:8081/api/flights?airline=american&code=PDX; you should now get a 500 Server error with no message instead of your 200 no flights to PDX response.



Review the API

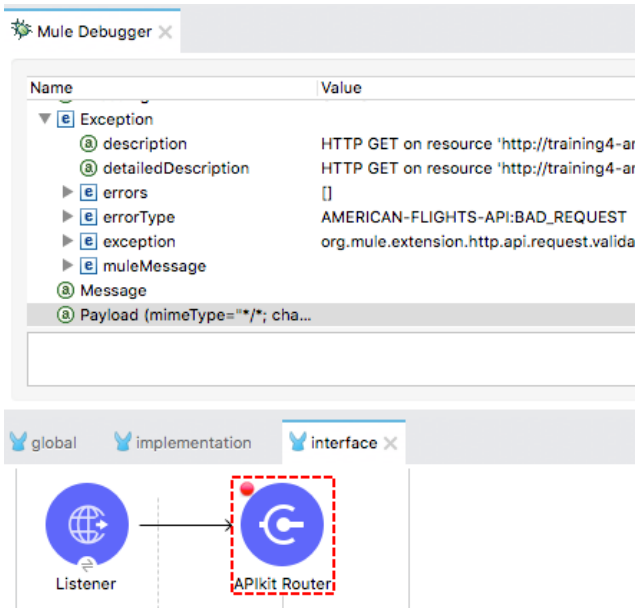
29. Return to Anypoint Studio and stop the project.
30. Return to mua-flights-api.raml and review the code; you should see the code query parameter is not required but it has allowed values enumerated.

```
queryParameters:
  code:
    displayName: Destination airport code
    required: false
    enum:
      - SFO
      - LAX
      - PDX
      - CLE
      - PDF
```

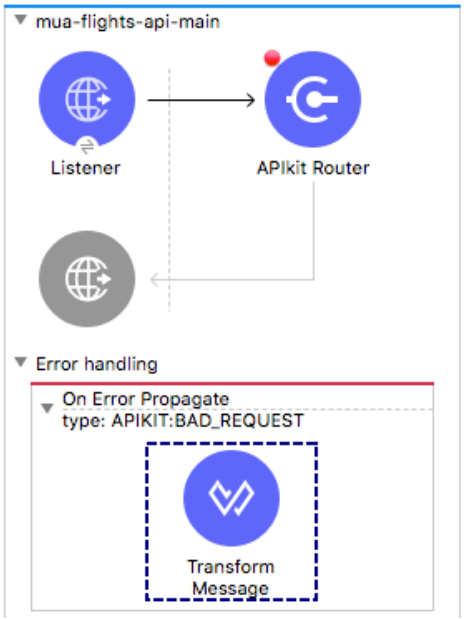
Debug the application

31. Return to interface.xml.
32. Add a breakpoint to the APIkit Router in mua-flights-api-main.
33. Debug the project.
34. In Advanced REST Client, make another request to http://localhost:8081/api/flights?airline=american&code=PDX.
35. In the Mule Debugger, watch the payload and variables and step through the application.
36. Step back to the APIkit router.

37. Review the exception, the payload, and the variables; you should no longer see any variables or your JSON payload message.

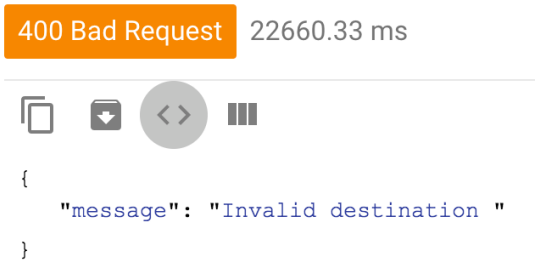


38. In Advanced REST Client, change the code to make a request to http://localhost:8081/api/flights?airline=american&code=FOO.
39. In the Mule Debugger, step through the application; the APIkit router should immediately throw an error.
40. Step again; you should see the error is handled by the that is handled by its APIKIT:BAD_REQUEST handler.



41. Click Resume to finish stepping through the application.

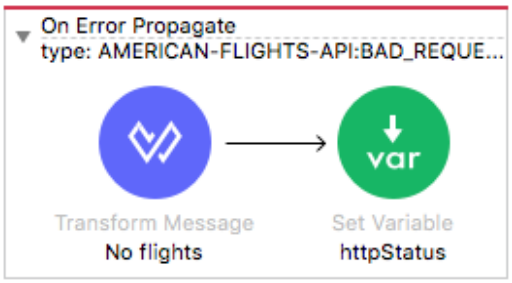
42. In Advanced REST Client, remove the airline and code to make a request to http://localhost:8081/api/flights.
43. In the Mule Debugger, step through the application; the validator should throw an error and execution should **not** return to the APIkit router.
44. Return to Advanced REST Client; you should successfully get a 400 response with the custom message.



Note: If you had specified the code query parameter to be required in the RAML file from which the interface was generated, the APIkit router would catch this immediately and respond with a 400 Bad Request response. The event would never get to the validator in your implementation.

Change the American flights error scope to On Error Continue

45. Return to Anypoint Studio and switch to the Mule Design perspective.
46. Return to implementation.xml.
47. Locate the error handler in the getAmericanFlights flow.

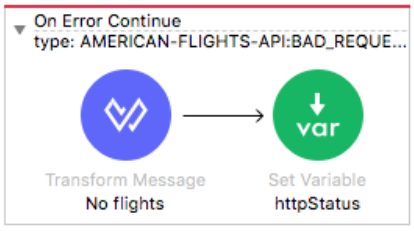


48. Right-click it and select Go To XML.

49. Change the on-error-propagate start and end tags in the getAmericaFlights flow to on-error-continue.

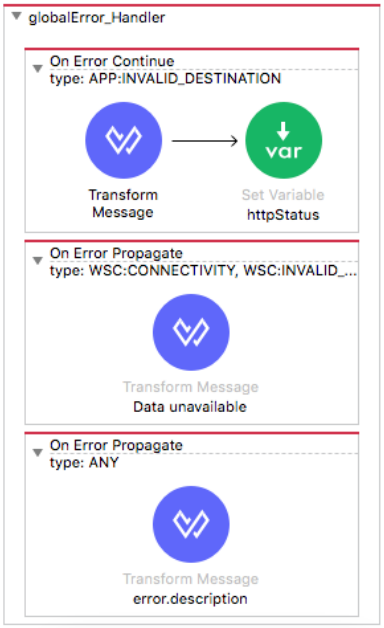
```
        <error-handler >
            <on-error-continue enab
                <ee:transform doc:
                    <ee:message>
                        <ee:set-pa
output application/json
---
{
    "message": "No flights to " ++
}]]></ee:set-payload>
                    </ee:message>
                </ee:transform>
                <set-variable valu
            </on-error-continue>
        </error-handler>
```

50. Change the doc:name in the start tag to On Error Continue.
51. Switch back to the Message Flow view; you should now see an On Error Continue.

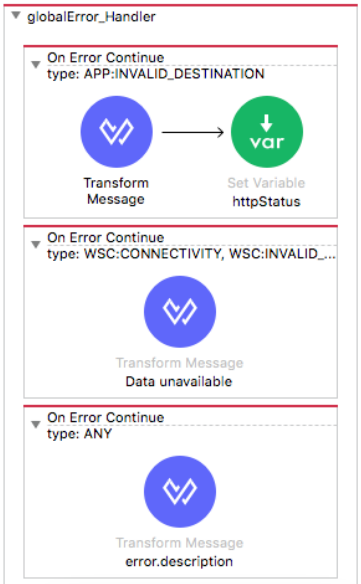


Change the global default error scopes to On Error Continue

52. Return to global.xml.
53. Review the types of error handler scopes.

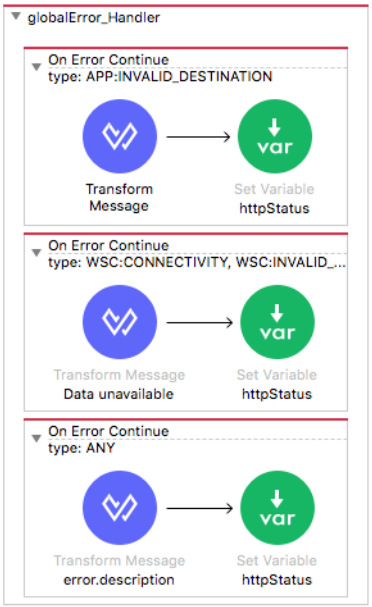


54. Right-click the WSC error handler and select Go To XML.
55. Change the four on-error-propagate start and end tags to on-error-continue.
56. Change the doc:names in both start tags to On Error Continue.
57. Switch back to the Message Flow view; you should now see both are On Error Continue scopes.



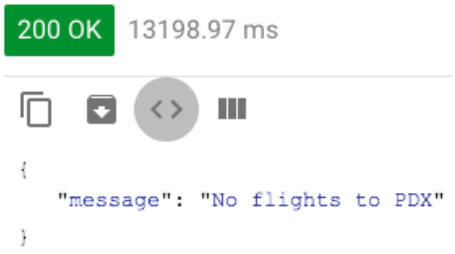
Set the HTTP status code for the On Error Continue scopes so you do not get 200

58. Add a Set Variable transformer to the WSC error handler.
59. In the Set Variable properties view, set the display name and name to httpStatus.
60. Set the value to 500.
61. Copy the Set Variable transformer and add it to the ANY error scope.



Test the application

62. Save the files to redeploy the project.
63. In Advanced REST Client, add the airline and code to make a request to http://localhost:8081/api/flights?airline=american&code=PDX.
64. In the Mule Debugger, step through the application; the application now does not return to the APIkit router.
65. Return to Advanced REST Client; you should now get a 200 response with the No flights to PDX message.



66. In Advanced REST Client, change the airline to make a request to http://localhost:8081/api/flights?airline=delta&code=PDX.
67. In the Mule Debugger, step through the application.
68. Return to Advanced REST Client; you should now get the 500 response with the data unavailable message.

69. Return to Anypoint Studio and switch perspectives.

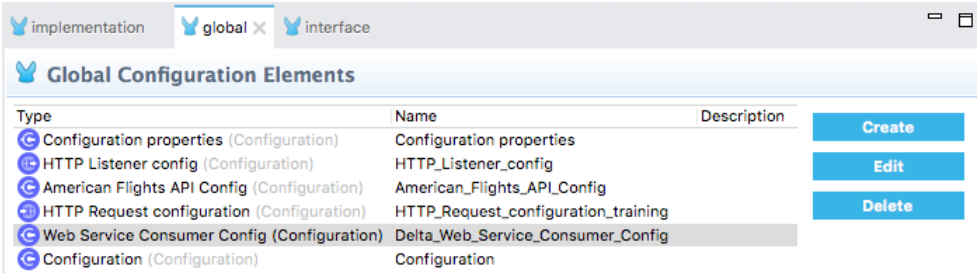
Walkthrough 10-8: Set a reconnection strategy for a connector

In this walkthrough, you will:

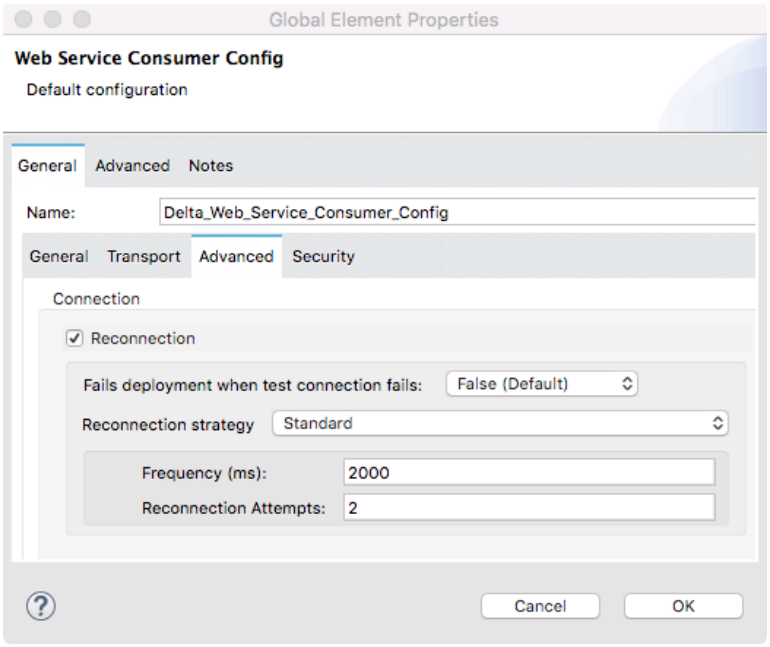
- Set a reconnection strategy for the Web Service Consumer connector.

Set a reconnection strategy for a connector

1. Return to global.xml in Anypoint Studio.
2. Switch to the Global Elements view.
3. Double-click the Web Service Consumer Config.



4. In the Global Element Properties dialog box, select the Advanced tab in the second tab bar.
5. Select the Reconnection checkbox.
6. Change the reconnection strategy to standard.
7. Review the default frequency and attempts.



8. Click OK.
9. Save the file.

Module 11: Writing DataWeave Transformations

At the end of this module, you should be able to:

- Write DataWeave expressions for basic XML, JSON, and Java transformations.
- Write DataWeave transformations for complex data structures with repeated elements.
- Define and use global and local variables and functions.
- Use DataWeave functions.
- Coerce and format strings, numbers, and dates.
- Define and use custom data types.
- Call Mule flows from DataWeave expressions.
- Store DataWeave scripts in external files.

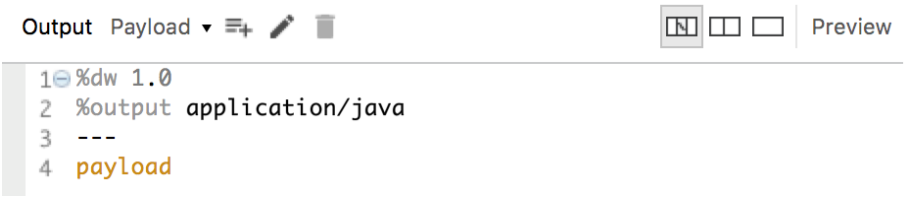
Walkthrough 11-1: Create transformations with the Transform Message component

In this walkthrough, you create a new flow that receives flight POST requests that you will use as the input for writing transformations in the next several walkthroughs. You will:

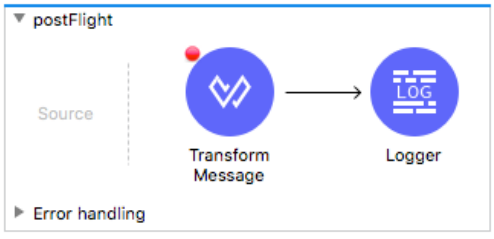
- Create a new flow that receives POST requests of JSON flight objects.
- Add sample data and use live preview.
- Create a second transformation that stores the output in a variable.
- Save a DataWeave script in an external file.
- Review DataWeave script errors.

Create a new flow that receives POST requests and returns the payload as Java

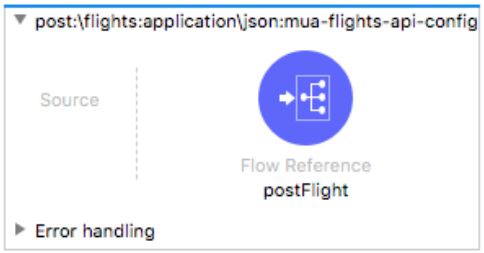
1. Return to implementation.xml.
2. Right-click in the canvas and select Collapse All.
3. Drag a Transform Message component from the Mule Palette to the bottom of the canvas; a new flow should be created.
4. Change the name of the flow to postFlight.
5. In the Transform Message properties view, set the expression to the payload.



6. Add a breakpoint to the Transform Message component.
7. Add a Logger to the flow.



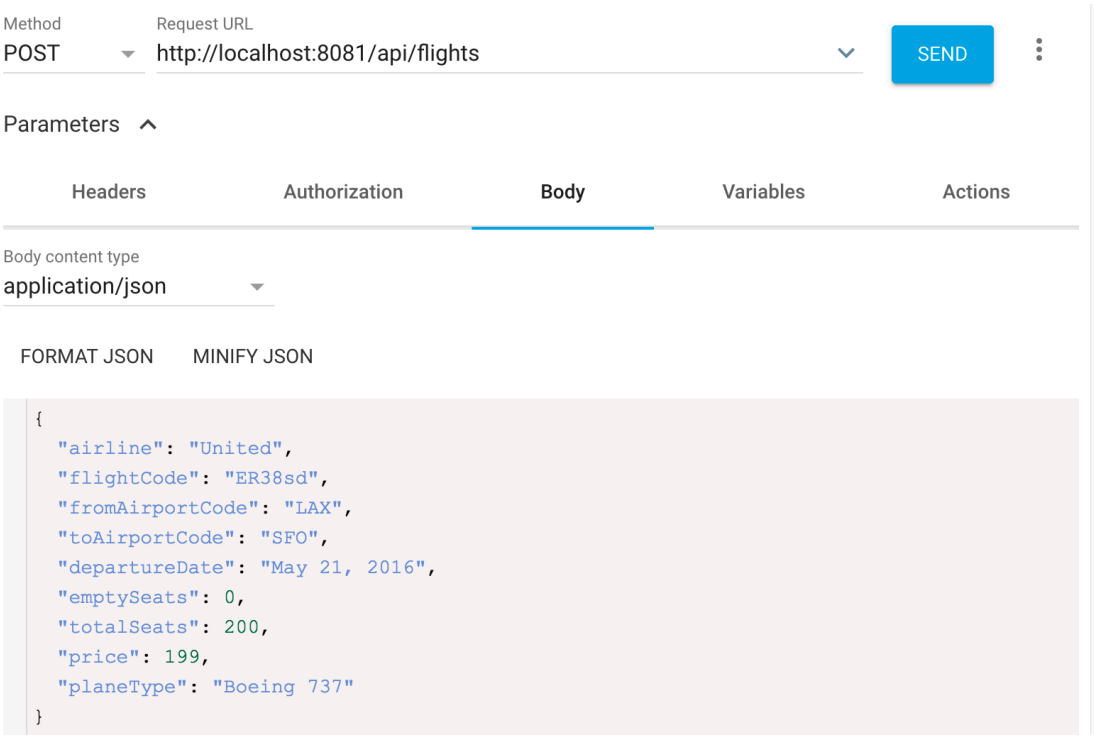
8. Return to interface.xml.
9. Locate the post:\flights flow.
10. Delete the Transform Message component and replace it with a Flow Reference.
11. In the Flow Reference properties view, set the flow to postFlight.



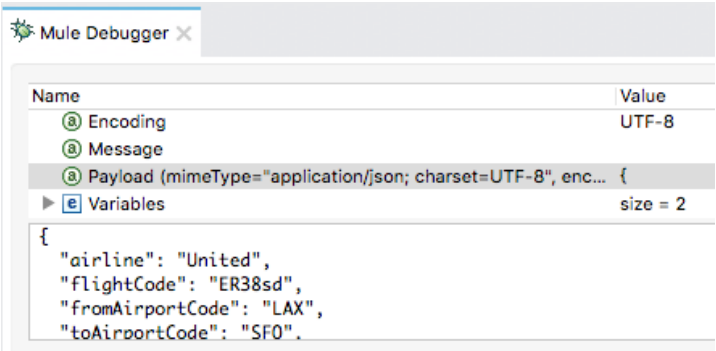
12. Save the files to redeploy the project in debug mode.
13. Close interface.xml.

Post a flight to the flow

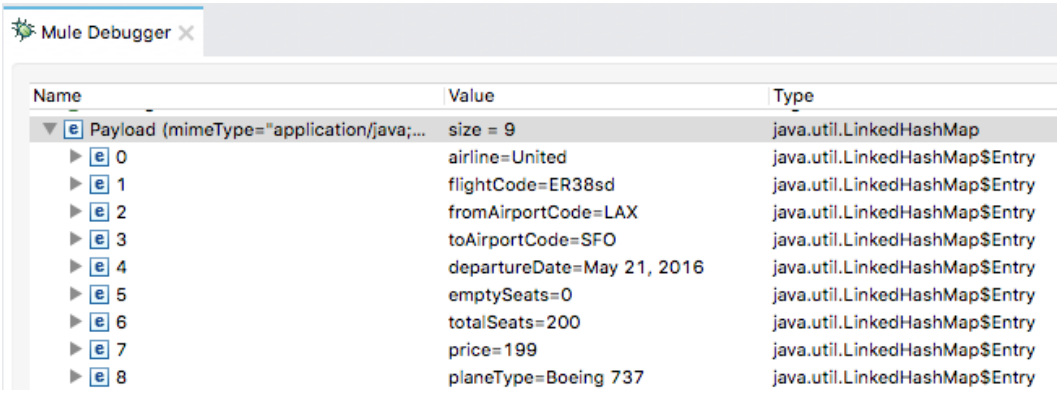
14. Open flight-example.json in src/test/resources.
15. Copy the code and close the file.
16. In Advanced REST Client, change the method to POST and remove any query parameters.
17. Add a request header called Content-Type and set it equal to application/json.
18. Set the request body to the value you coped from flight-example.json.
19. Make the request to http://localhost:8081/api/flights.



20. In the Mule Debugger, expand Attributes and locate the content-type header.
21. Review the payload; you should see it has a mime-type of application/json.



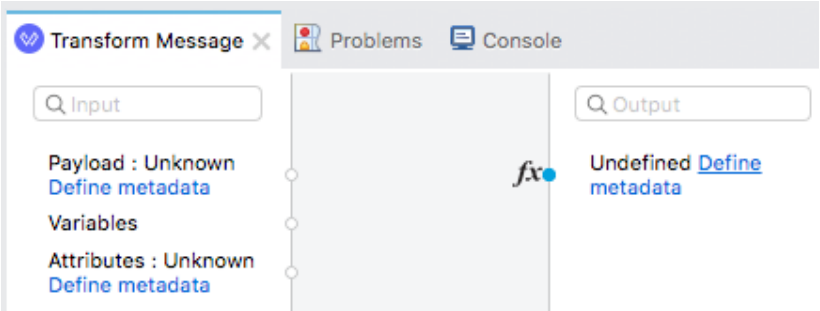
22. Step to the Logger; you should see the payload now has a mime-type of application/java and is a LinkedHashMap.



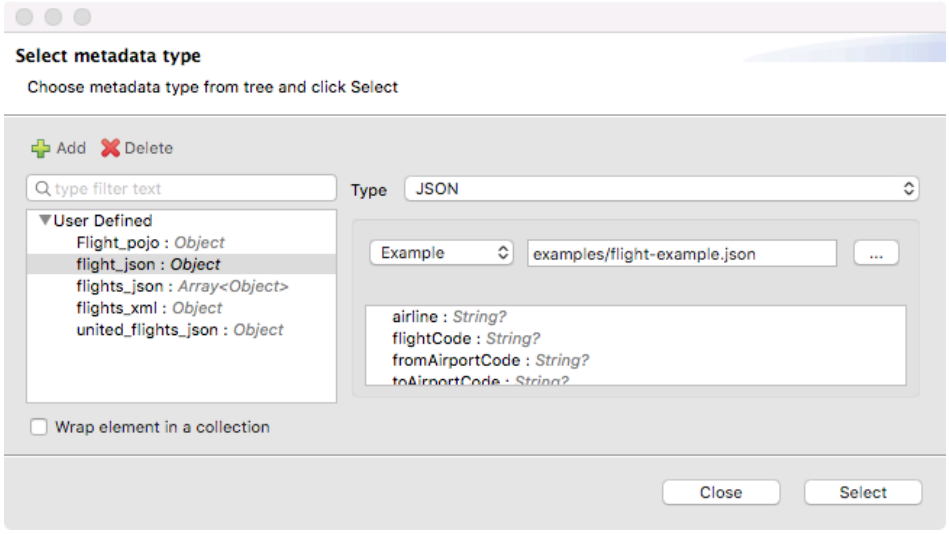
23. Step to the end of the application and switch perspectives.

Add input metadata for the transformation

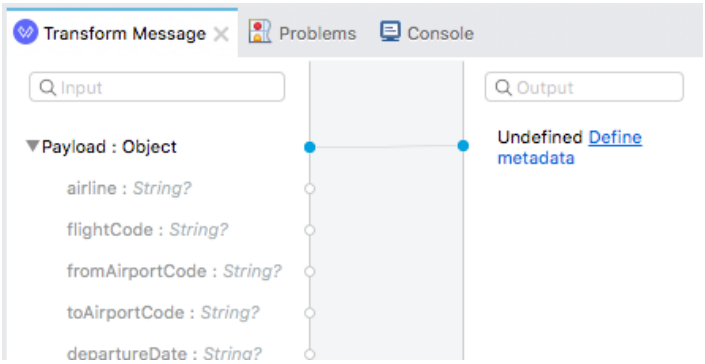
24. In the Transform Message properties view, click Define metadata in in the input section.



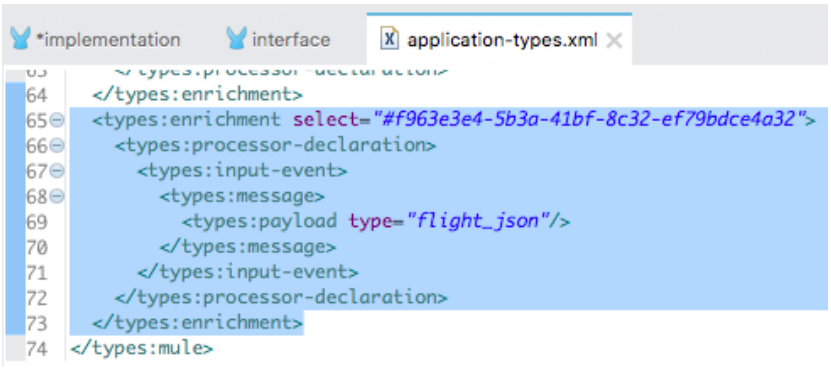
25. In the Select metadata type dialog box, select flight_json and click Select.



26. In the Transform Message Properties view, you should now see metadata for the input.



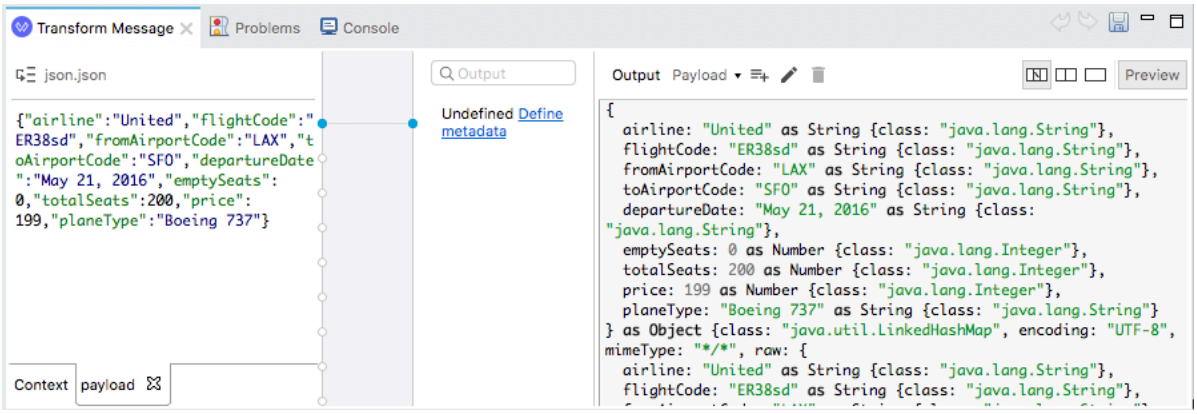
27. Open application-types.xml in src/main/resources.
28. Review the enrichment elements.



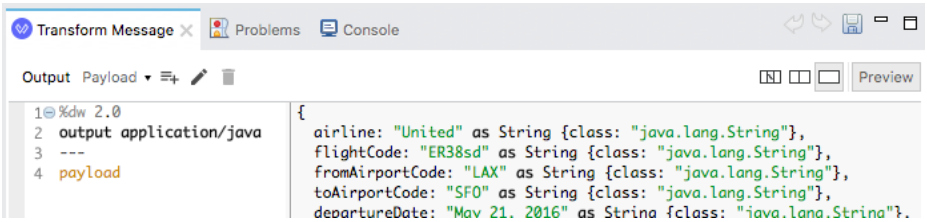
29. Close the file.

Preview sample data and sample output

30. In the Transform Message properties view, click the Preview button.
31. Click the Create required sample data to execute preview link.
32. Look at the input section; you should see a new tab called payload and it should contain the sample data.
33. Look at the preview section; you should see the sample output there of type Java.

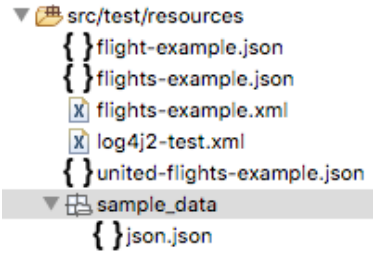


34. Click the Source Only button to the left of the Preview button.



Locate the new sample data file

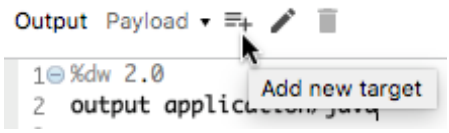
35. In the project explorer, locate the new sample_data folder in src/test/resources.



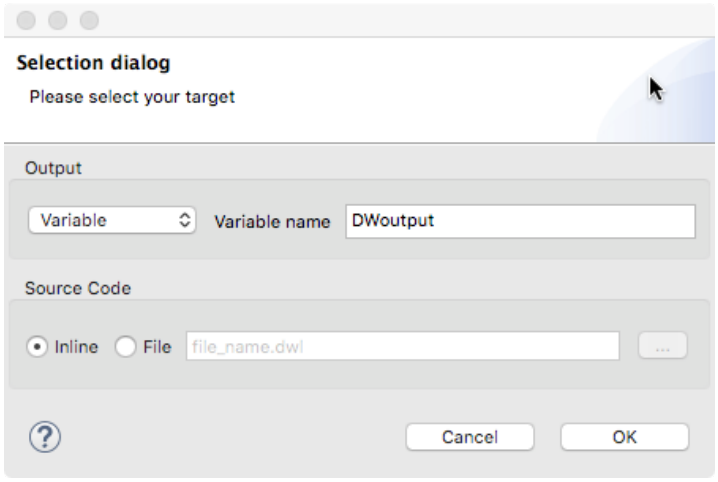
36. Open json.json.
37. Review the code and close the file.

Create a second transformation with the same component

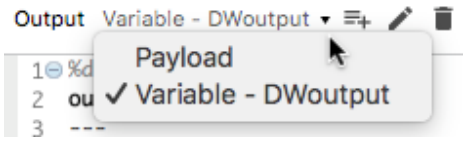
38. Click the Add new target button.



39. In the Selection dialog box, change the output to variable.
40. Set the variable name to DWoutput.



41. Click OK.
42. In the Transform Message properties view, click the drop-down menu button for the output; you should see and can switch between the two transformations.

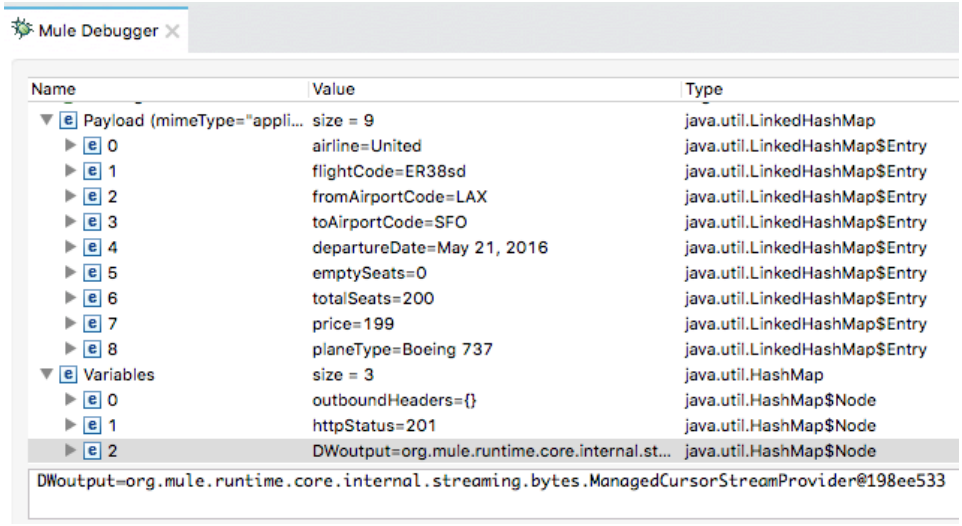


43. For the new transformation, set the output type to json and set the expression to payload.

Test the application

44. Save the file to redeploy the project in debug mode.
45. In Advanced REST Client, post the same request to http://localhost:8081/api/flights.

46. In the Mule Debugger, step to the Logger.
47. Review Payload; it should be Java as before.
48. Expand Variables; you should see the new DWoutput variable.



49. Step through the rest of the application and switch perspectives.

Review the Transform Message XML code

50. Right-click the Transform Message component and select Go to XML.
51. Locate and review the code for both DataWeave transformations.

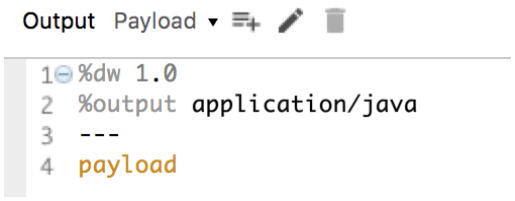
```
    <flow name="postFlight" doc:id="493369a6-2d0d-4dfd-a241-210504c4d5eb" >
        <ee:transform doc:name="Transform Message" doc:id="be12bc7d-22f8-49
            <ee:message >
                <ee:set-payload ><![CDATA[%dw 2.0
output application/java
---
payload]]></ee:set-payload>
            </ee:message>
            <ee:variables >
                <ee:set-variable variableName="DWoutput" ><![CDATA[%dw 2.0
output application/json
---
payload]]></ee:set-variable>
            </ee:variables>
        </ee:transform>
        <logger level="INFO" doc:name="Logger" doc:id="03c0bf95-38e9-45d2-a
    </flow>
```

52. Switch back to the Message Flow view.

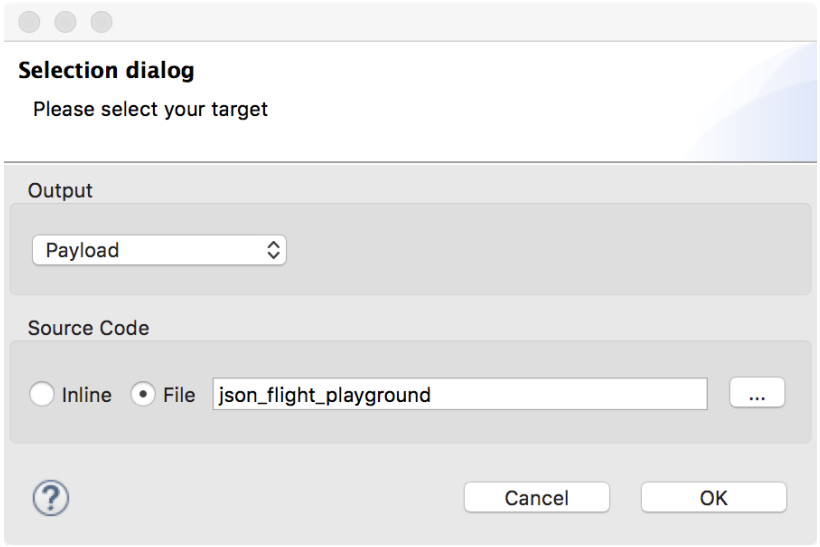
Save a DataWeave script to an external file

53. In the Transform Message Properties view, make sure the payload output expression is selected.

54. Click the Edit current target button (the pencil) above the code.



55. In the Selection dialog box, change the source code selection from inline to file.
56. Set the file name to json_flight_playground.



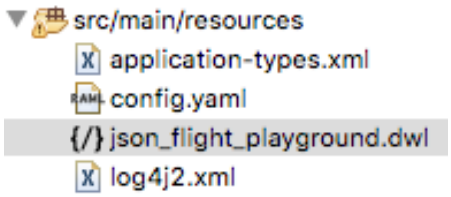
57. Click OK.

Locate and review the code that uses an external DataWeave script

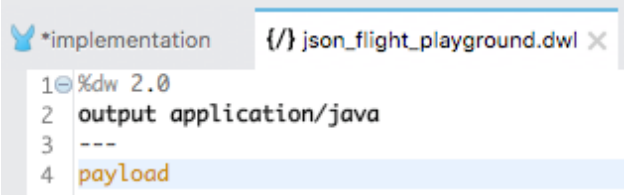
58. Switch to Configuration XML view.
59. Locate the new code for the transformation in postFlight.

```
    <flow name="postFlight" doc:id="847ead89-f898-4cd5-9ec0-1a1fdde0ed6d" >
        <ee:transform doc:name="Transform Message" doc:id="f963e3e4-5b3a-41
            <ee:message >
                <ee:set-payload resource="json_flight_playground.dwl" />
            </ee:message>
            <ee:variables >
                <ee:set-variable variableName="DWoutput" ><![CDATA[%dw 2.0
output application/json
---
payload]]></ee:set-variable>
            </ee:variables>
        </ee:transform>
        <logger level="INFO" doc:name="Logger" doc:id="360ce24e-4dd3-4cef-b
    </flow>
```

60. In the Package Explorer, expand src/main/resources.



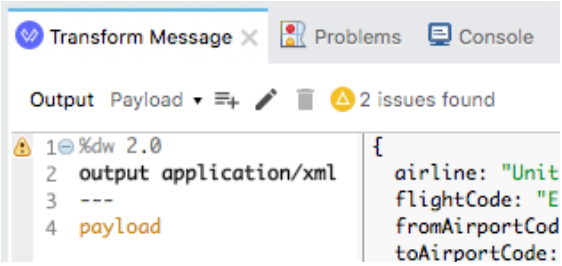
61. Open json_flight_playground.dwl.
62. Review and then close the file.



63. Return to Message Flow view in implementation.xml.

Change the output type to XML and review errors

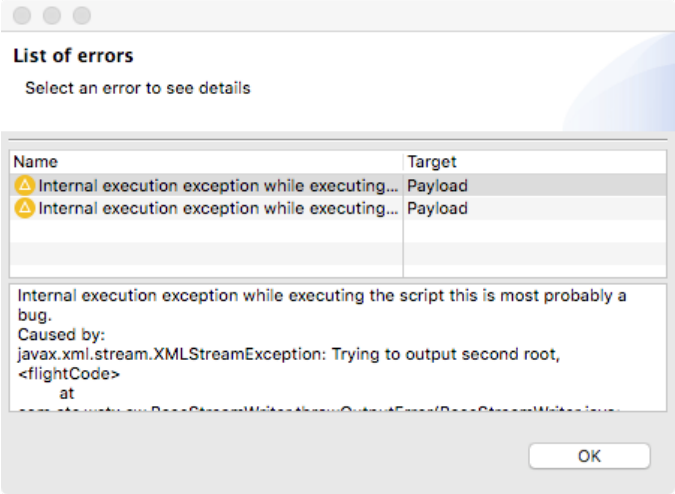
64. In the Transform Message properties view, select the Variable – DWoutput output.
65. Change the output type from application/json to application/xml.
66. Locate the warning icons indicating that there is a problem.



Note: If you do not see the warnings, click the Preview button. If you do not see a data preview, select the payload tab in the input section. If you don't see the payload tab, right-click payload in the input section and select Edit sample data.

67. Mouse over the icon located to the left of the code; you should see a message that there was an exception trying to output the second root <flightCode>.
68. Click the icon above the code; a List of errors dialog box should open.

69. Select and review the first issue.

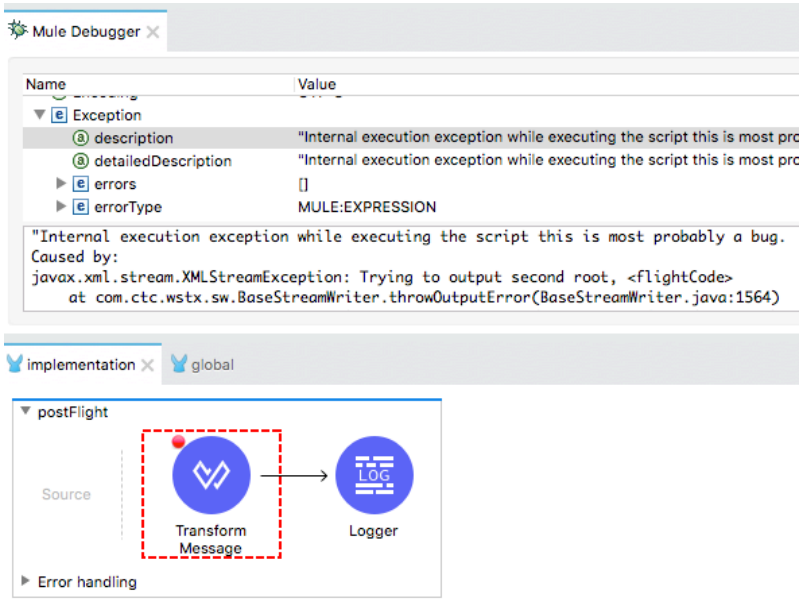


70. In the List of errors dialog box, click OK.

    Note: You will learn how to successfully transform to XML in the next walkthrough.

Test the application

71. Save the file to redeploy the project in debug mode.
72. In Advanced REST Client, post the same request to http://localhost:8081/api/flights.
73. In the Mule Debugger, step once; you should get an error.
74. Expand Exception and review the error information; you should see there is a DataWeave error when trying to output the second root.



75. Step to the end of the application and switch perspectives.

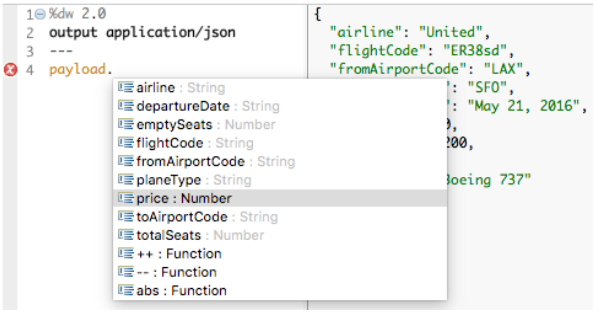
Walkthrough 11-2: Transform basic JSON, Java, and XML data structures

In this walkthrough, you continue to work with the JSON flight data posted to the flow. You will:

- Write scripts to transform the JSON payload to various JSON and Java structures.
- Write scripts to transform the JSON payload to various XML structures.

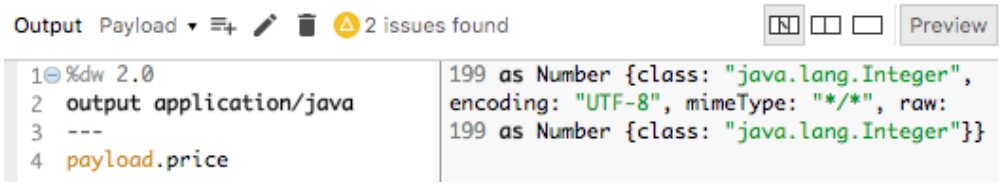
Write expressions to transform JSON to various Java structures

1. Return to the Transform Message properties view for the transformation in postFlight.
2. In the output drop-down menu, select Payload.
3. Change the output type from java to json.
4. Type a period after payload and select price from the auto-completion menu.

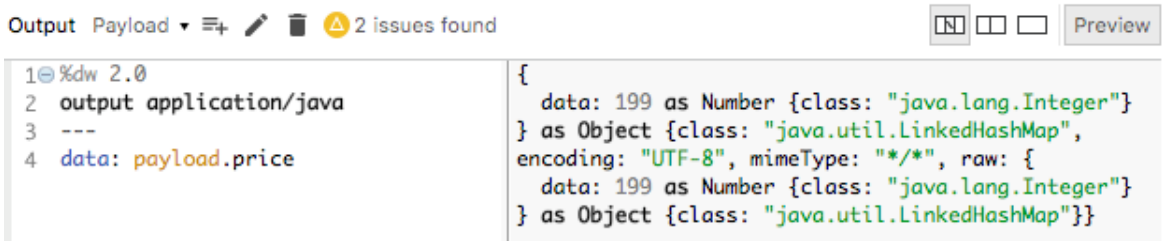


5. Look at the preview.

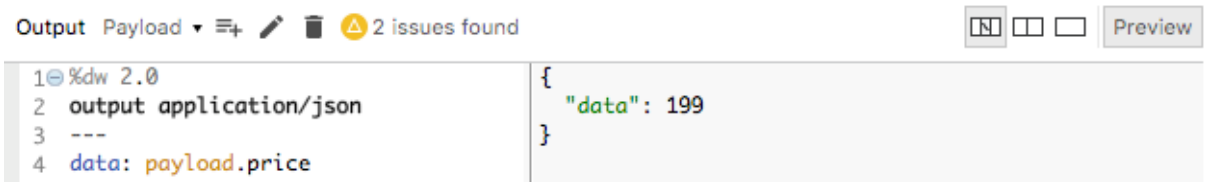
6. Change the output type from json to java.



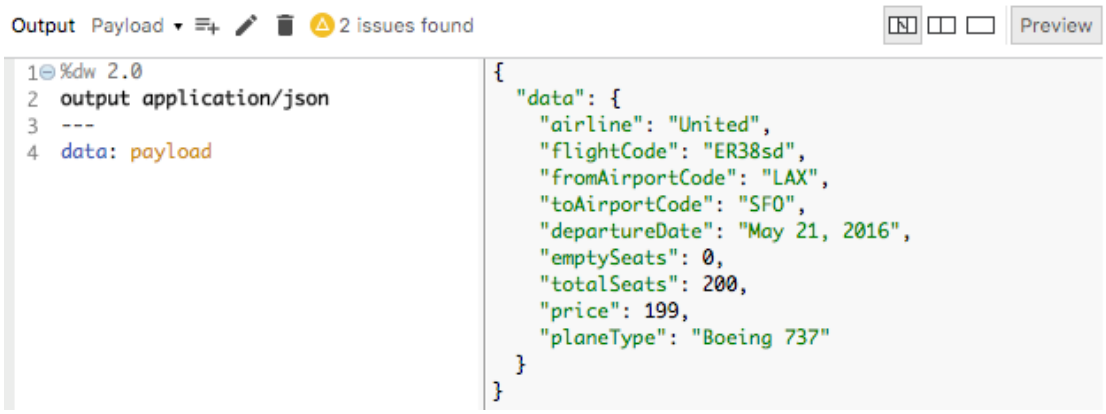
7. Change the DataWeave expression to data: payload.price.



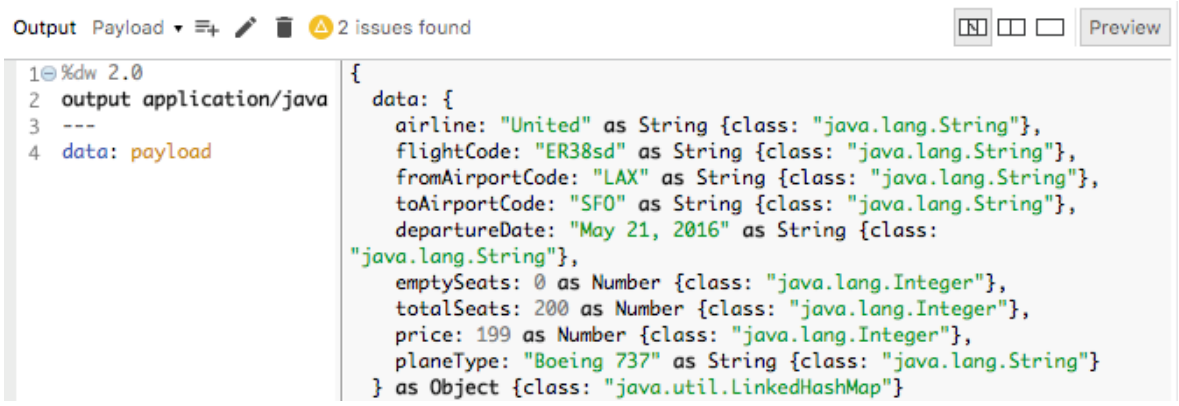
8. Change the output type from java to json.



9. Change the DataWeave expression to data: payload.

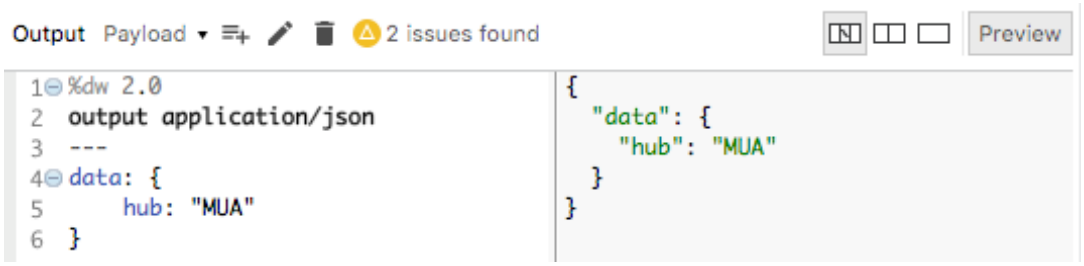


10. Change the output type from json to java.



11. Change the DataWeave expression to data: {}.
12. Add a field called hub and set it to “MUA”.

13. Change the output type from java to json.

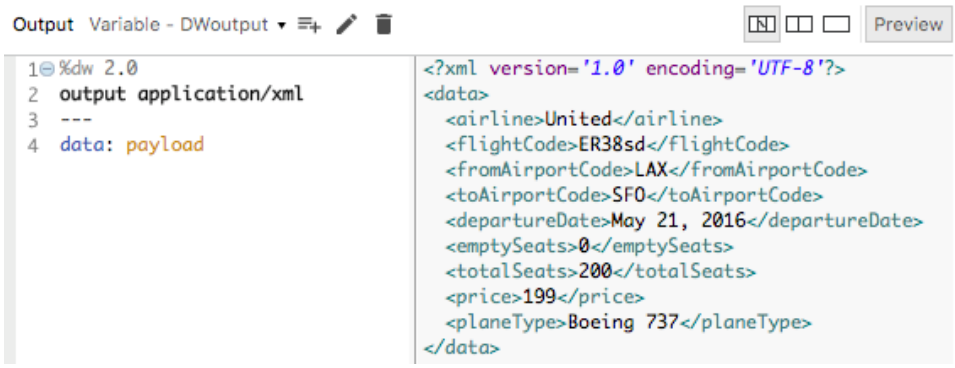


14. Add a field called code and use auto-completion to set it to the toAirportCode property of the payload.
15. Add a field called airline and set it to the airline property of the payload.

Write an expression to output data as XML

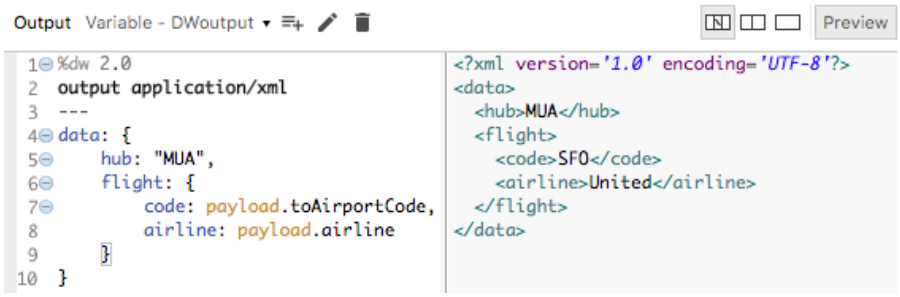
16. In the output drop-down menu, select Variable – DW output.
17. Change the DataWeave expression to data: payload.

18. Look at the preview; you should now see XML.

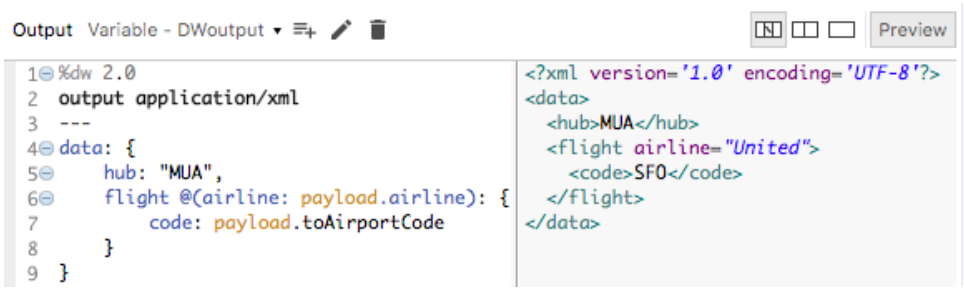


19. In the output drop-down menu, select Payload.
20. Copy the DataWeave expression.
21. Switch back to Variable – DWoutput and replace the DataWeave expression with the one you just copied.

22. Modify the expression so the code and airline properties are child elements of a new element called flight.

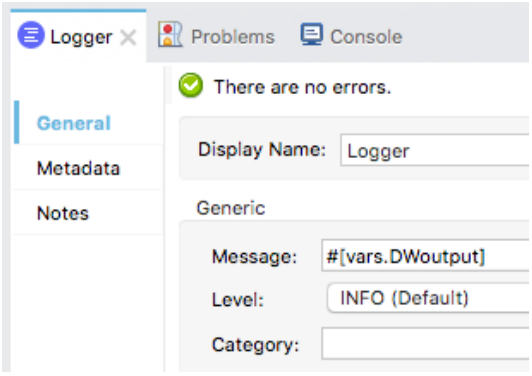


23. Modify the expression so the airline is an attribute of the flight element.

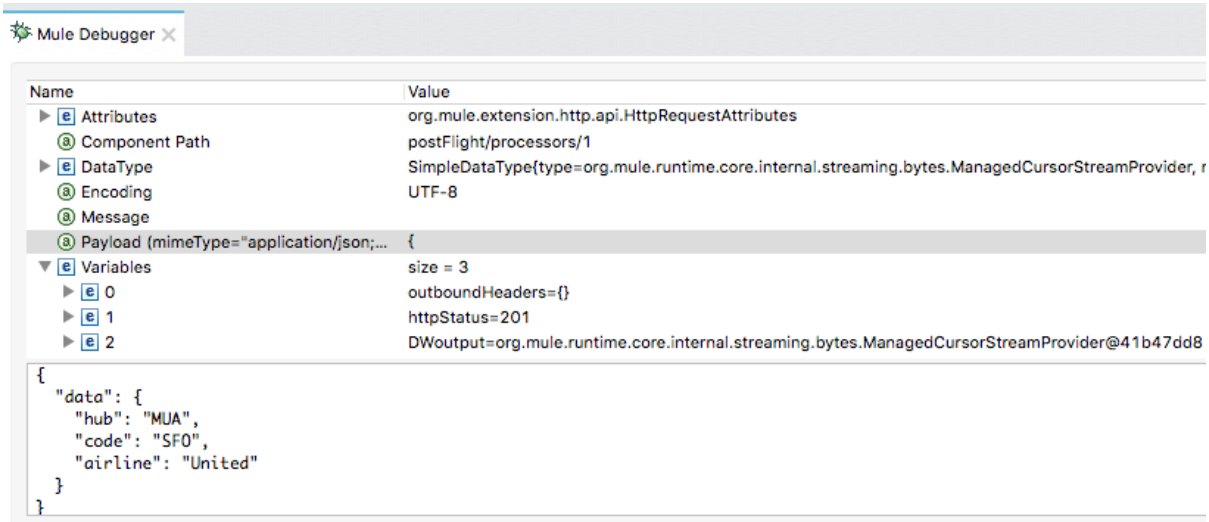


Debug the application

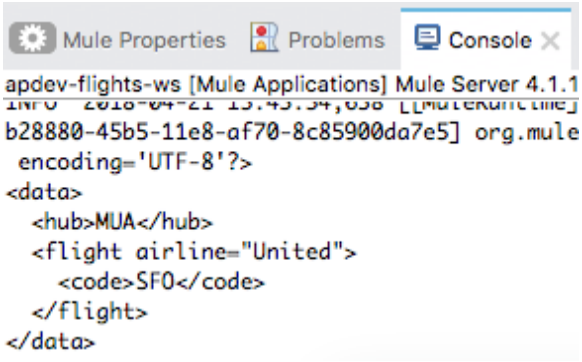
24. In the Logger properties view, set the message to the value of the DWoutput variable.



25. Save the file to redeploy the project in debug mode.
26. In Advanced REST Client, post the same request to http://localhost:8081/flights.
27. In the Mule Debugger, step to the Logger; you should see the transformed JSON payload and the DWoutput variable as a StreamProvider.



28. Step through the application and switch perspectives.
29. Look at the Logger output in the console; you should see the XML.



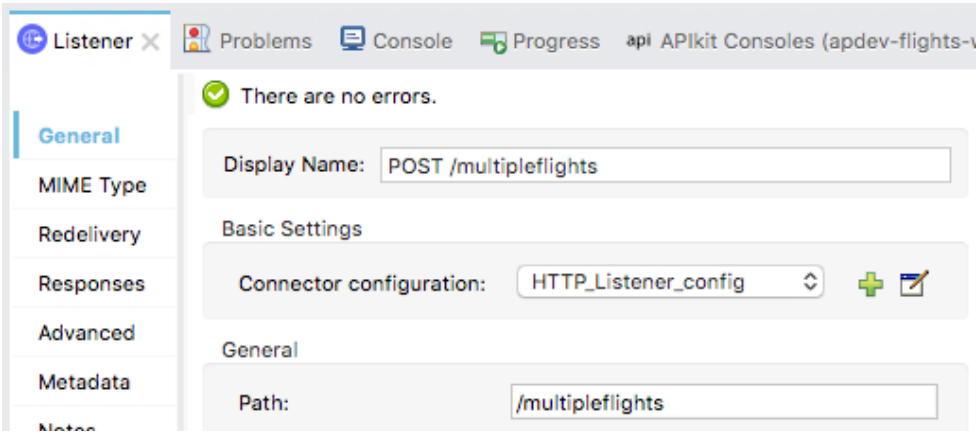
Walkthrough 11-3: Transform complex data structures with arrays

In this walkthrough, you create a new flow that allows multiple flights to be posted to it, so you have more complex data with which to work. You will:

- Create a new flow that receives POST requests of a JSON array of flight objects.
- Transform a JSON array of objects to DataWeave, JSON, and Java.

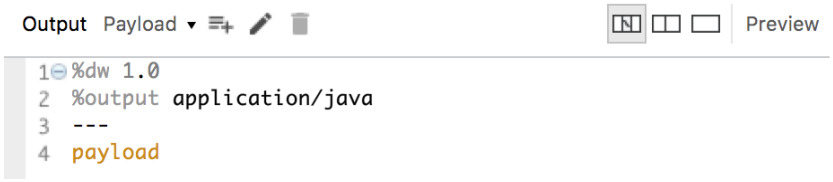
Create a new flow that receives POST requests of a JSON array of flight objects

1. Return to implementation.xml.
2. Drag out an HTTP Listener from the Mule Palette to the bottom of the canvas.
3. Change the flow name to postMultipleFlights.
4. In the HTTP Listener properties view, set the display name to POST /multipleflights.
5. Set the connector configuration to the existing HTTP_Listener_config.
6. Set the path to /multipleflights and set the allowed methods to POST.

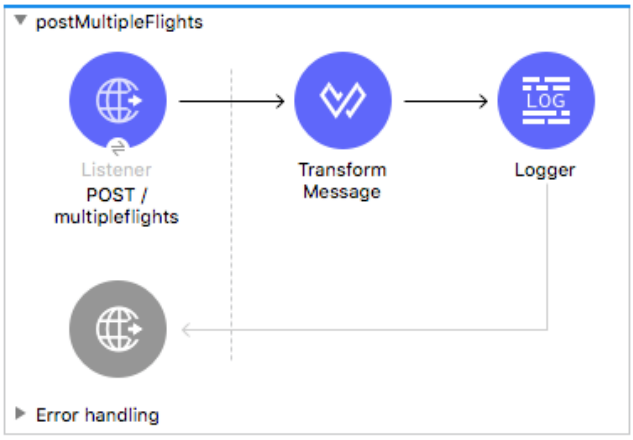


Note: You are bypassing the APIkit router because a corresponding resource/method pair is not defined in the API.

7. Add a Transform Message component to the flow.
8. In the Transform Message properties view, set the expression to the payload.

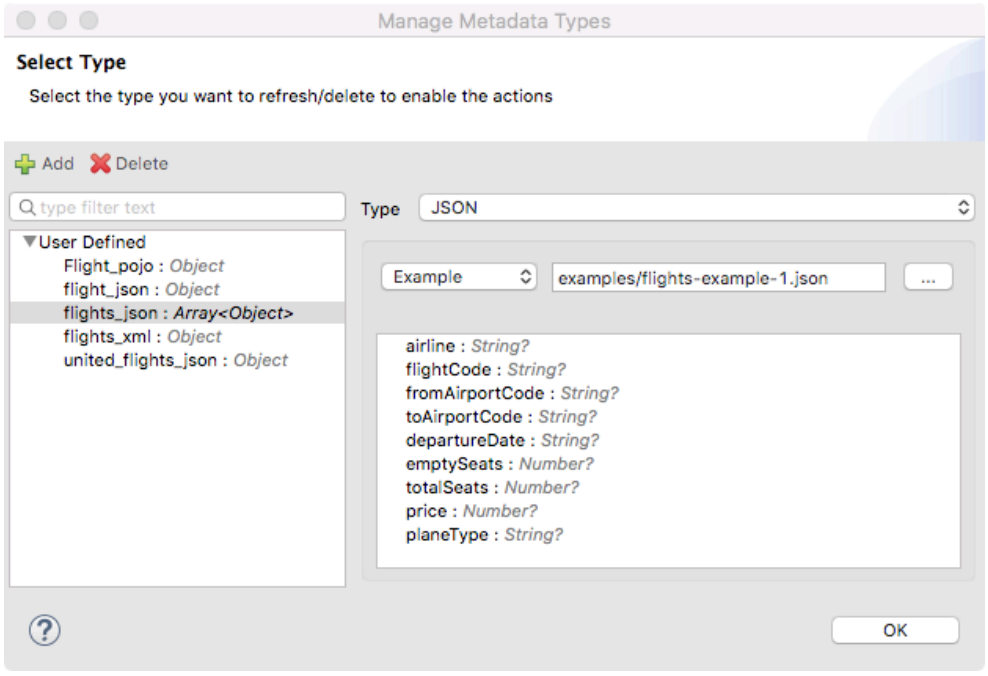


9. Add a Logger to the flow.

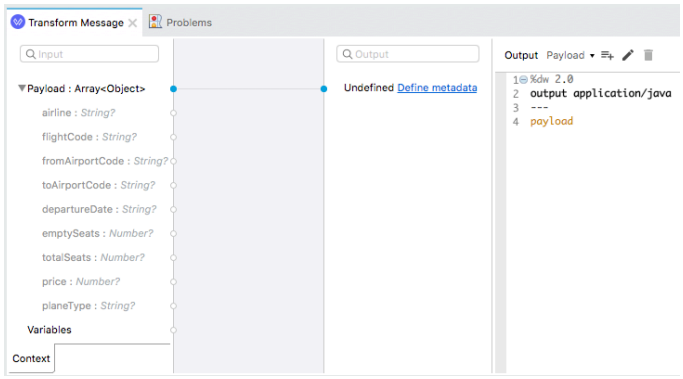


Add input metadata for the transformation

10. In the Transform Message properties view, click Define metadata in in the input section.
11. In the Select metadata type dialog box, select flights_json and click Select.

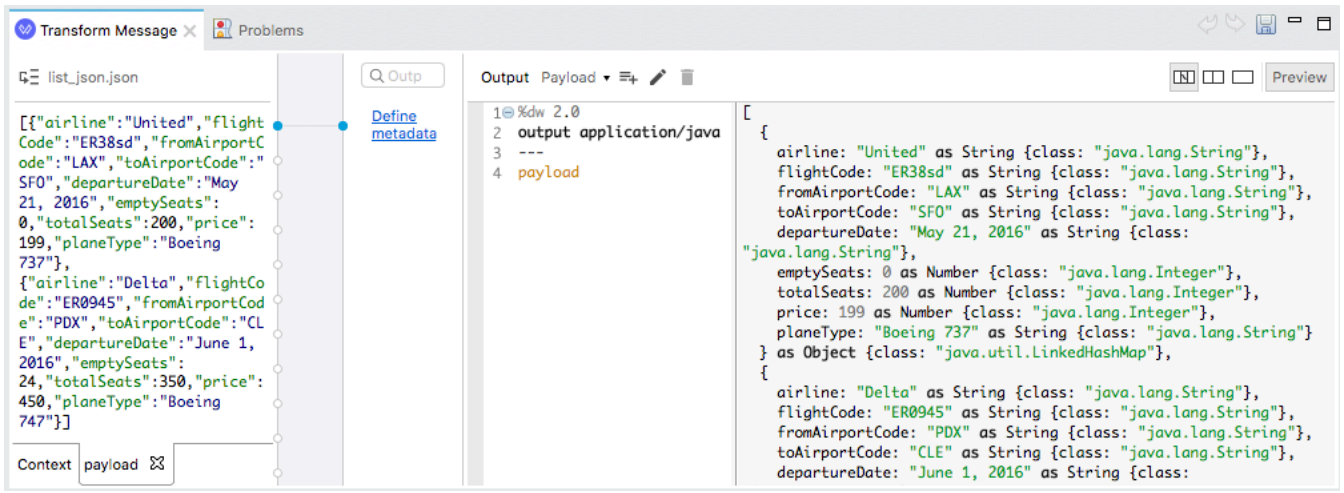


12. In the Transform Message Properties view, you should now see metadata for the input.



Preview sample data and sample output

13. Click the Preview button.
14. In the preview section, click the Create required sample data to execute preview link.
15. Look at the preview section; the sample output should be an ArrayList of LinkedHashMaps.



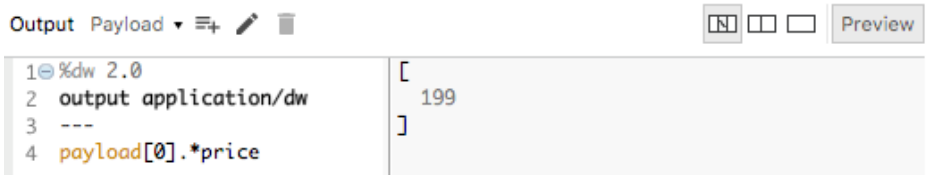
16. Change the output type to application/dw.

Return values for the first object in the collection

17. Change the DataWeave expression to payload[0].

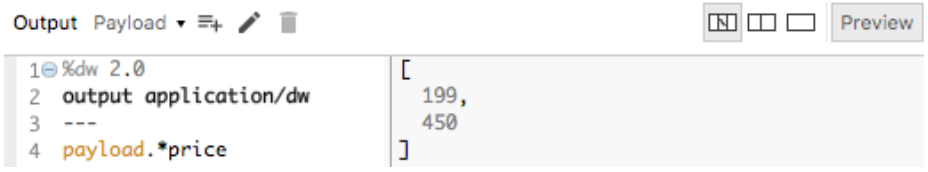
18. Change the DataWeave expression to payload[0].price.

19. Change the DataWeave expression to payload[0].*price.

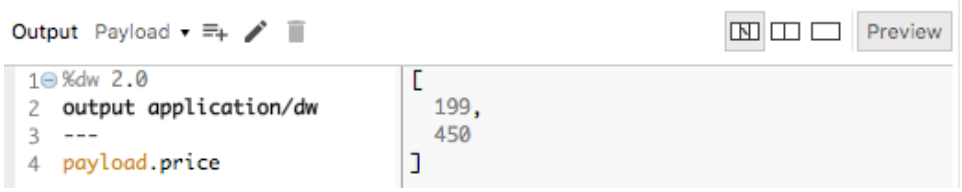


Return collections of values

20. Change the DataWeave expression to return a collection of all the prices.

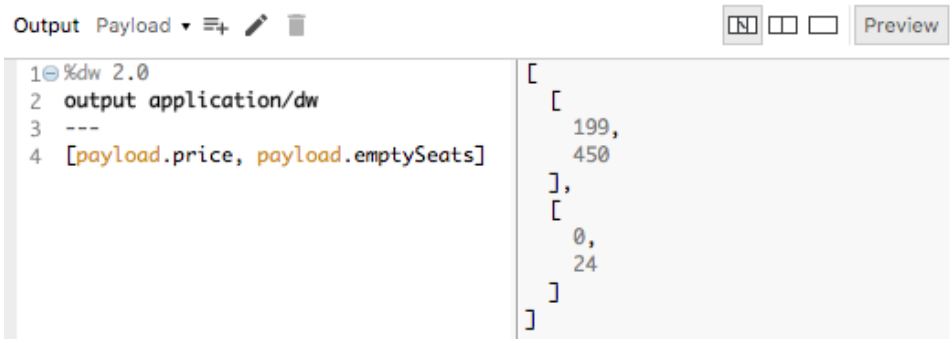


21. Change the DataWeave expression to payload.price; you should get the same result.



22. Change the DataWeave expression to return a collection of prices and available seats.

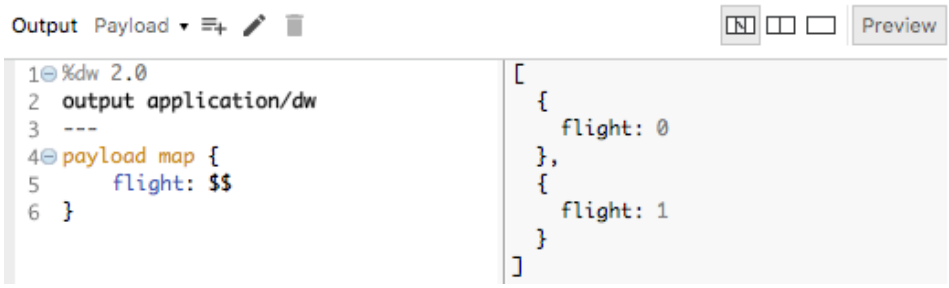
```
[payload.price, payload.emptySeats]
```



Use the map operator to return object collections with different data structures

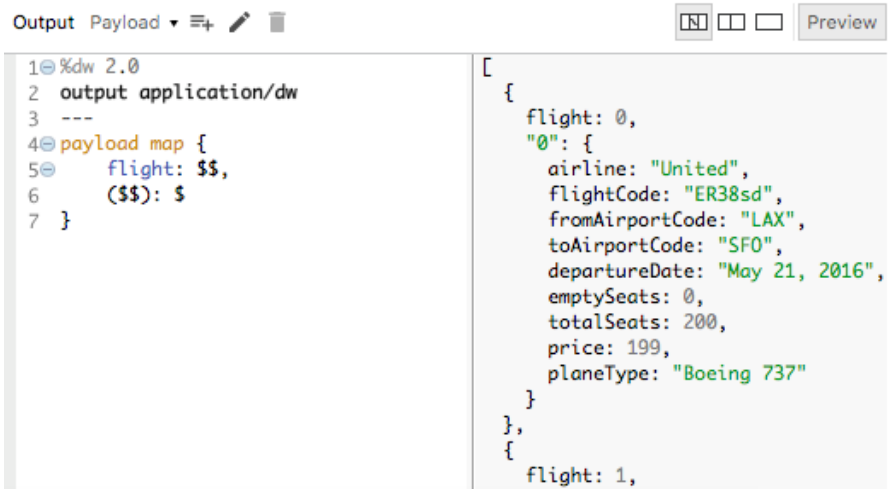
23. Change the DataWeave expression to payload map $.

24. Change the DataWeave expression to set a property called flight for each object that is equal to its index value in the collection.



25. Change the DataWeave expression to create a property called destination for each object that is equal to the value of the toAirportCode field in the payload collection.

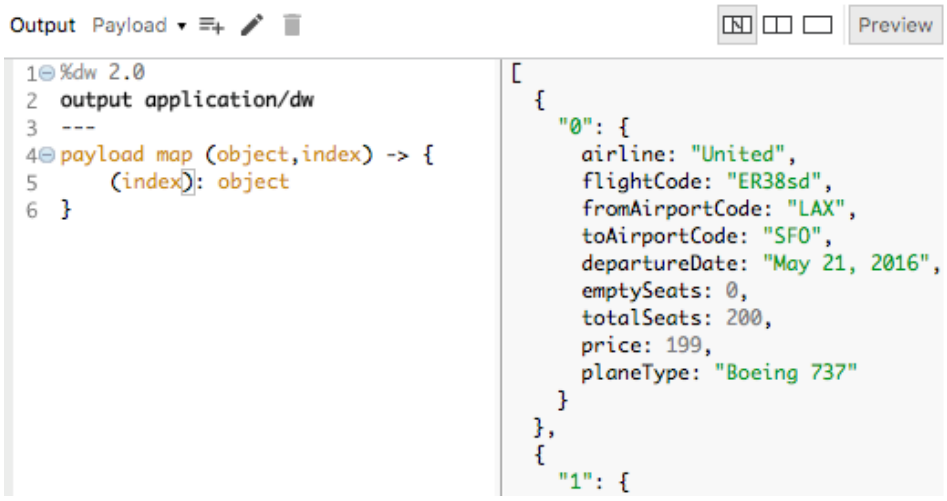
26. Change the DataWeave expression to dynamically add each of the properties present on the payload objects.



27. Remove the first flight property assignment.
28. Change the DataWeave expression so the name of each object is flight+index and you get flight0, flight1, and flight2.

Use explicit named parameters with the map operator

29. Change the map operator so that it uses two parameters called object and index and set the field name to just the index value.



30. Change the DataWeave expression so the name of each object is flight+index and you get flight0, flight1, and flight2.

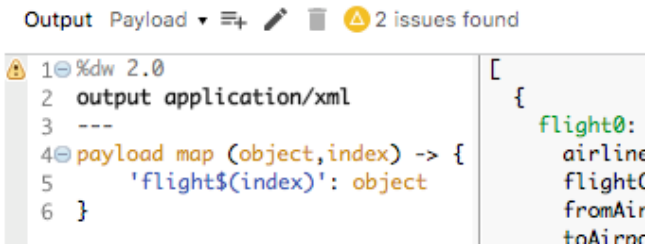
Note: If you want to test the application, change the output type to application/json and then in Advanced REST Client, make a request to http://localhost:8081/multipleflights and replace the request body with JSON from the flights-example.json file.

Change the expression to output different data types

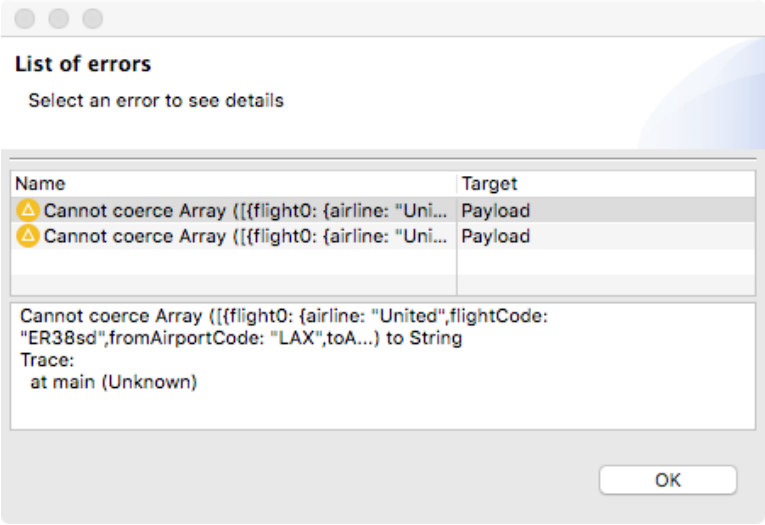
31. Change the DataWeave expression output type from application/dw to application/json.

32. Change the output type to application/java.

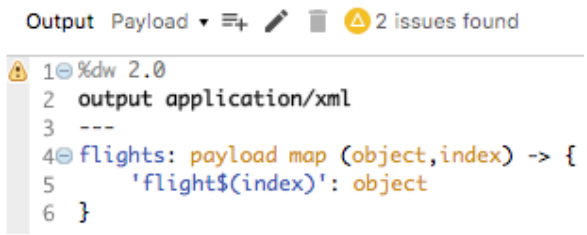
33. Change the output type to application/xml; you should get an issue displayed.



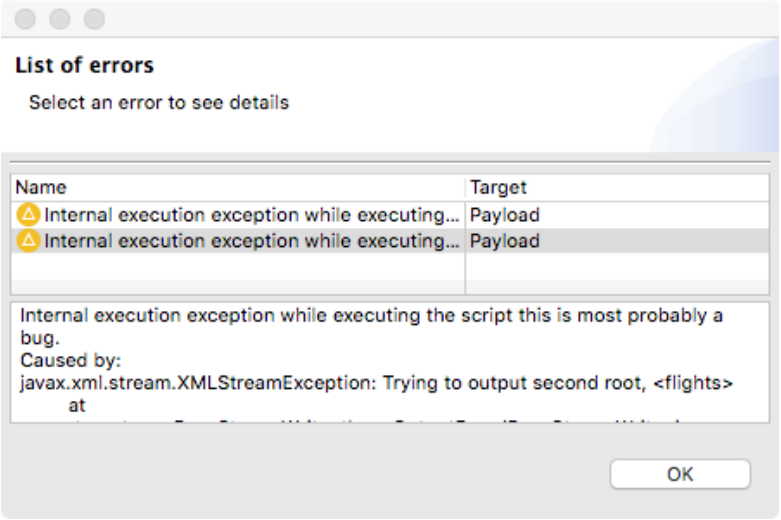
34. Click the warning icon and review the issues.



35. Change the DataWeave expression to create a root node called flights; you should still get an issue.



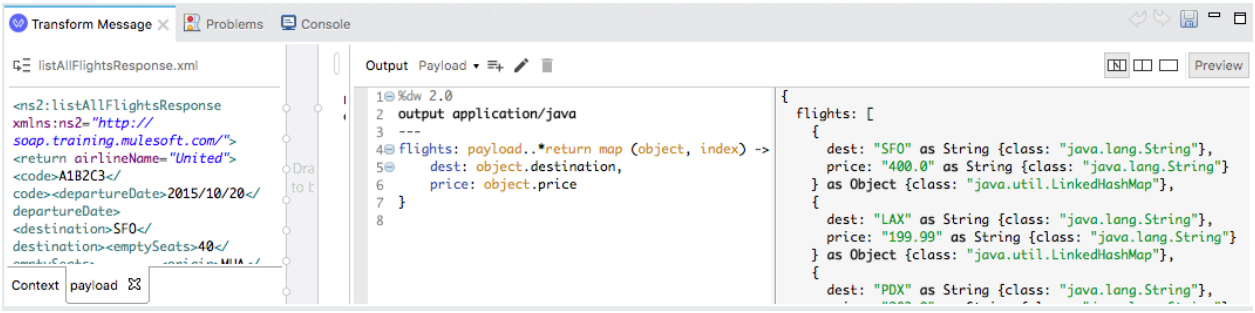
36. Click the warning icon and review the issues.



Walkthrough 11-4: Transform to and from XML with repeated elements

In this walkthrough, you continue to work with the JSON data for multiple flights posted to the flow. You will:

- Transform a JSON array of objects to XML.
- Replace sample data associated with a transformation.
- Transform XML with repeated elements to different data types.



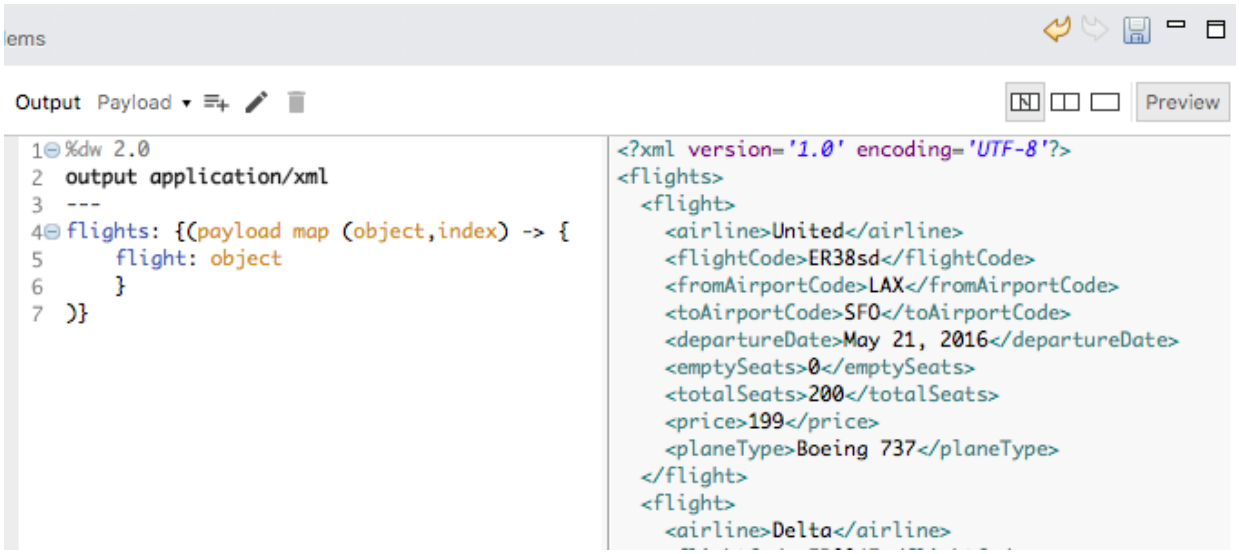
Transform the JSON array of objects to XML

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Change the expression to map each item in the input array to an XML element.

```
flights: {(payload map (object,index) -> {
    'flight$(index)': object
    }
)}
```

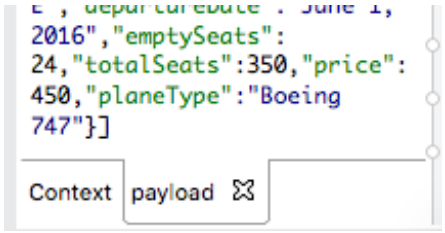
3. Look at the preview; the JSON should be transformed to XML successfully.

4. Modify the expression so the flights object has a single property called flight.

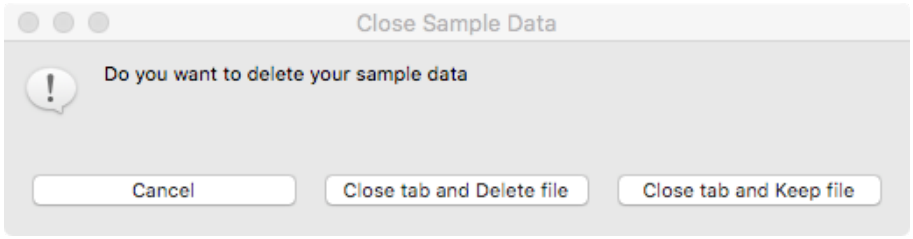


Change the input metadata to XML

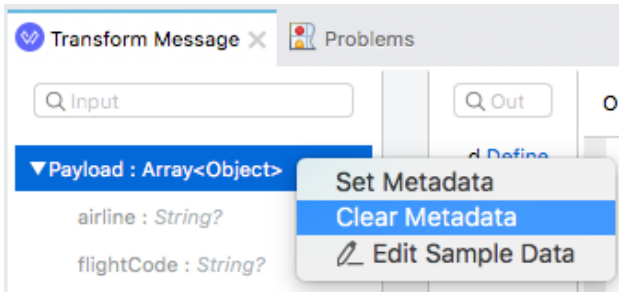
5. In the input section, click the x on the payload tab to close it.



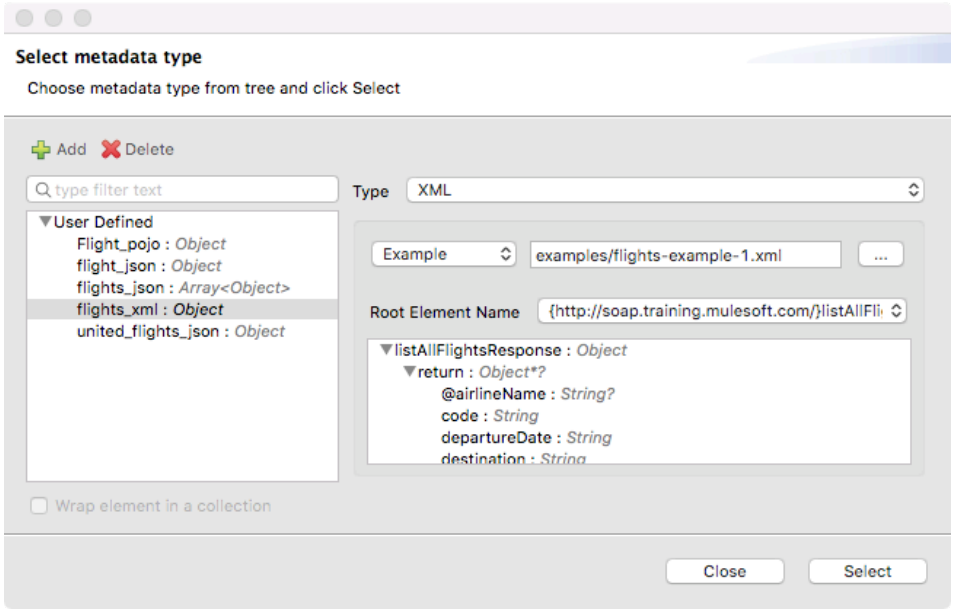
6. In the Close Sample Data dialog box, click Close tab and Keep file.



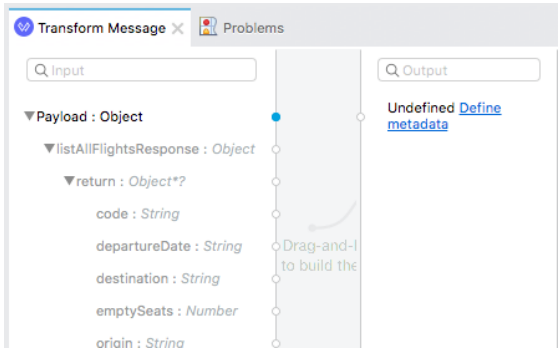
7. In the input section, right-click Payload and select Clear Metadata.



8. Click the Define metadata link.
9. In the Select metadata type dialog box, select flights_xml and click Select.

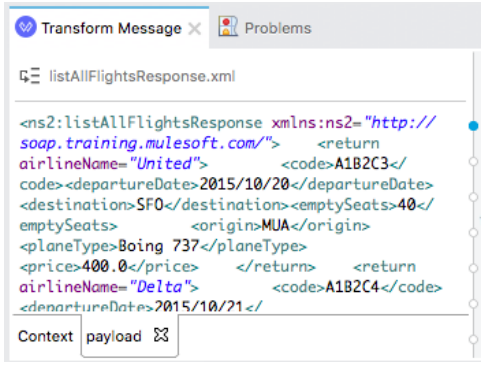


10. In the Transform Message Properties view, you should now see new metadata for the input.

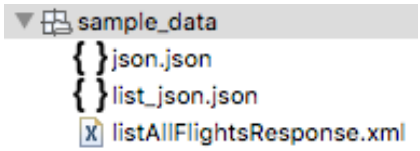


Preview sample data and sample output

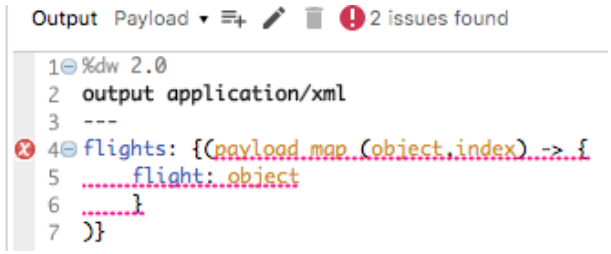
11. In the preview section, click the Create required sample data to execute preview link.
12. Look at the XML sample payload in the input section.



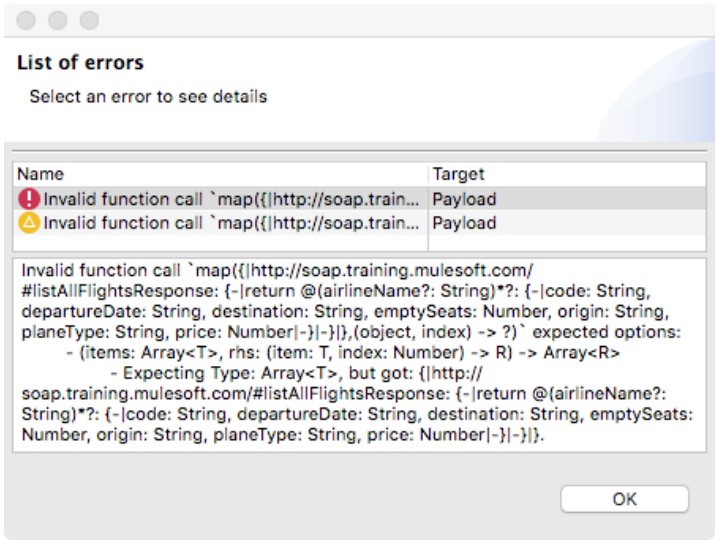
13. Look at the sample_data folder in src/test/resources.



14. Review the transformation expression; you should get an error.

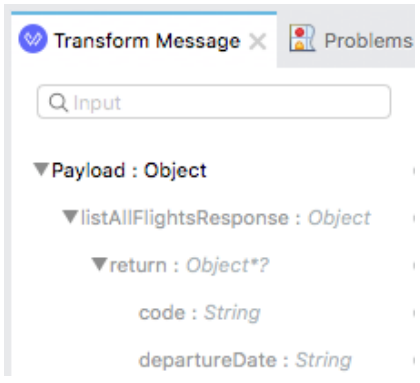


15. Look at the error.



Transform XML with repeated elements to a different XML structure

16. Look at the payload metadata in the input section.



17. Change the DataWeave expression so that the map is iterating over payload..return.

18. Change the DataWeave expression so that the map is iterating over the repeated return elements of the input.

19. Change the DataWeave expression to return flight elements with dest and price elements that map to the corresponding destination and price input values.

Note: If you want to test the application with Advanced REST Client, be sure to change the content-type header to application/XML and replace the request body with XML from the flights-example.xml file.

Change the expression to output different data types

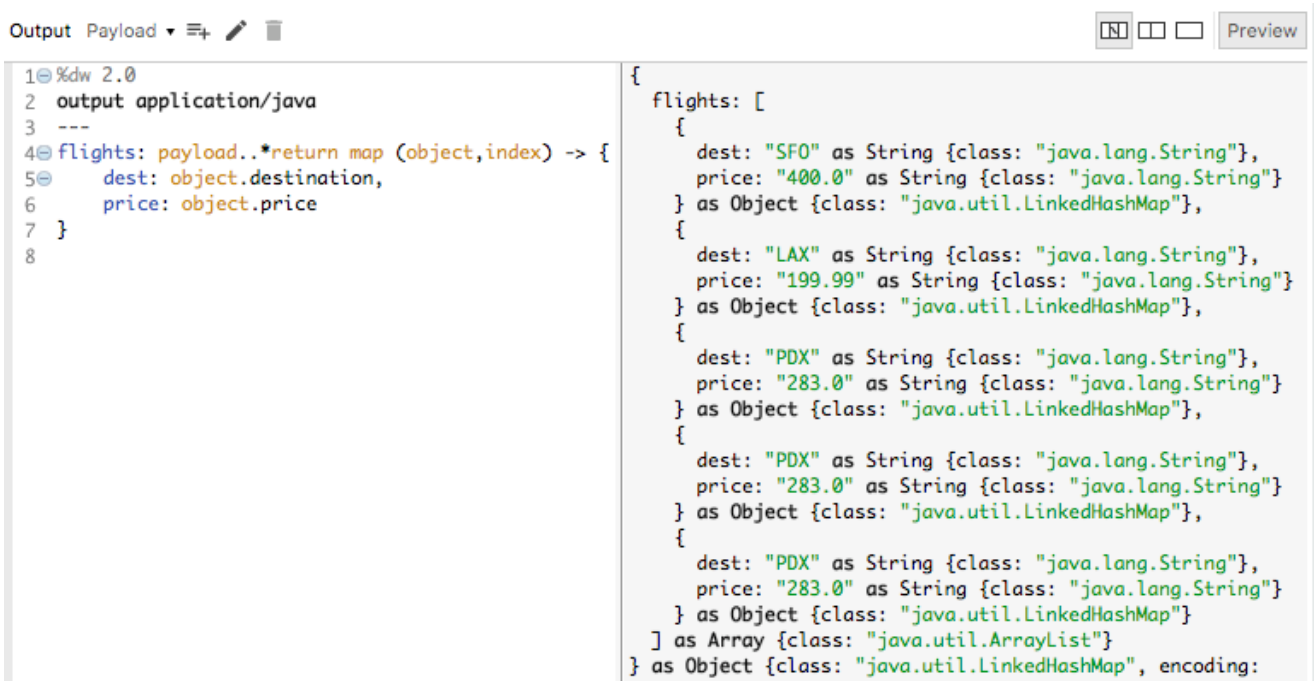
20. Change the output type from application/xml to application/dw.

21. Change the output type to application/json.

22. Change the output type to application/java.

23. In the DataWeave expression, remove the {(}) around the map expression.

24. Change the DataWeave expression to remove the flight property; you should get an ArrayList with five LinkedHashMaps.



Walkthrough 11-5: Define and use variables and functions

In this walkthrough, you continue to work with the DataWeave transformation in postMultipleFlights. You will:

- Define and use a global constant.
- Define and use a global variable that is equal to a lambda expression.
- Define and use a lambda expression assigned to a variable as a function.
- Define and use a local variable.

Define and use a global constant

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Change the output type to application/dw.
3. In the header, define a global variable called numSeats that is equal to 400.

   ```
   var numSeats = 400
   ```

4. In the body, add a totalSeats property that is equal to the numSeats constant.

   ```
   totalSeats: numSeats
   ```

```
var numSeats = 400
---
flights: payload..*return map (object,index) -> {
    dest: object.destination,
    price: object.price,
    totalSeats: numSeats
}
```

5. Look at the preview.

6. Comment out the variable declaration in the header.

```
//var numSeats = 400
```

Define and use a global variable that is equal to a lambda expression that maps to a constant

7. In the header, define a global variable called numSeats that is equal to a lambda expression with an input parameter x equal to 400.
8. Inside the expression, set numSeats to x.

```
var numSeats = (x=400) -> x
```

9. Add parentheses after numSeats in the totalSeats property assignment.

```
totalSeats: numSeats()
```

10. Look at the preview.

Add a plane property to the output

11. Add a plane property to the DataWeave expression equal to the value of the input planeType values.
12. Look at the values of plane type in the preview.

Define and use a global variable that is that is equal to a lambda expression with input parameters

13. Comment out the numSeats variable declaration in the header.
14. In the header, define a global variable called numSeats that is equal to a lambda expression with an input parameter called planeType of type String.

```
var numSeats = (planeType: String) ->
```

15. Inside the expression, add an if/else block that checks to see if planeType contains the string 737 and sets numSeats to 150 or 300.

```
var numSeats = (planeType: String) ->
    if (planeType contains('737'))
        150
    else
        300
```

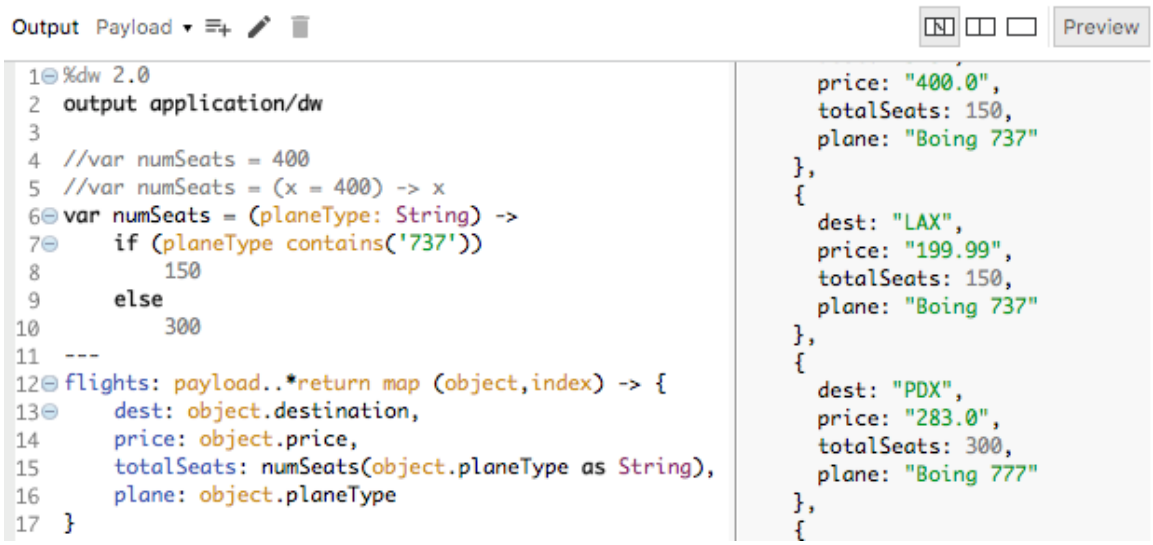
16. Change the totalSeats property assignment to pass the object's planeType property to numSeats.

```
totalSeats: numSeats(object.planeType)
```

17. Force the argument to a String.

```
totalSeats: numSeats(object.planeType as String)
```

18. Look at the preview; you should see values of 150 and 300.



19. Surround the variable declaration in the header with /* and */ to comment it out.

Define and use a lambda expression assigned to a variable as a function

20. In the header, use the fun keyword to define a function called getNumSeats with a parameter called planeType of type String.

```
fun getNumSeats(planeType: String)
```

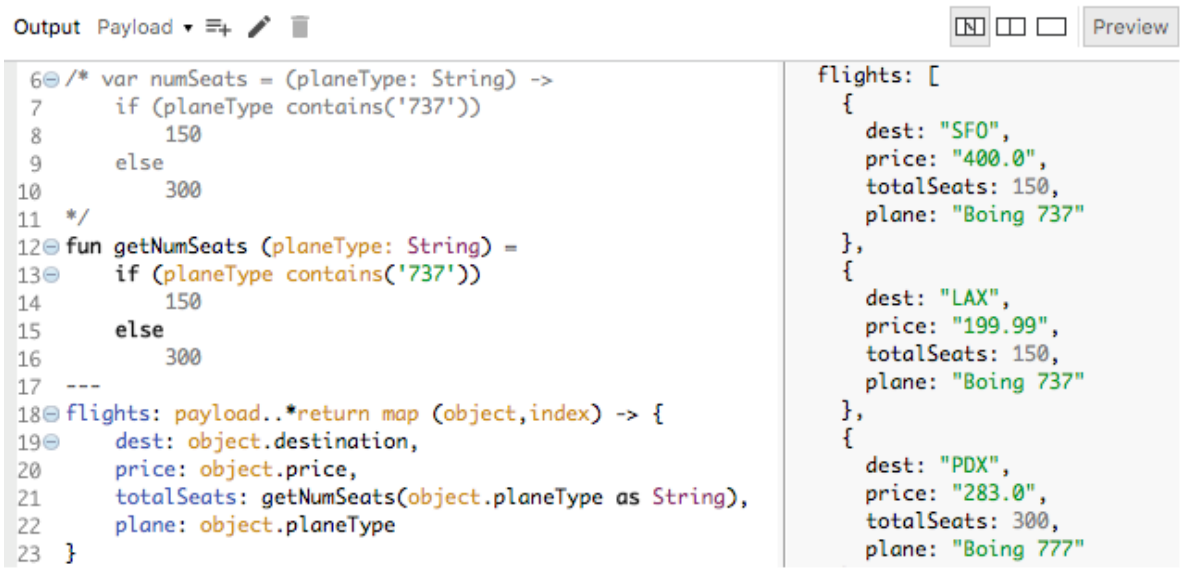
21. Copy the if/else expression in the numSeats declaration.
22. Set the getNumSeats function equal to the if/else expression.

```
fun getNumSeats(planeType: String) =
      if (planeType contains('737'))
            150
      else
            300
```

23. In the body, change the totalSeats property to be equal to the result of the getNumSeats function.

```
totalSeats: getNumSeats(object.planeType as String)
```

24. Look at the preview.



Create and use a local variable

25. In the body, surround the existing DataWeave expression with parentheses and add the using keyword in front of it.

```
---
using (flights: payload..*return map (object,index) -> {
    dest: object.destination,
    price: object.price,
    totalSeats: getNumSeats(object.planeType as String),
    plane: object.planeType
}
)
```

26. Change the colon after flights to an equal sign.
27. Modify the indentation to your liking.
28. After the local variable declaration, set the transformation expression to be the local flights variable.

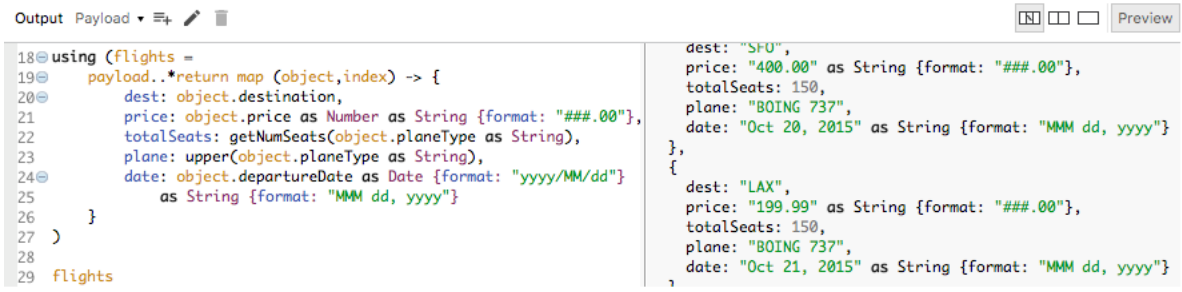
```
using (flights =
    payload..*return map (object,index) -> {
        dest: object.destination,
        price: object.price,
        totalSeats: getNumSeats(object.planeType as String),
        plane: object.planeType
    }
)

flights
```

29. Look at the preview.

Walkthrough 11-6: Coerce and format strings, numbers, and dates

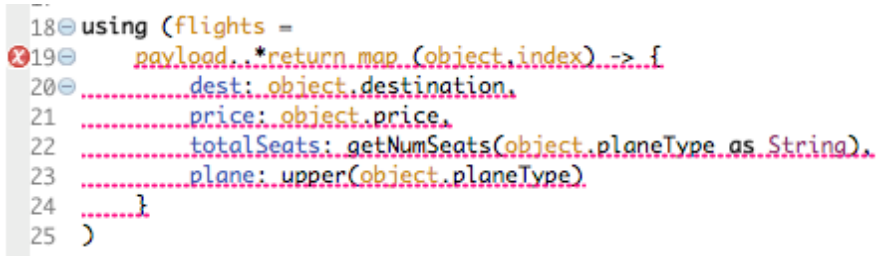
In this walkthrough, you continue to work with the XML flights data posted to the postMultipleFlights flow. You will:

- Coerce data types.
- Format strings, numbers, and dates.



Format a string

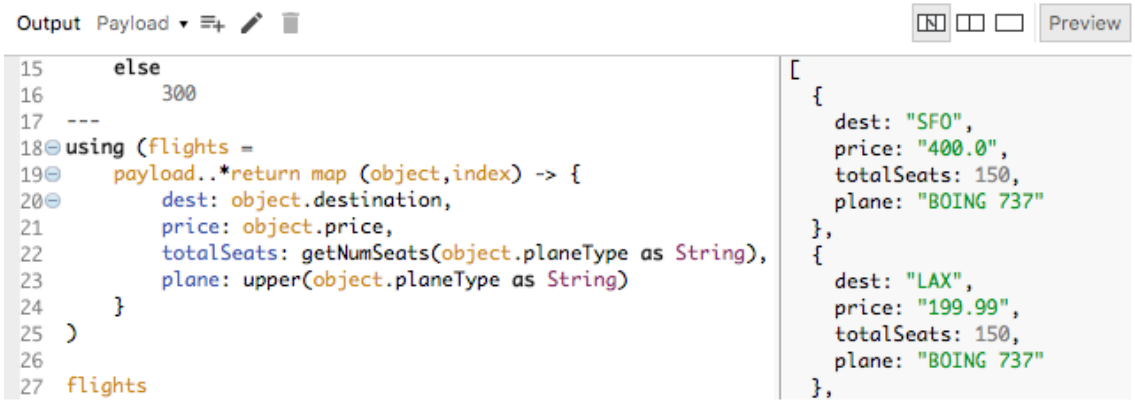
1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Use the upper function to return the plane value in uppercase.



3. Coerce the argument to a String.

```
plane: upper(object.planeType as String)
```

4. Look at the preview.

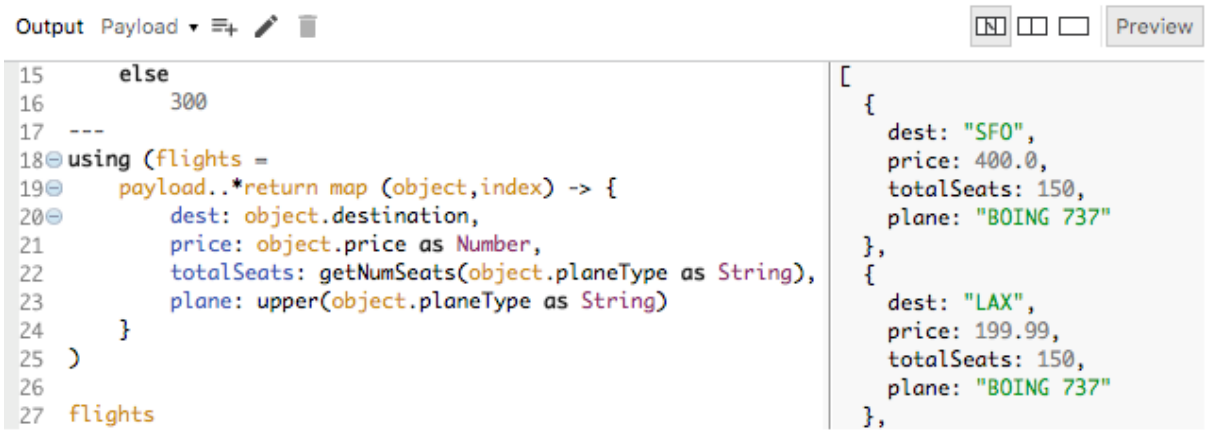


Coerce a string to a number

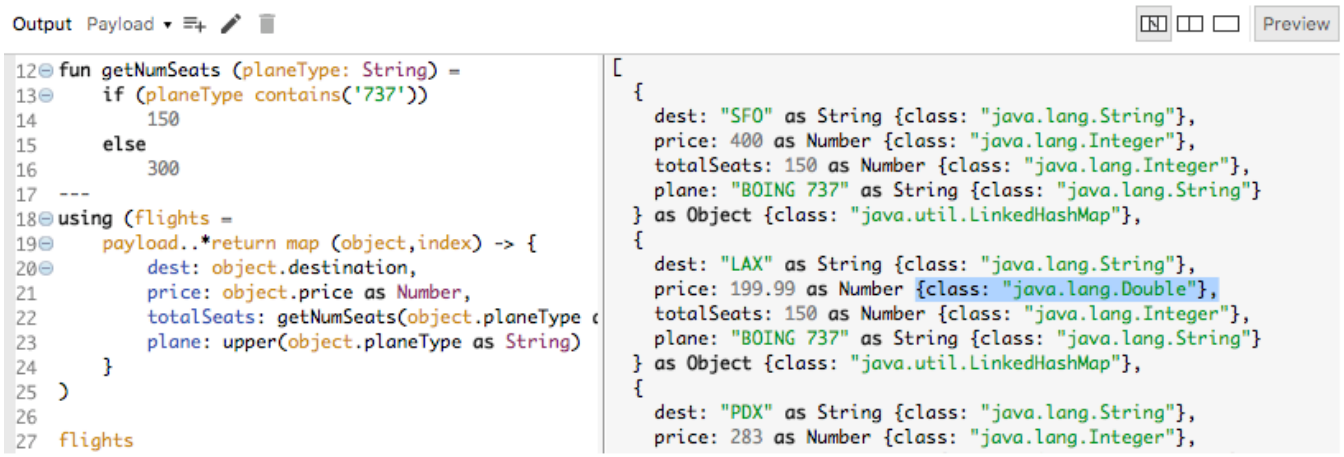
5. In the preview, look at the data type of the prices; you should see that they are strings.
6. Change the DataWeave expression to use the as keyword to return the prices as numbers.

```
price: object.price as Number,
```

7. Look at the preview.



8. Change the output type from application/dw to application/java.
9. Look at the preview; you should see the prices are now either Integer or Double objects.

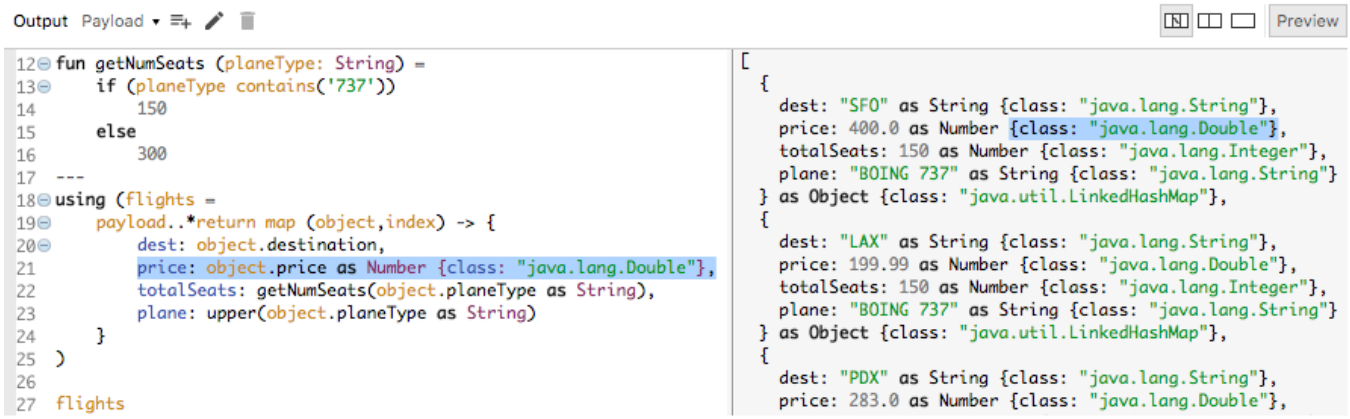


Coerce a string to a specific type of number object

10. Change the DataWeave expression to use the class metadata key to coerce the prices to java.lang.Double objects.

```
price: object.price as Number {class:"java.lang.Double"},
```

11. Look at the preview; you should see the prices are now all Double objects.



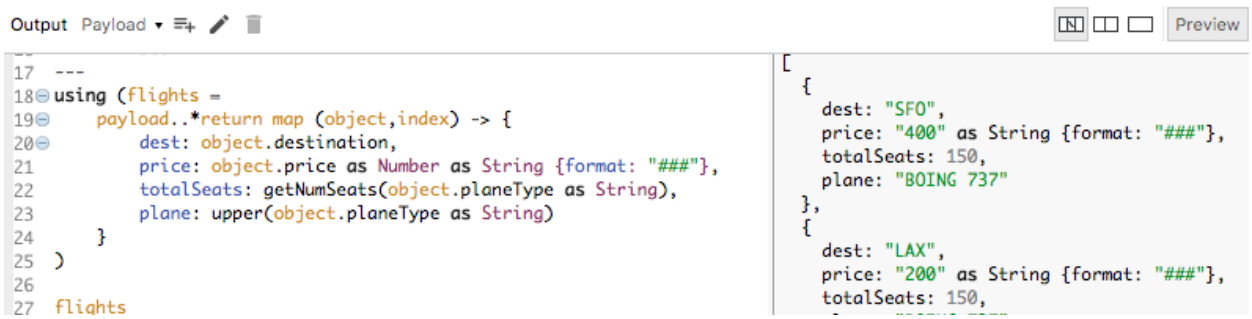
12. Change the output type from application/java back to application/dw.

Format a number

13. Remove the coercion of the price to a java.lang.Double.
14. Coerce the price to a String and use the format schema property to format the prices to zero decimal places.

```
price: object.price as Number as String {format: "###"},
```

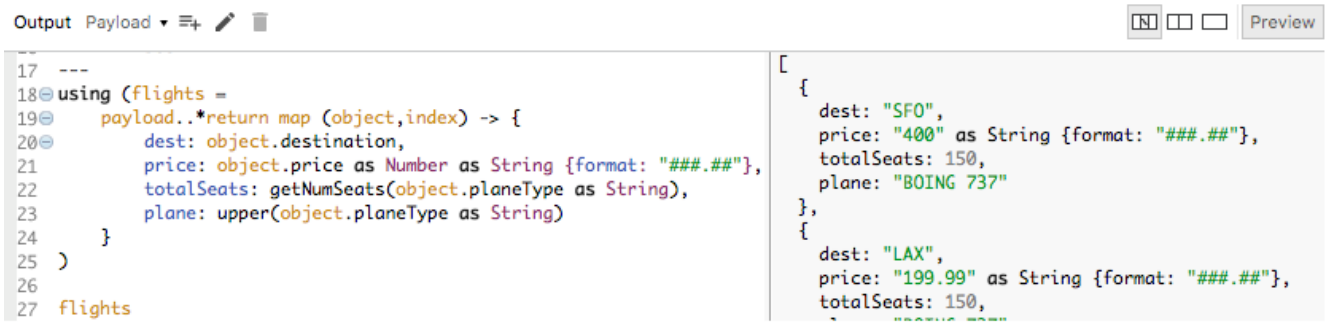
15. Look at the preview; the prices should now be formatted.



16. Change the DataWeave expression to format the prices to two decimal places.

```
price: object.price as Number as String {format: "###.##"},
```

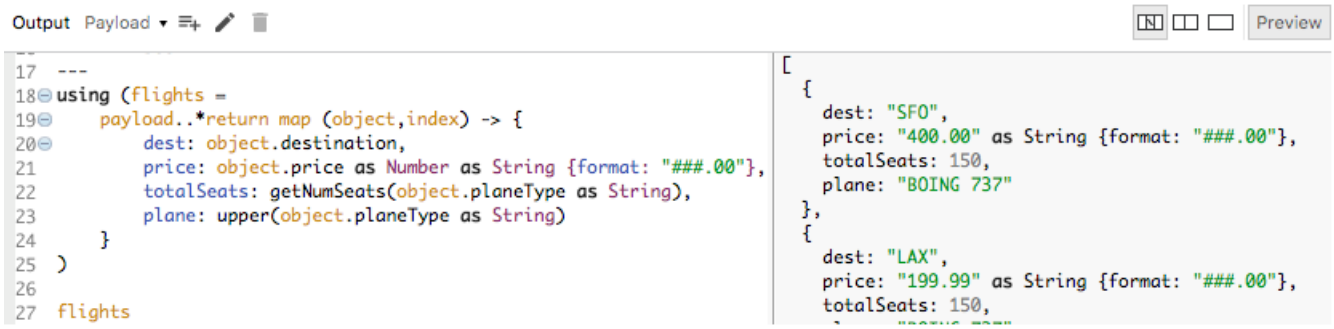
17. Look at the preview.



18. Change the DataWeave expression to format the prices to two minimal decimal places.

```
price: object.price as Number as String {format: "###.00"},
```

19. Look at the preview; all the prices should now be formatted to two decimal places.



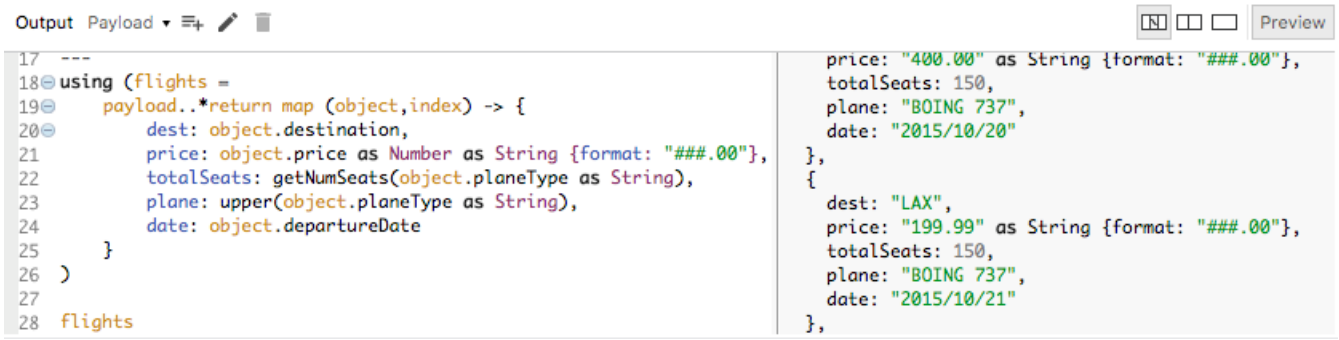
Note: If you are not a Java programmer, you may want to look at the Java documentation for the DecimalFormatter class to review documentation on patterns.
https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html

Coerce a string to a date

20. Add a date field to the return object.

```
date: object.departureDate
```

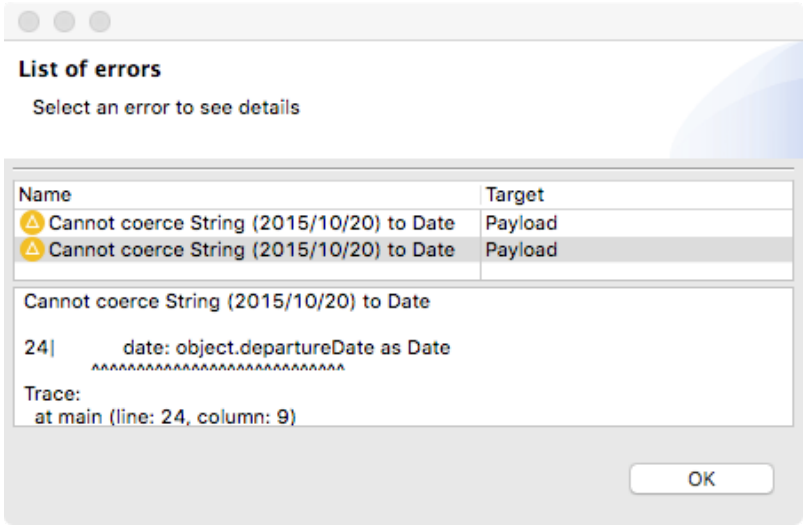
21. Look at the preview; you should see the date property is a String.



22. Change the DataWeave expression to use the as operator to convert departureDate to a date object; you should get an exception.

```
date: object.departureDate as Date
```

23. Look at the exception.



24. Look at the format of the dates in the preview.
25. Change the DataWeave expression to use the format schema property to specify the pattern of the input date strings.

```
date: object.departureDate as Date {format: "yyyy/MM/dd"}
```

Note: If you are not a Java programmer, you may want to look at the Java documentation for the DateTimeFormatter class to review documentation on pattern letters.
https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

26. Look at the preview; you should see date is now a Date object.

Format a date

27. Use the as operator to convert the dates to strings, which can then be formatted.

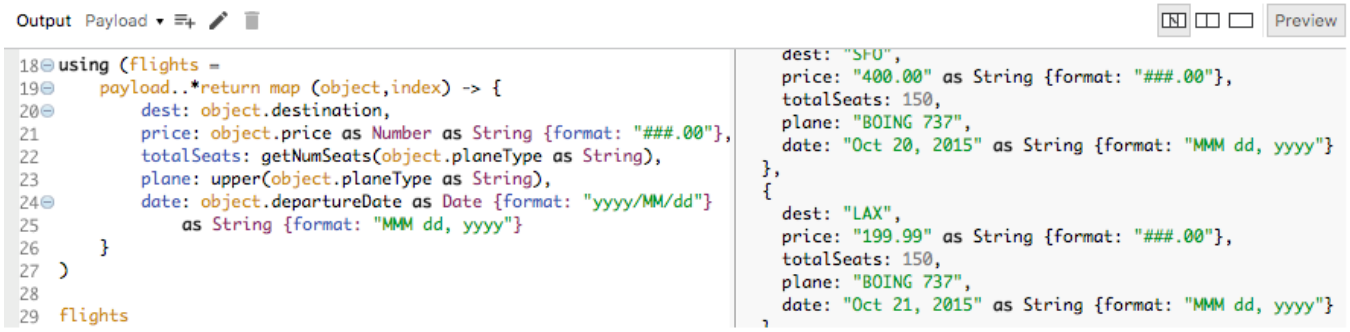
```
date: object.departureDate as Date {format: "yyyy/MM/dd"} as String
```

28. Look at the preview section; the date is again a String – but with a different format.

29. Use the format schema property with any pattern letters and characters to format the date strings.

```
date: object.departureDate as Date {format: "yyyy/MM/dd"} as String
{format: "MMM dd, yyyy"}
```

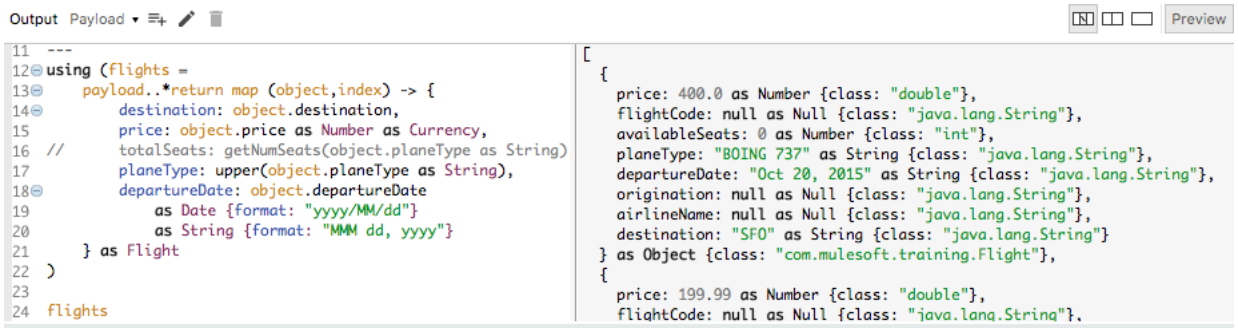
30. Look at the preview; the dates should now be formatted according to the pattern.



Walkthrough 11-7: Define and use custom data types

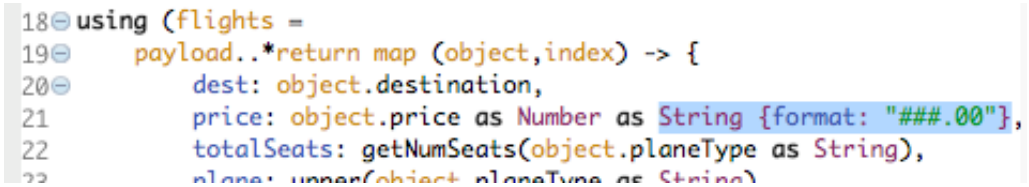
In this walkthrough, you continue to work with the flight JSON posted to the flow. You will:

- Define and use custom data types.
- Transform objects to POJOs.



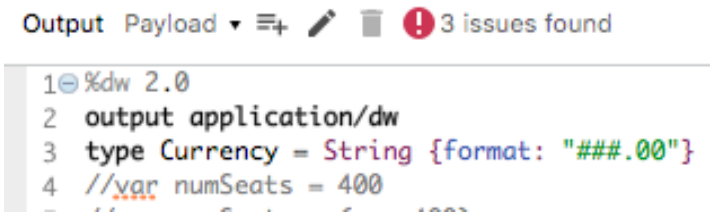
Define a custom data type

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Select and cut the string formatting expression for the prices.



3. In the header section of the expression, define a custom data type called Currency.
4. Set it equal to the value you copied.

```
type Currency = String {format: "###.00"}
```

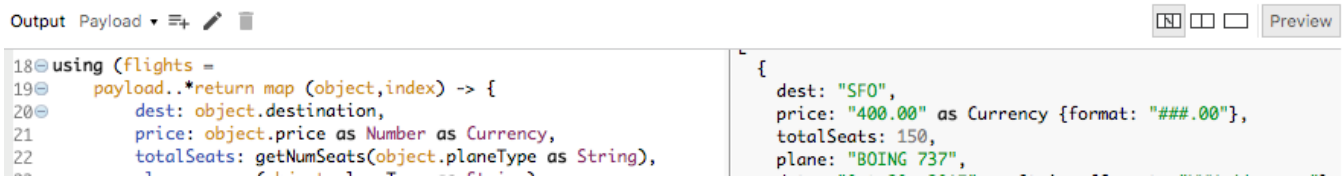


Use a custom data type

5. In the DataWeave expression, set the price to be of type currency.

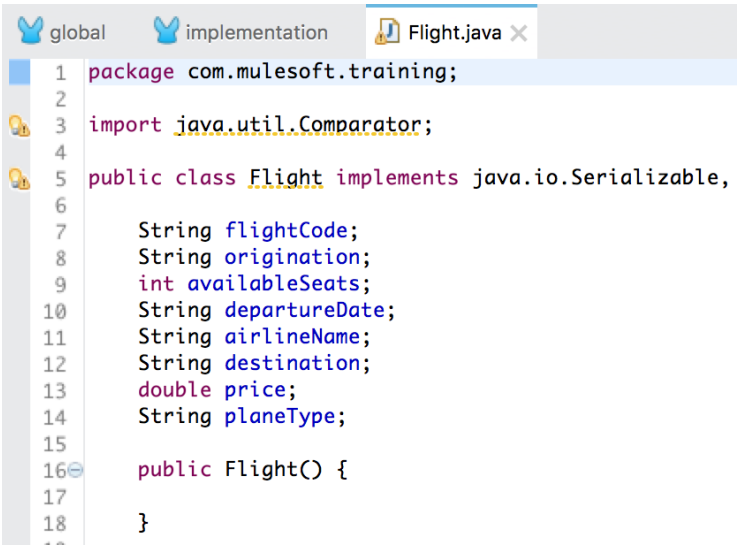
```
price: object.price as Number as Currency,
```

6. Look at the preview; the prices should still be formatted as strings to two decimal places.



Transform objects to POJOs

7. Open the Flight.java class in the project's src/main/java folder and look at the names of the properties.



8. Return to postMultipleFlights in implementation.xml.
9. In the header section of the expression, define a custom data type called flight that is of type com.mulesoft.traning.Flight.

```
type Flight = Object {class: "com.mulesoft.training.Flight"}
```

10. In the DataWeave expression, set the map objects to be of type Flight.

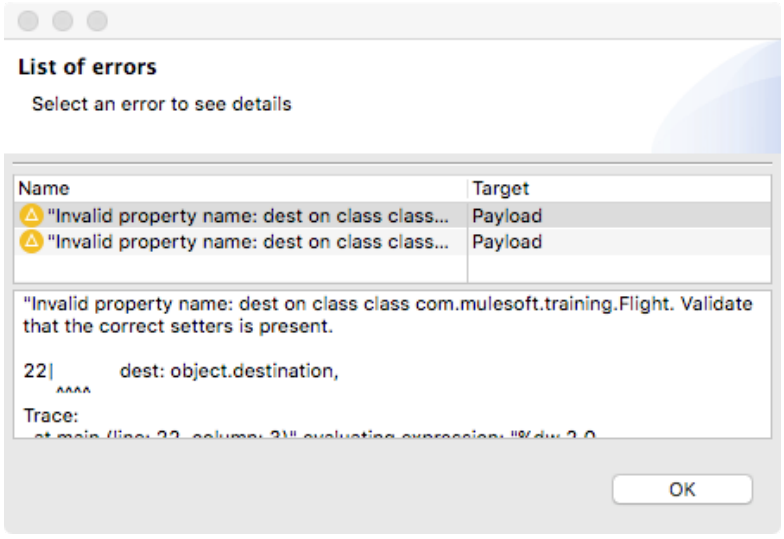
```
using (flights =
    payload..*return map (object,index) -> {
        dest: object.destination,
        price: object.price as Number as Currency,
        totalSeats: getNumSeats(object.planeType as String),
        plane: upper(object.planeType as String),
        date: object.departureDate as Date {format: "yyyy/MM/dd"}
            as String {format: "MMM dd, yyyy"}
    } as Flight
)
```

11. Look at the preview.

12. Change the output type from application/dw to application/java.

```
%dw 2.0
output application/java
type Currency = String {format: "###.00"}
type Flight = Object {class: "com.mulesoft.training.Flight"}
```

13. Look at the issue you get.



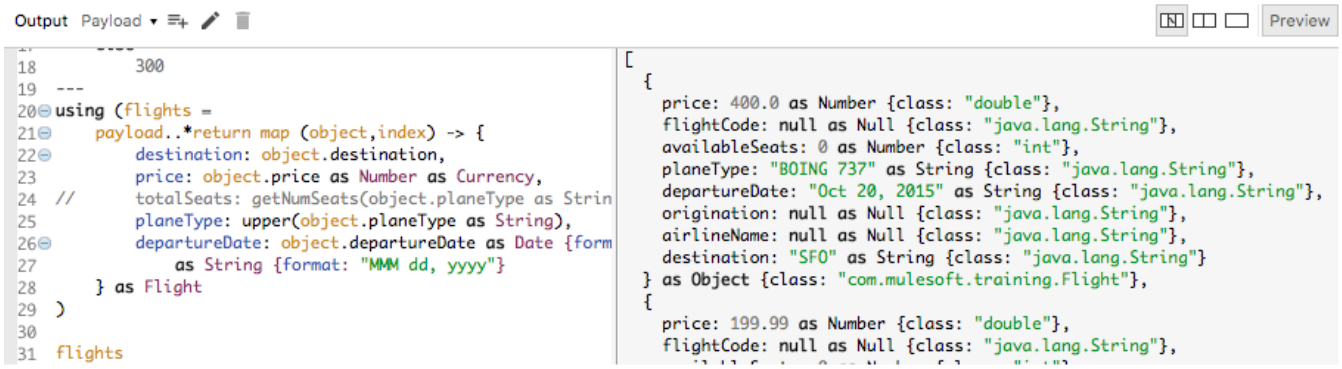
14. Change the name of the dest key to destination.

```
destination: object.destination,
```

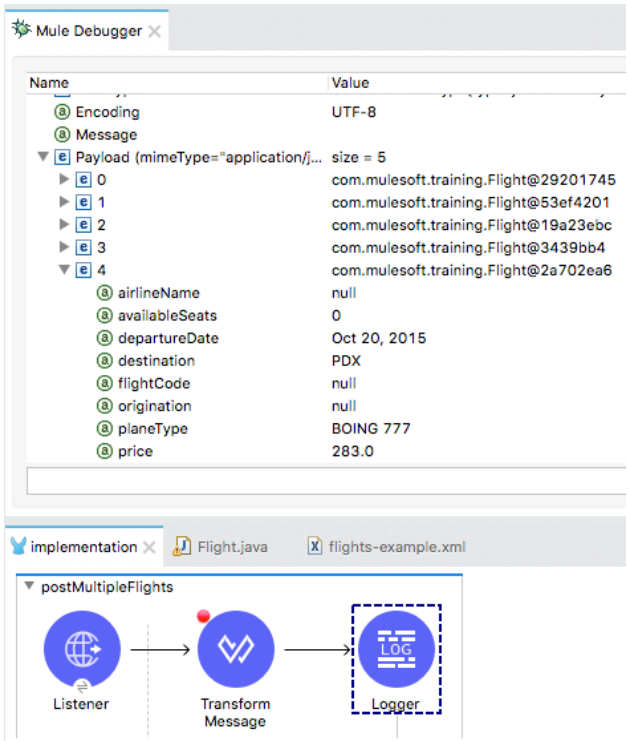
15. Change the other keys to match the names of the Flight class properties:

    - plane to planeType
    - date to departureDate

16. Comment out the totalSeats property.
17. Look at the preview.



Note: If you want to test the application with Advanced REST Client, be sure to change the content-type header to application/XML and replace the request body with XML from the flights-example.xml file.



Walkthrough 11-8: Use DataWeave functions

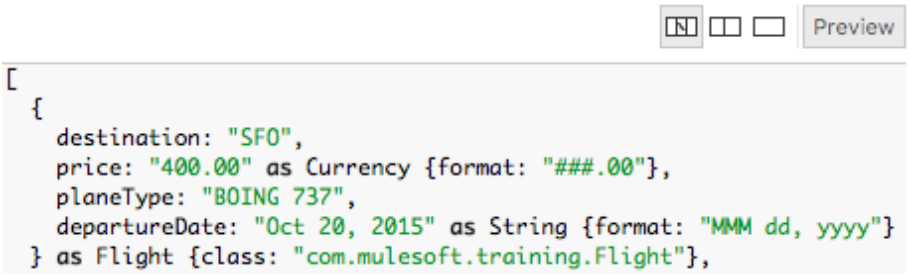
In this walkthrough, you continue to work with the flights JSON posted to the flow. You will:

- Use functions in the Core module that are imported automatically.
- Replace data values using pattern matching.
- Order data, remove duplicate data, and filter data.
- Use a function in another module that you must explicitly import into a script to use.
- Dasherize data.

```
34 flights orderBy $.departureDate
35         orderBy $.price
36         distinctBy $
37         filter ($.availableSeats !=0)
```

Use the replace function

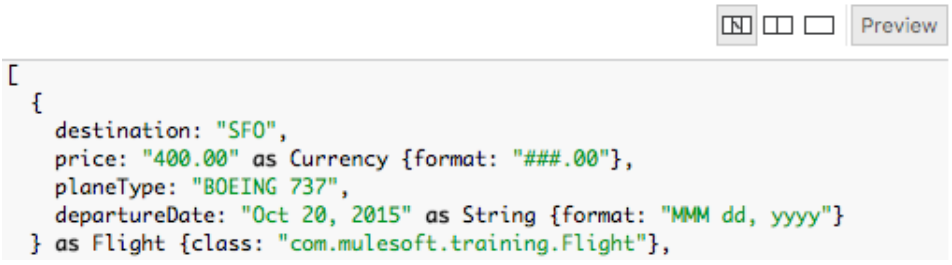
1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Change the output type from application/java to application/dw.
3. Look at the preview and see that Boeing is misspelled.



4. For planeType, use the replace function to replace the string Boing with Boeing.

```
planeType: upper(replace(object.planeType,/(Boing)/) with "Boeing"),
```

5. Look at the preview; Boeing should now be spelled correctly.

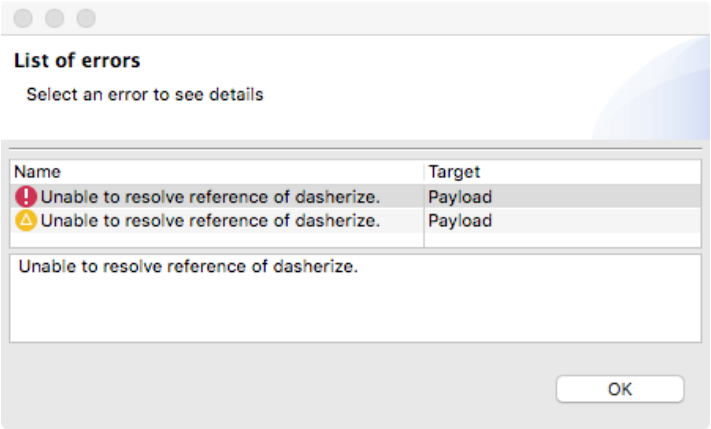


Use the dasherize function in the Strings module

6. For planeType, replace the upper function with the dasherize function.

```
 planeType: dasherize(replace(object.planeType,/(Boing)/) with
"Boeing"),
```

7. Look at the error you get.

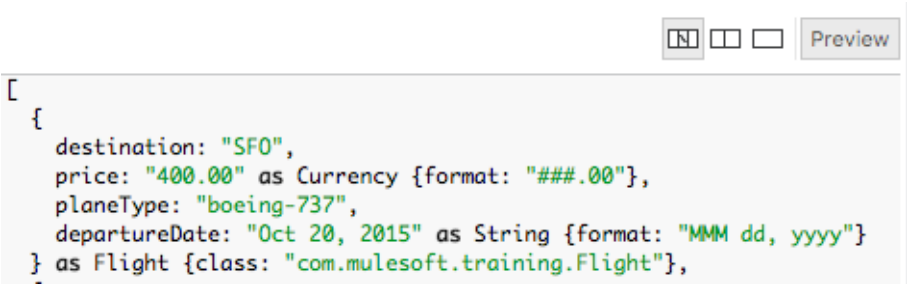


8. In the script header, add an import statement to import dasherize from the Strings module.

```
import dasherize from dw::core::Strings
```

```
%dw 2.0
output application/dw
import dasherize from dw::core::Strings
type Currency = String {format: "###.00"}
type Flight = Object {class: "com.mulesoft.training.Flight"}
```

9. Look at the preview.



Use the orderBy function

10. In the preview section, look at the flight prices; the flights should not be ordered by price.
11. Use the orderBy function to order the flights object by price.

```
flights orderBy $.price
```

12. Look at the preview; the flights should now be ordered by price.

13. Look at the three $283 flights; there is a duplicate and they are not ordered by date.
14. Change the DataWeave expression to sort flights by price and then date.

```
flights    orderBy $.departureDate
           orderBy $.price
```

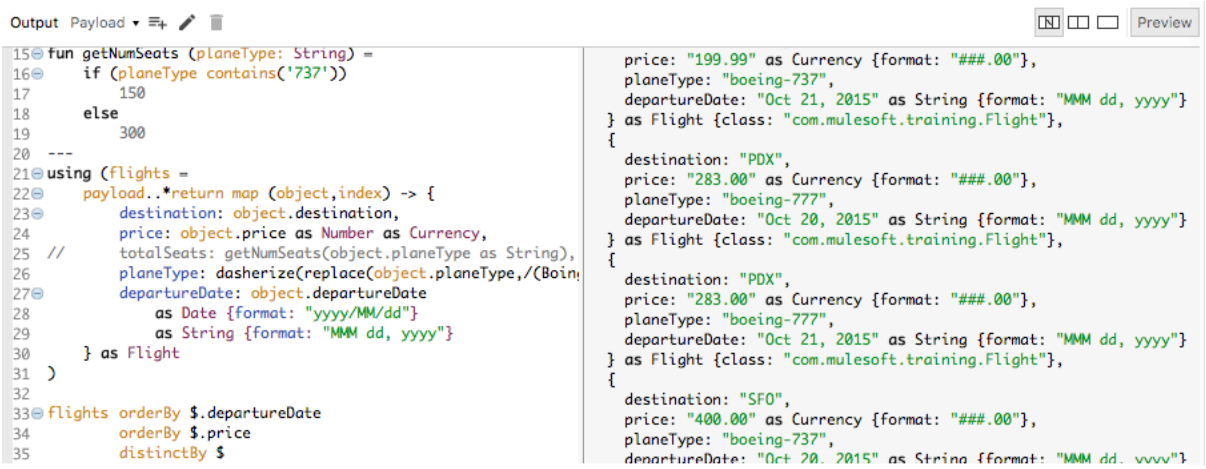
15. Look at the preview; the flights should now be ordered by price and flights of the same price should be sorted by date.

Remove duplicate data

16. Use the distinctBy function to first remove any duplicate objects.

```
flights    orderBy $.departureDate
           orderBy $.price
           distinctBy $
```

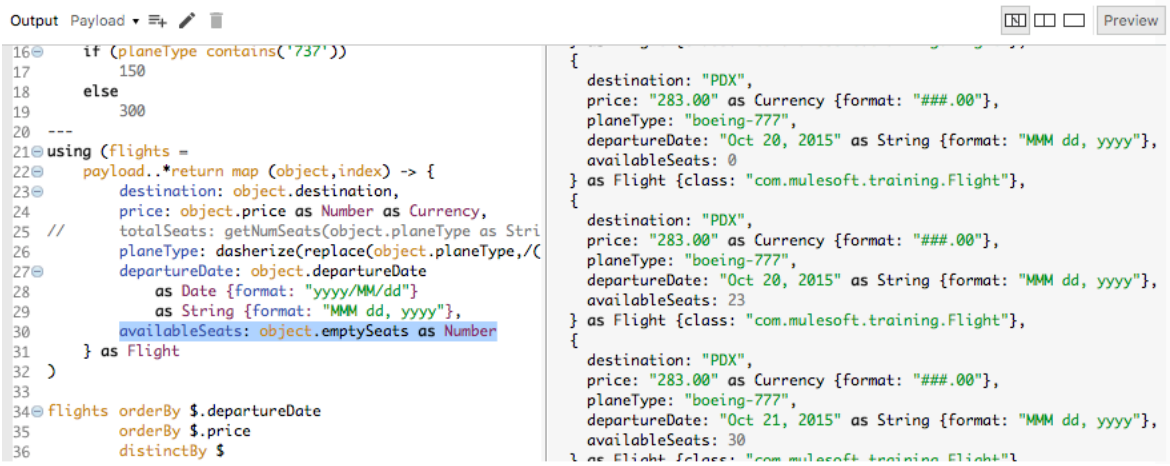
17. Look at the preview; you should now see only two $283 flights to PDX instead of three.



18. Add an availableSeats field that is equal to the emptySeats field and coerce it to a number.

```
availableSeats: object.emptySeats as Number
```

19. Look at the preview; you should get three $283 flights to PDX again.

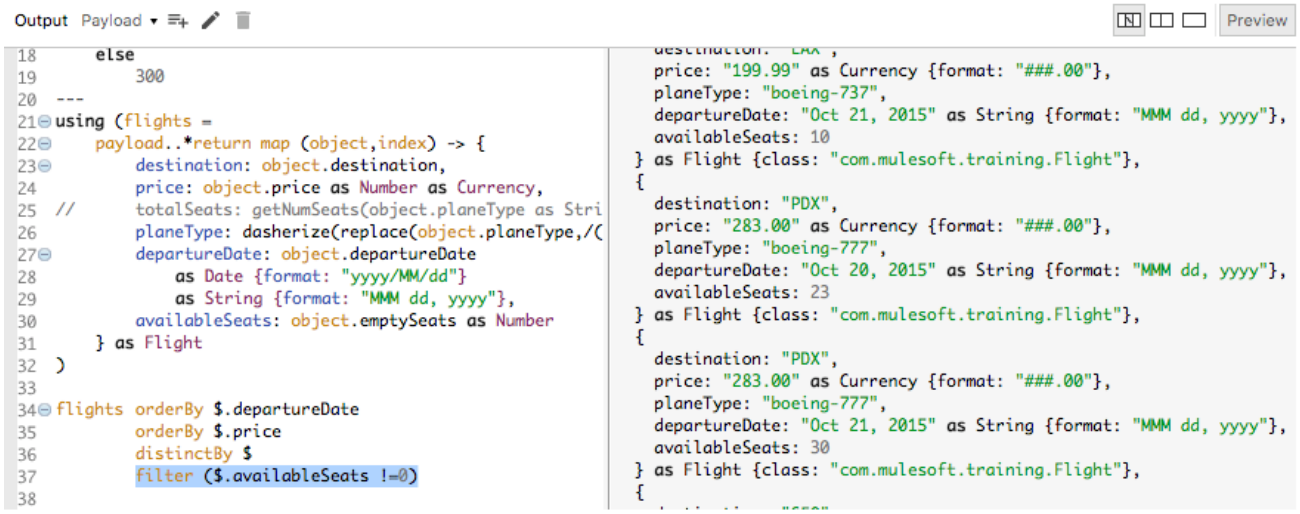


Use the filter function

20. In the preview, look at the values of the availableSeats properties; you should see some are equal to zero.
21. Use the filter function to remove any objects that have availableSeats equal to 0.

```
flights    orderBy $.departureDate
           orderBy $.price
           distinctBy $
           filter ($.availableSeats !=0)
```

22. Look at the preview; you should no longer get the flight that had no available seats.

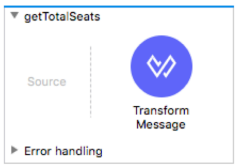


Walkthrough 11-9: Look up data by calling a flow

In this walkthrough, you continue to work with the transformation in getMultipleFlights. You will:

- Call a flow from a DataWeave expression.

```
totalSeats: lookup("getTotalSeats", {planeType: object.planeType}),
```

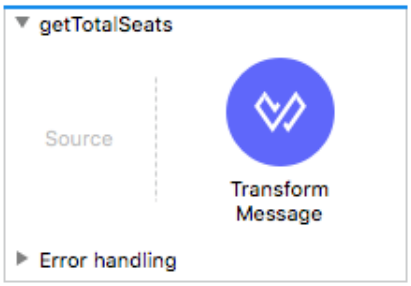


Create a lookup flow

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Copy the getNumSeats function.

```
fun getNumSeats (planeType: String) =
    if (planeType contains('737'))
        150
    else
        300
```

3. Drag a Transform Message component from the Mule Palette and drop it at the bottom of the canvas to create a new flow.
4. Change the name of the flow to getTotalSeats.



5. In the Transform Message properties view of the component in getTotalSeats, paste the function you copied into the script header.

6. In the script body, call the function, passing to it the payload.

Output Payload

```
%dw 2.0
output application/java

fun getNumSeats(planeType: String) =
    if (planeType contains ("737"))
        150
    else
        300

---
getNumSeats(payload.planeType)
```

Call a flow from a DataWeave expression

7. Return to the Transform Message properties view of the component in postMultipleFlights.
8. Add a property called totalSeats inside the expression (keep the other one commented out).
9. Change the return type of the map function from Flight to Object.

```
using (flights =
    payload..*return map (object,index) -> {
        destination: object.destination,
        price: object.price as Number as Currency,
//      totalSeats: getNumSeats(object.planeType as String),
        totalSeats:
        planeType: dasherize(replace(object.planeType,/(Boin
        departureDate: object.departureDate
            as Date {format: "yyyy/MM/dd"}
            as String {format: "MMM dd, yyyy"},
        availableSeats: object.emptySeats as Number
    } as Object
)
```

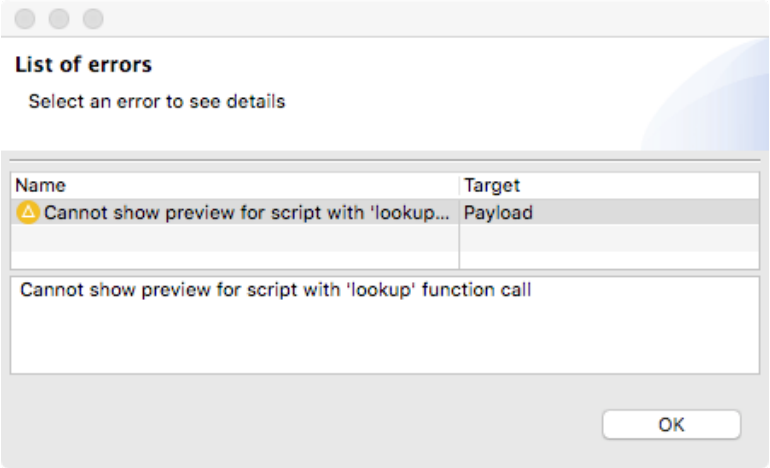
10. Set totalSeats equal to the return value from the DataWeave lookup() function.

```
totalSeats: lookup()
```

11. Pass to lookup() an argument that is equal to the name of the flow to call: "getTotalSeats".
12. Pass an object to lookup() as the second argument.
13. Give the object a field called type (to match what the flow is expecting) and set it equal to the value of the planeType field.

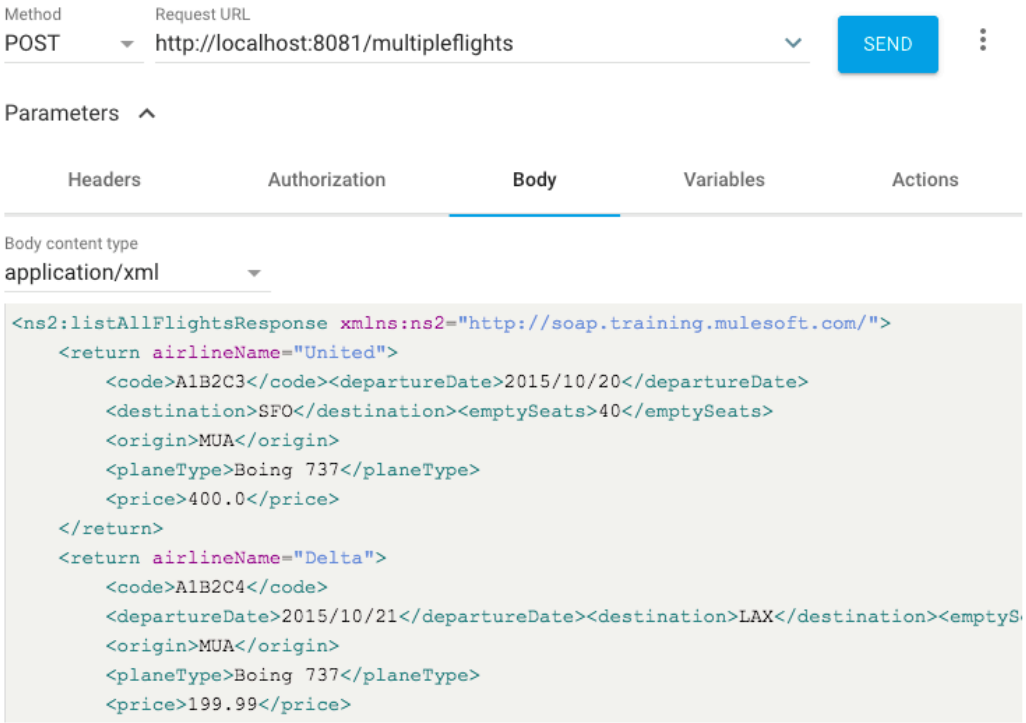
```
totalSeats: lookup("getTotalSeats",{planeType: object.planeType})
```

14. Look at the preview.
15. Look at the warning you get.



Test the application

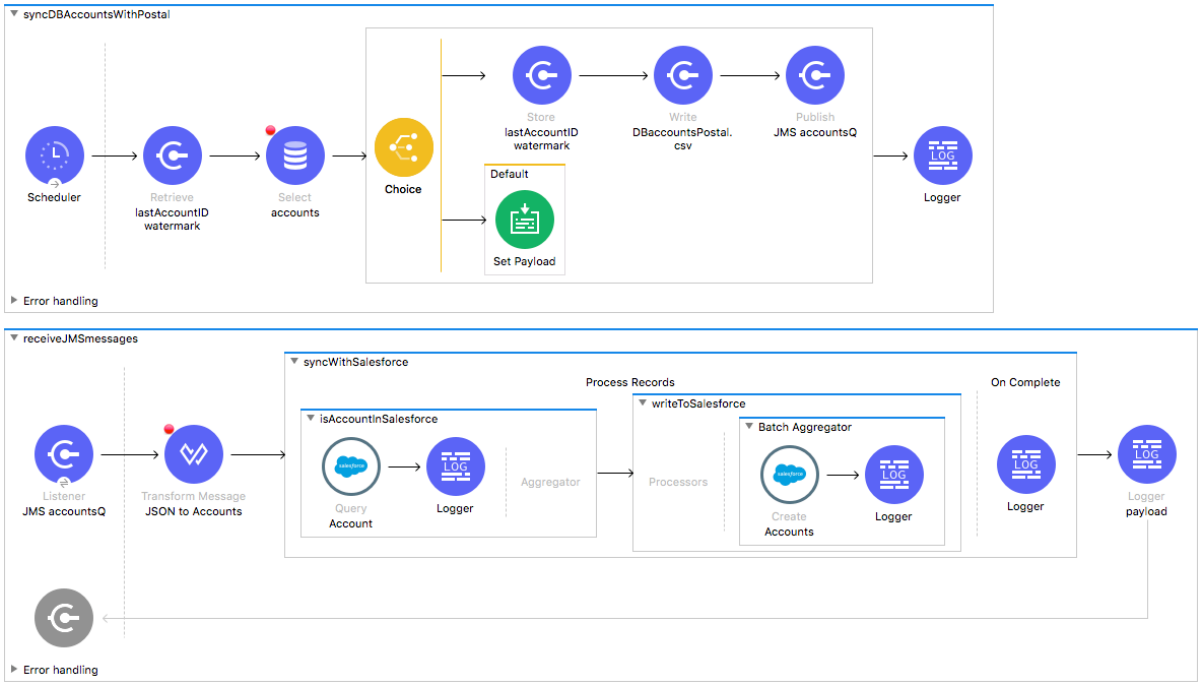
16. Run or debug the project.
17. In Advanced REST Client, make sure the method is set to POST and the request URL is set to http://localhost:8081/multipleflights.
18. Set a Content-Type header to application/xml.
19. Set the request body to the value contained in the flights-example.xml file in the src/test/resources folder.



20. Send the request; you should get the DataWeave representation of the return flight data and each flight should have a totalSeats property equal to 150 or 300.

21. Return to Anypoint Studio.
22. Stop the project.
23. Close the project.

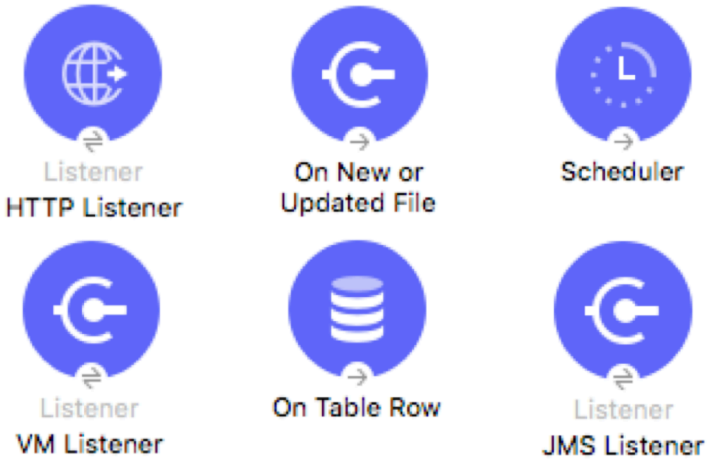
PART 3: Building Applications to Synchronize Data



At the end of this part, you should be able to:

- Trigger flows when files or database records are added or updated.
- Schedule flows.
- Persist and share data across flow executions.
- Publish and consume JMS messages.
- Process items in a collection sequentially.
- Process records asynchronously in batches.

Module 12: Triggering Flows



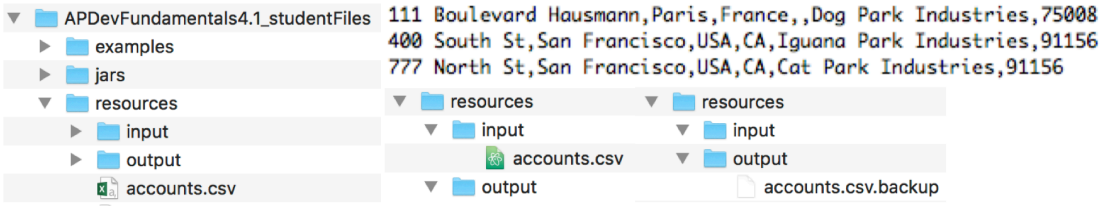
At the end of this module, you should be able to:

- Read and write files.
- Trigger flows when files are added, created, or updated.
- Trigger flows when new records are added to a database table.
- Schedule flows to run at a certain time or frequency.
- Persist and share data in flows using the Object Store.
- Publish and consume JMS messages.

Walkthrough 12-1: Trigger a flow when a new file is added to a directory

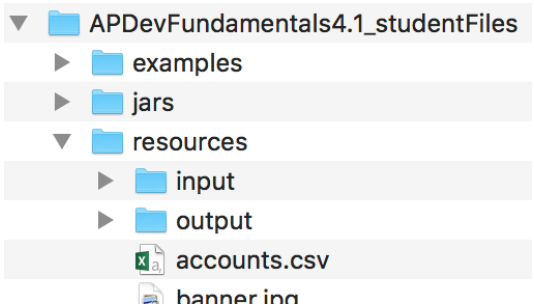
In this walkthrough, you load data from a local CSV file when a new file is added to a directory. You will:

- Add and configure a File listener to watch an input directory.
- Restrict the type of file read.
- Rename and move the processed files.



Locate files and folders

1. In your computer's file browser, return to the student files for the course.
2. Open the resources folder and locate the accounts.csv file and the input and output folders.

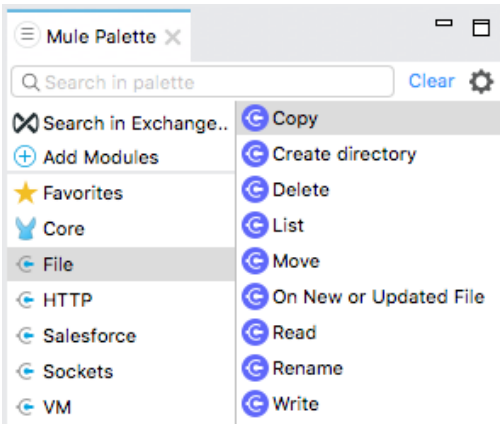


3. Leave this folder open.

Add the File module to the project

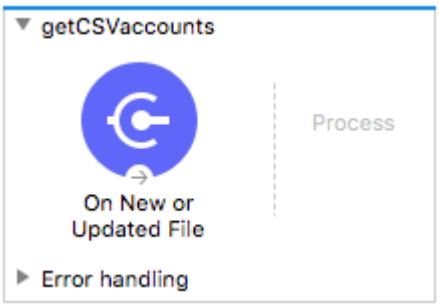
4. Return to Anypoint Studio and open the apdev-examples project.
5. Open accounts.xml.
6. In the Mule Palette, select Add Modules.
7. Select the File connector in the right side of the Mule Palette and drag and drop it into the left side.

8. If you get a Select module version dialog box, select the latest version and click Add.



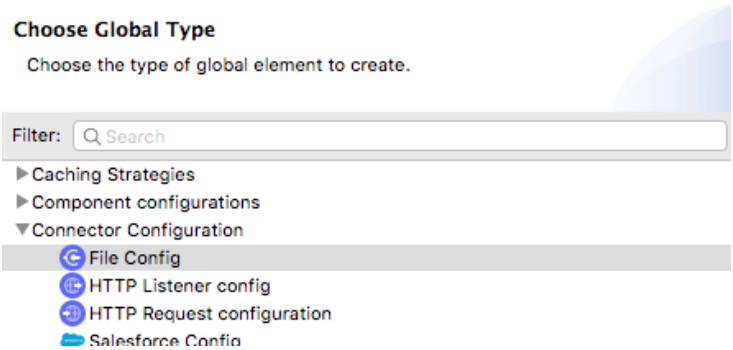
Create a flow that monitors a location for new files

9. Locate the On New or Updated File operation for the File connector in the right side of the Mule Palette and drag and drop it at the top of the canvas to create a new flow.
10. Rename the flow to getCSVaccounts.

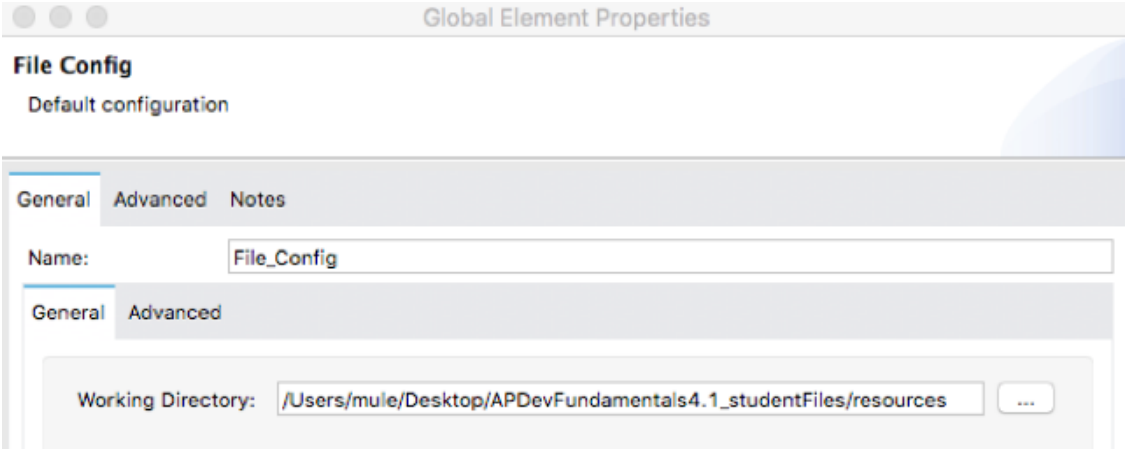


Configure the File connector

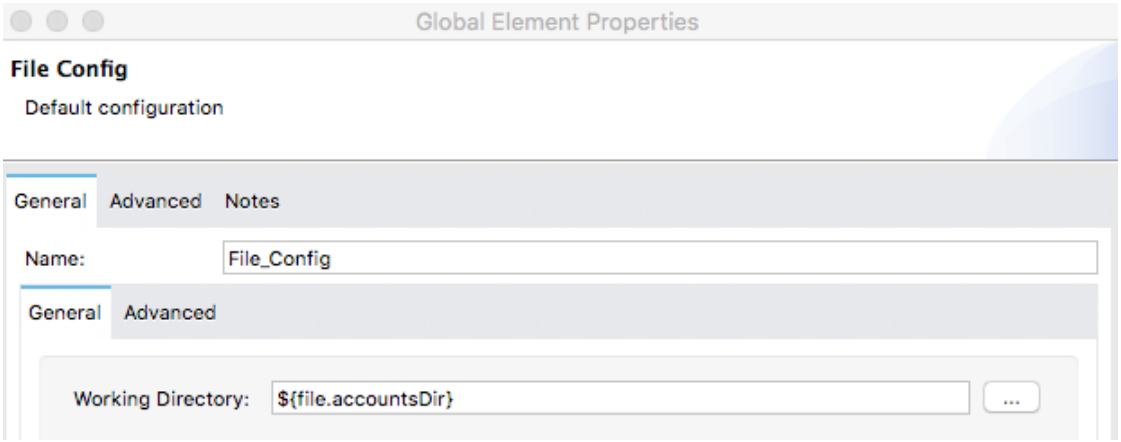
11. Open global.xml and switch to the Global Elements view.
12. Click Create.
13. In the Choose Global Type dialog box, select Connector Configuration > File Config and click OK.



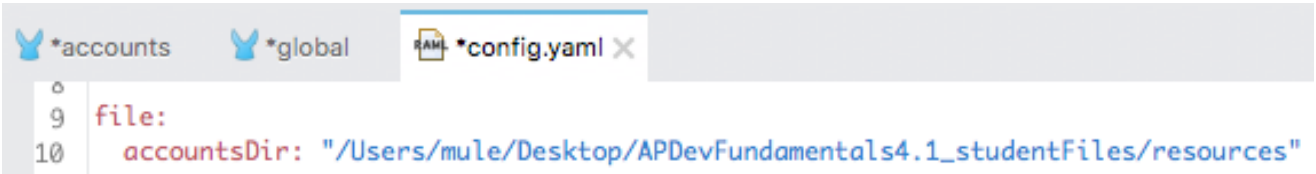
14. In the Global Element Properties dialog box, click the browse button next to working directory.
15. Browse to and select the student files folder for the course.
16. Select the resources folder and click Open.



17. Select and cut the value of the working directory that got populated and replace it with a property ${file.accountsDir}.

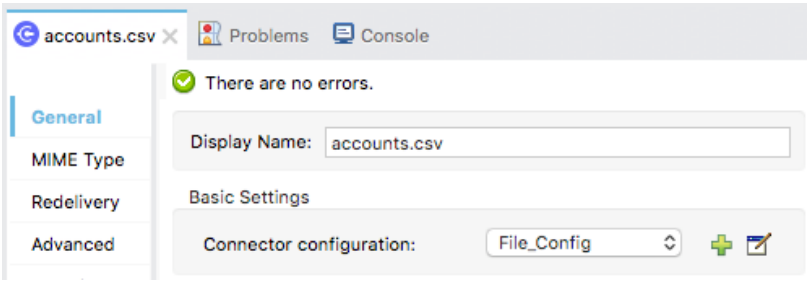


18. In the Global Element Properties dialog box, click OK.
19. Open config.yaml in src/main/resources.
20. Create a new property file.accountsDir and set it equal to the value you copied.

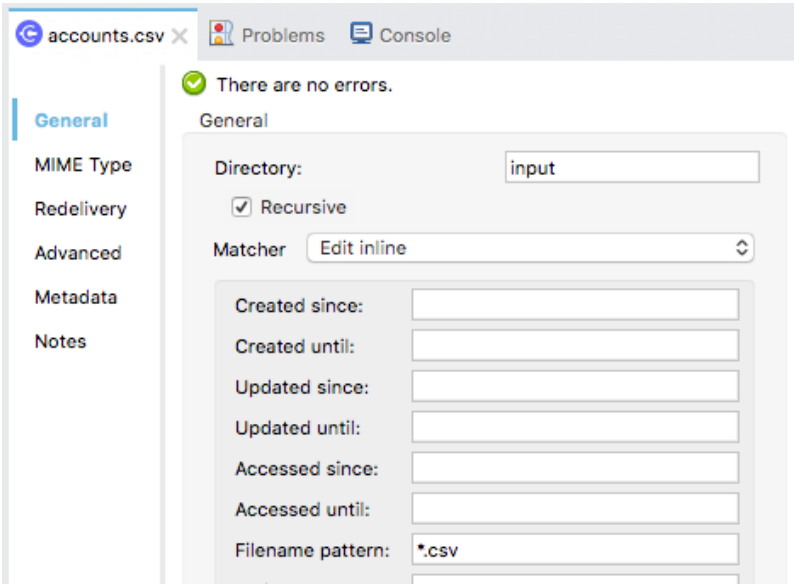


Configure the On New or Updated File operation

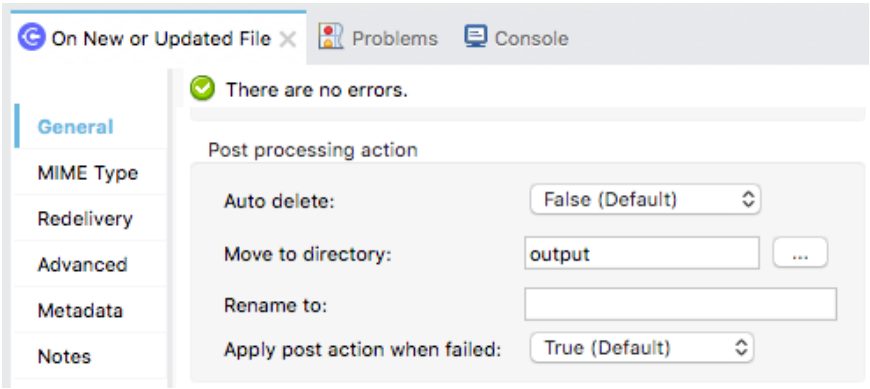
21. Return to accounts.xml.
22. In the On New or Updated File properties view, change the display name to accounts.csv.
23. Set the connector configuration to the existing File_Config.



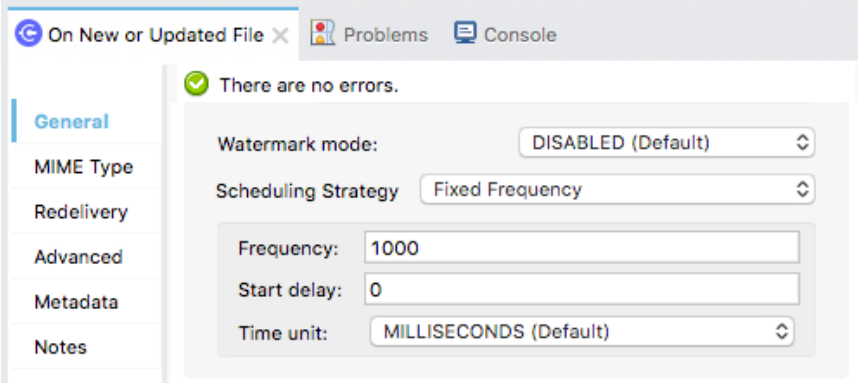
24. In the General section, set the directory to input.
25. Set the matcher to Edit inline.
26. Set filename pattern to *.csv.



27. In the post processing action section, set the move to directory to output.

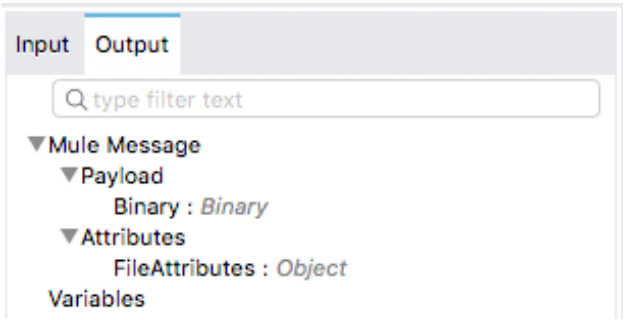


28. In the General section, review the default scheduling information; the endpoint checks for new files every 1000 milliseconds.



Review event metadata in the DataSense Explorer

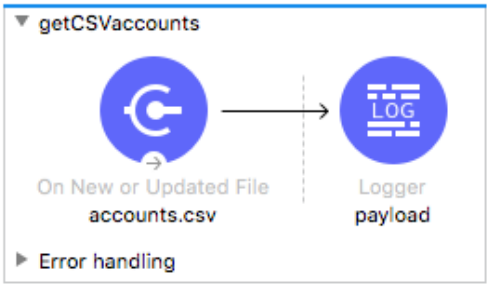
29. Select the Output tab in the DataSense Explorer; you should see no metadata about the structure of the CSV file.



Display the file contents

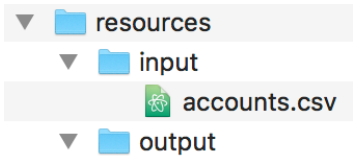
30. Add a Logger to the flow.
31. In the Logger properties view, set the display name to payload and the message to display the payload.

```
#[payload]
```

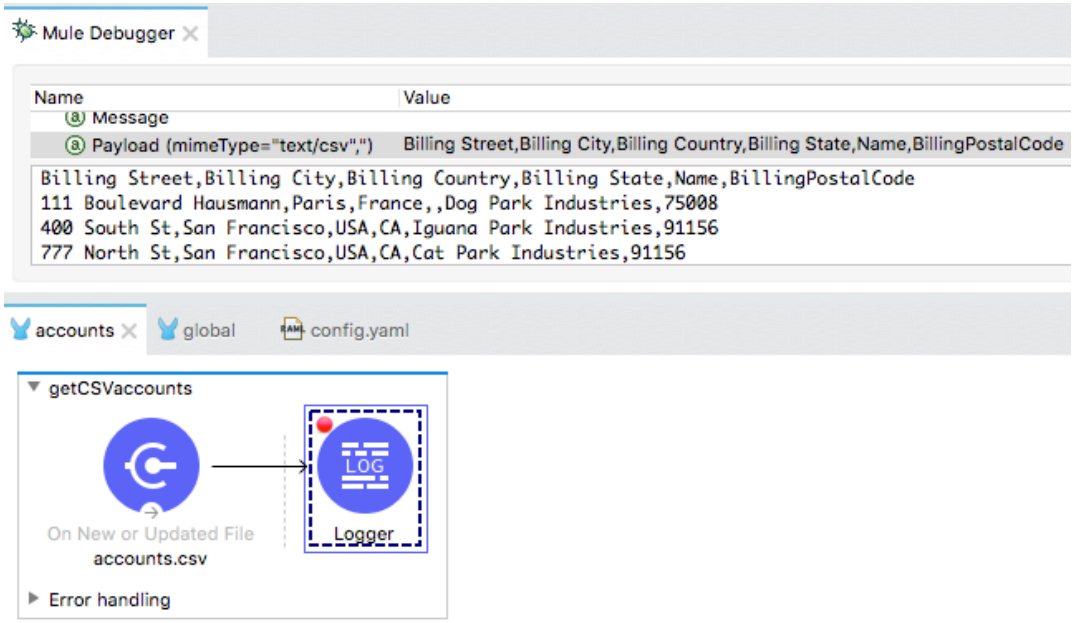


Debug the application

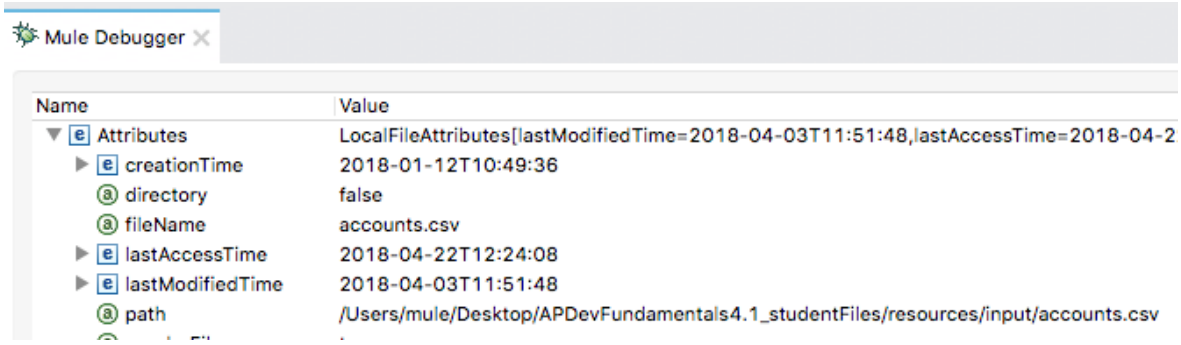
32. Add a breakpoint to the Logger.
33. Debug the project.
34. In your computer's file browser, return to the student files for the course.
35. Move the accounts.csv file to the input folder.



36. Return to the Mule Debugger and look at the payload.

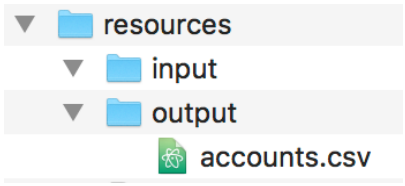


37. Expand Attributes and locate the fileName and path attributes.

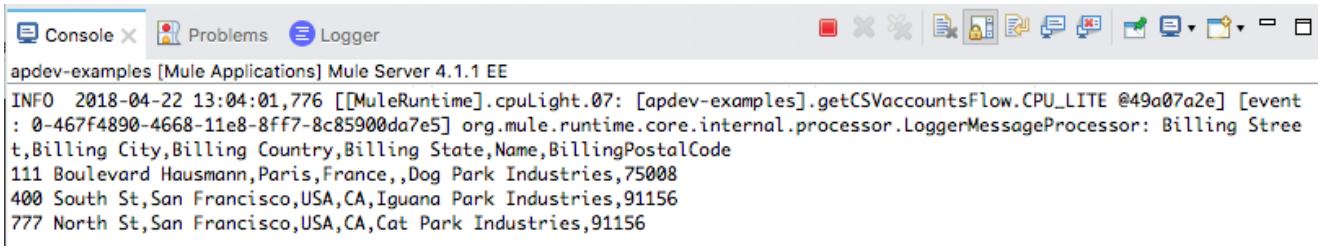


38. Step to the end of the application.

39. In your computer's file browser, return to the student files for the course; you should see accounts.csv has been moved to the output folder.



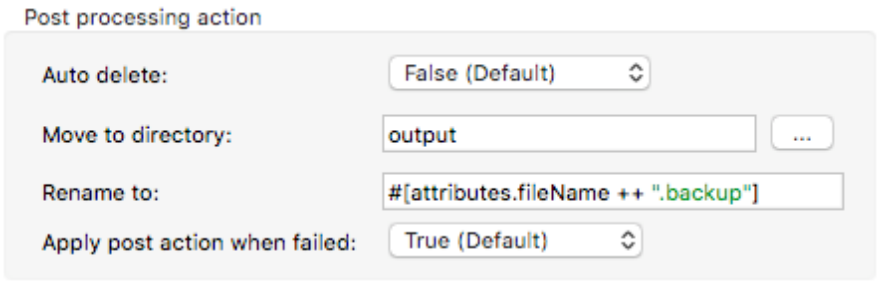
40. Return to Anypoint Studio and look at the console; you should see the file contents displayed.



Rename the file

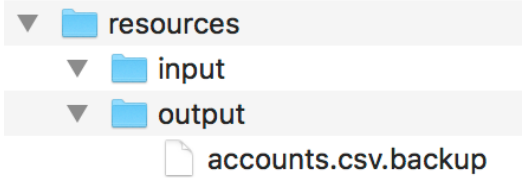
41. Return to the File properties view and in the Post processing action section, set the rename to property to an expression that appends .backup to the original filename.

```
#[attributes.fileName ++ ".backup"]
```

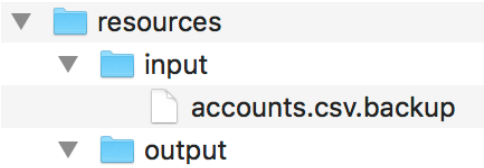


Test the application

42. Save the file to redeploy the project.
43. Remove the breakpoint from the Logger.
44. In your computer's file explorer, move the accounts.csv file from the output folder back to the input folder; you should see it appear in the output folder with its new name.



45. Move the accounts.csv.backup file from the output folder back to the input folder; it should not be processed and should stay in the input folder.

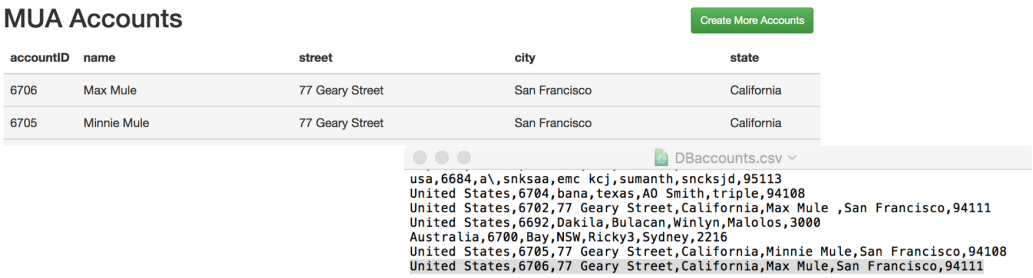


46. Return to Anypoint Studio and switch perspectives.
47. Stop the project.

Walkthrough 12-2: Trigger a flow when a new record is added to a database and use automatic watermarking

In this walkthrough, you work with the accounts table in the training database. You will:

- Add and configure a Database listener to check a table on a set frequency for new records.
- Use the listener's automatic watermarking to track the ID of the latest record retrieved and trigger the flow whenever a new record is added.
- Output new records to a CSV file.
- Use a form to add a new account to the table and see the CSV file updated.

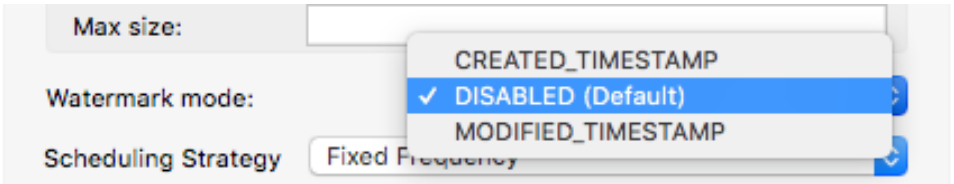


View accounts data

1. In a web browser, navigate to http://mu.mulesoft-training.com/accounts/show.
2. Review the data and the names of the columns—which match those of database table.

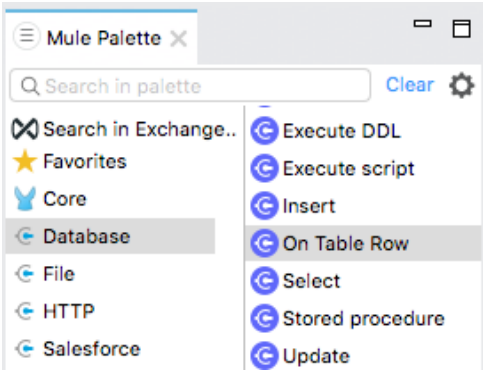
Look at the File listener settings for watermarking

3. Return to accounts.xml in Anypoint Studio.
4. Return to the On New or Updated File properties view for the listener in getCSVaccounts.
5. Locate the watermark mode setting in the General section and look at its possible values.



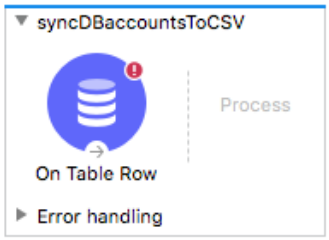
Add the Database module to the project

6. In the Mule Palette, select Add Modules.
7. Select the Database connector in the right side of the Mule Palette and drag and drop it into the left side.
8. If you get a Select module version dialog box, select the latest version and click Add.



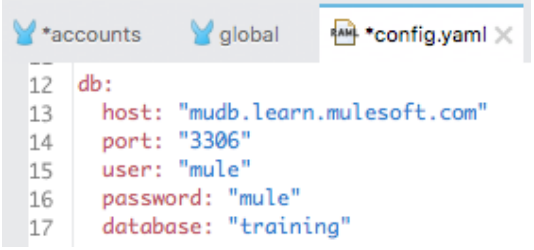
Create a flow to monitor a database

9. Locate the On Table Row operation for the Database connector in the right side of the Mule Palette and drag and drop it at the top of the canvas to create a new flow.
10. Rename the flow to syncDBaccountsToCSV.



Configure a Database connector

11. Return to the course.snippets.txt file.
12. Locate and copy the MySQL (or Derby) database properties in the Module 4 section.
13. Return to config.yaml and paste the properties.

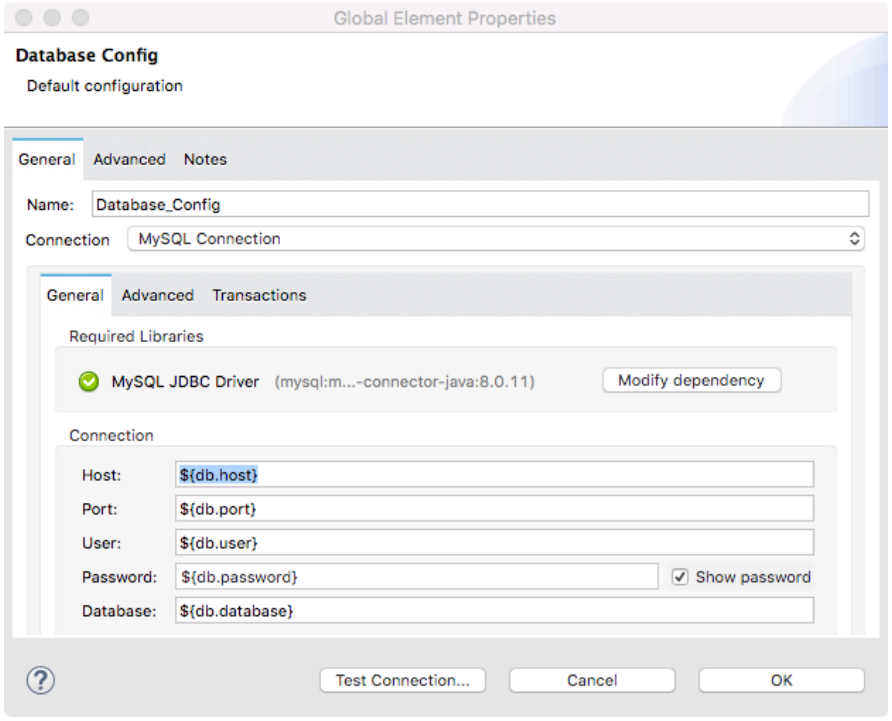


14. Save the file.

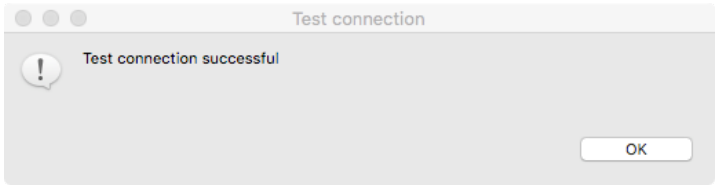
15. Return to global.xml and create a new Database Config that uses these properties.

    Note: If necessary, refer to the detailed steps in Walkthrough 4-2.

16. Add the MySQL (or Derby) JDBC driver.



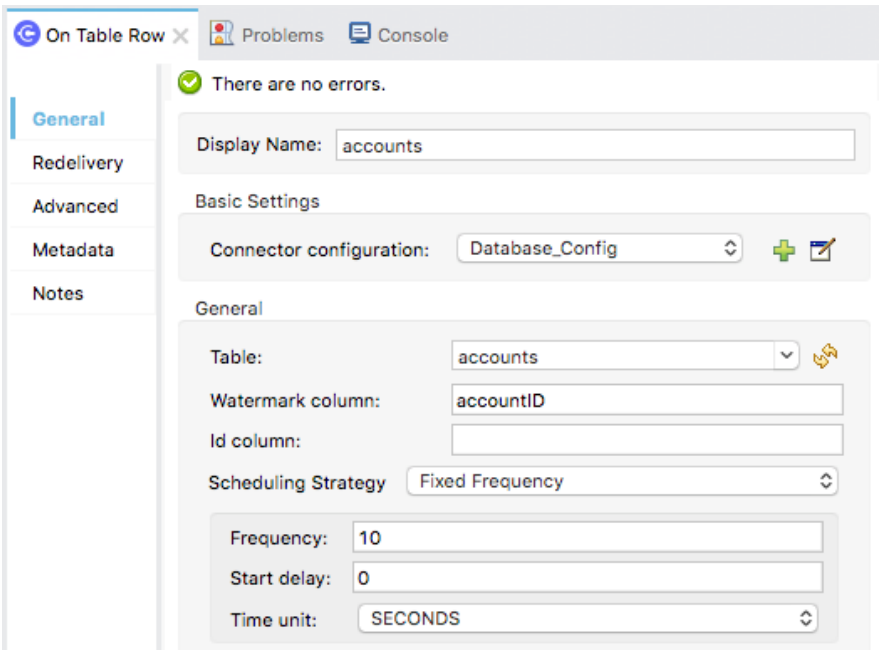
17. Test the connection and make sure it is successful.



Configure the Database listener

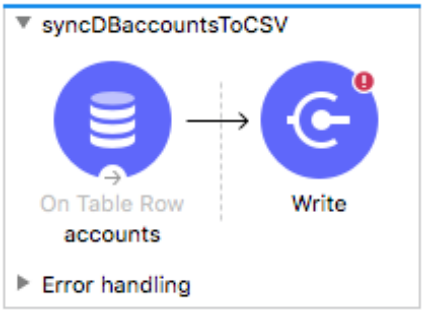
18. Return to accounts.csv.
19. In the On Table Row properties view, set the following values:

    - Display name: accounts
    - Connector configuration: Database_Config
    - Table: accounts
    - Watermark column: accountID
    - ID column: accountID
    - Frequency: 10
    - Time unit: SECONDS



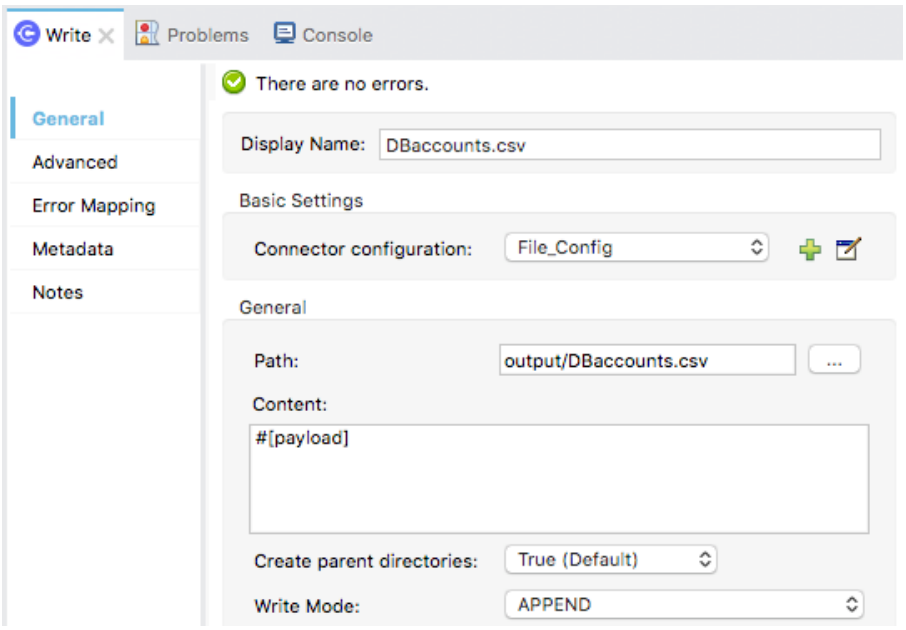
Write new records to a CSV file

20. Add a File Write operation to the flow.



21. In the Write properties view, set the following values:

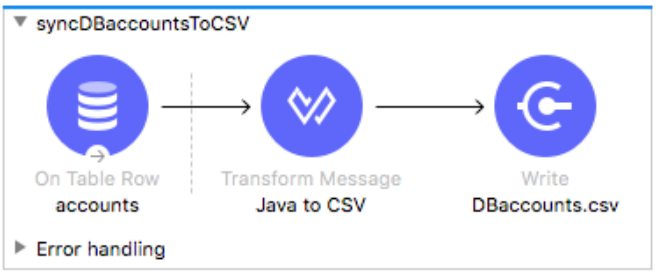
- Display name: DBaccounts.csv
- Connector configuration: File_Config
- Path: output/DBaccounts.csv
- Write mode: APPEND



22. Select the input tab in the DataSense Explorer.
23. Expand Payload; you should see the operation will get an Array of Objects from the Database operation.

Transform the records to CSV

24. Add a Transform Message component before the Write operation.
25. Change the display name to Java to CSV.



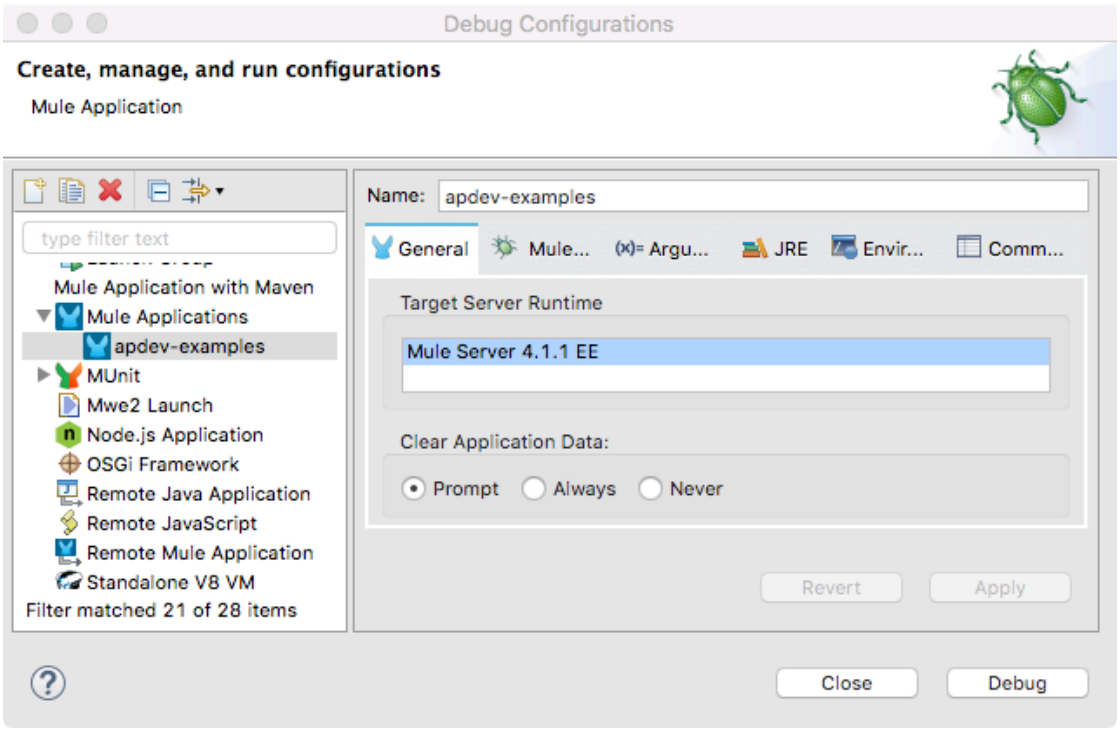
26. In the Transform Message properties view, change the output type to application/csv and add a header property equal to false.
27. Set the body expression to [payload].

Add a Logger to display the records

28. Add a Logger to the flow.
29. Change the display name to payload.
30. Set the message to display the payload.

Set the application to prompt to clear application data when it starts

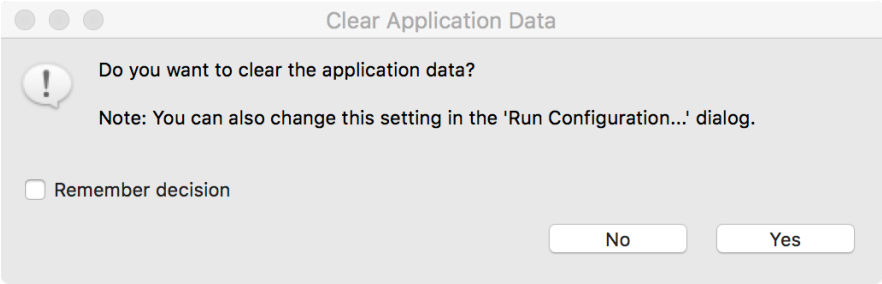
31. Add a breakpoint to the Transform Message component.
32. In the main menu, select Run > Debug Configurations.
33. In the Debug Configurations dialog box, locate the Clear Application Data section in the General tab and change it to Prompt.



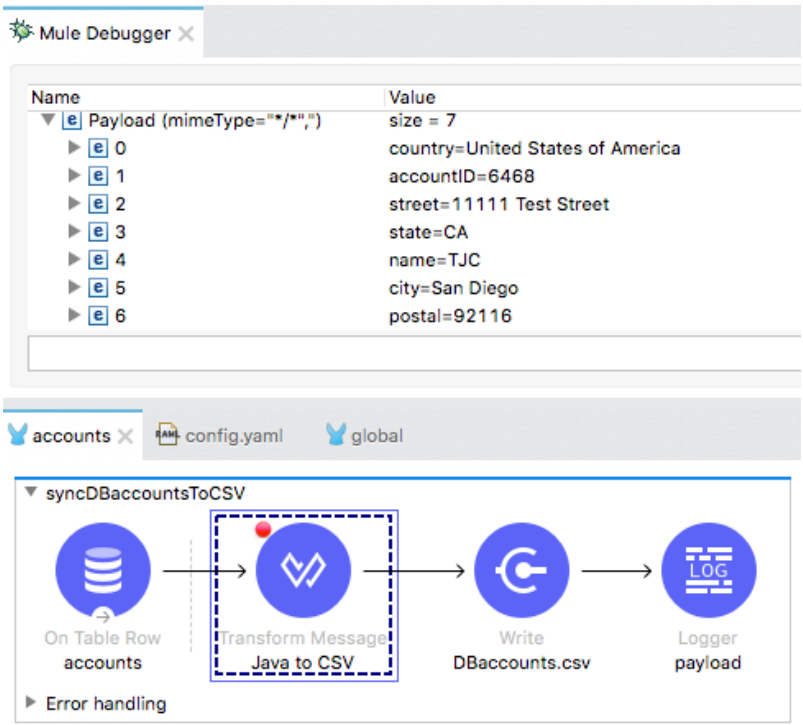
Note: You need to rerun or debug an application to get the prompt; you cannot just save to redeploy.

Debug the project

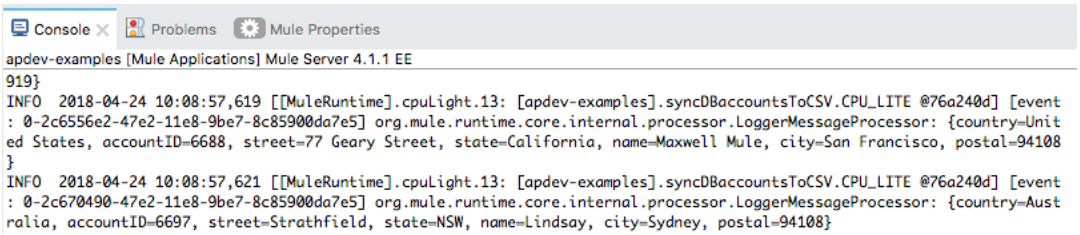
34. Click Debug.
35. In the Clear Application dialog box, select Yes.



36. In the Mule Debugger, expand Payload.



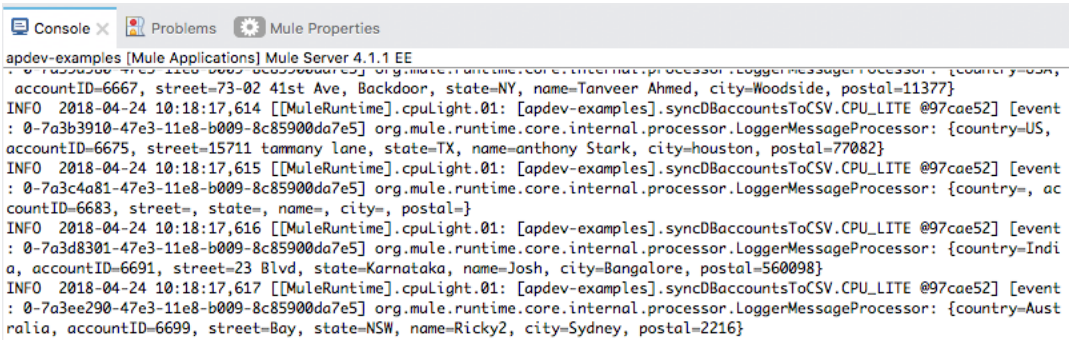
37. Click Resume and step through several of the records.
38. Look at the console, you should records displayed.



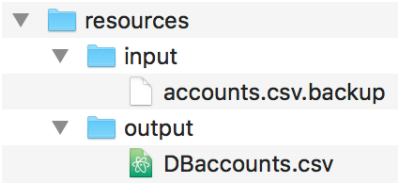
39. Stop the project and switch perspectives.

Clear application data and test the application

40. Run the project.
41. In the Clear Application dialog box, select Yes.
42. Look at the console; you should see many records.



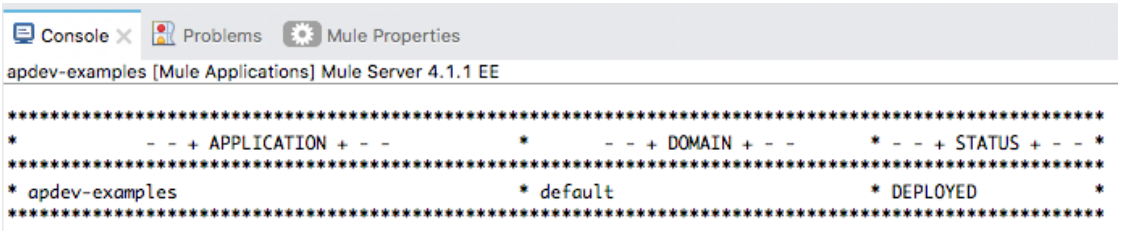
43. Stop the project.
44. Return to the resource folder in your computer's file explorer; you should see a new DBaccounts.csv file appear in the output folder.



45. Open the DBaccounts.csv file with a text editor; you should see the records.

Test the application again without clearing application data

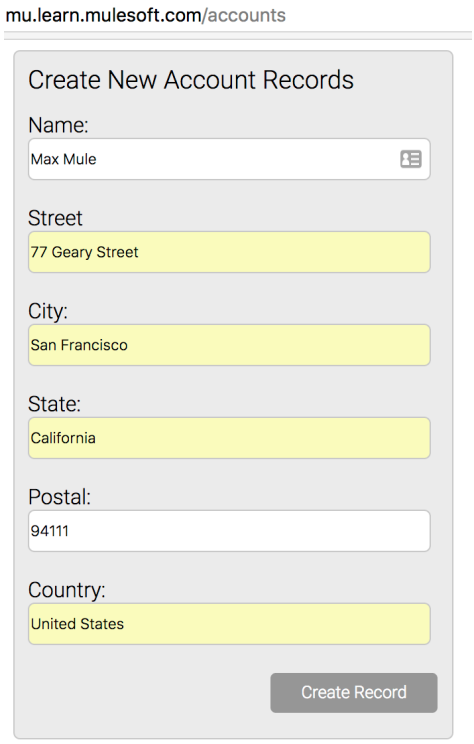
46. Return to Anypoint Studio and run the project.
47. In the Clear Application dialog box, select No.
48. Look at the console; you should see no new records output.



Add a new account to the database

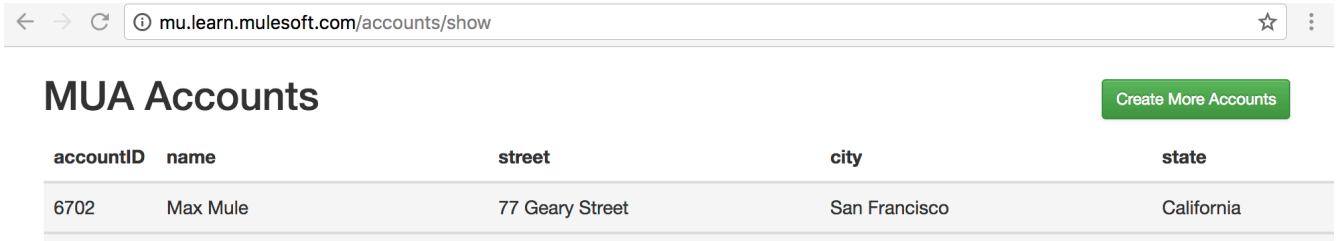
49. Return to the account data in the web browser at http://mu.mulesoft-training.com/accounts/show.
50. Click Create More Accounts.

51. Fill out the form with data and click Create Record.

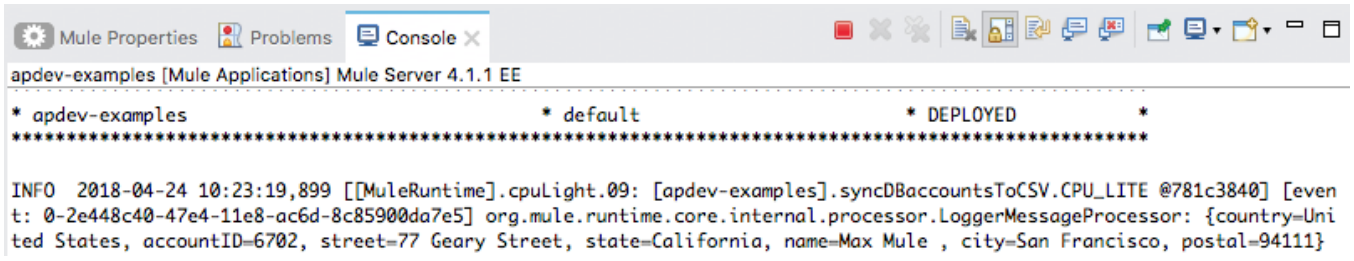


Note: Set the postal code to a specific value. In the next walkthrough, you will retrieve only new records with this specific postal code, so you do not get all of the records.

52. Click View Existing Accounts; you should see your new account.

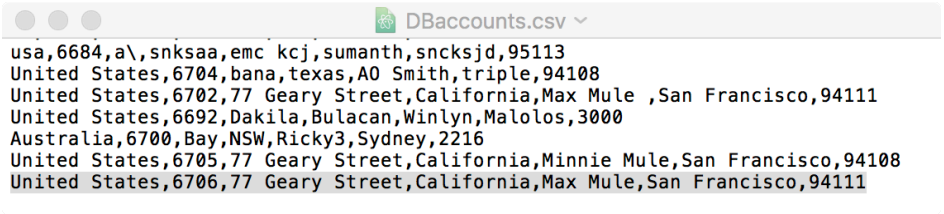


53. Return to the console in Anypoint Studio; you should see your new record.



54. Stop the project.

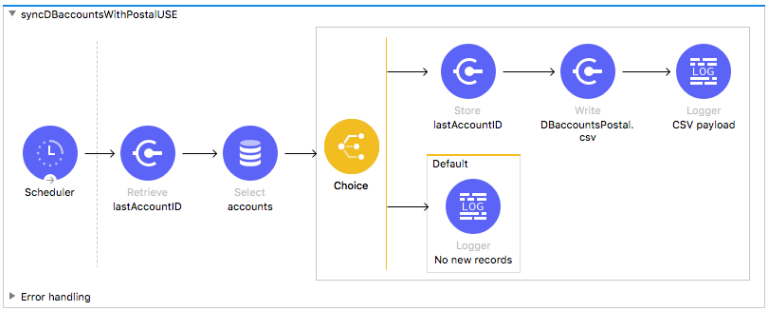
55. Return to your computer's file explorer; the modified date on the DBAccounts.csv file should have been updated.
56. Open DBaccounts.csv and locate your new record at the end of the file.



Walkthrough 12-3: Schedule a flow and use manual watermarking

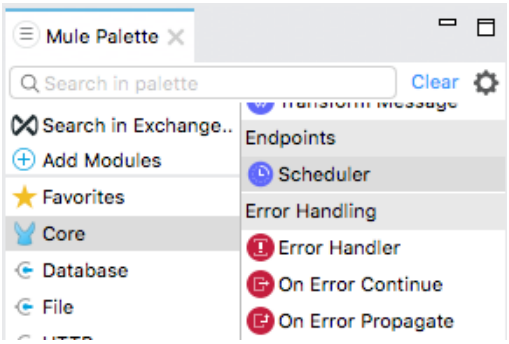
In this walkthrough, you continue to work with the accounts table in the training database. You will:

- Use the Scheduler component to create a new flow that executes at a specific frequency.
- Retrieve accounts with a specific postal code from the accounts table.
- Use the Object Store component to store the ID of the latest record and then use it to only retrieve new records.

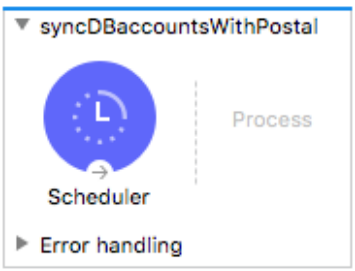


Create a flow that executes at a specific frequency

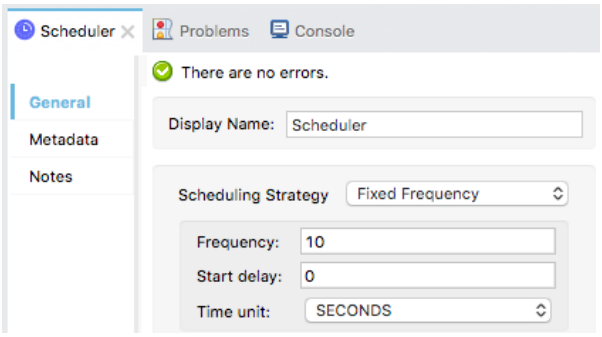
1. Return to accounts.xml in Anypoint Studio.
2. Locate the Scheduler component in the Core section of the Mule Palette.



3. Drag a Scheduler component from the Mule Palette and drop it at the top of the canvas to create a new flow.
4. Change the name of the flow to syncDBaccountsWithPostal.

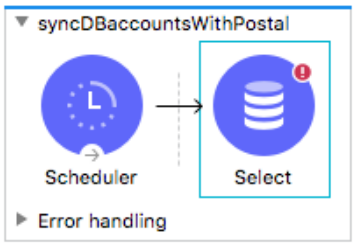


5. In the Scheduler properties view, set the frequency to 10 seconds.



Retrieve records with a specific postal code from the database

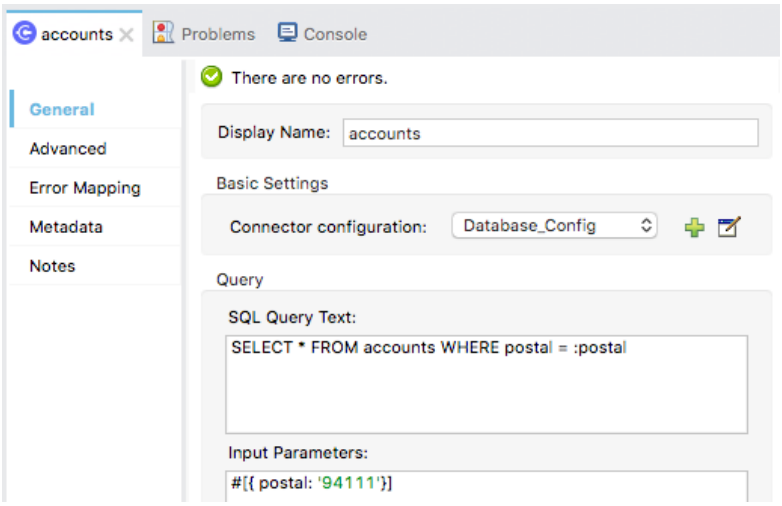
6. From the Mule Palette, drag a Database Select operation and drop it in the flow.



7. In the Select properties view, set the following:

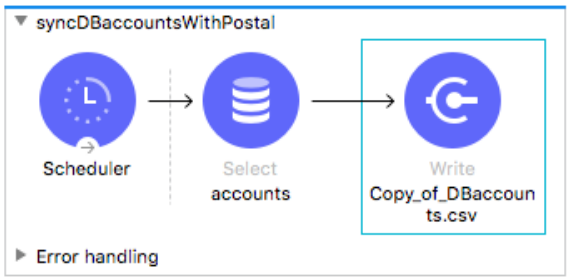
   - Display name: accounts
   - Connector configuration: Database _Config
   - SQL query text: SELECT * FROM accounts WHERE postal = :postal
   - Input parameters: #[{ postal: *'yourPostalValue'*}]

   Note: If you want, you can store the postal code as a property in config.yaml and then reference it here in the DataWeave expression as #[{ postal: p('propertyName')}].



Output the records to a CSV file

8. Copy the Write operation in syncDBaccountsToCSV and paste it at the end of syncDBaccountsWithPostal.

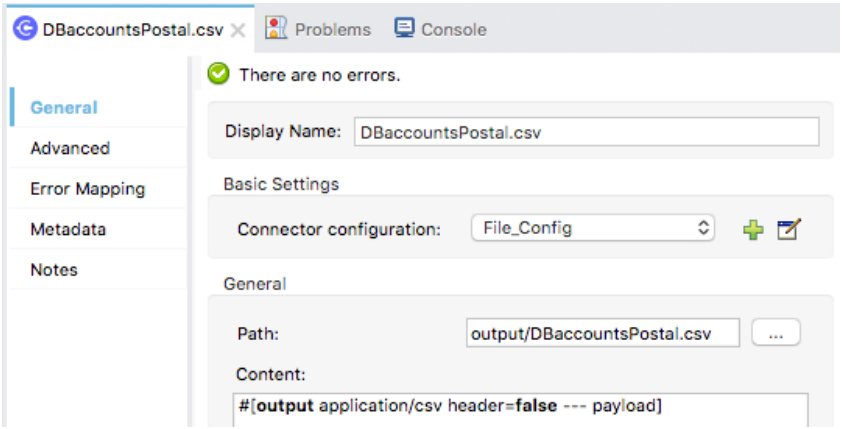


9. In the properties view for the Write operation, set the following values:

   - Display name: DBaccountsPostal.csv
   - Path: output/DBaccountsPostal.csv

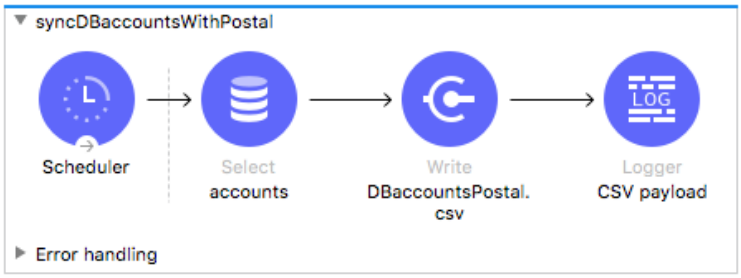
10. Change the content to output the payload as application/csv with a header property equal to false.

```
#[output application/csv header=false --- payload]
```

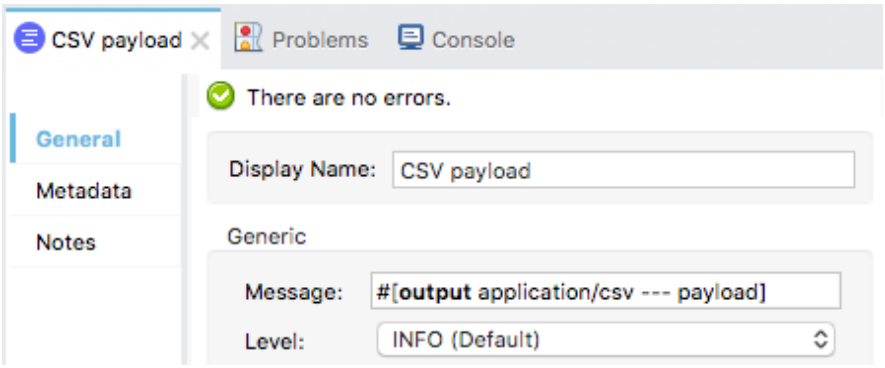


Log the payload

11. Add a Logger at the end of the flow and change the display name to CSV payload.

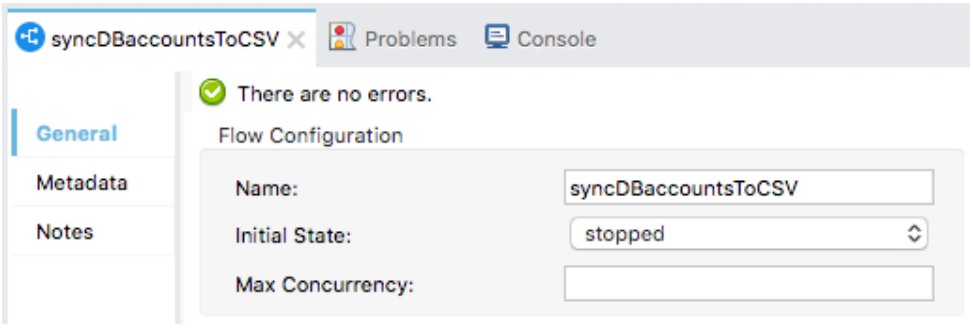


12. Set the message to the payload as type application/csv.



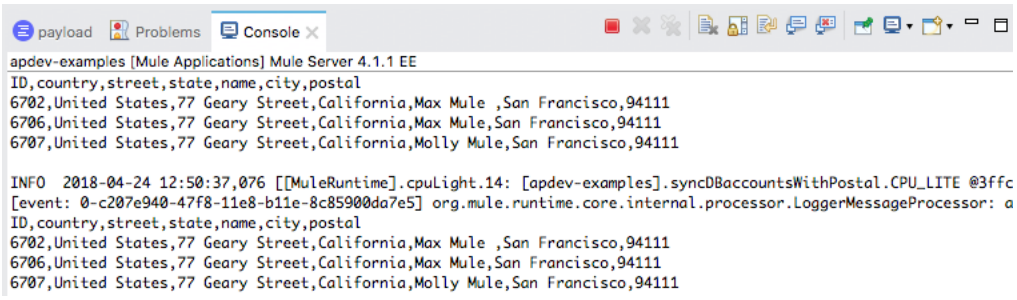
Stop the other syncDBaccountsToCSV flow so it does not run

13. In the properties view for the syncDBaccountsToCSV flow, set the initial state to stopped.



Test the application

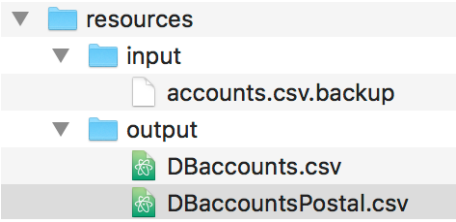
14. Run the project.
15. In the Clear Application dialog box, select Yes.
16. Watch the console, you should see the same records displayed every 10 seconds.



Note: Right now, all records with matching postal code are retrieved – over and over again. Next, you will modify this so only new records with the matching postal code are retrieved.

17. Stop the project.

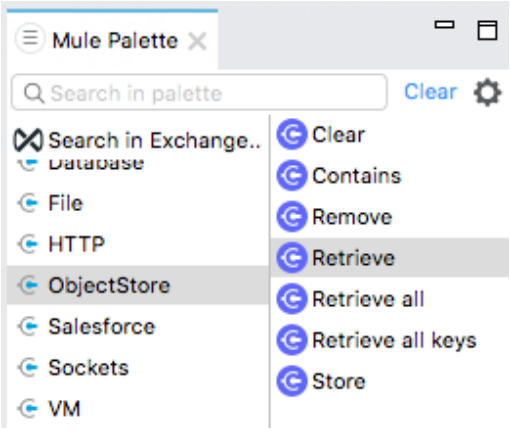
18. Return to the resources folder in your computer's file browser; you should see the new DBaccountsPostal.csv file.



19. Open the file and review the contents.
20. Delete the file.

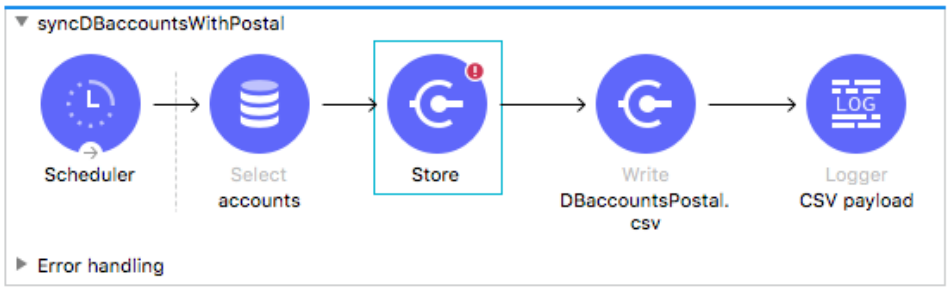
Add the ObjectStore module to the project

21. In the Mule Palette, select Add Modules.
22. Select the ObjectStore connector in the right side of the Mule Palette and drag and drop it into the left side.
23. If you get a Select module version dialog box, select the latest version and click Add.



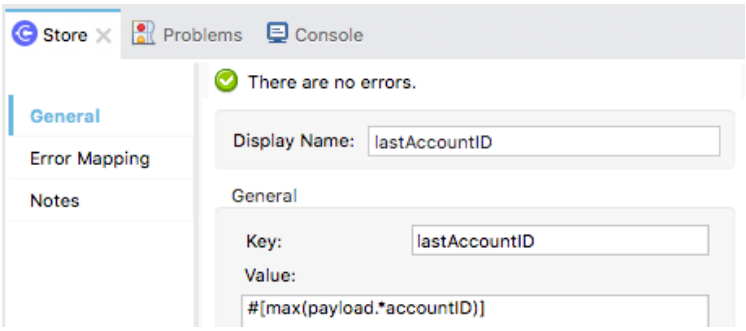
Store the ID of the last record retrieved

24. Drag the Store operation for the ObjectStore connector from the Mule Palette and drop it after the Database Select operation.



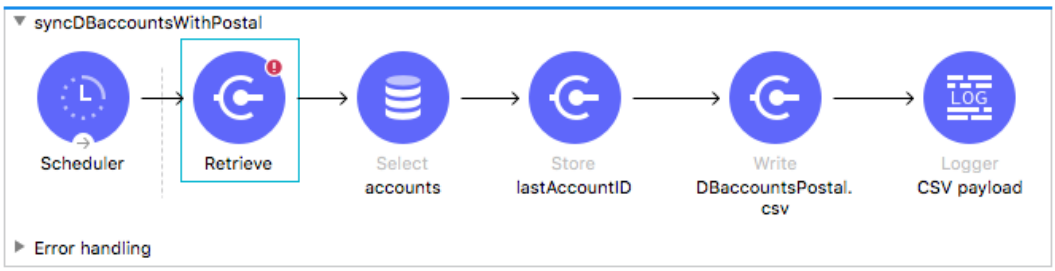
25. In the Store properties view, set the display name and key to lastAccountID.
26. Set the value to an expression for the maximum value of lastAccountID in the payload, the retrieved records.

```
#[max(payload.*accountID)]
```



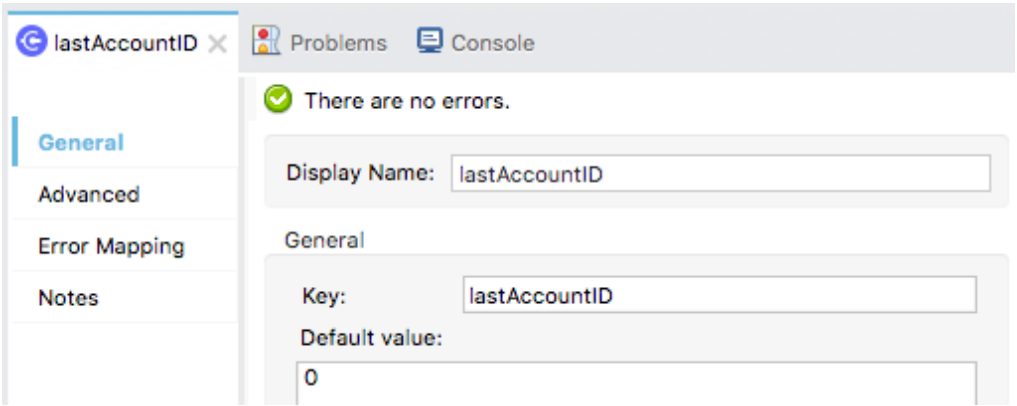
Before the query, retrieve the ID of the last record retrieved

27. Drag the Retrieve operation for the ObjectStore connector from the Mule Palette and drop it before the Database Select operation.

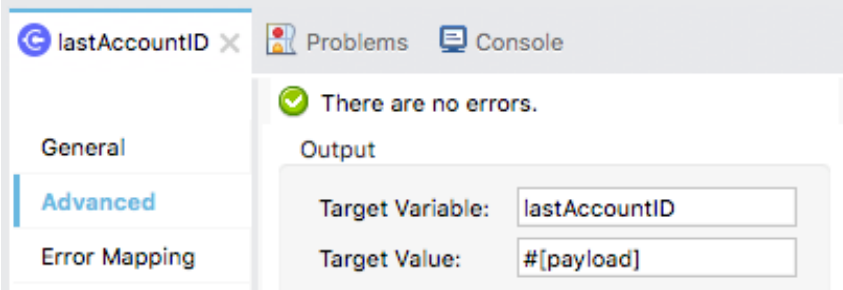


28. In the Retrieve properties view, set the following values:

- Display name: lastAccountID
- Key: lastAccountID
- Default value: 0



29. Select the Advanced tab and set the target variable to lastAccountID so the value is stored in a variable called lastAccountID.



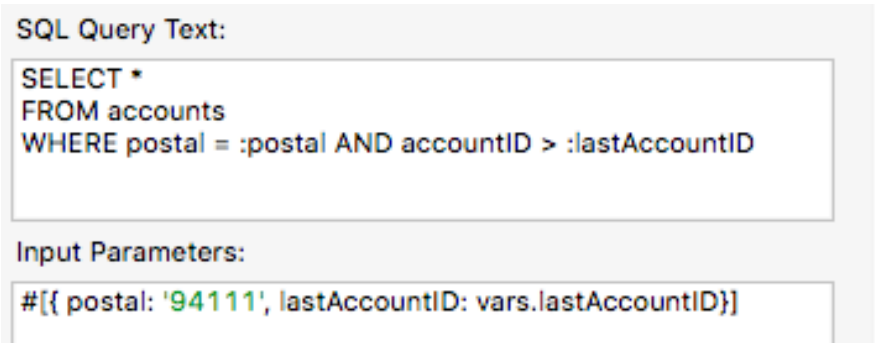
Modify the query to only retrieve new records with a specific postal code

30. In the accounts Select properties view, add a second query input parameter called lastAccountID that is equal to the watermark value.

```
#[{postal: 'yourValue', lastAccountID: vars.lastAccountID}]
```

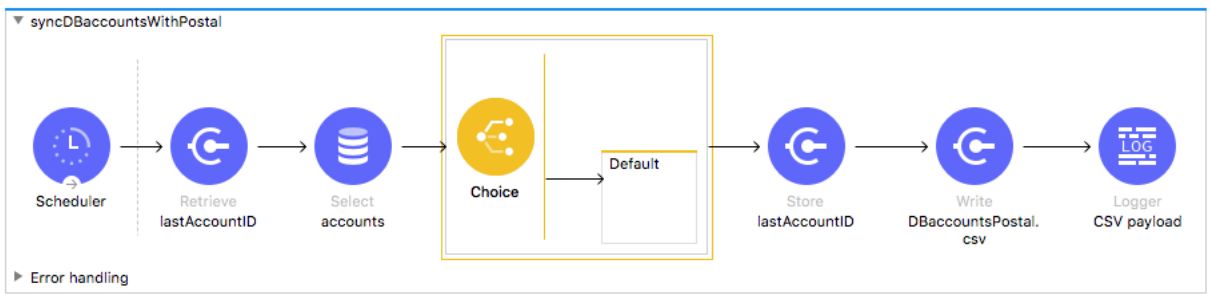
31. Modify the SQL query text to use this parameter to only retrieve new records.

```
SELECT * FROM accounts WHERE postal = :postal AND accountID
> :lastAccountID
```

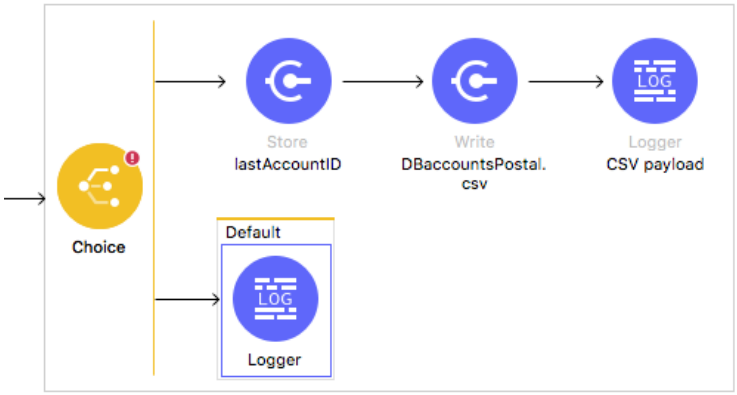


Only store the value if new records were retrieved

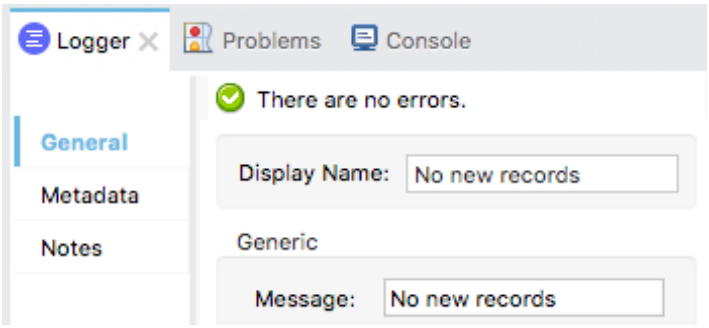
32. Add a Choice router after the Select operation.



33. Move the Store, Write, and Logger operations into the main branch of the router.
34. Add a Logger to the default branch.

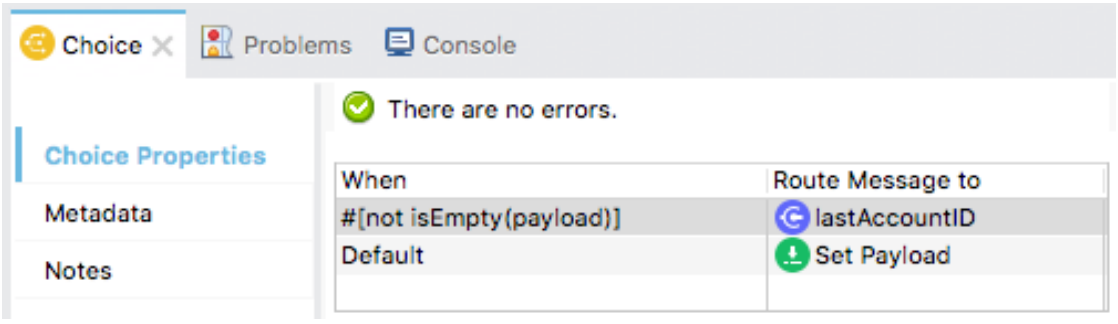


35. In the Logger properties view, set the display name and value to No new records.



36. In the Choice properties view, route the event to the first branch when the payload is not empty.

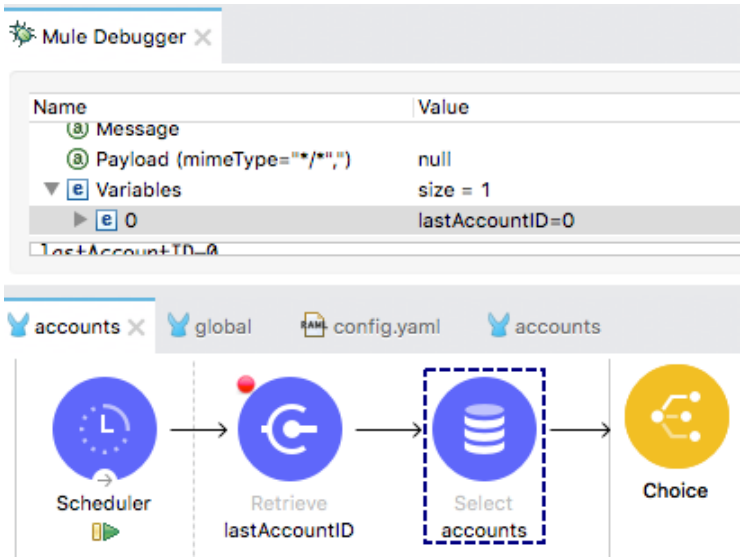
```
#[not isEmpty(payload)]
```



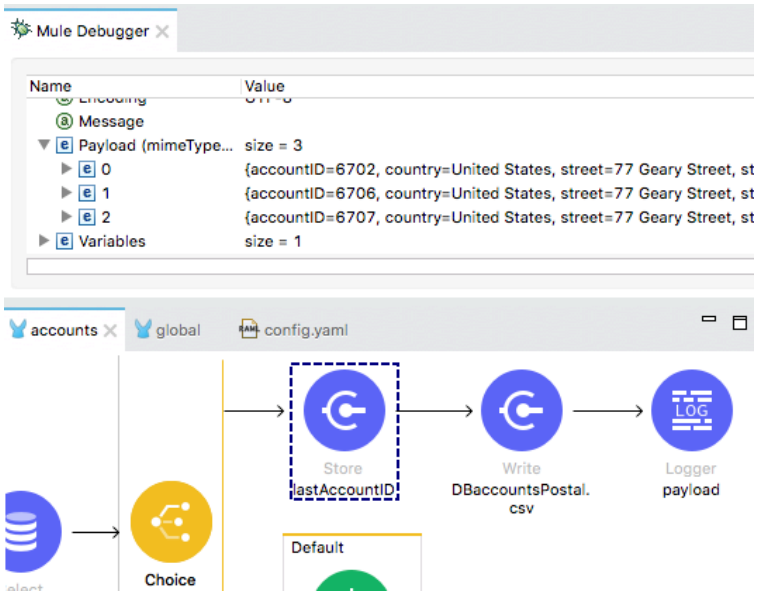
Clear the application data and debug the application

37. Add a breakpoint to the Retrieve operation.
38. Debug the project.
39. In the Clear Application dialog box, select Yes.

40. In the Mule Debugger, step to the Select operation; you should see a lastAccountID variable with a value of 0.

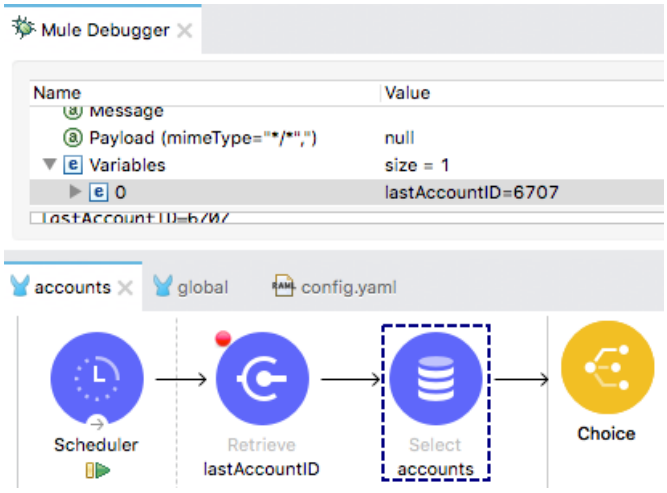


41. Step to the Choice router; you should see record(s) were retrieved from the database.

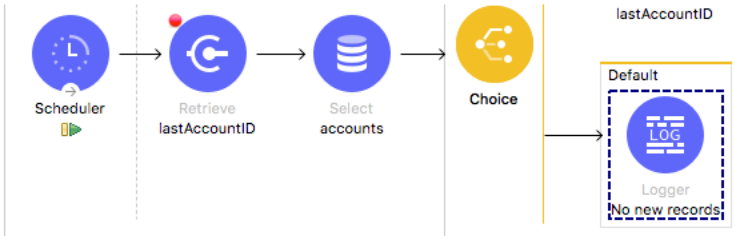


42. Step through the rest of the application and click resume; the flow should be executed again, and execution should be stopped back at the beginning of the flow.

43. Step to the Select operation; you should see the lastAccountID variable now has a non-zero value.



44. Step into the Choice router; you should see no records were retrieved from the database (unless someone else just added one with the same postal code!).

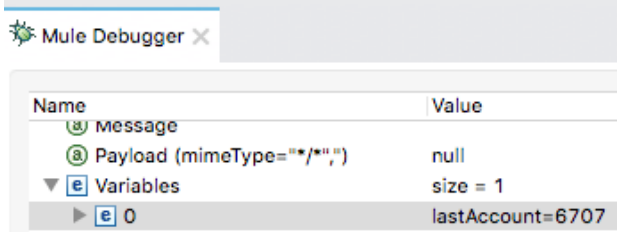


45. Stop the project and switch perspectives.
46. Return to your computer's file explorer and locate and open the new DBaccountsPostal.csv file.

    Note: If you see the same records more than once, it was because during debugging, the flow was executed again before the watermark was stored.

Debug the application and do not clear application data

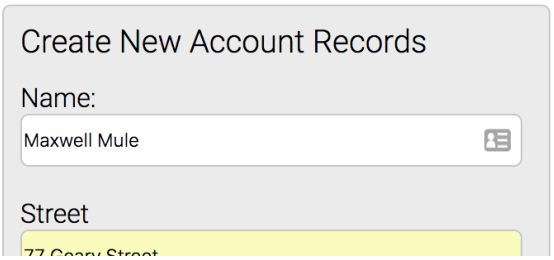
47. Return to Anypoint Studio and debug the project.
48. In the Clear Application dialog box, select No.
49. In the Mule Debugger, step to the Select operation; you should see a lastAccountID variable with the same non-zero value.



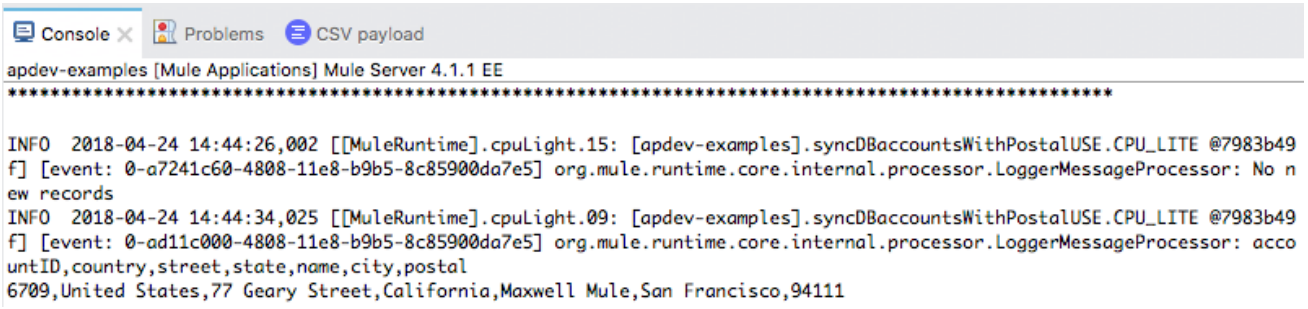
50. Step to the Choice router; no new records should have been returned.
51. Remove the breakpoint from the Retrieve operation.
52. Click Resume until application execution is no longer stopped anywhere.
53. Make sure there are no other breakpoints in the flow.

Add a new account to the database

54. Return to the account data in the web browser at http://mu.mulesoft-training.com/accounts/show.
55. Click Create More Accounts.
56. Fill out the form with data and use the same postal code.



57. Click Create Record.
58. Click View Existing Accounts; you should see your new account.
59. Return to Anypoint Studio; you should see your new record in the console.

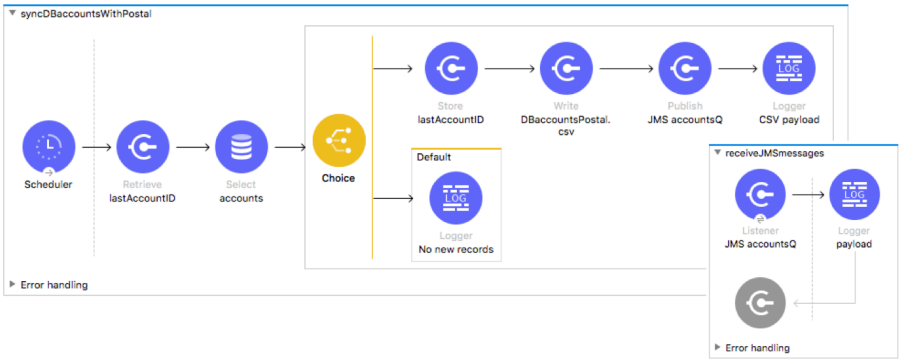


60. Return to your computer's file explorer; the modified date on the DBAccountsPostal.csv file should have been updated.
61. Open DBaccountsPostal.csv and locate your new record at the end of the file.
62. Return to Anypoint Studio and switch perspectives.
63. Stop the project.

Walkthrough 12-4: Publish and listen for JMS messages

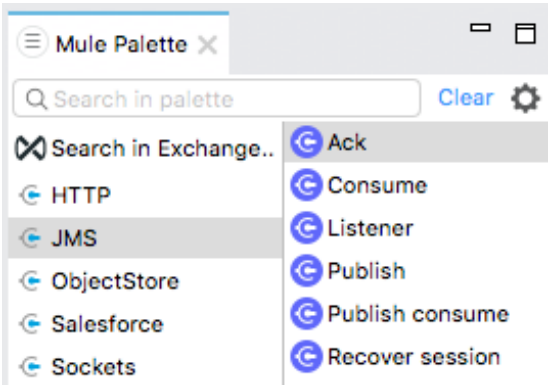
In this walkthrough, you send a JMS message for each new database record so it can be processed asynchronously. You will:

- Add and configure a JMS connector for ActiveMQ (that uses an in-memory broker).
- Send messages to a JMS queue.
- Listener for and process messages from a JMS queue.



Add the JMS module to the project

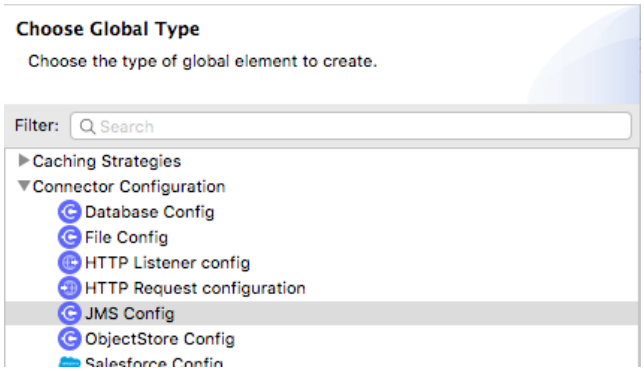
1. Return to accounts.xml.
2. In the Mule Palette, select Add Modules.
3. Select the JMS connector in the right side of the Mule Palette and drag and drop it into the left side.
4. If you get a Select module version dialog box, select the latest version and click Add.



Configure the JMS connector

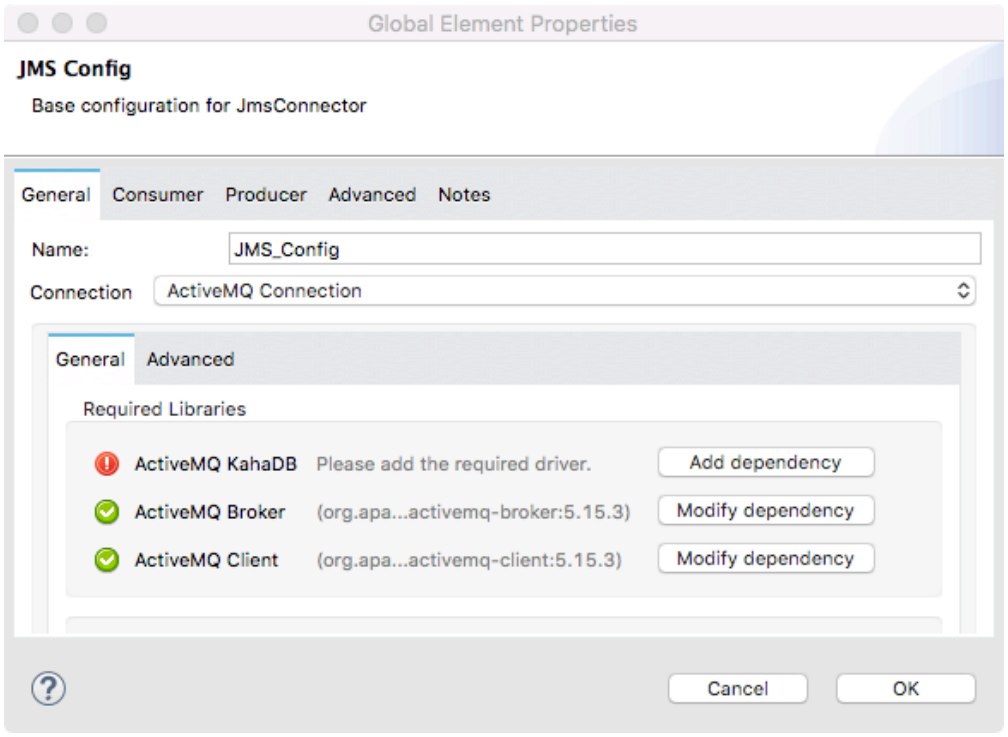
5. Return to the Global Elements view of global.xml.
6. Click Create.

7. In the Choose Global Type dialog box, select Connector Configuration > JMS Config and click OK.

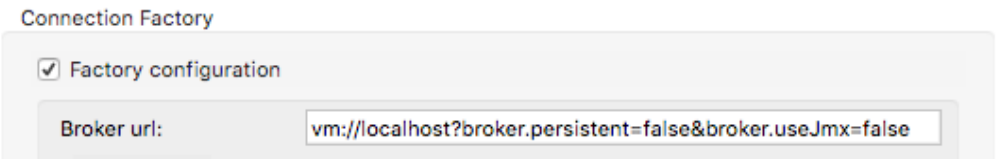


8. In the Global Element Properties dialog box, make sure the connection is set to ActiveMQ Connection.
9. Click the Add dependency button for the ActiveMQ Broker.
10. In the Maven dependency dialog box, enter activemq-broker in the Search Maven Central field.
11. Select the org.apache.activemq:activemq-broker entry and select Finish.
12. Click the Add dependency button for the ActiveMQ Client.
13. In the Maven dependency dialog box, enter activemq-client in the Search Maven Central field.
14. Select the org.apache.activemq:activemq-client entry and select Finish.

Note: There are local versions of the drivers available in the student files.



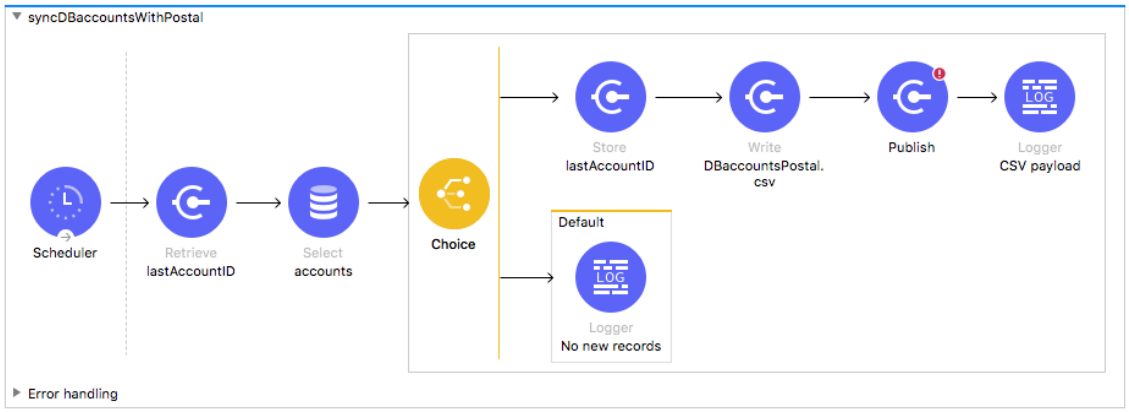
15. Scroll down and locate the Connection Factory section.
16. Select Factory configuration; the Broker url should already be populated.



17. Click OK.

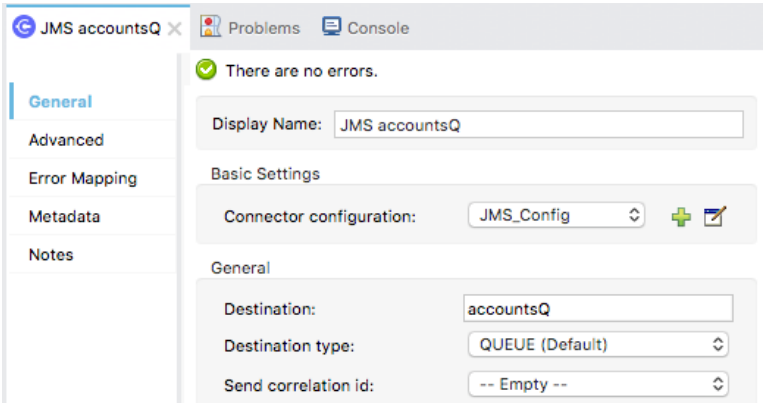
Use the JMS Publish operation to send a message to a JMS queue

18. Return to accounts.xml.
19. Drag out a JMS Publish operation from the Mule Palette and drop it after the Write operation in syncDBaccountsWithZip.



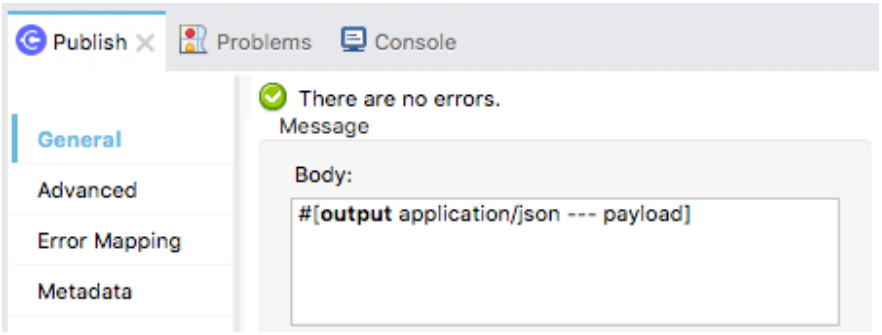
20. In the Publish properties view, set the following values:

- Display name: JMS accountsQ
- Connector configuration: JMS_Config
- Destination: accountsQ
- Destination type: QUEUE



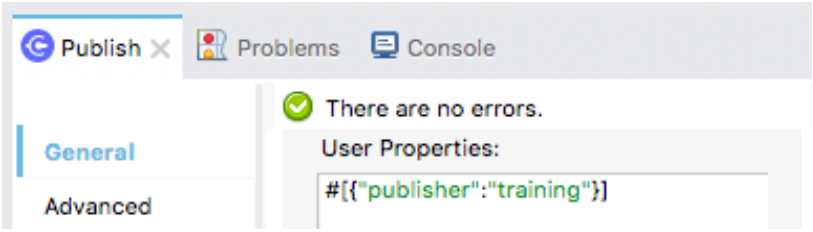
21. Modify the message body to be of type application/json.

```
#[output application/json --- payload]
```



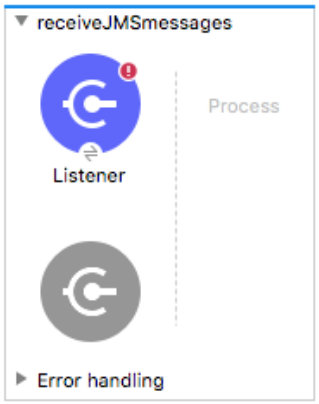
22. Scroll down and locate the User Properties field.
23. Set it equal to an object a key called publisher with a value of training.

```
#[{"publisher":"training"]
```



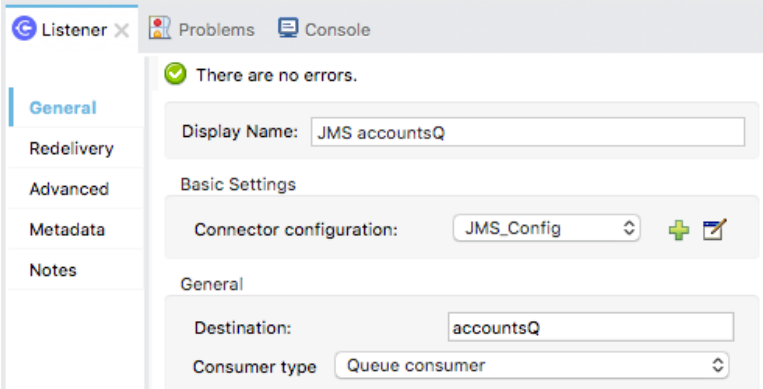
Create a flow to listen to a JMS topic

24. Drag out the JMS Listener operation from the Mule Palette and drop it in the canvas to create a new flow.
25. Give the flow a new name of receiveJMSMessages.

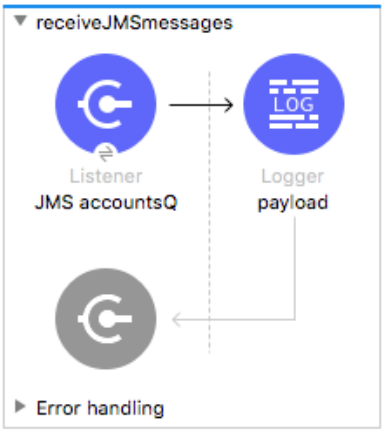


26. In the Listener properties view, set the following values:

    - Display name: JSM accountsQ
    - Connector configuration: JMS_Config
    - Destination: accountsQ
    - Consumer type: Queue consumer



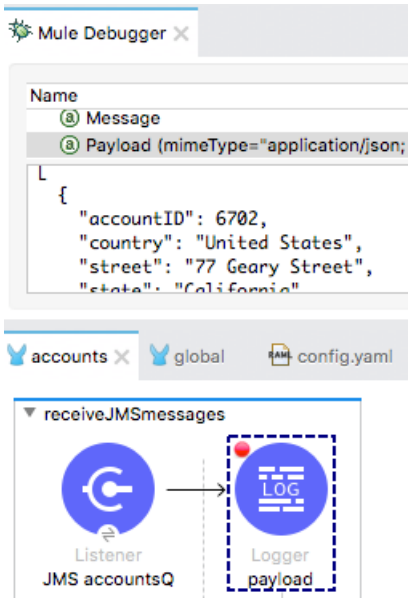
27. Add a Logger to the flow.
28. Set its display name to payload and its message to #[payload].



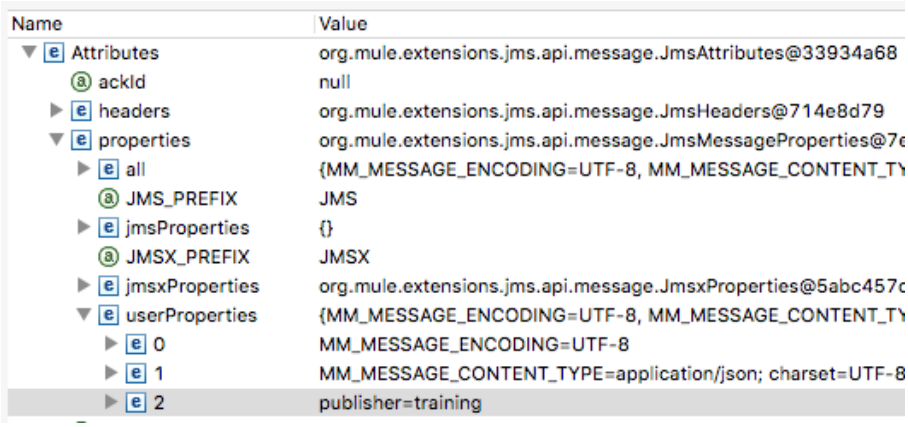
Clear application data and debug the application

29. Add a breakpoint to the Logger in receiveJMSmessages.
30. Debug the project.
31. In the Clear Application Data dialog box, select Yes.

32. Wait until the project deploys; application execution should stop in receiveJMSMessages.
33. In the Mule Debugger view, look at the value of the payload.



34. Expand the Attributes and locate the publisher property in the userProperties.

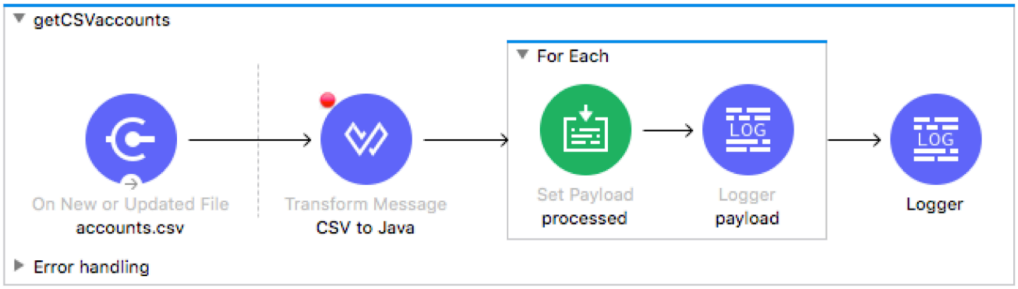


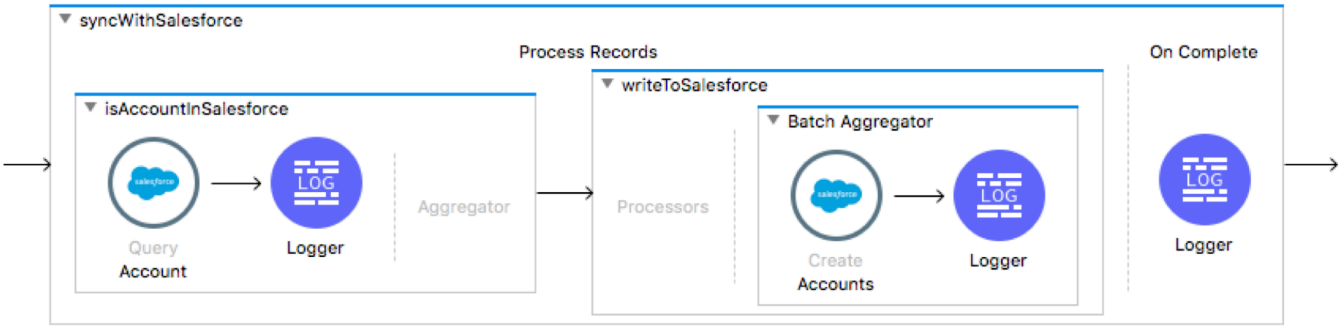
35. Step through the rest of the application; you should see the JSON message in the console.

    Note: If you want to test further, return to the account data in the web browser at http://mu.mulesoft-training.com/accounts/show and add a new record with the same postal code.

36. Stop the project and switch perspectives.

Module 13: Processing Records





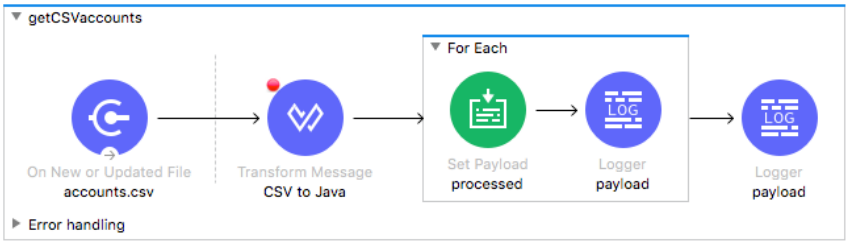
At the end of this module, you should be able to:

- Process items in a collection using the For Each scope.
- Process records using the Batch Job scope.
- Use filtering and aggregation in a batch step.

Walkthrough 13-1: Process items in a collection using the For Each scope

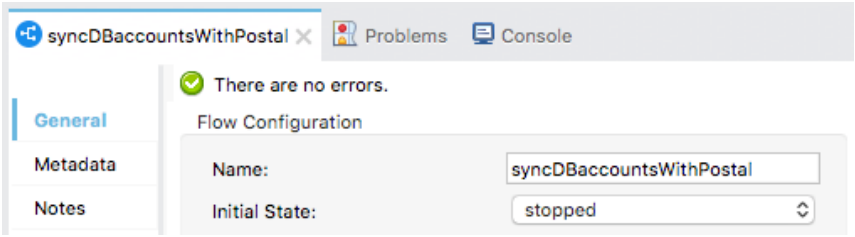
In this walkthrough, you split a collection and process each item in it. You will:

- Use the For Each scope element to process each item in a collection individually.
- Change the value of an item inside the scope.
- Examine the payload before, during, and after the scope.
- Look at the thread used to process each item.



Stop the syncDBaccountsWithPostal flow so it does not run

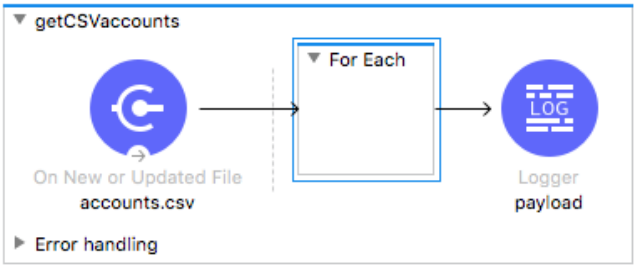
1. Return to accounts.xml.
2. In the properties view for the syncDBaccountsWithPostal flow, set the initial state to stopped.



3. Right-click in the canvas and select Collapse All.

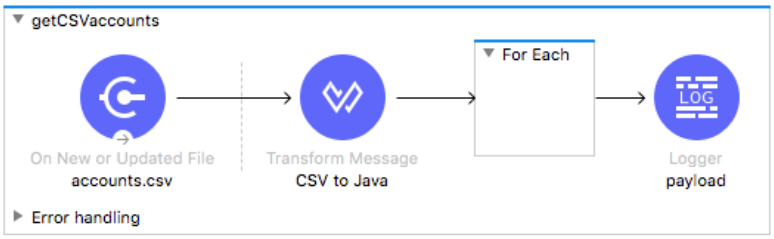
Add a For Each scope

4. Expand getCSVaccounts.
5. In the Mule Palette, select Core.
6. Locate the For Each scope and drag and drop it before the Logger.

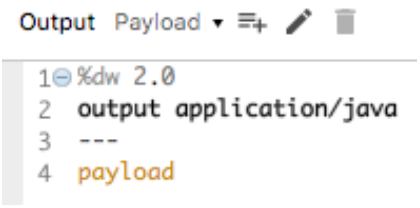


Transform the input to a collection

7. Add a Transform Message component before the For Each scope.
8. Set its display name to CSV to Java.

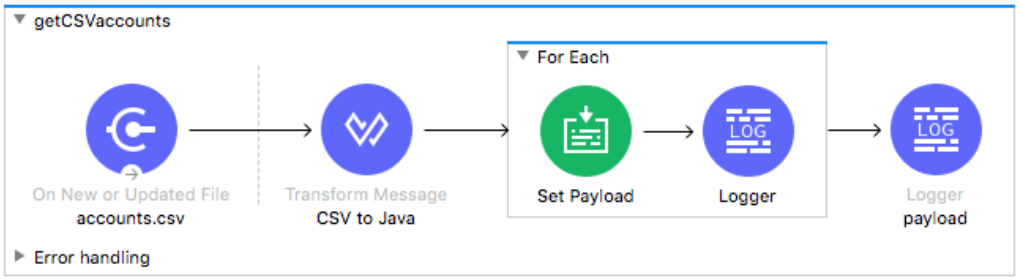


9. In the Transform Message properties view, leave the output type set to java and set the expression to payload.

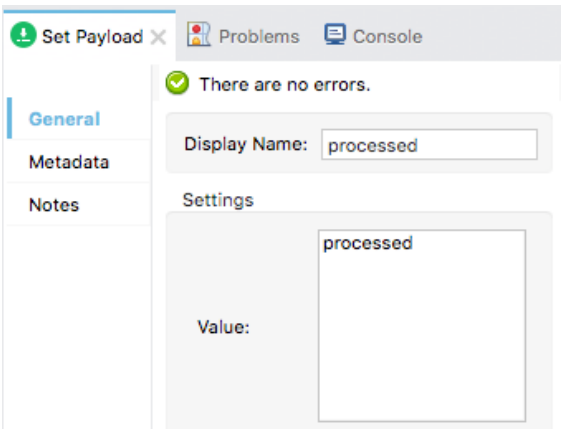


Process each element in the For Each scope

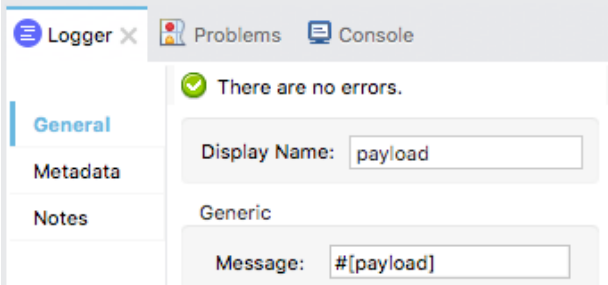
10. Add a Set Payload transformer and a Logger to the For Each scope.



11. In the Set Payload properties view, set the display name and value to the string: processed.

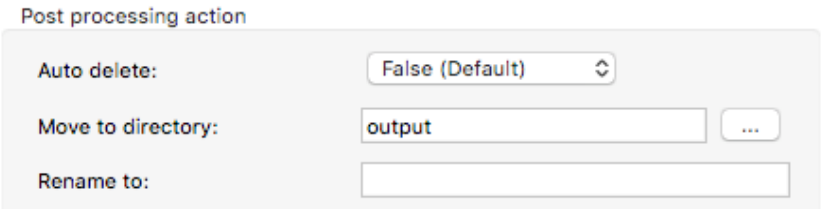


12. Set the Logger display name to payload and have it display the payload.



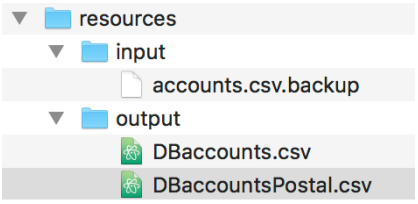
Change the File listener so it does not rename files

13. In the On New or Updated File properties view, delete the rename to value.



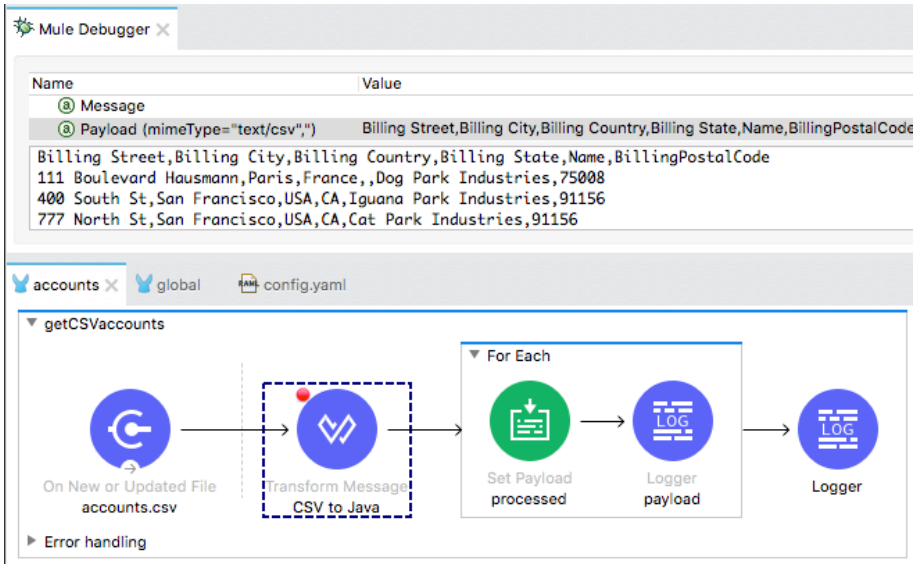
Debug the application

14. Add a breakpoint to the Transform Message component.
15. Debug the project and do not clear application data.
16. Return to the course student files in your computer's file browser.

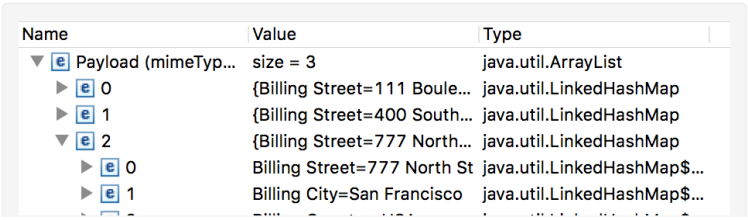


17. Rename accounts.csv.backup to accounts.csv.
18. Return to Anypoint Studio; application execution should have stopped at the Transform Message component.

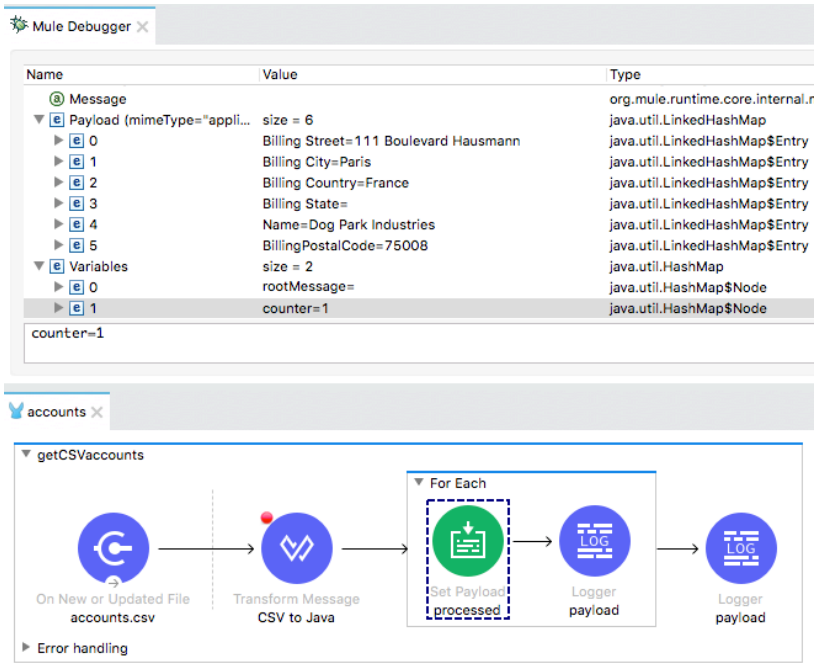
19. In the Mule Debugger view, look at the payload type and value.



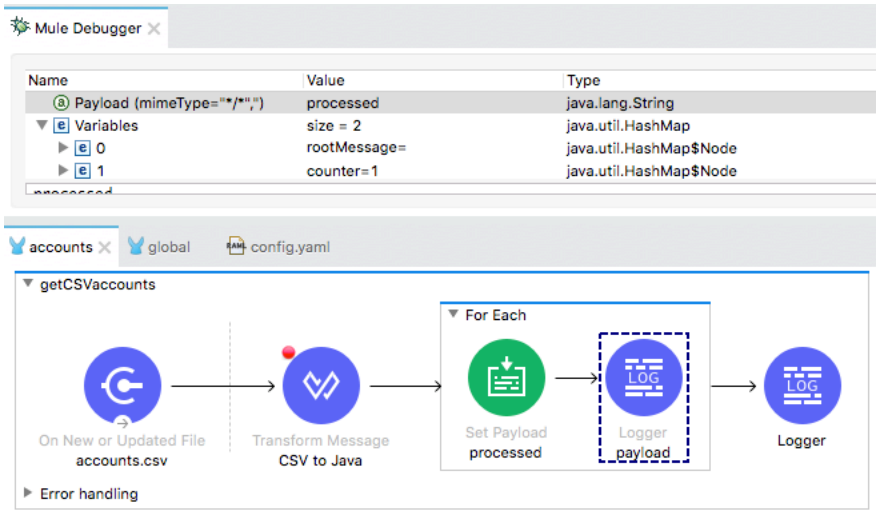
20. Step to the For Each scope; the payload should now be an ArrayList of LinkedHashMaps.



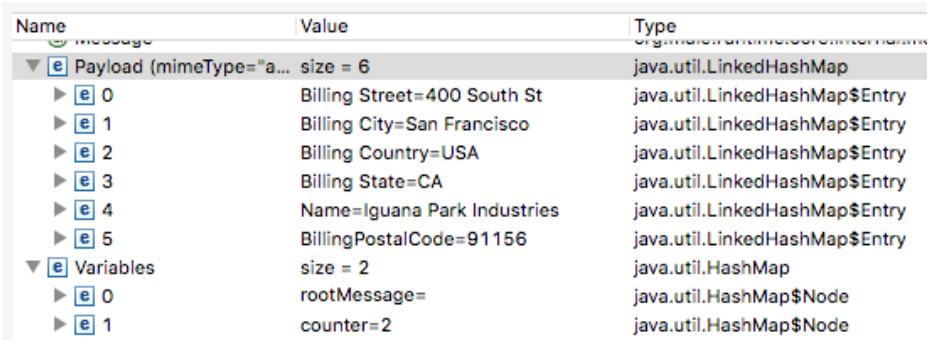
21. Step into the For Each scope; the payload should be a LinkedHashMap.
22. Expand Variables; you should see a counter variable.



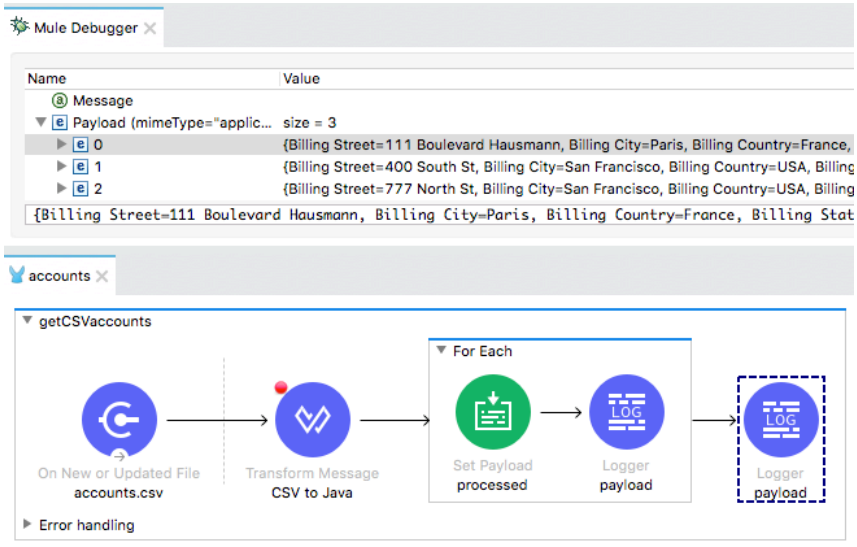
23. Step again; you should see the payload for this record inside the scope has been set to the string, processed.



24. Step again and look at the payload and counter for the second record.



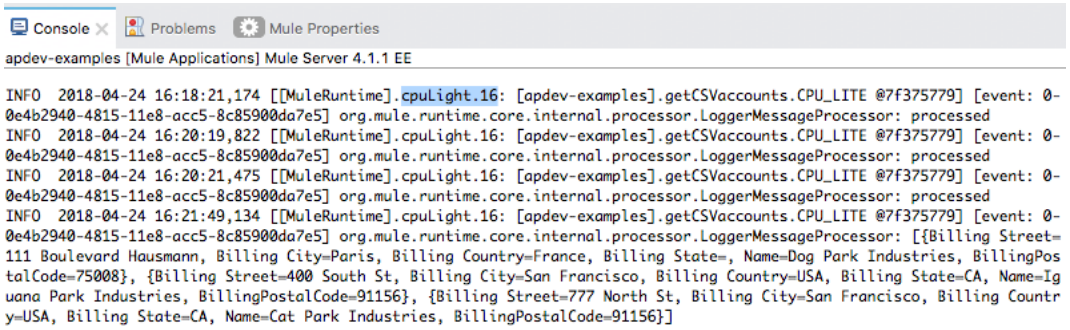
25. Step through the application to the Logger after the For Each scope; the payload should be equal to the original ArrayList of HashMaps and not a list of processed strings.



26. Step to the end of the application.
27. Stop the project and switch perspectives.

Look at the processing threads

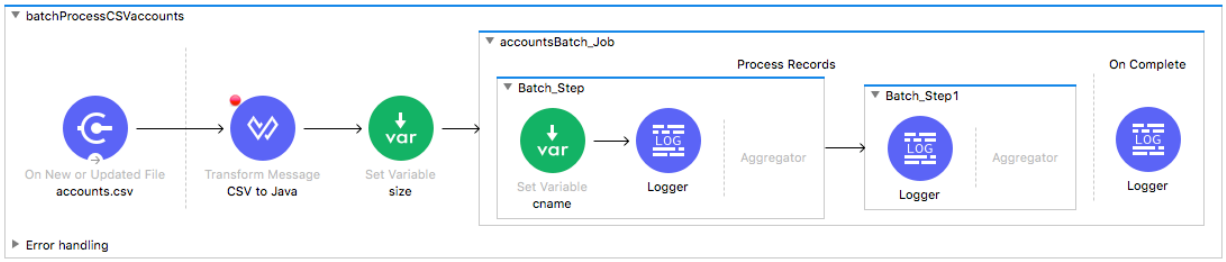
28. In the console, locate the thread number used to process each item in the collection; the same thread should be used for each: cpuLight.16 in the following screenshot.



Walkthrough 13-2: Process records using the Batch Job scope

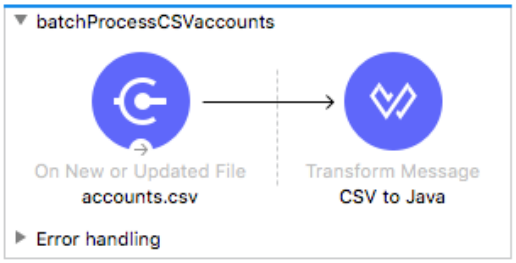
In this walkthrough, you create a batch job to process the records in a CSV file. You will:

- Use the Batch Job scope to process items in a collection.
- Examine the payload as it moves through the batch job.
- Explore variable persistence across batch steps and phases.
- Examine the payload that contains information about the job in the on complete phase.
- Look at the threads used to process the records in each step.

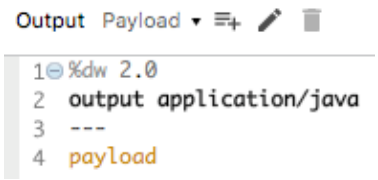


Create a new flow to read CSV files

1. Return to accounts.xml.
2. Drag a Flow scope from the Mule Palette and drop it above getCSVaccounts.
3. Change the flow name to batchProcessCSVaccounts.
4. Select the On New or Updated File operation in getCSVaccounts and select Edit > Copy.
5. Click the Source section of batchProcessCSVaccounts and select Edit > Paste.
6. Modify the display name.
7. Add a Transform Message component to the flow.
8. Set the display name to CSV to Java.

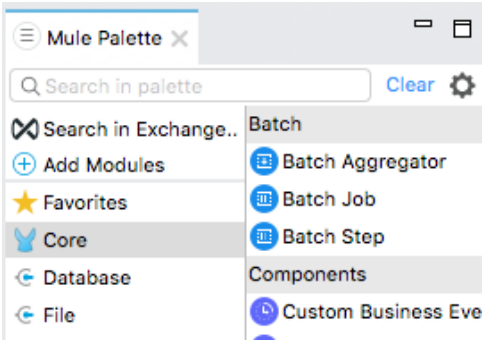


9. In the Transform Message properties view, change the body expression to payload.

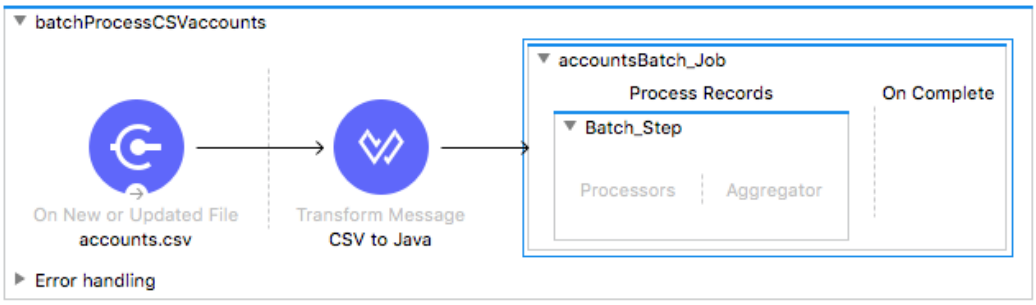


Add a Batch Job scope

10. In the Mule Palette, select Core and locate the Batch elements.

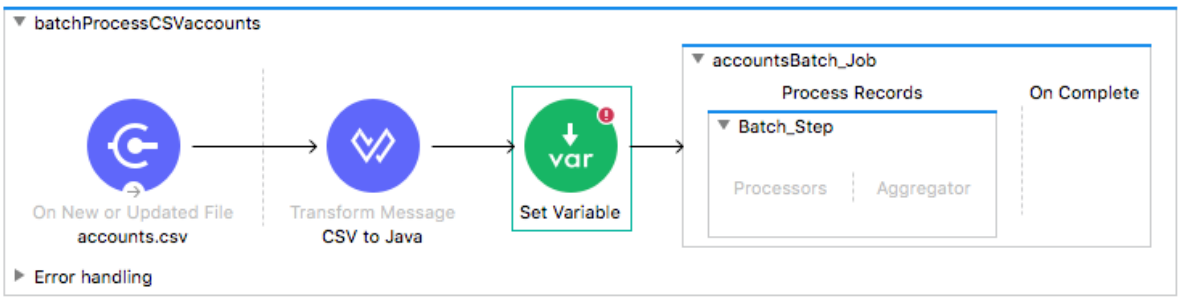


11. Drag a Batch Job scope and drop it after the Transform Message component.



Set a variable before the Batch Job scope

12. Add a Set Variable transformer before the Batch Job scope.

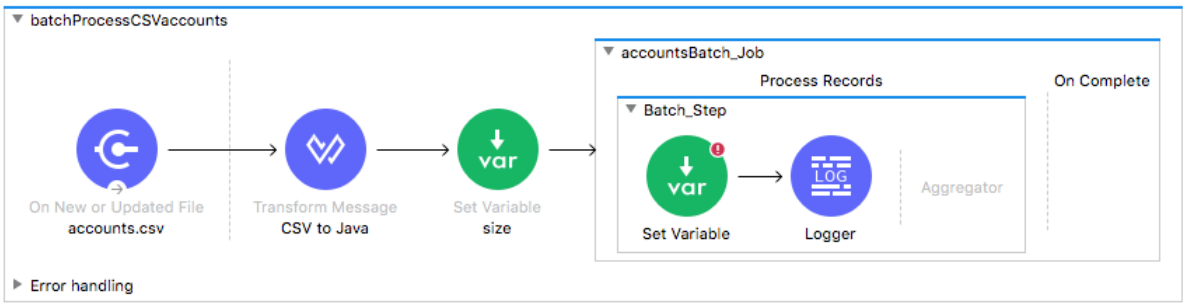


13. In the Set Variable properties view, set the following:

- Display name: size
- Name: size
- Value: #[sizeOf(payload)]

Set a variable inside the Batch Job scope

14. Add a Set Variable transformer to the batch step in the processors phase of the batch step.
15. Add a Logger component to the batch step after the transformer.

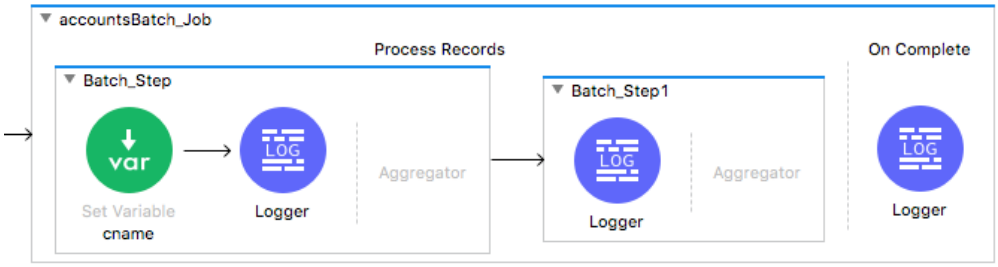


16. In the Set Variable properties view, set the following:

- Display name: cname
- Name: cname
- Value: #[payload.Name]

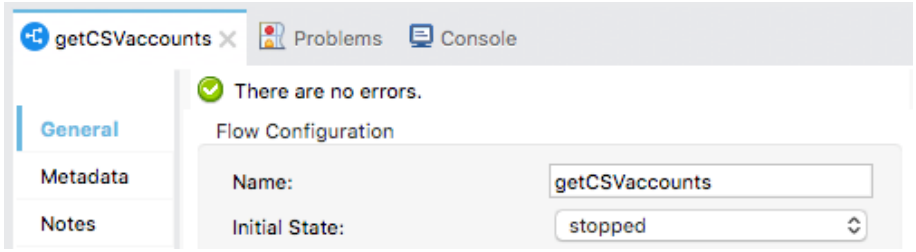
Create a second batch step

17. Drag a Batch Step scope from the Mule Palette and drop it in the process records phase of the Batch Job scope after the first batch step.
18. Add a Logger to the processors section of the second batch step.
19. Add a Logger to the On Complete phase.



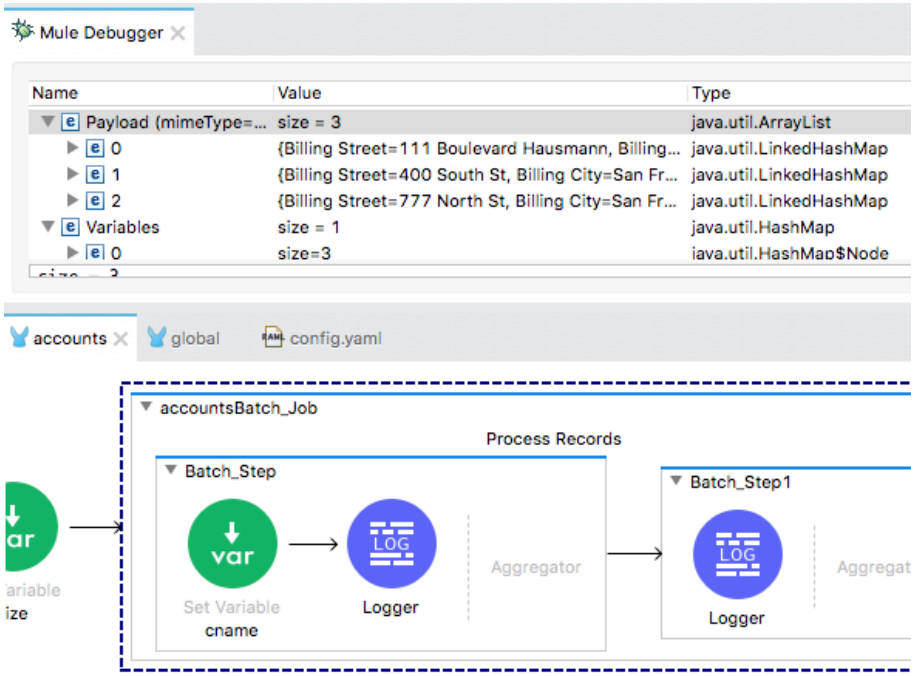
Stop the other flow watching the same file directory from being executed

20. In the properties view for the getCSVaccounts flow, set the initial state to stopped.

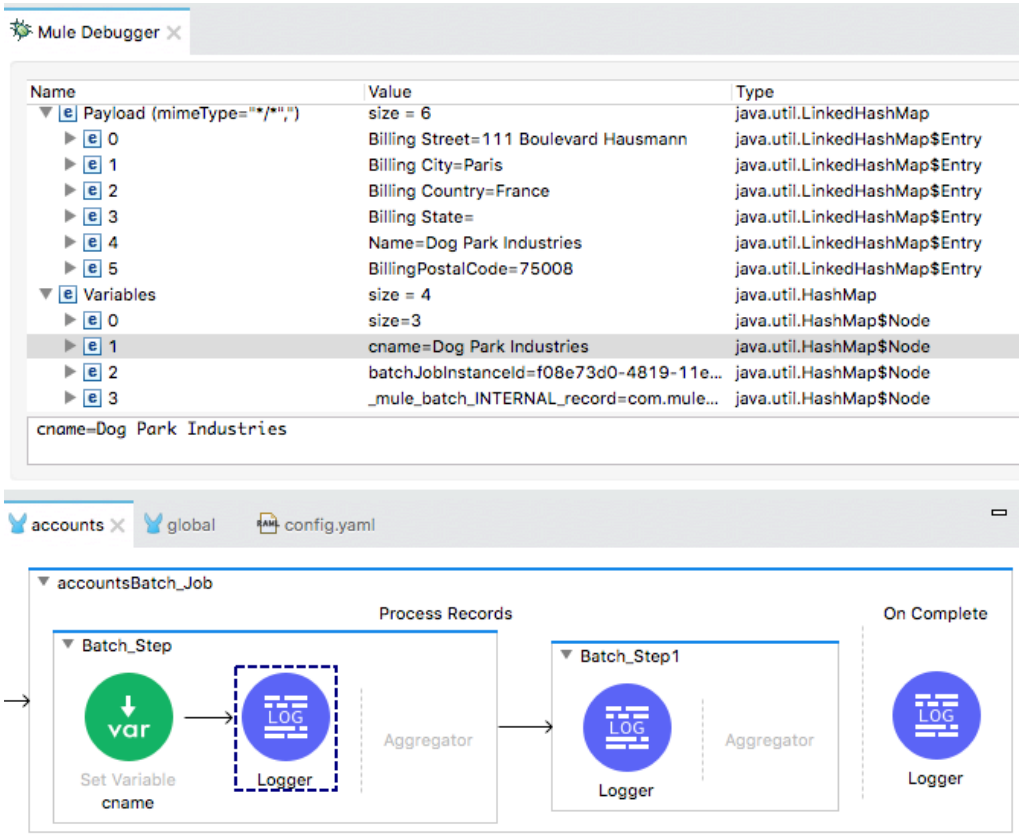


Debug the application

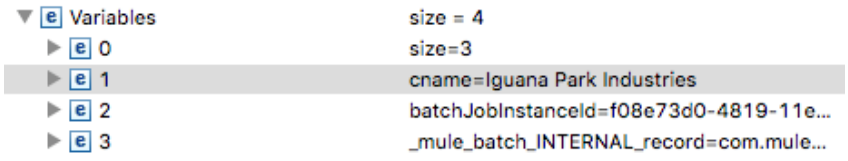
21. In batchProcessCSVaccounts, add a breakpoint to the Transform Message component.
22. Save the file to redeploy the application in debug mode.
23. Return to the course student files in your computer's file browser and move accounts.csv from the output folder to the input folder.
24. In the Mule Debugger view, watch the payload as you step to the Batch Job scope; it should go from CSV to an ArrayList.
25. In the Mule Debugger view, expand Variables; you should see the size variable.



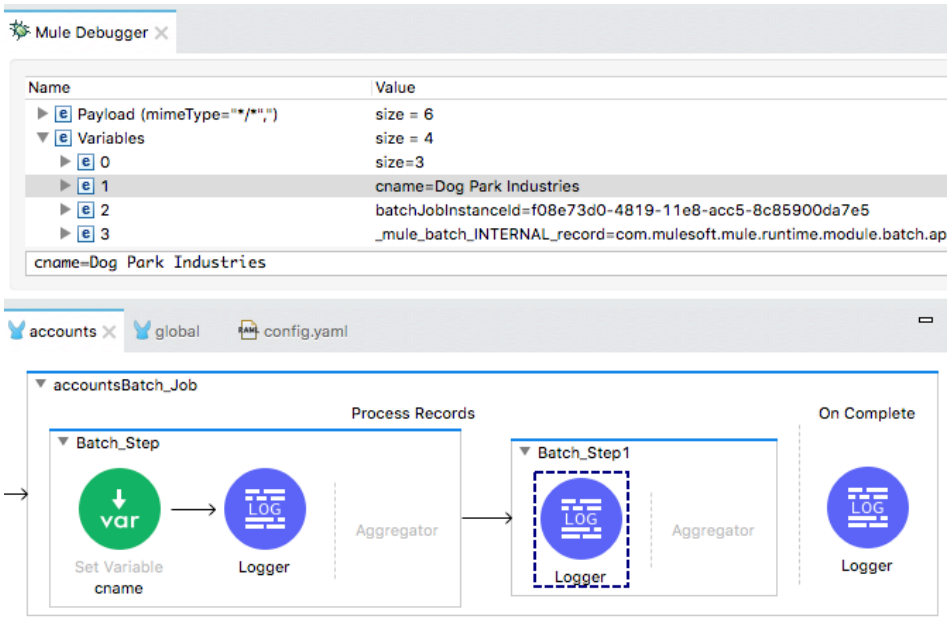
26. Step to the Logger in the first batch step.
27. In the Mule Debugger view, look at the value of the payload; it should be a HashMap.
28. Expand Variables; you should see the size variable and the cname variable specific for that record.



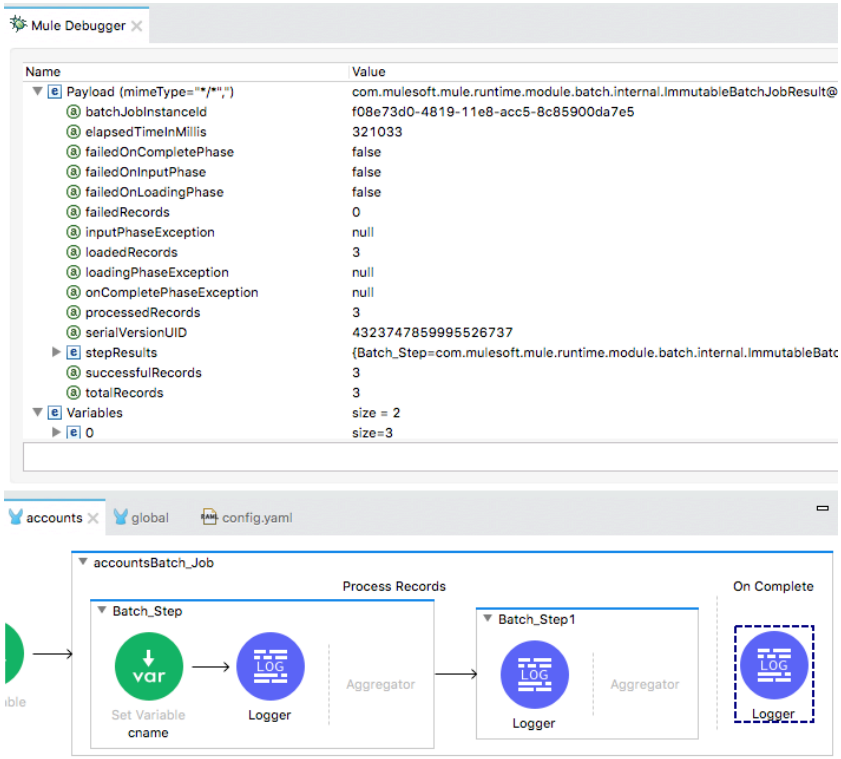
29. Step through the rest of the records in the first batch step and watch the payload and the cname variable change.



30. Step into the second batch step and look at the payload and the cname variable; you should see the cname variable is defined and has a value.



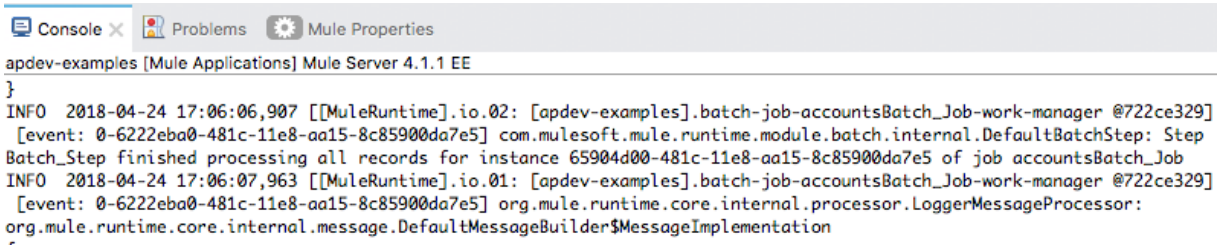
31. Step through the rest of the records in the second batch step and watch the value of cname.
32. Step into the on complete phase; you should see the payload is an ImmutableBatchJobResult.
33. Expand the payload and locate the values for totalRecords, successfulRecords, and failedRecords.



34. Step through the rest of the application and switch perspectives.
35. Stop the project.

Look at the processing threads

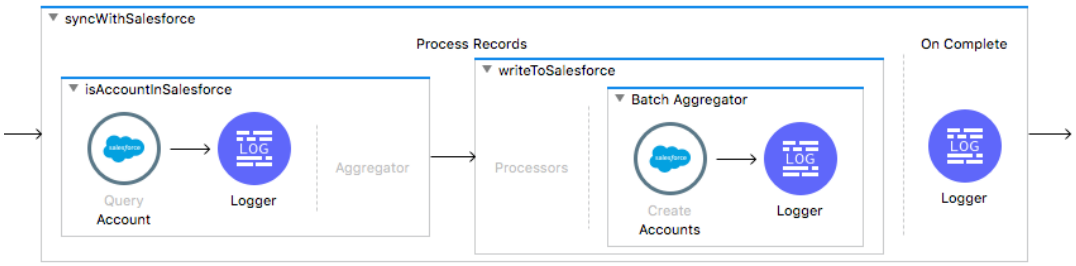
36. In the console, locate the thread number used to process each record in the collection in each step of the batch process; you should see more than one thread used.



Walkthrough 13-3: Use filtering and aggregation in a batch step

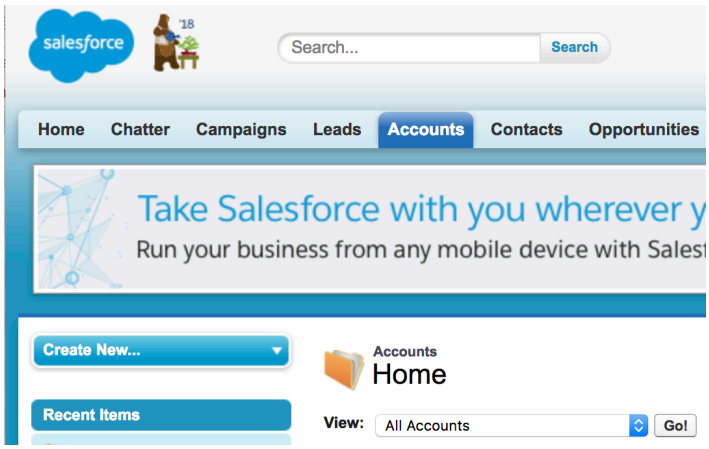
In this walkthrough, you use a batch job to synchronize account database records to Salesforce. You will:

- Use a batch job to synchronize database records (with your postal code) to Salesforce.
- In a first batch step, check to see if the record already exists in Salesforce.
- In a second batch step, add the record to Salesforce.
- Use a batch step filter so the second batch step is only executed for specific records.
- Use a Batch Aggregator scope to commit records in batches.

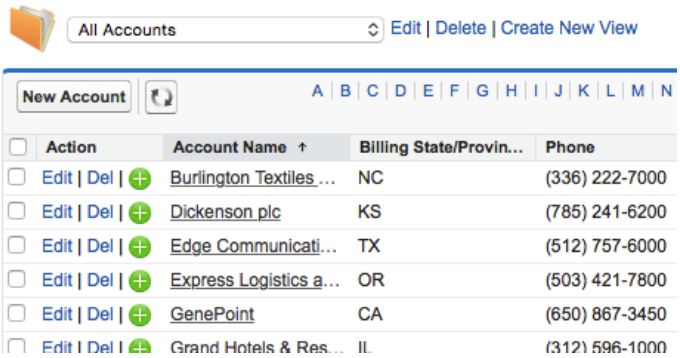


Look at existing Salesforce account data

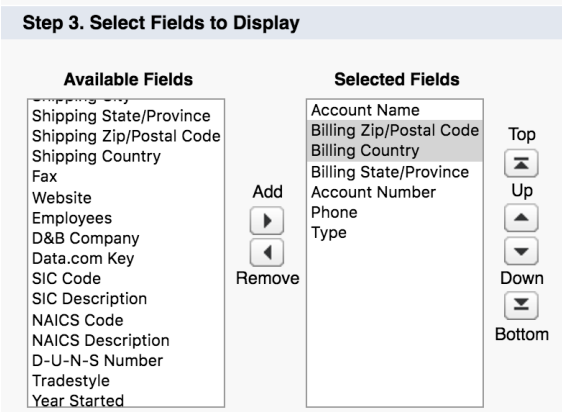
1. In a web browser, navigate to http://login.salesforce.com/ and log in with your Salesforce Developer account.
2. Click the Accounts link in the main menu bar.
3. In the view drop-down menu, select All Accounts and click the Go button.



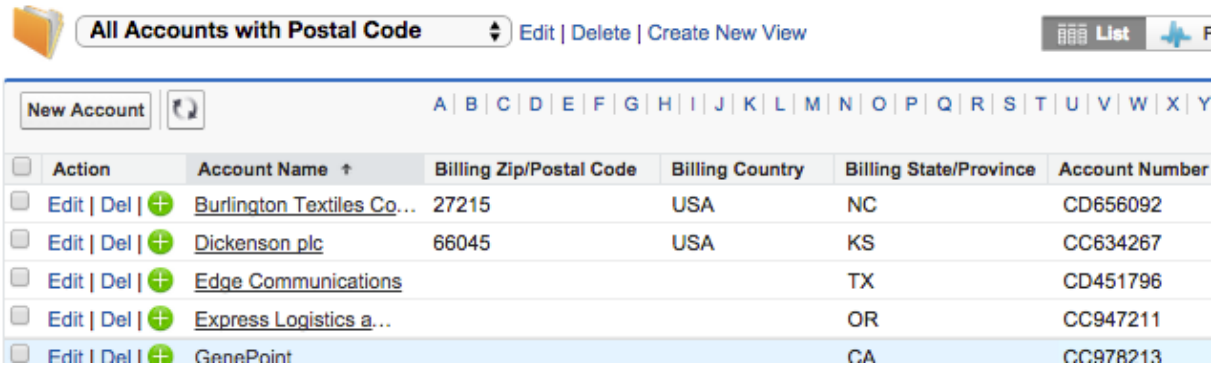
4. Look at the existing account data; a Salesforce Developer account is populated with some sample data.



5. Notice that countries and postal codes are not displayed by default.
6. Click the Create New View link next to the drop-down menu displaying All Accounts.
7. Set the view name to All Accounts with Postal Code.
8. Locate the Select Fields to Display section.
9. Select Billing Zip/Postal Code as the available field and click the Add button.
10. Add the Billing Country field.
11. Use the Up and Down buttons to order the fields as you prefer.

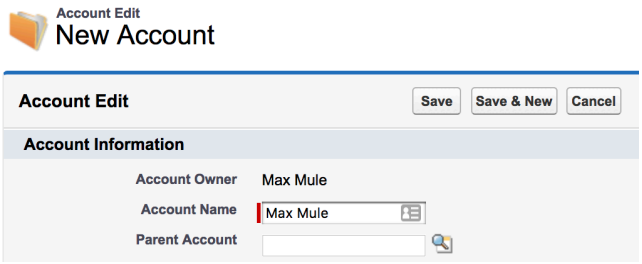


12. Click the Save button; you should now see all the accounts with postal codes and countries.



Add an account to Salesforce with a name matching one of the database records

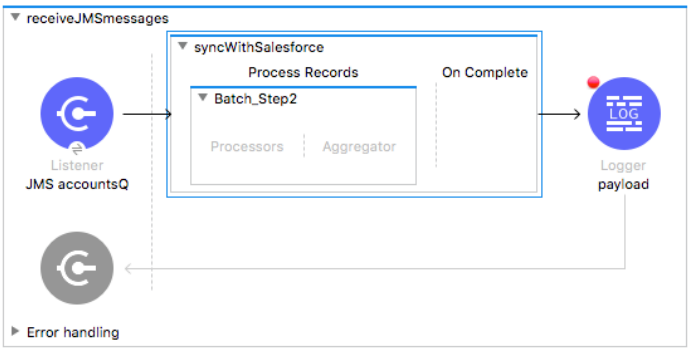
13. Click the New Account button.
14. Enter an account name (one that matches one of the accounts you added with your postal code to the database) and click Save.



15. Leave this window open.

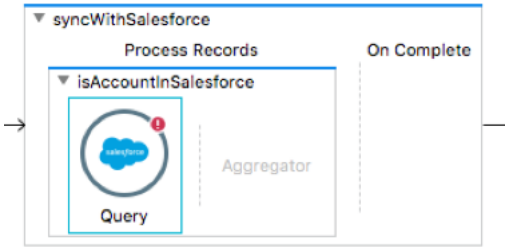
Add a Batch Job scope to the receiveJMSmessages flow

16. Return to accounts.xml in Anypoint Studio.
17. Drag a Batch Job scope from the Mule Palette and drop it before the Logger in the receiveJMSmessages flow.
18. Change the display name of the batch job to syncWithSalesforce.



Query Salesforce to see if an account already exists

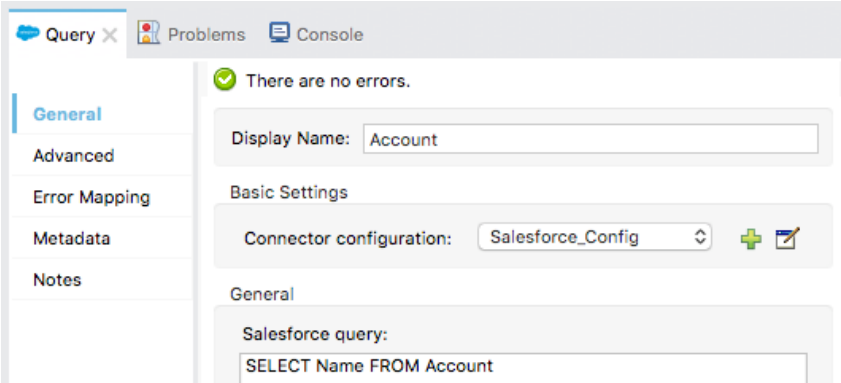
19. Change the name of the batch step inside the batch job to isAccountInSalesforce.
20. Drag a Salesforce Query operation from the Mule Palette and drop it in the processors phase in the batch step.



21. In the Query properties view, set the following values:

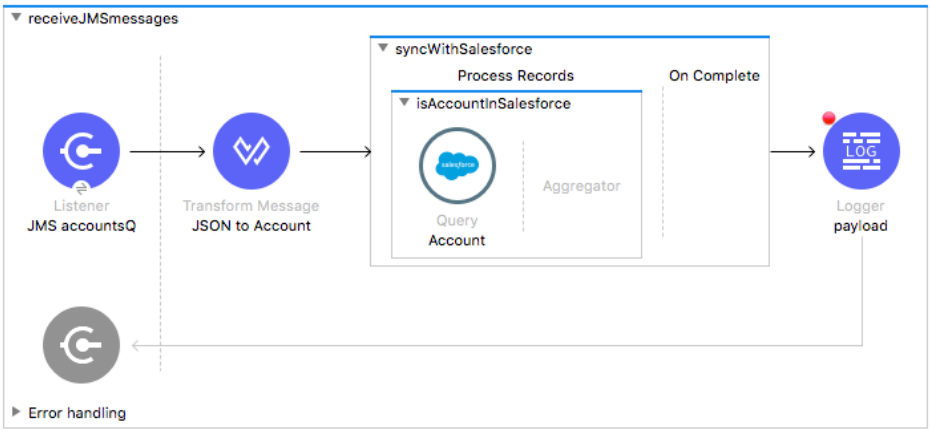
    - Display name: Account
    - Connector configuration: Salesforce_Config
    - Salesforce query: SELECT Name FROM Account

    Note: You will add to this query shortly.

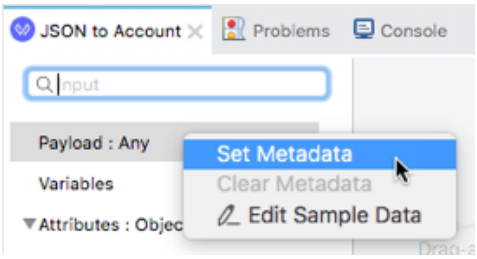


Transform the input JSON data to Salesforce Account objects

22. Add a Transform Message component before the Batch Job.
23. Change the display name to JSON to Accounts.

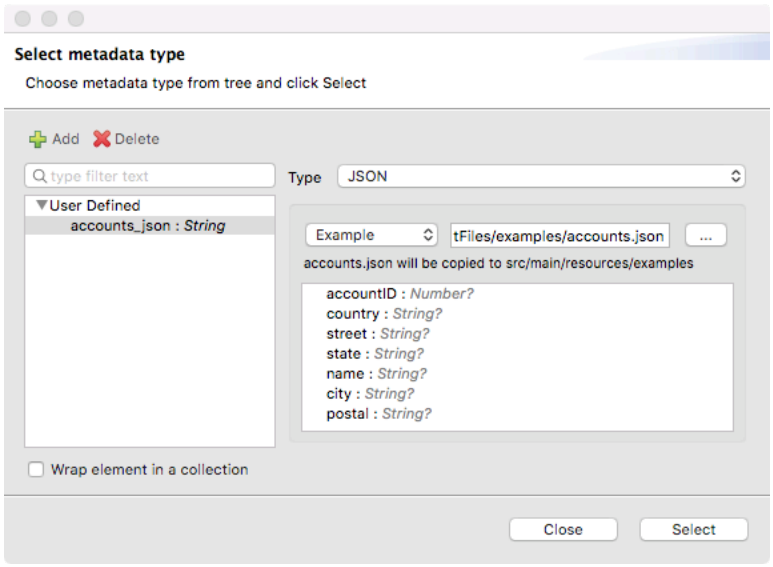


24. In the Transform Message properties view, look at the metadata in the input section.
25. Right-click Payload and select Set Metadata.



26. Create a new metadata type with the following information:

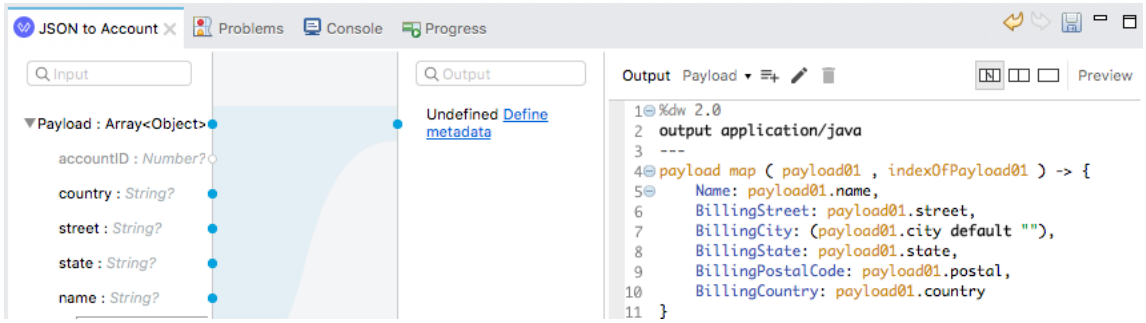
- Type id: accounts_json
- Type: JSON
- Example: *Use the accounts.json file in the examples folder of the course student files*



27. In the Transform Message properties view, map the following fields:

- country: BillingCountry
- street:BillingStreet
- state: BillingState
- name: BillingPostalCode
- city: BillingCity
- postal:BillingPostalCode

Note: If you do not get Salesforce metadata for the Account object, you can copy the DataWeave transformation from the course snippets.txt file.



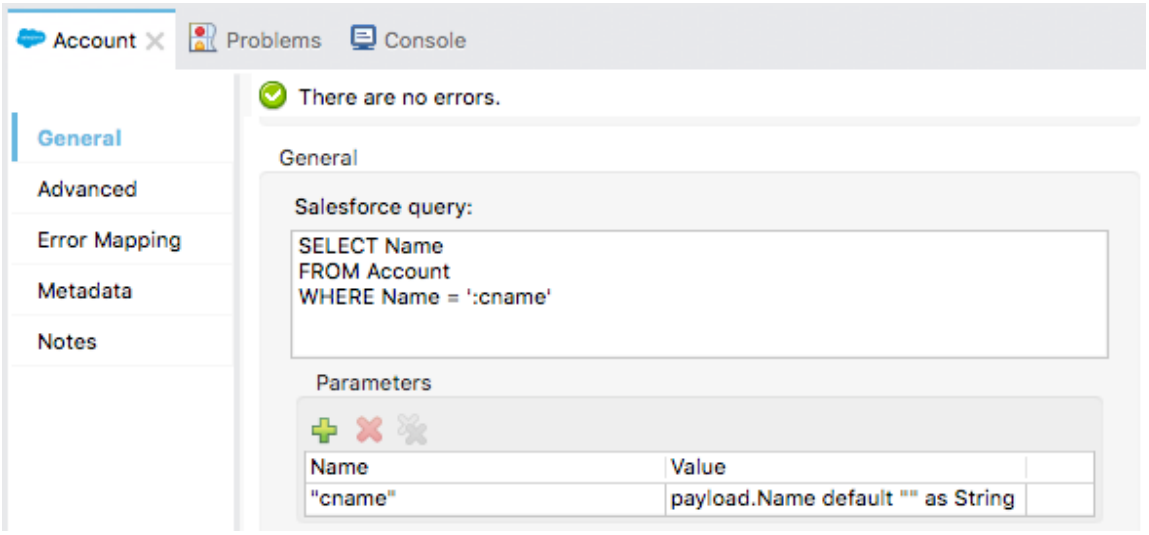
Finish the batch step to check if an account already exists in Salesforce

28. In the Query properties view, add an input parameter named cname.
29. Set the parameter value to payload.Name, ensure it is a String, and give it a default value.

```
payload.Name default "" as String
```

30. Modify the Salesforce query to look for accounts with this name.

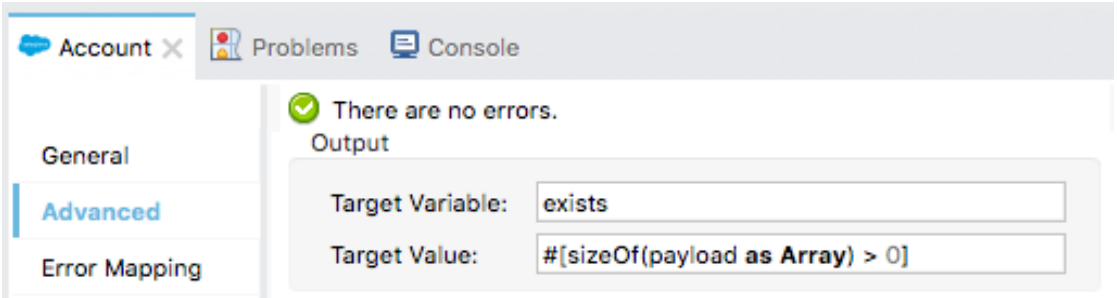
```
SELECT Name
FROM Account
WHERE Name= ':cname'
```



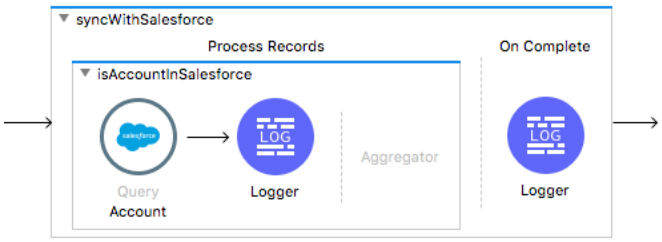
Store the result in a variable instead of overriding the payload

31. Select the Advanced tab.
32. Set the target variable to exists.
33. Set the target value to

```
#[(sizeOf(payload as Array) > 0)]
```

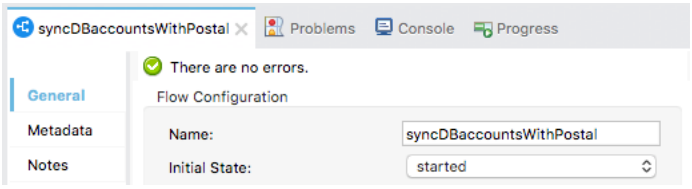


34. Add a Logger after the Query operation.
35. Add a Logger to the on complete phase.



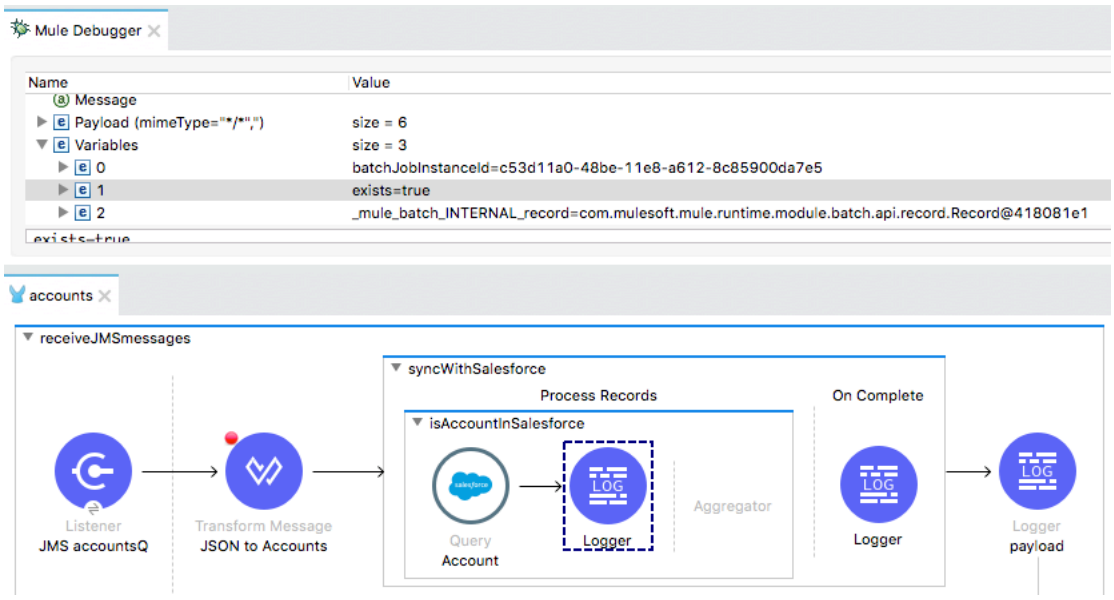
Change the initial state of the flow

36. In the properties view for the syncDBaccountsWithPostal flow, change the initial state to started.



Debug the application

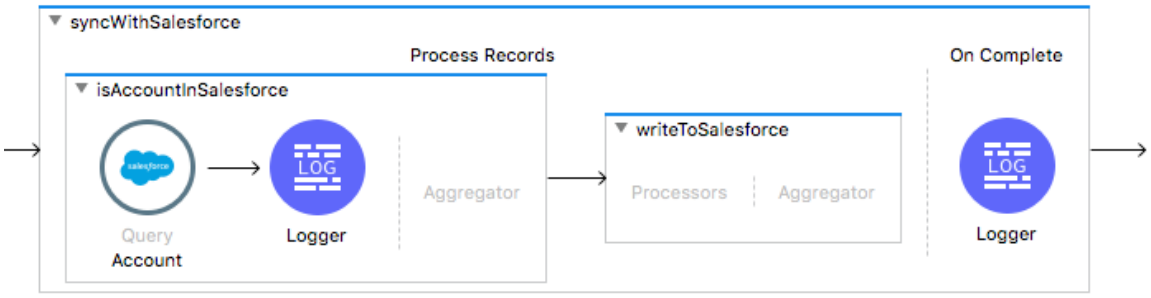
37. Add a breakpoint to the Transform Message component.
38. Debug the project.
39. Clear the application data.
40. In the Mule Debugger, wait until application execution stops in the receiveJMSmessages flow.
41. Step to the Logger in the first batch step and expand Variables; you should see the exists variable set to true or false.



42. Step through the application; you should see the exists variable set to false for records with names that don't exist in Salesforce and true for those that do.
43. Stop the project and switch perspectives.

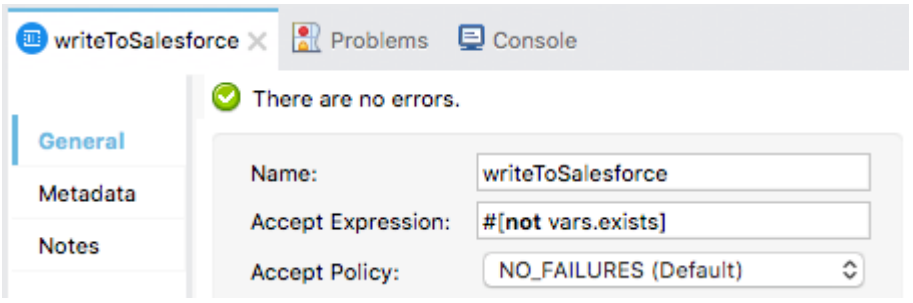
Set a filter for the insertion step

44. Add a second batch step to the batch job.
45. Change its display name to writeToSalesforce.



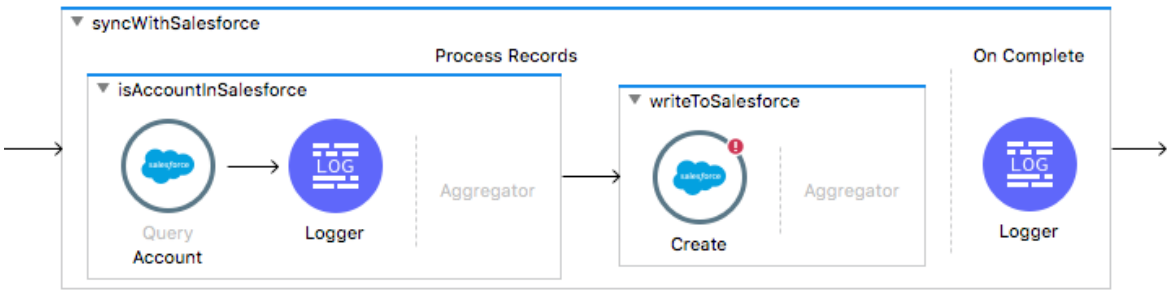
46. In the properties view for the second batch step, set the accept expression so that only records that have the variable exists set to false are processed.

```
#[not vars.exists]
```



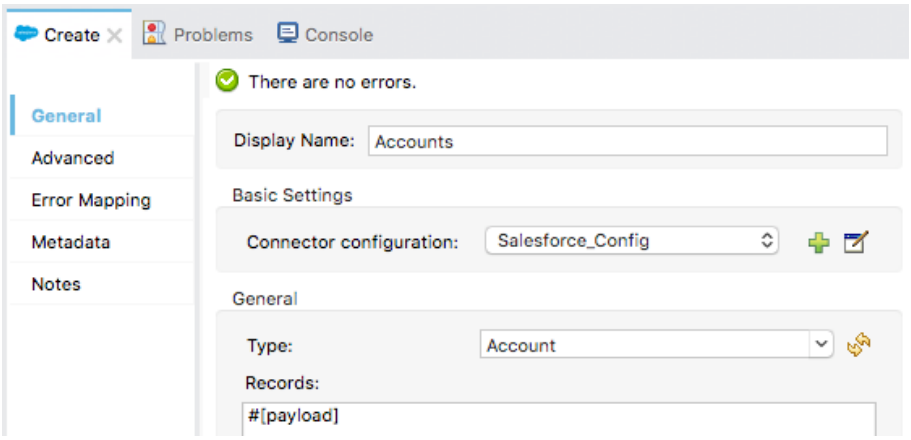
Use the Salesforce Create operation to add new account records to Salesforce

47. Add Salesforce Create operation to the second batch step in the processing phase.

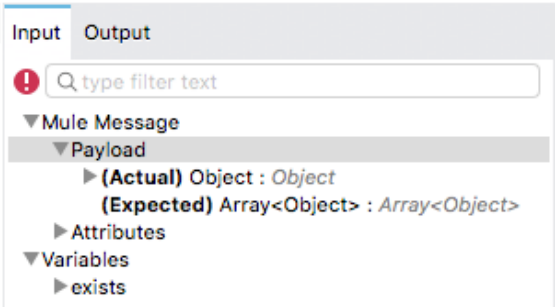


48. In the Create properties view, set the following:

    - Display name: Accounts
    - Connector configuration: Salesforce_Config
    - Type: Account
    - Records: #[payload]

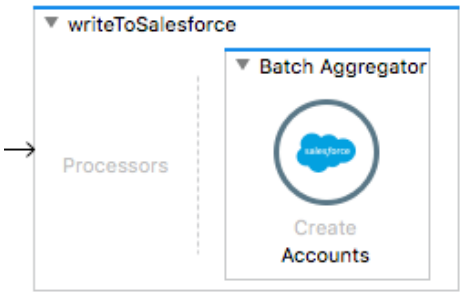


49. Select the Input tab in the DataSense Explorer; you should see this operation expects an Array of Account objects – not just one.

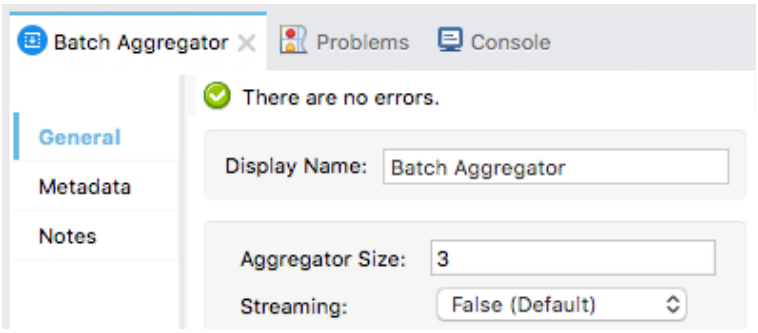


Create the Array of objects that the operation is expecting

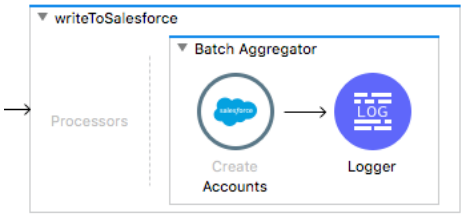
50. Drag a Batch Aggregator scope from the Mule Palette and drop it in the aggregator section of the second batch step.
51. Move the Salesforce Create operation into the Batch Aggregator.



52. In the Batch Aggregator properties view, leave the aggregator size set to 3.

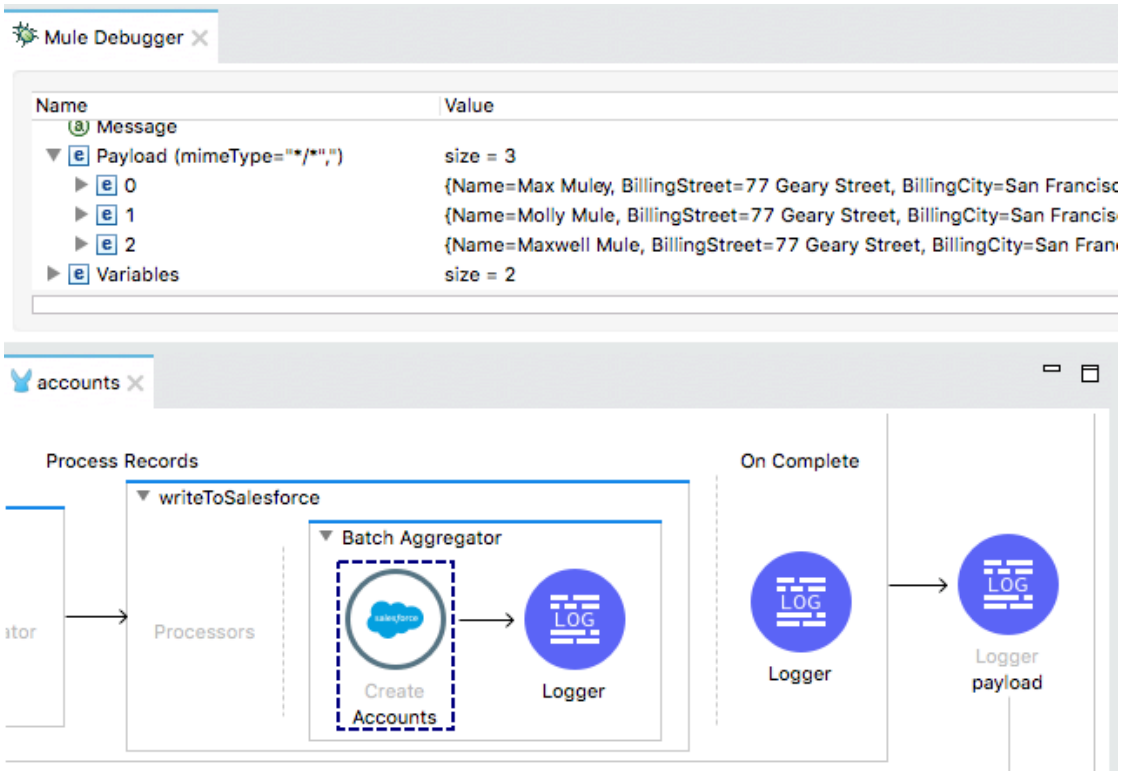


53. Add a Logger after the Create operation.



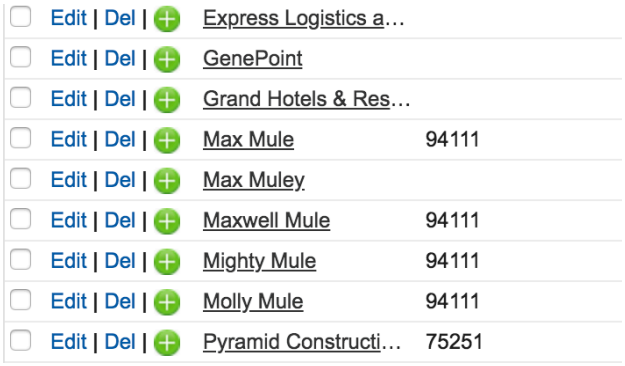
Test the application

54. Debug the project and clear the application data.
55. Step through the application until you step into the Batch Aggregator.
56. Expand Payload.



57. Step through the rest of the flow.
58. Return to Salesforce and look at the accounts; you should see the records from the legacy MySQL database are now in Salesforce.

    Note: You could also check for the records by making a request to http://localhost:8081/sfdc.



59. Run the application again but do not clear the application data; no records should be processed.
60. Return to salesforce.com and locate your new record(s); they should have been inserted only once.
61. Return to Anypoint Studio and stop the project.