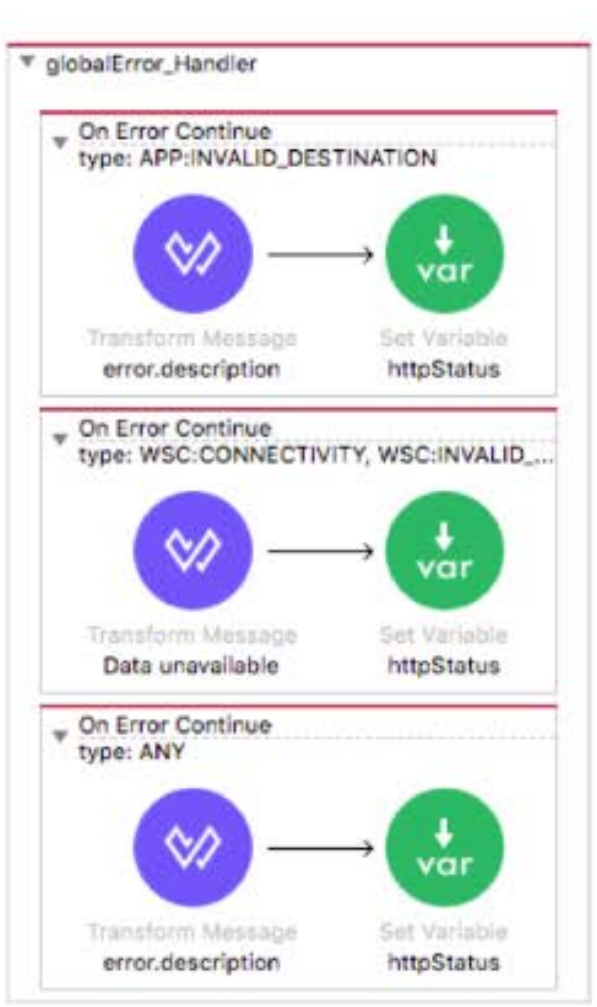
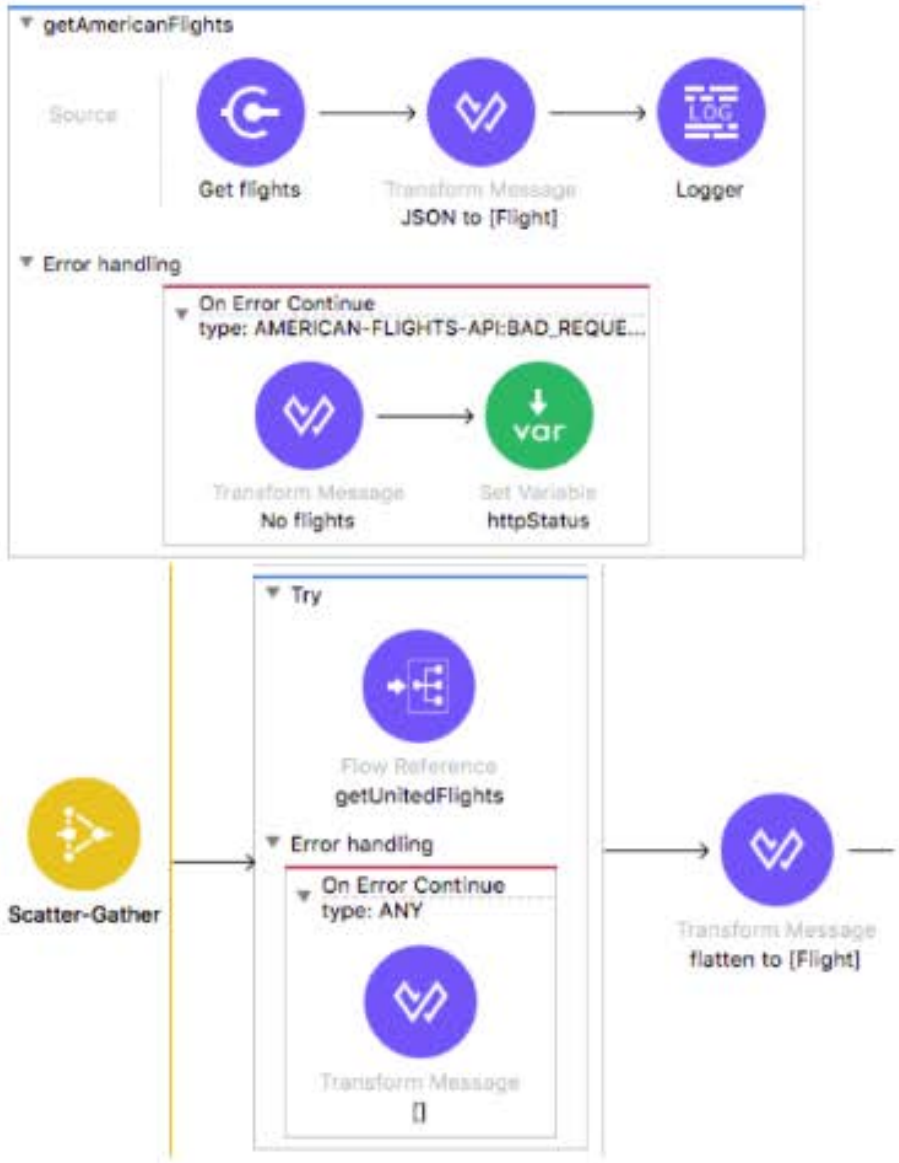

MuleSoft
Module 10: Handling Errors

Goal





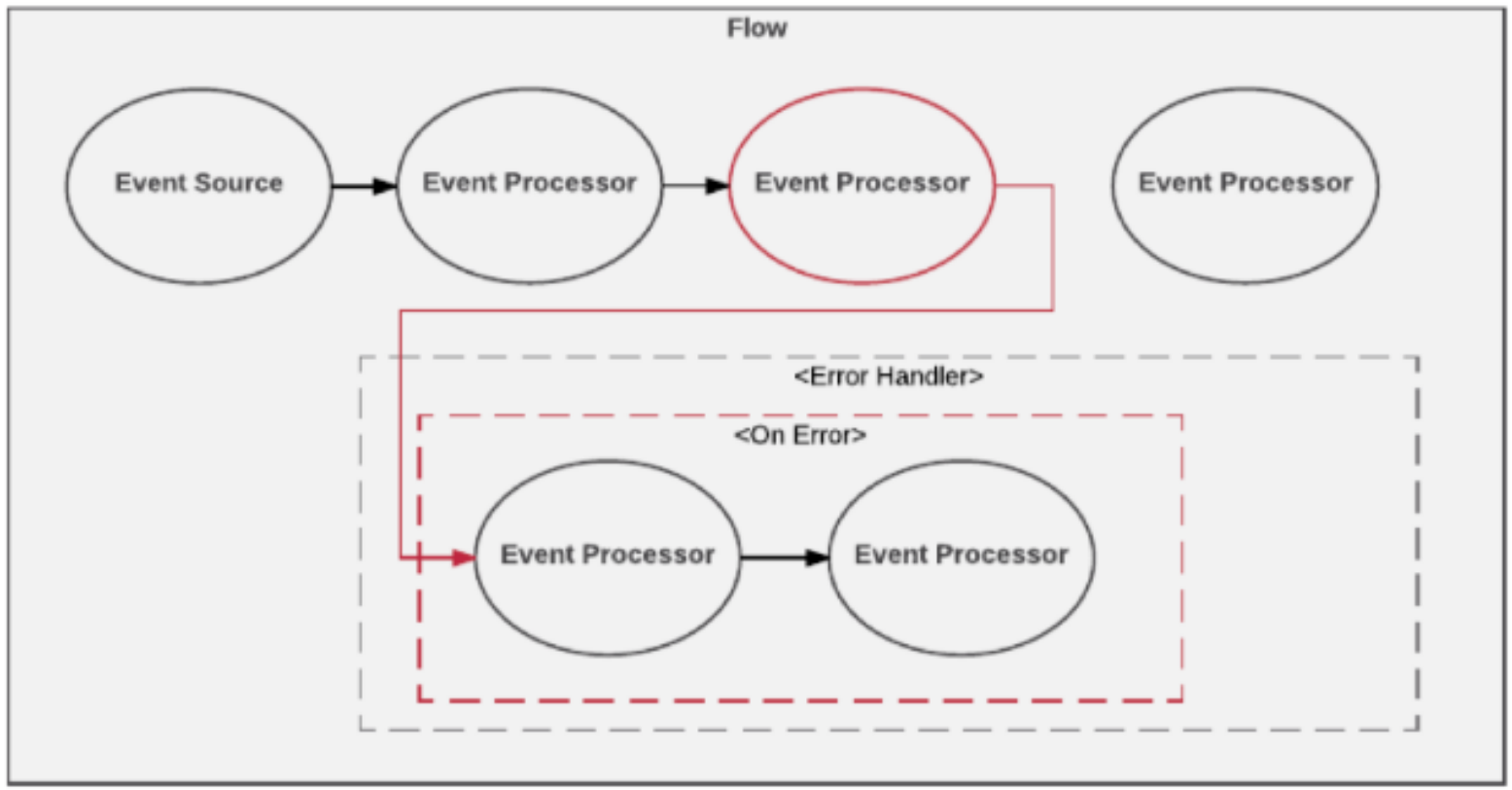
At the end of this module, you should be able to

- 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

Reviewing the default handling of messaging errors

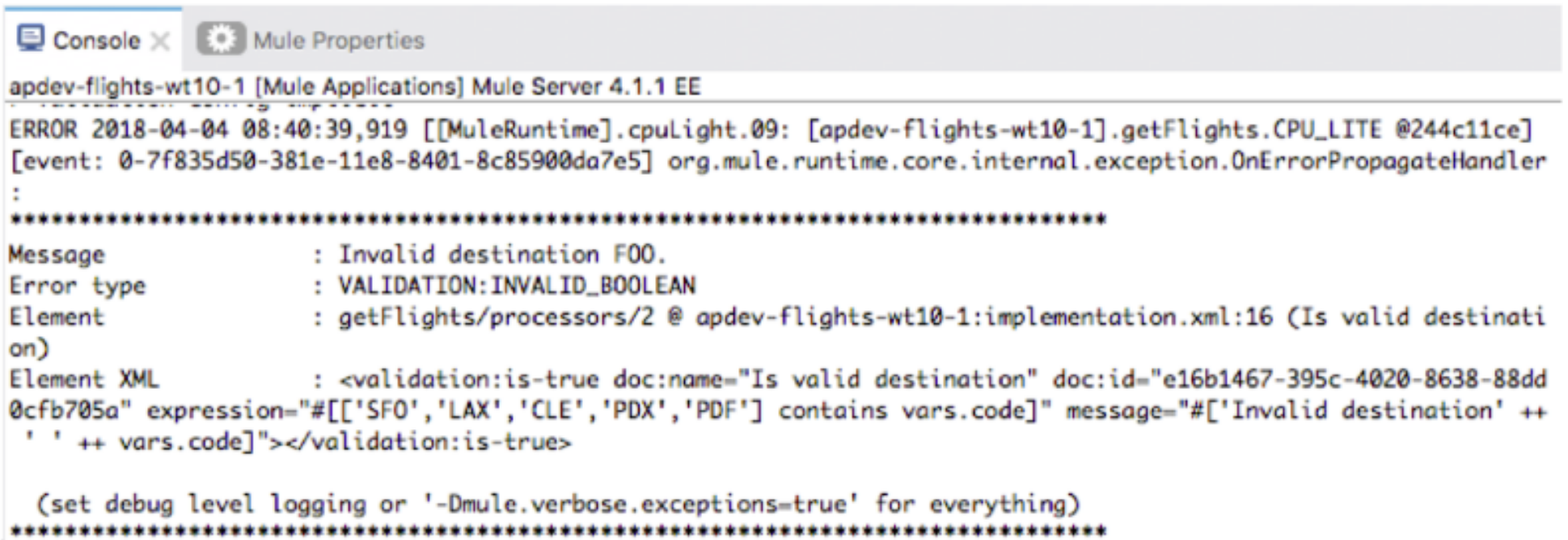
Handling messaging errors

- When an event is being processed through a Mule flow that throws an error
  - Normal flow execution stops
  - The event is passed to the first processor in an error handler



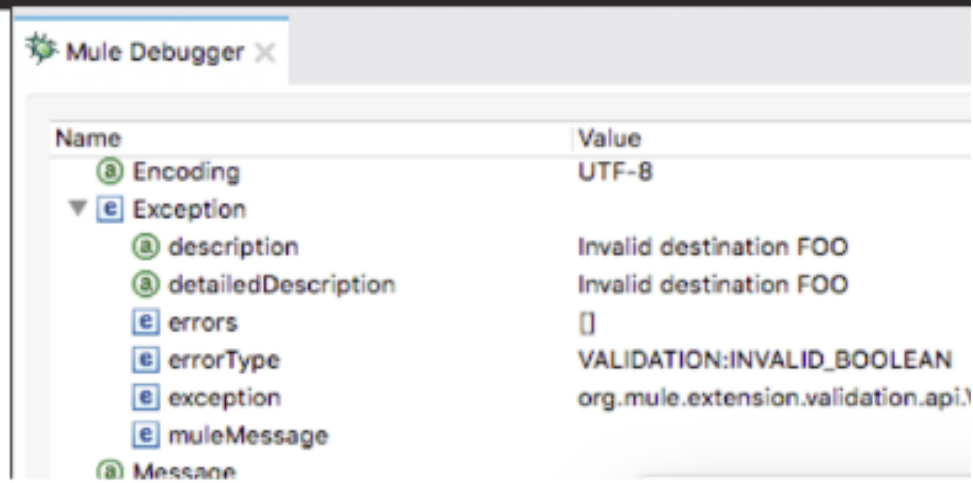
Default error handler behavior

- If there is no error handler defined, a **Mule default error handler** is used
  - Implicitly and globally handles all messaging errors thrown in Mule applications
  - Stops execution of the flow and logs information about the error
  - Cannot be configured



Information about the error

- When an error is thrown, an **error** object is created



- Two of its properties include
  - error.**description** – a string
  - error.**errorType** – an object

- Error types are identified by a namespace and an identifier
  - HTTP:UNAUTHORIZED, HTTP:CONNECTIVITY, VALIDATION:INVALID_BOOLEAN

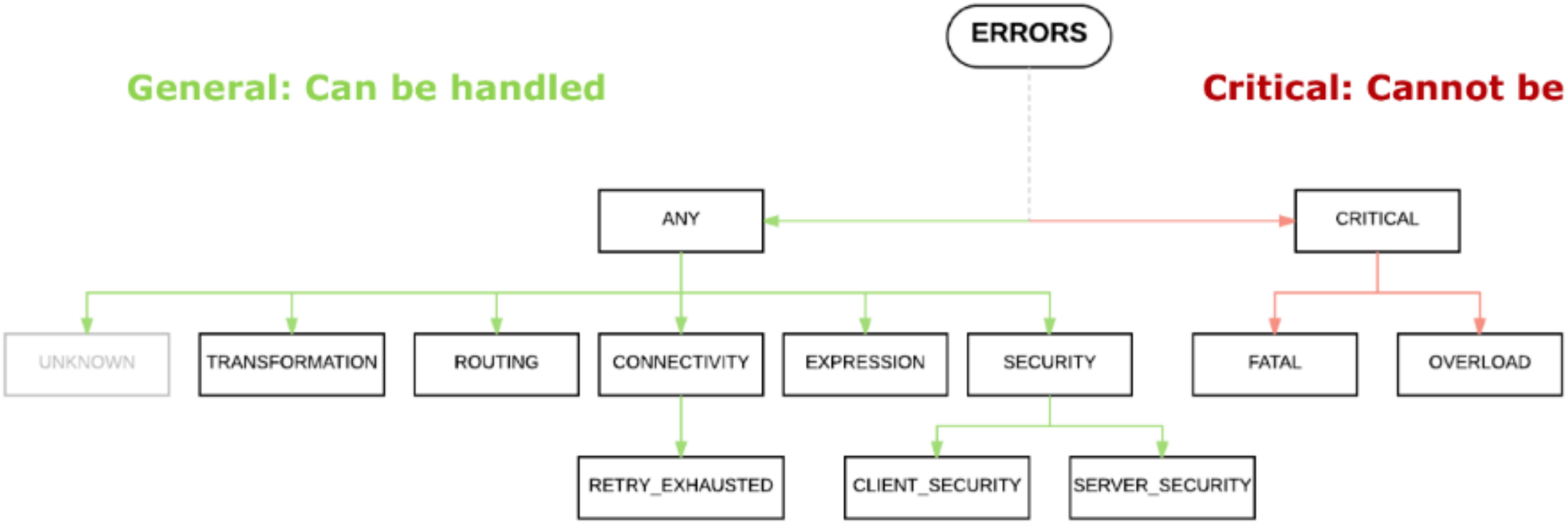
namespace identifier

Error types follow a hierarchy

- Each error type has a parent
  - HTTP:UNAUTHORIZED has MULE:CLIENT_SECURITY as the parent, which has MULE:SECURITY as the parent
  - VALIDATION:INVALID_BOOLEAN has VALIDATION:VALIDATION as the parent, which has MULE:VALIDATION as the parent
- The error type ANY is the most general parent

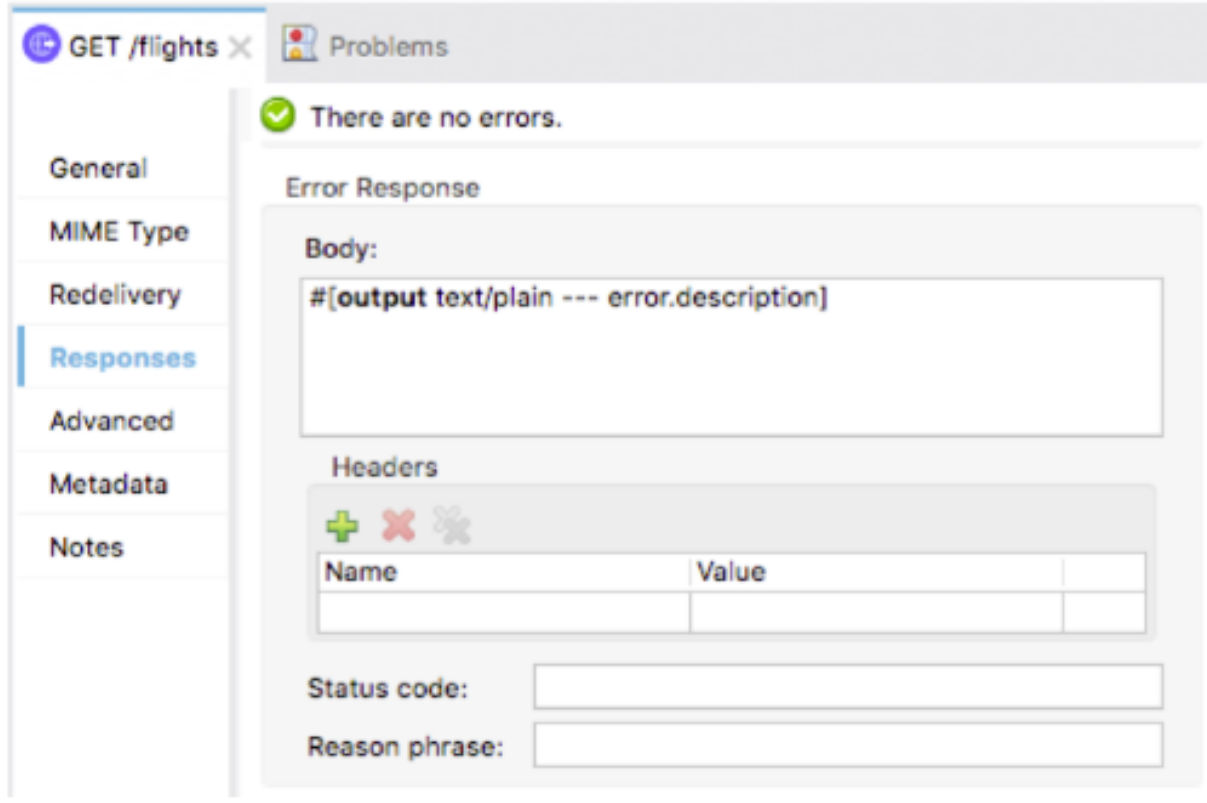
```
{
  "parentErrorType": {
    "parentErrorType": {
      "parentErrorType": {
        "parentErrorType": null,
        "namespace": "MULE",
        "identifier": "ANY"
      },
      "namespace": "MULE",
      "identifier": "VALIDATION"
    },
    "namespace": "VALIDATION",
    "identifier": "VALIDATION"
  },
  "namespace": "VALIDATION",
  "identifier": "INVALID_BOOLEAN"
}
```

Error type hierarchy reference



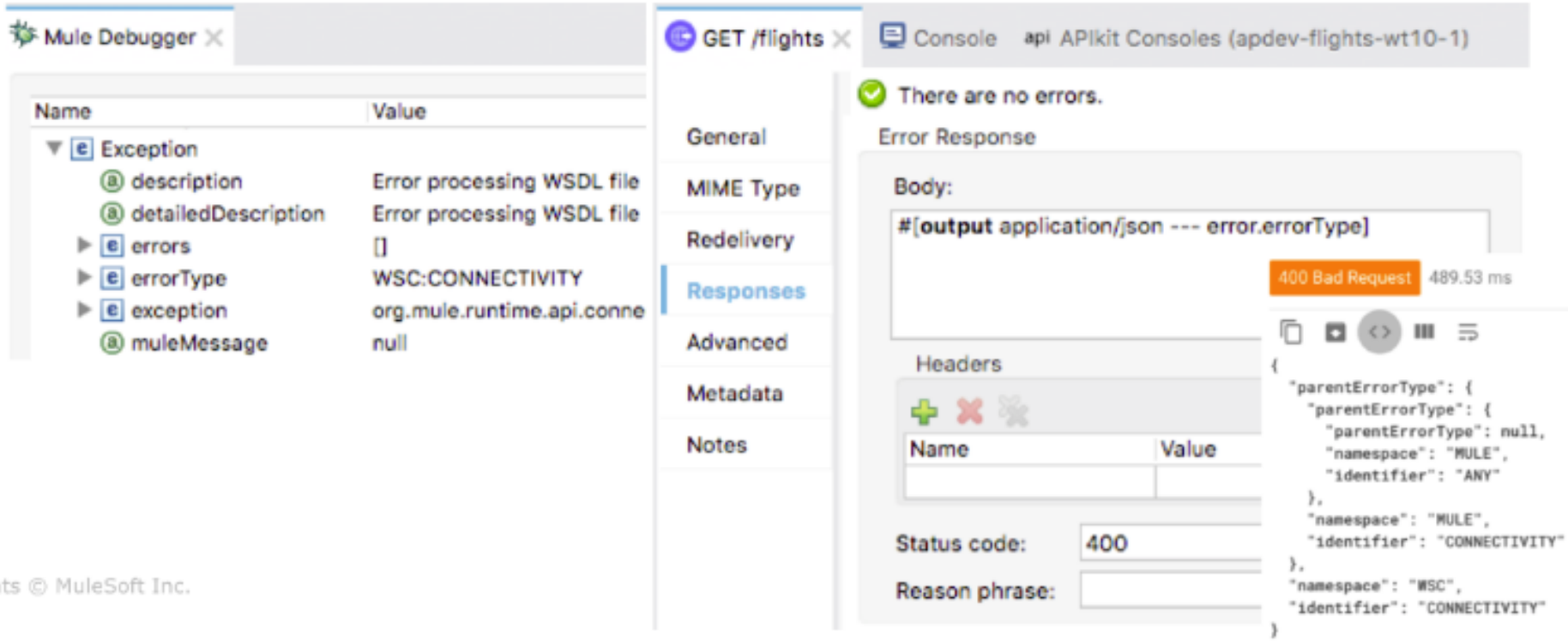
Information returned from HTTP Listeners

- By default, for a **success response**
  - The payload
  - A status code of 200
- By default, for an **error response**
  - The error description
  - A status code of 500
- You can override these values for an HTTP Listener



Walkthrough 10-1: Explore default error handling

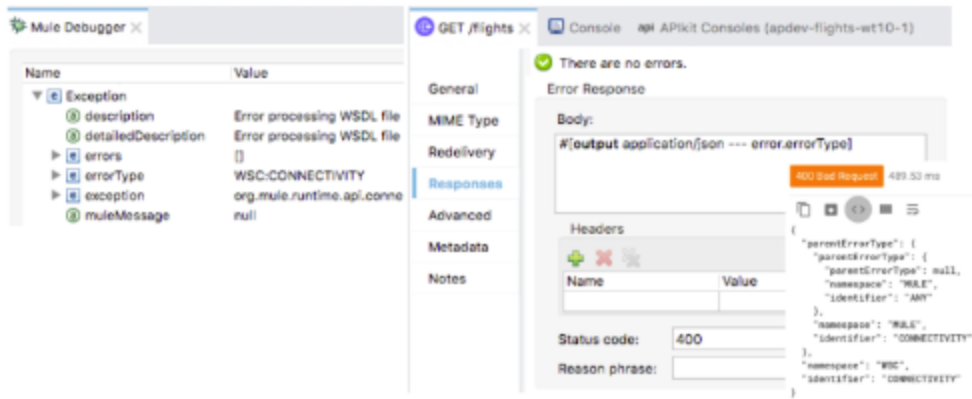
- Explore information about different types of errors in the Mule Debugger and the console
- Review the default error handling behavior
- Review and modify the error response settings for an HTTP Listener



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.

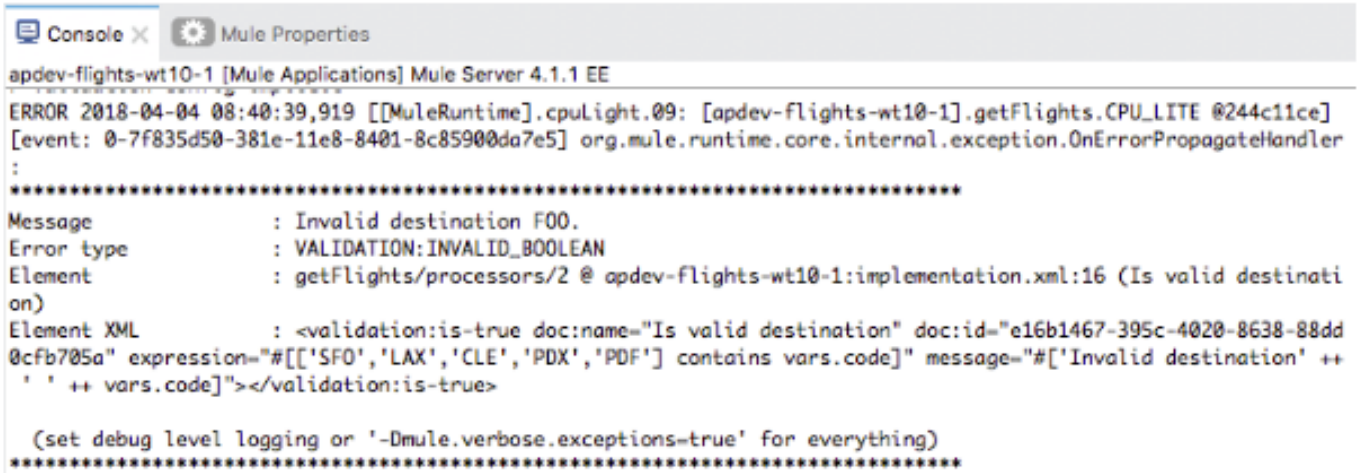
```
delta:
  wsdl: "http://mu.learn.mulesoft.com/deltas?wsdl"
  service: "TicketServiceService"
  port: "TicketServicePort"
```

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.

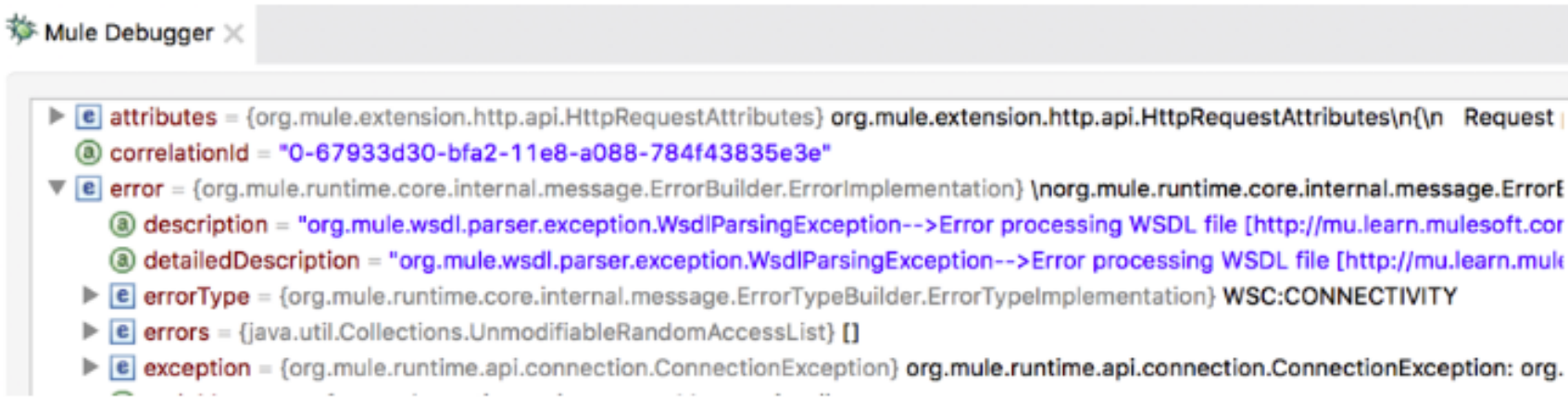
8. Step again and review the error information logged in the console.



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.

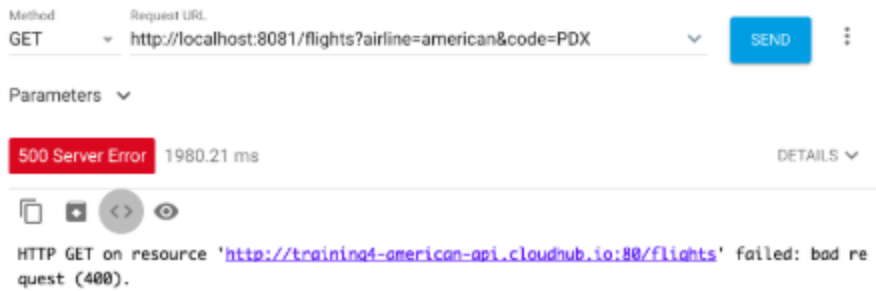


13. Review the error information logged in the console.
14. Step through the rest of the application.
15. Return to Advanced REST Client; you should see a 500 Server Error with the error description.

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.

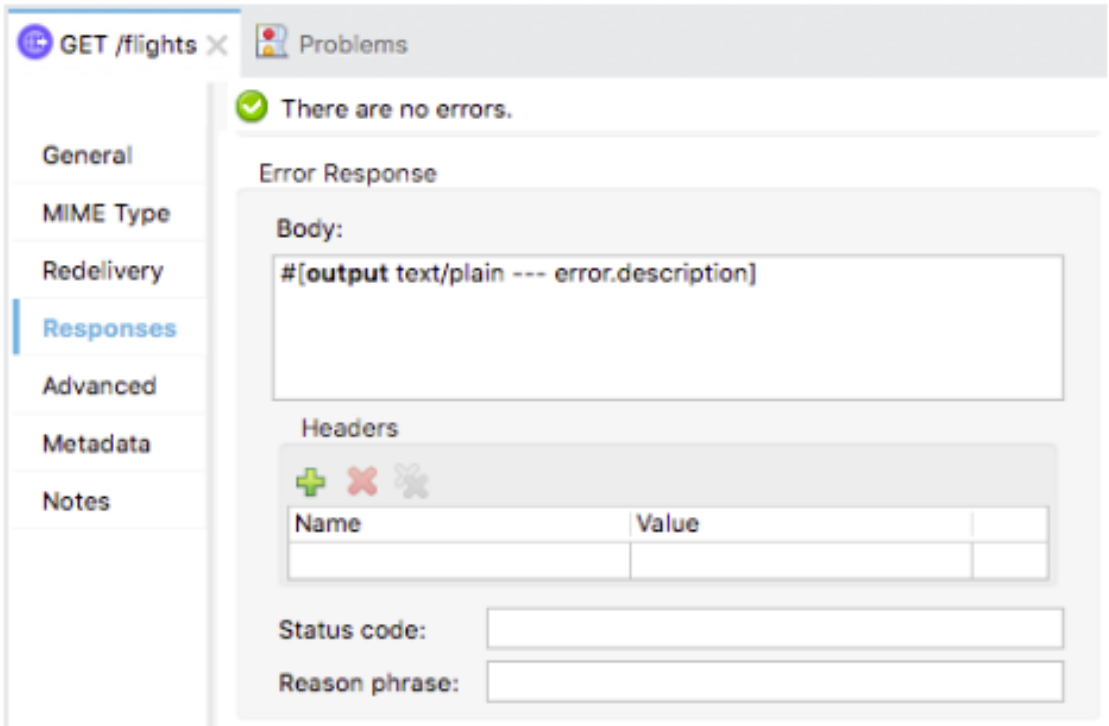
18. Step through the rest of the application.
19. Review the error information logged in the console.
20. Return to Advanced REST Client; you should see a 500 Server Error with the error description.



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

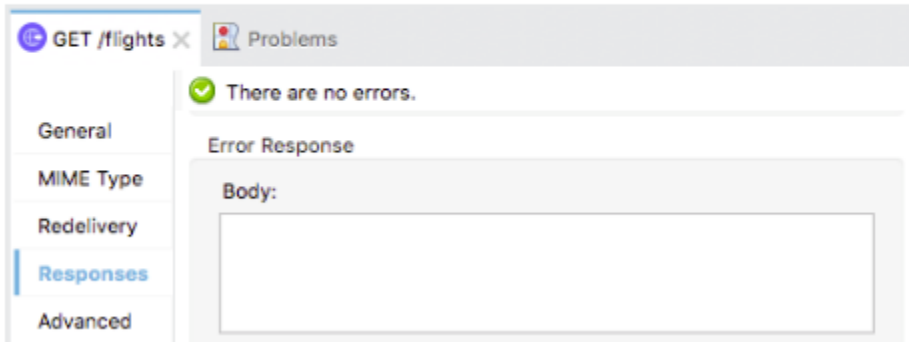
Review the default error response settings for an HTTP Listener

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.



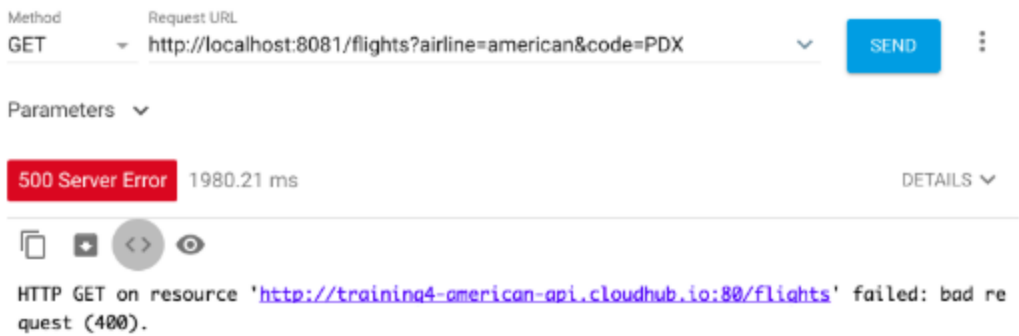
Remove the default error response settings for the HTTP Listener

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

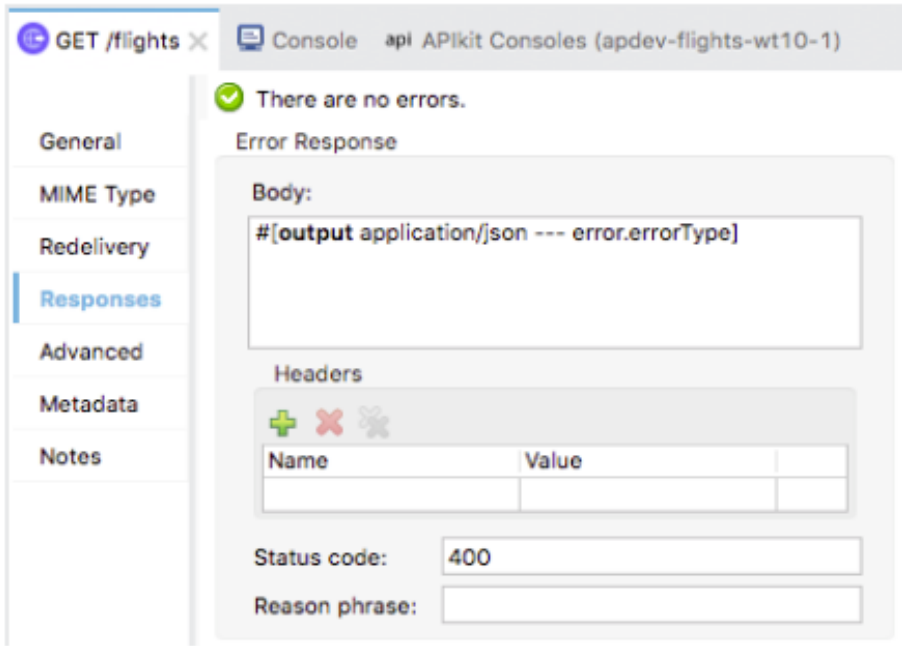
29. Return to Anypoint Studio.
30. Return to the Responses tab in the properties view for the GET /flights Listener.
31. In the error response body, paste back the original value.

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

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

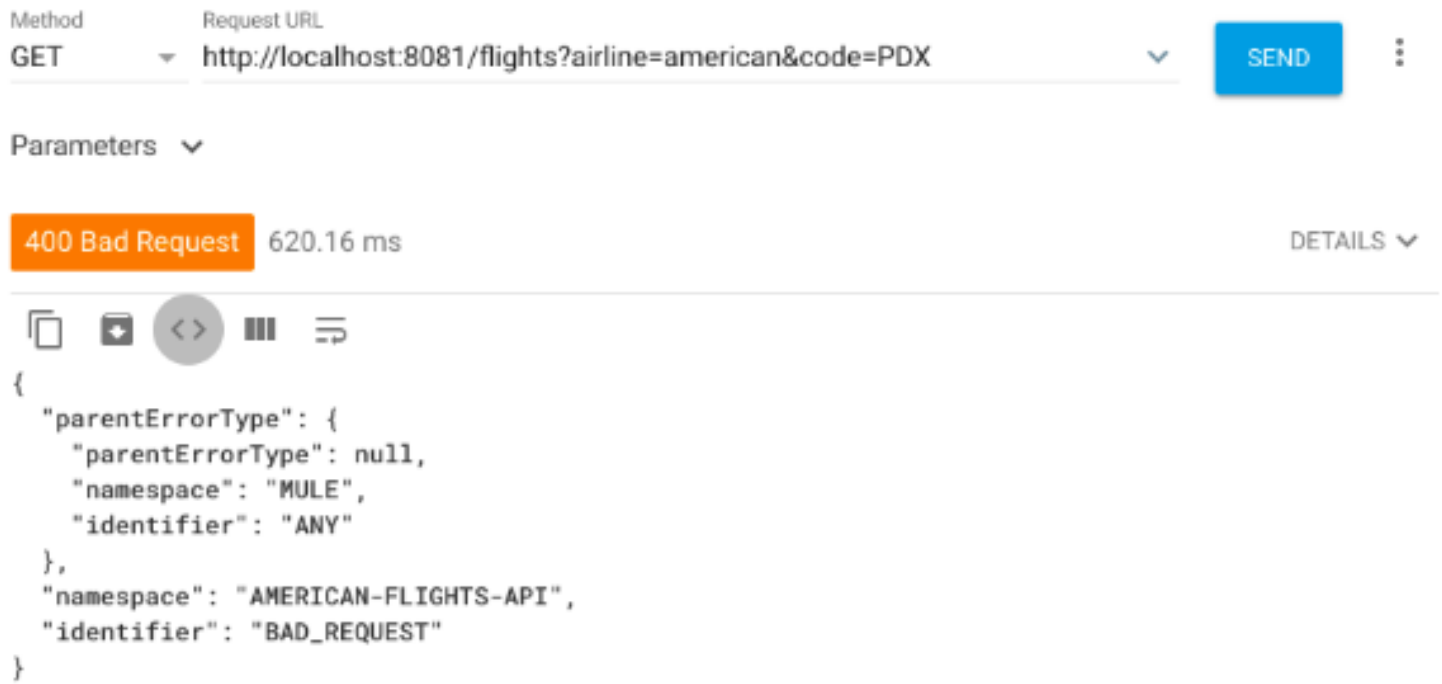
```
#[output application/json --- error.errorType]
```

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=F 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.

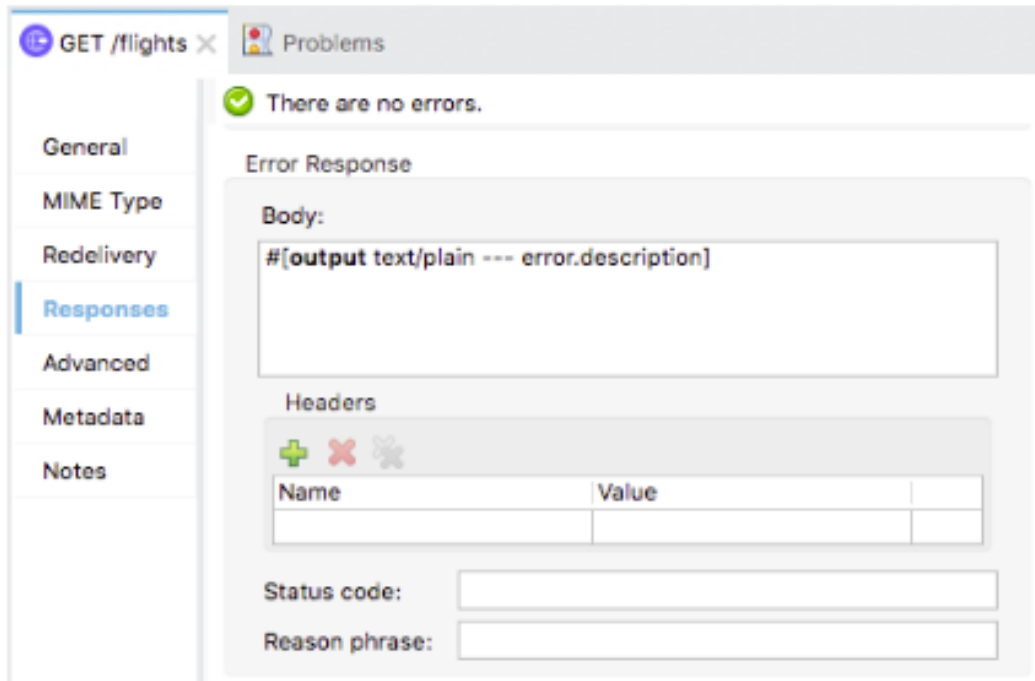
40. Look at the namespace and identifier.

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.

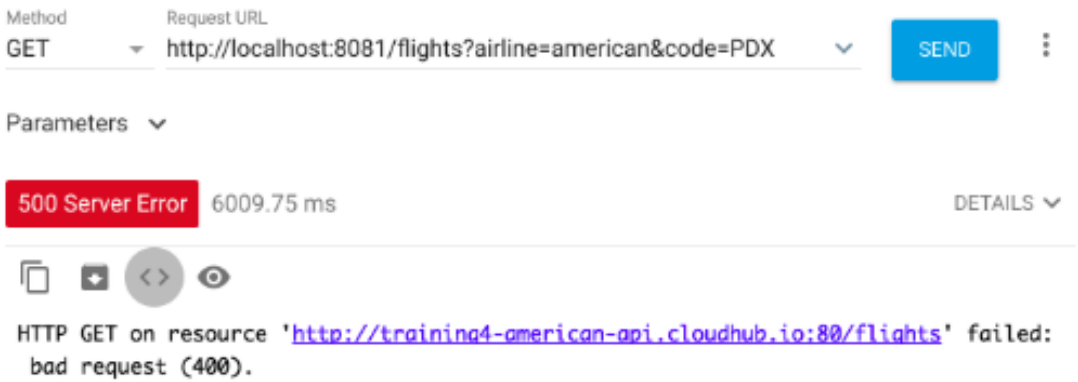
```
#[output text/plain --- error.description]
```

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.

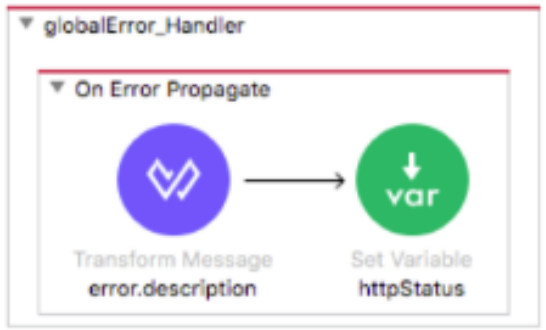


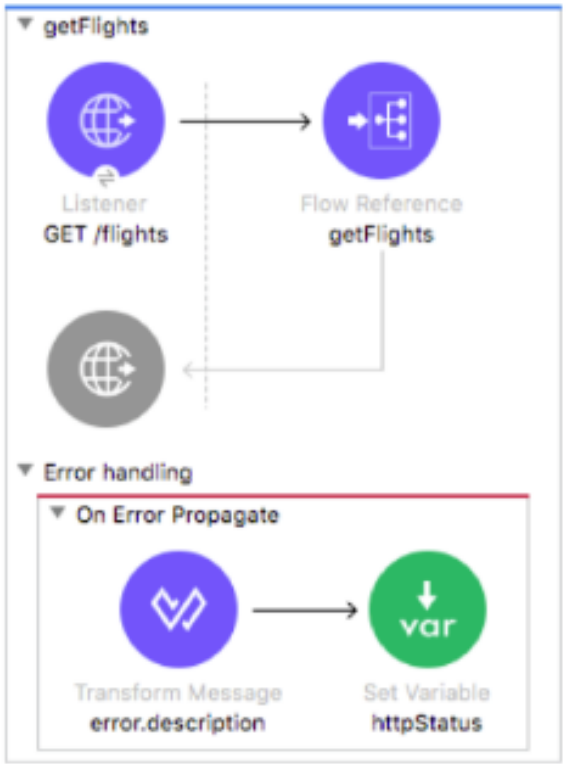
47. Return to Anypoint Studio.
48. Stop the project.

Creating error handlers

Creating error handlers

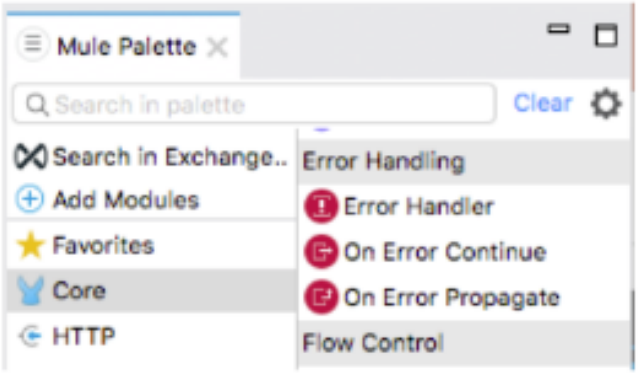
- Error handlers can be added to
  - An application (outside of any flows)
  - A flow
  - A selection of one or more event processors



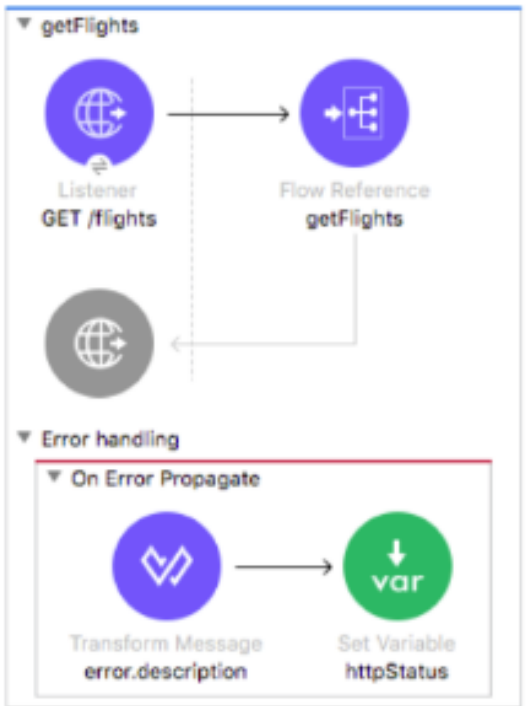


Adding error handler scopes

- Each error handler can contain one or more error handler scopes
  - **On Error Continue**
  - **On Error Propagate**



- Each error scope can contain any number of event processors

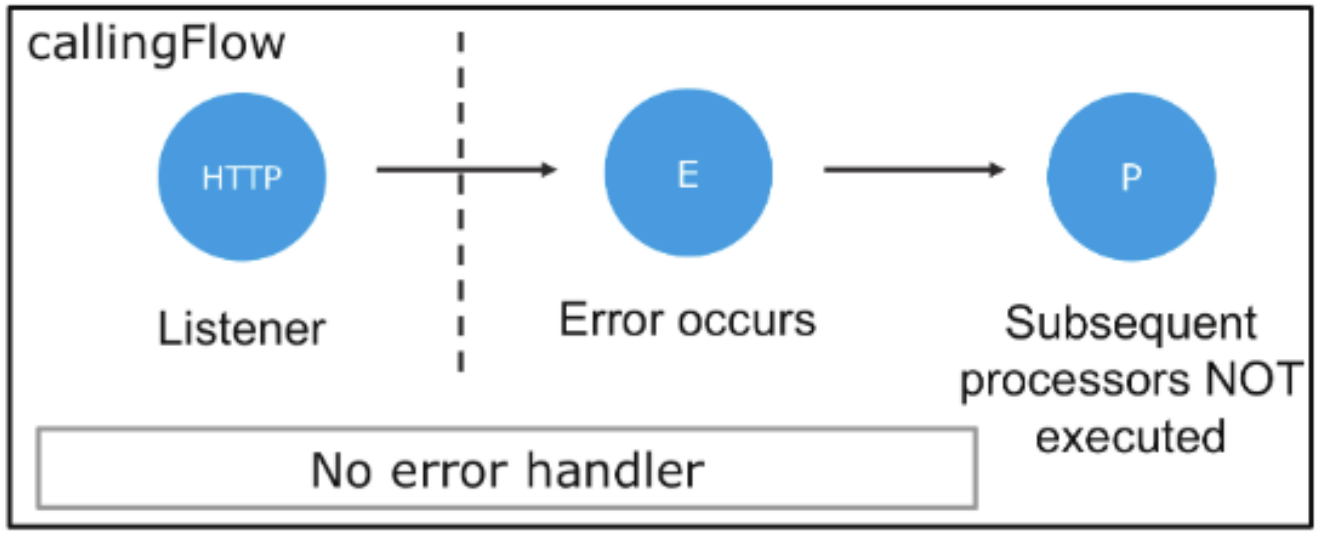


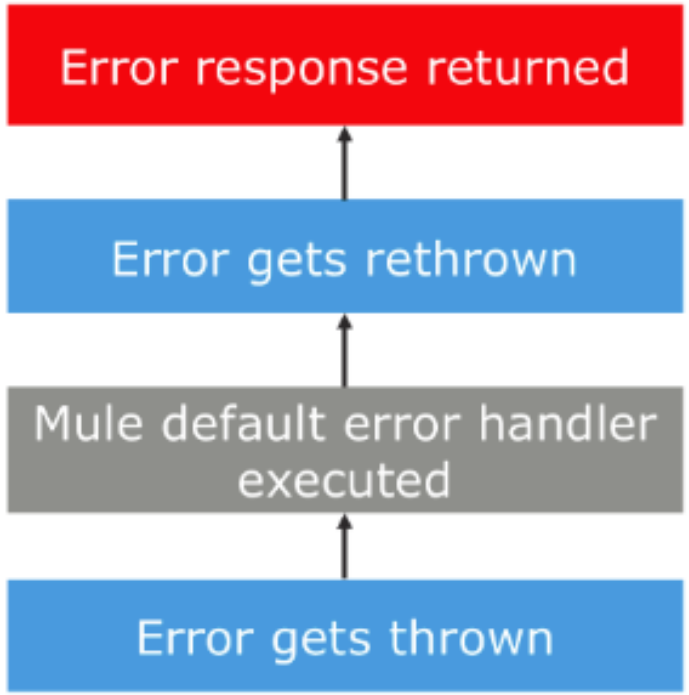
Two types of error handling scopes

- **On Error Propagate**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the flow that threw the error is not executed
    - *The error is rethrown up to the next level and handled there*
  - An HTTP Listener returns an ***error*** response

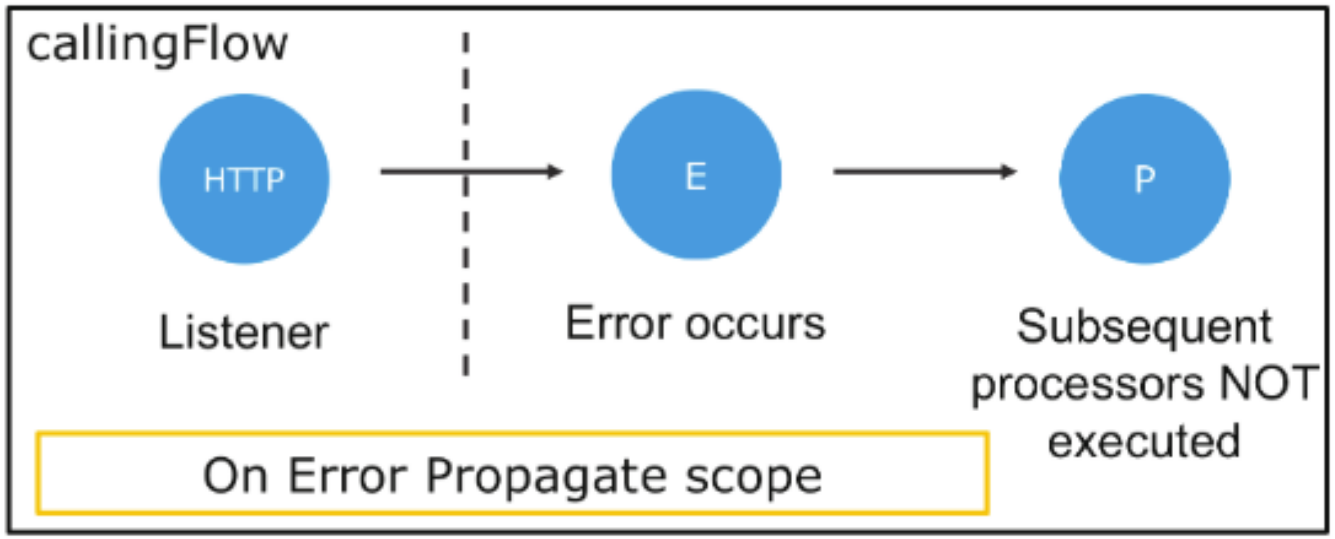
- **On Error Continue**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the flow that threw the error is not executed
    - *The event is passed up to the next level as if the flow execution had completed successfully*
  - An HTTP Listener returns a ***successful*** response

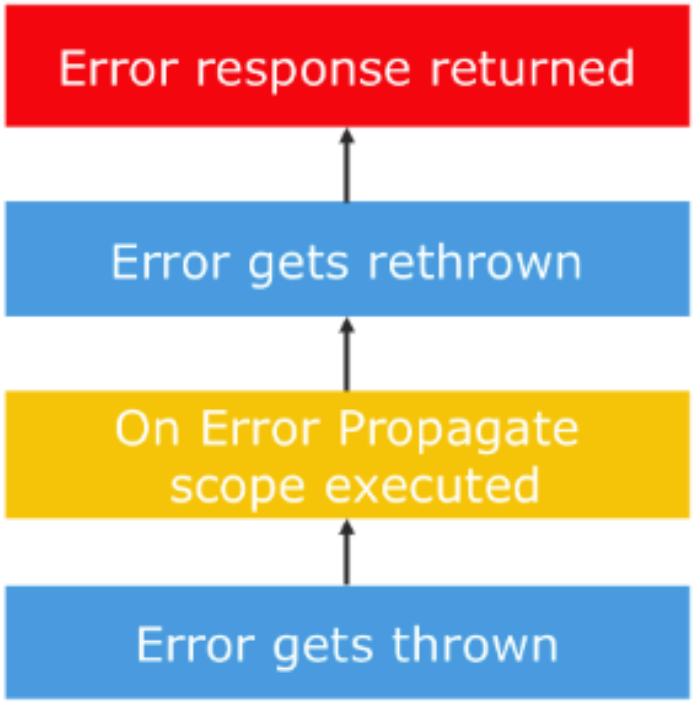
Error handling scenario 1



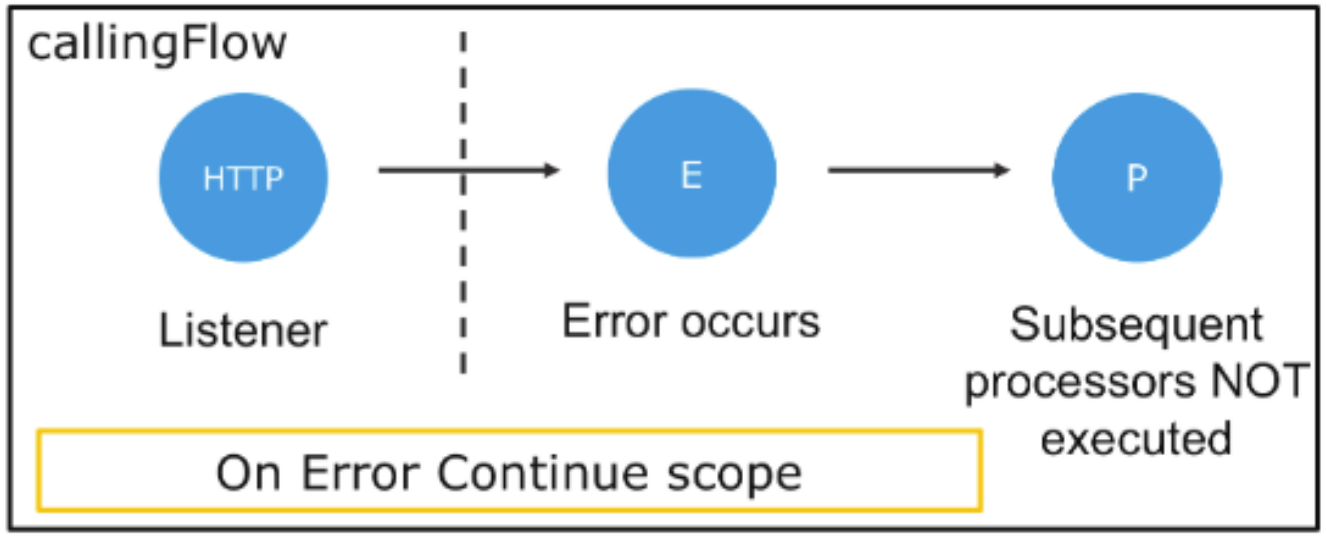


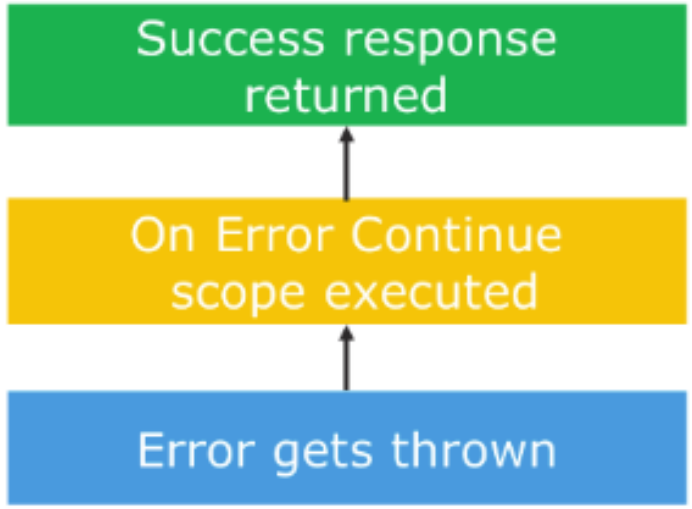
Error handling scenario 2



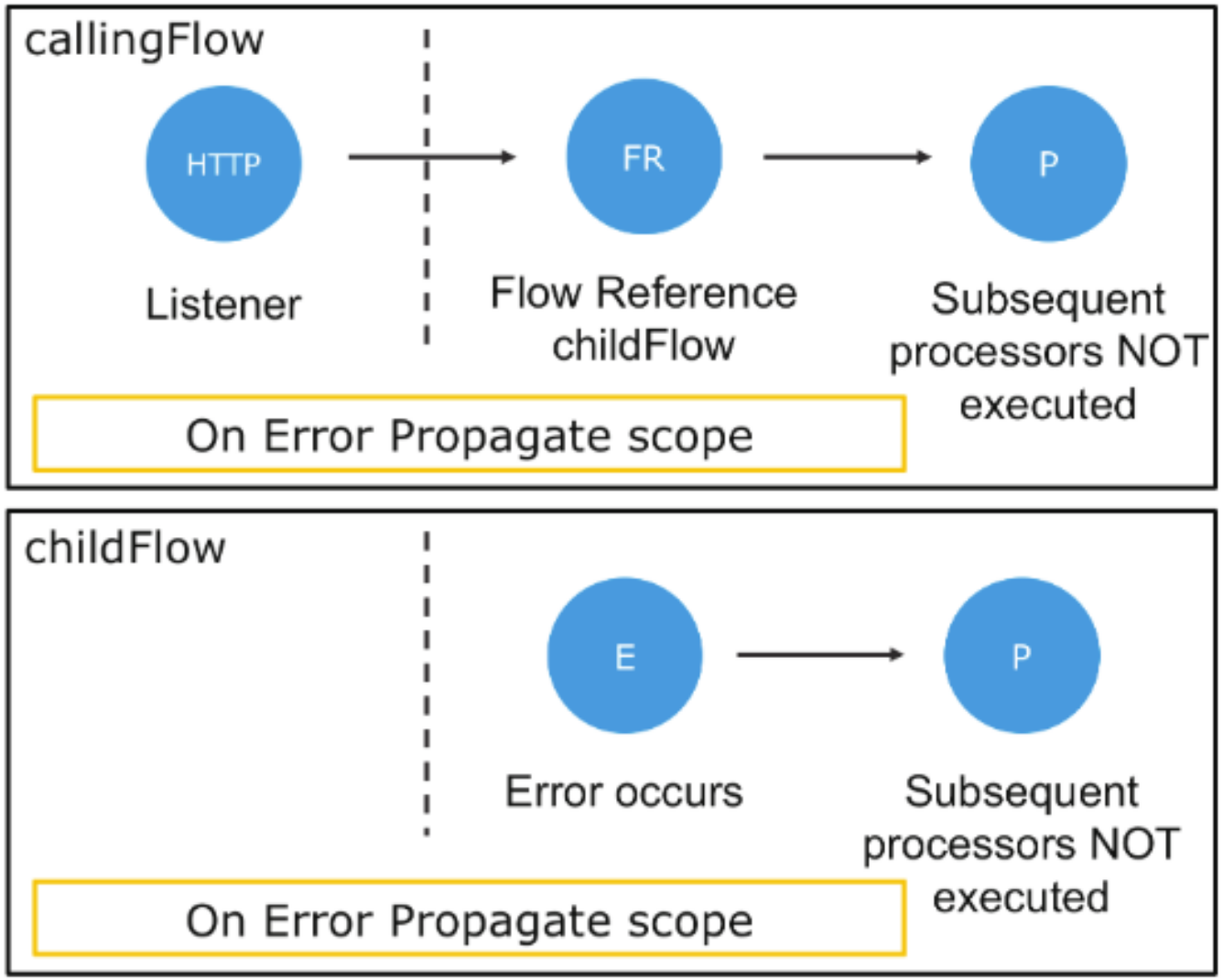


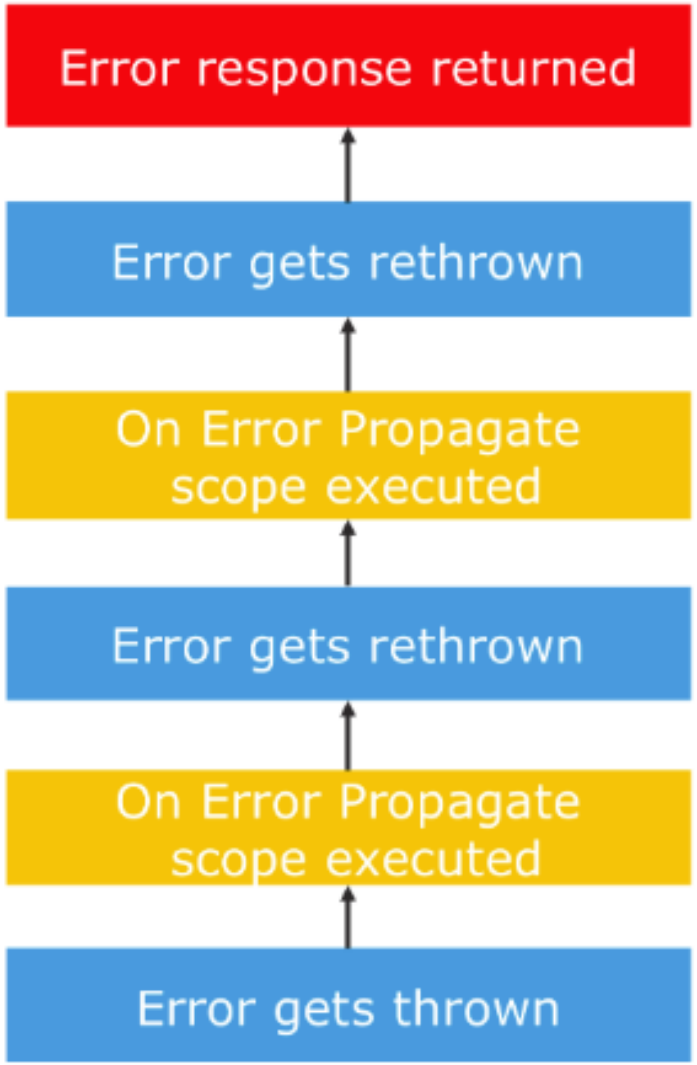
Error handling scenario 3



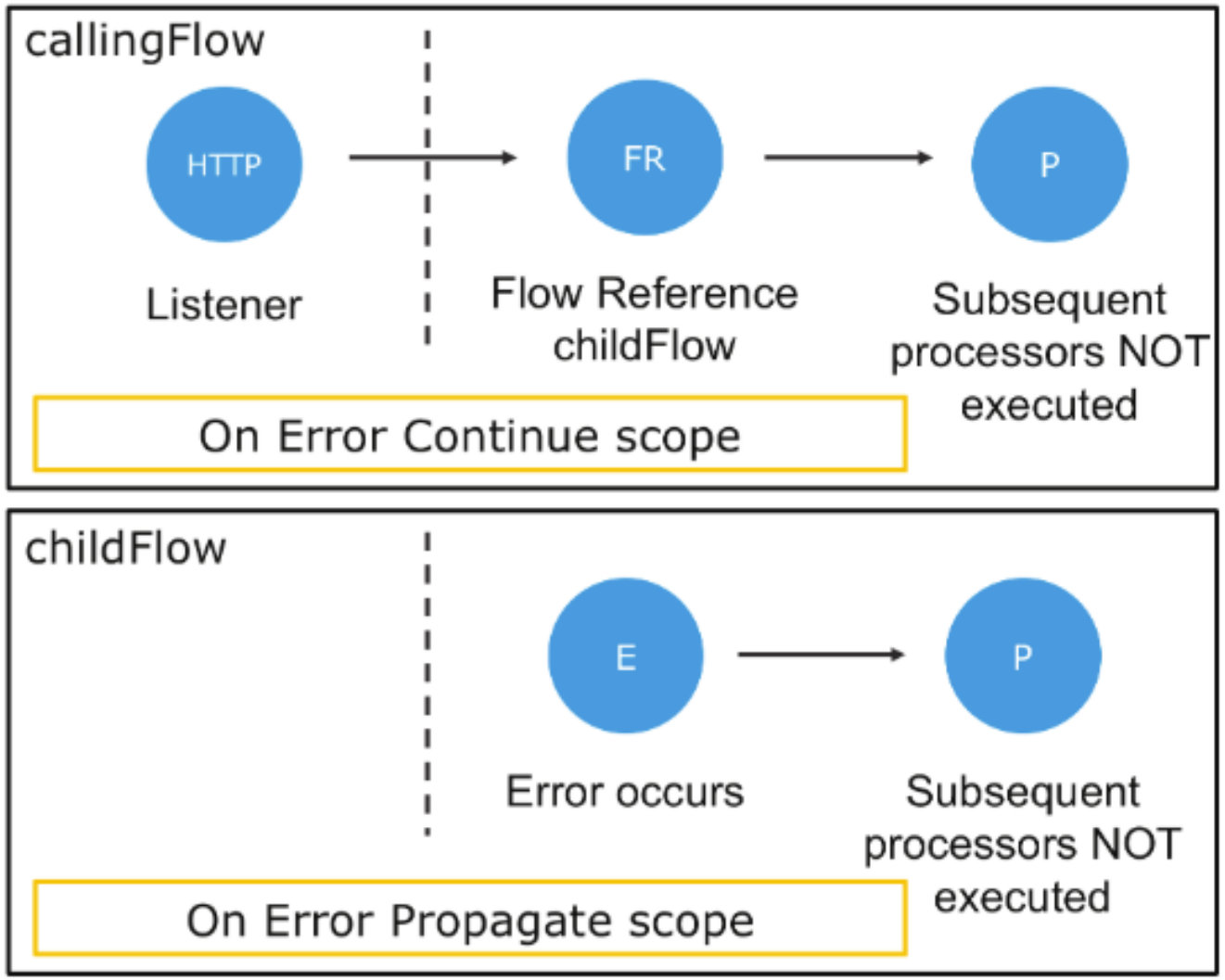


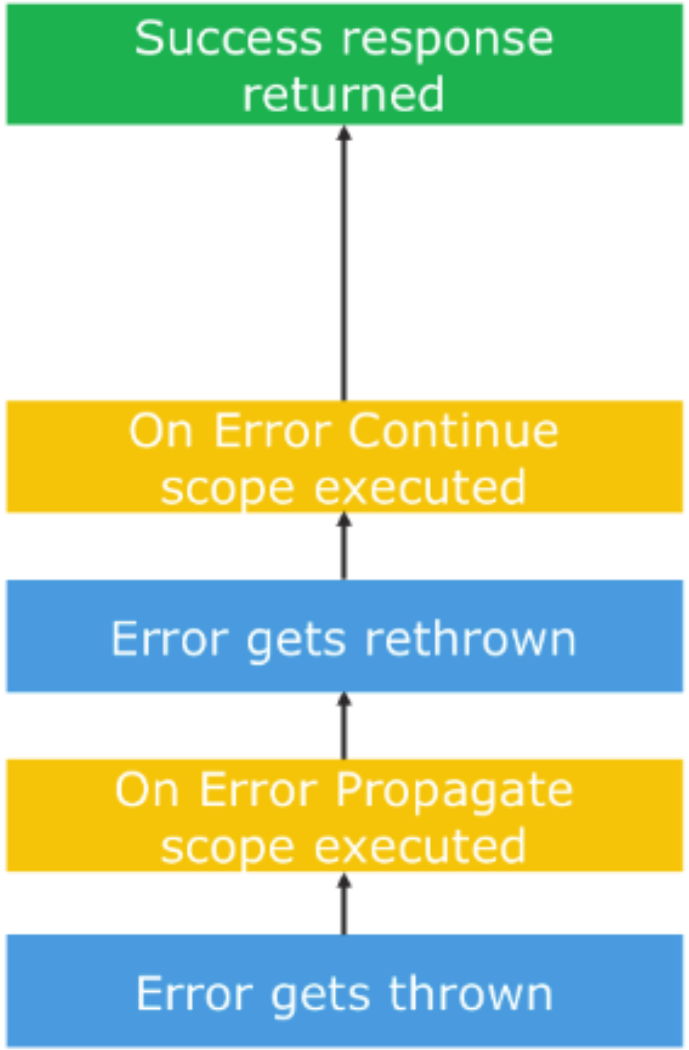
Error handling scenario 4



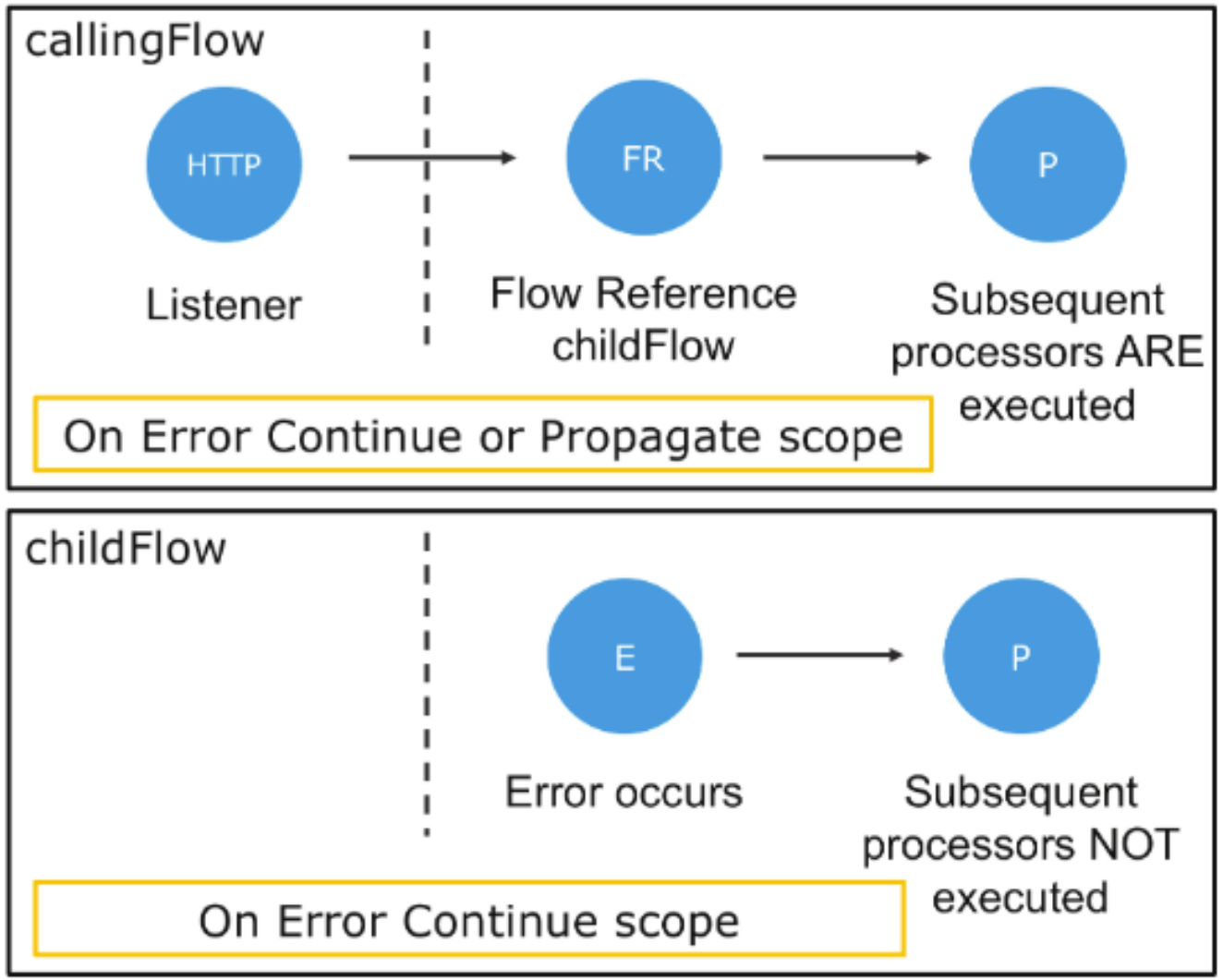


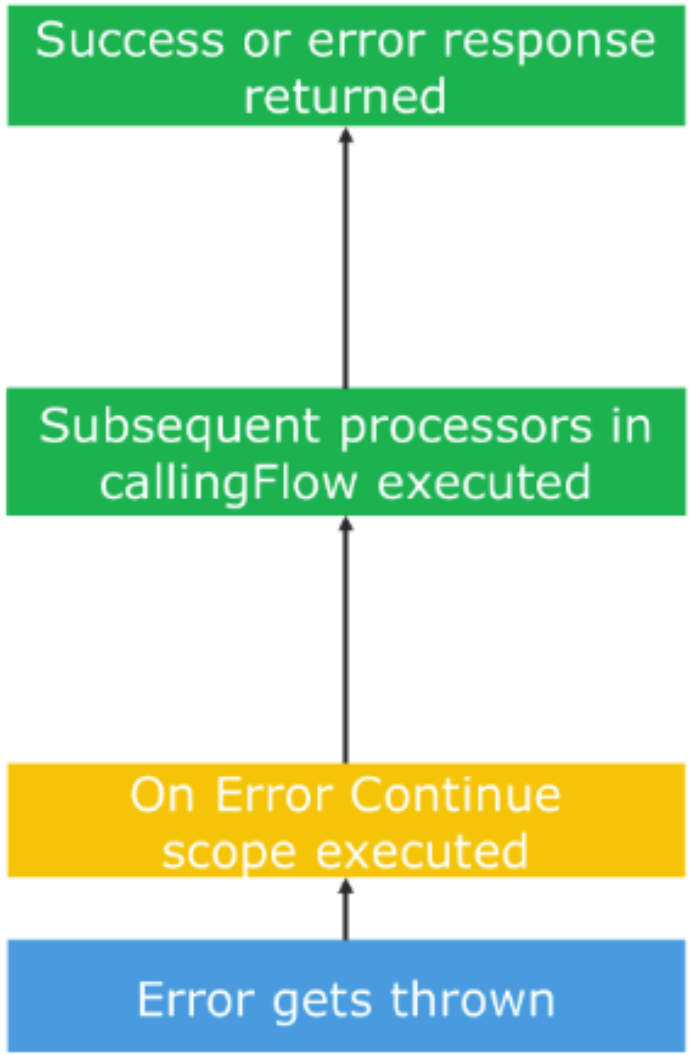
Error handling scenario 5





Error handling scenario 6

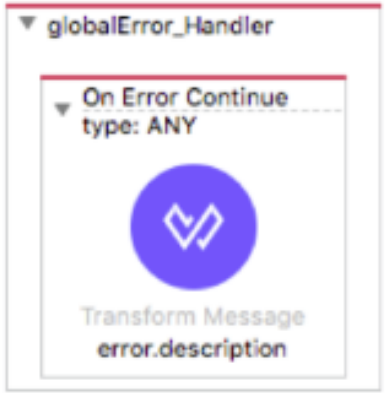




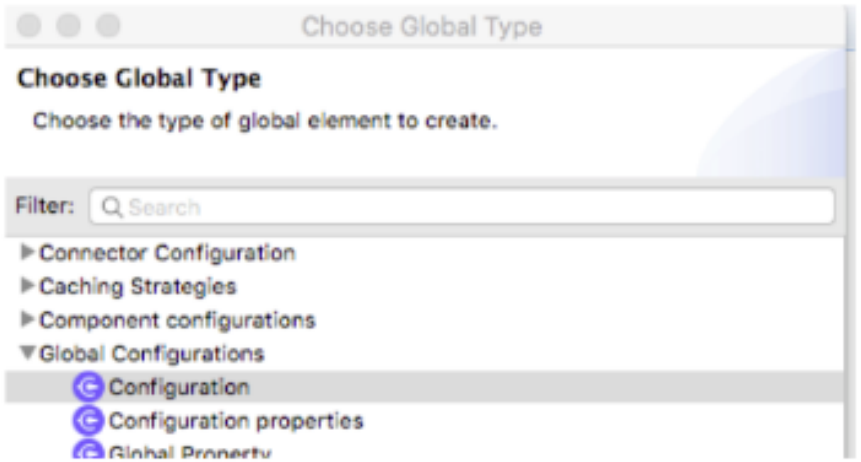
Handling errors at the application level

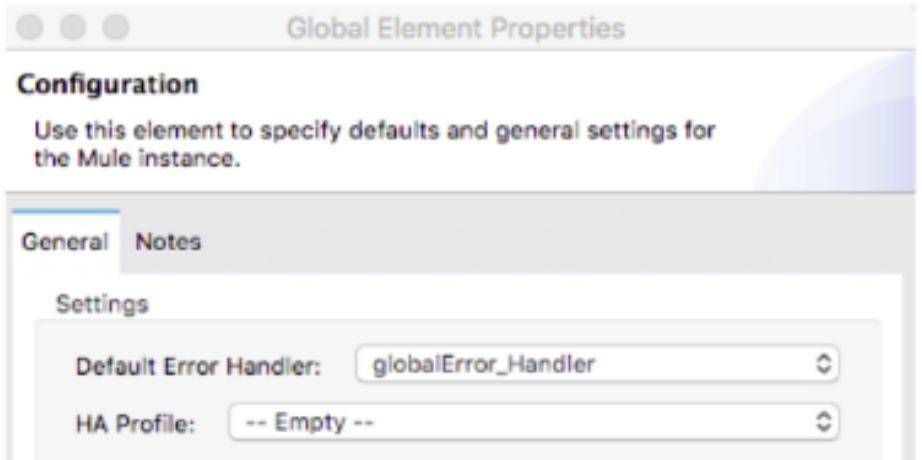
Defining a default error handler for an application

- Add an error handler outside a flow
  - Typically, put it in the global configuration file



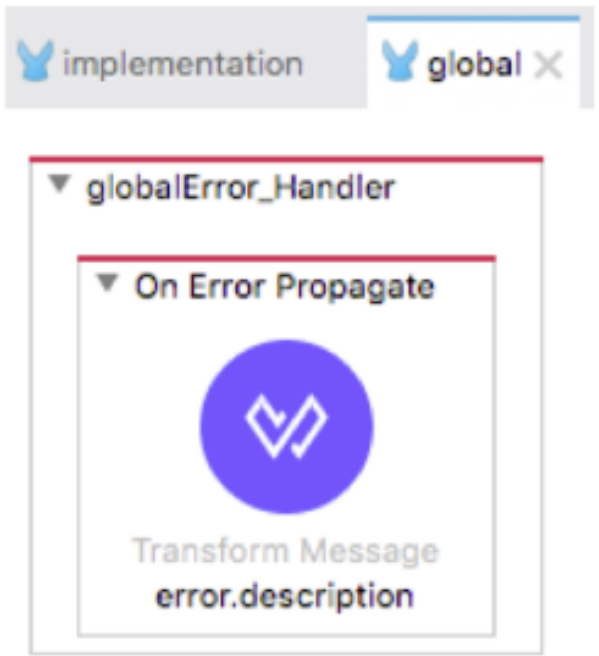
- Specify this handler to be the application's default error handler





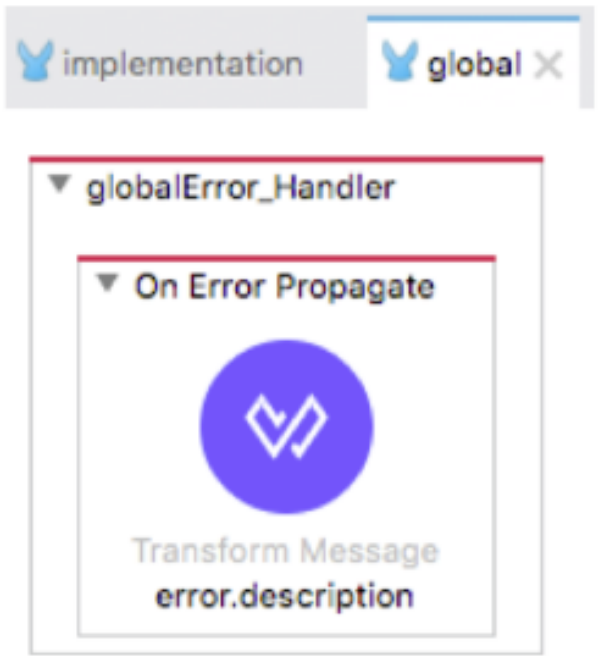
Walkthrough 10-2: Handle errors at the application level

- 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



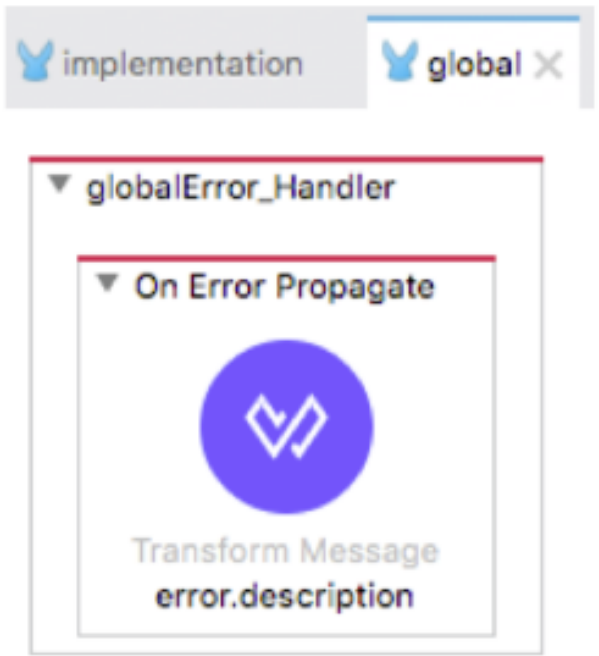
Walkthrough 10-2: Handle errors at the application level

- 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



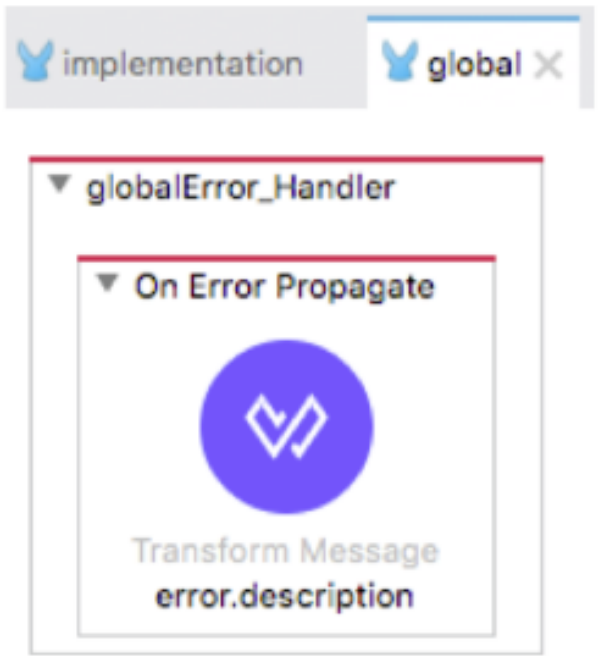
Walkthrough 10-2: Handle errors at the application level

- 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



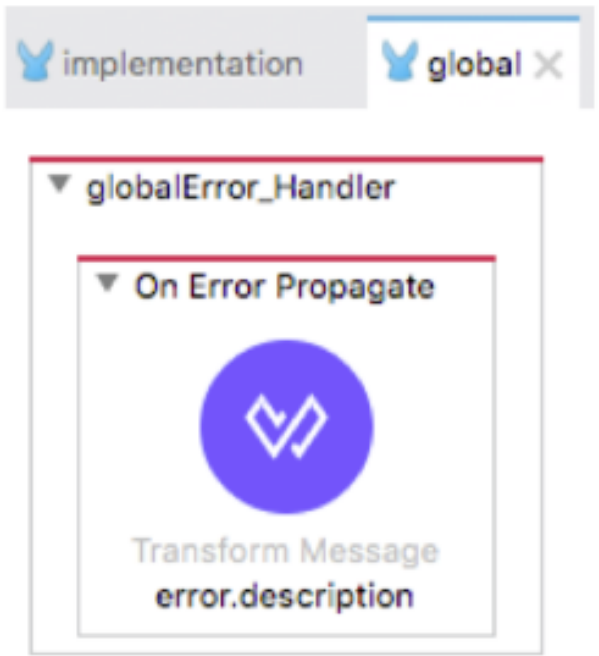
Walkthrough 10-2: Handle errors at the application level

- 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



Walkthrough 10-2: Handle errors at the application level

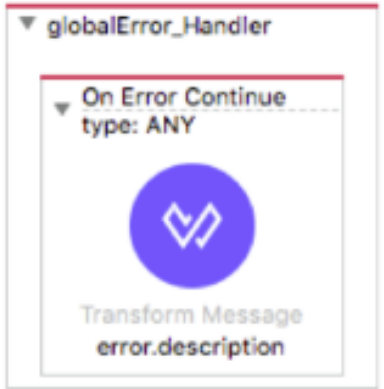
- Create a global error handler in an application
- Configure an application to use a global default error handler
- Explore the differences between the On Error Continue and On Error Propagate scopes
- Modify the default error response settings for an HTTP Listener



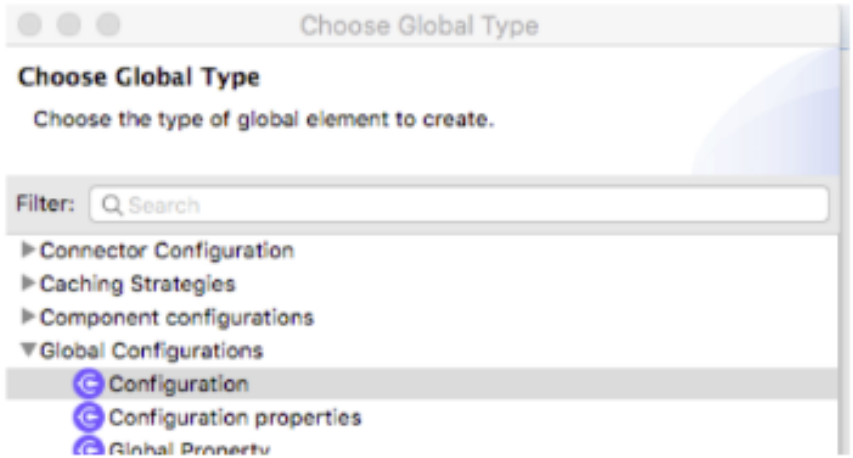
Handling errors at the application level

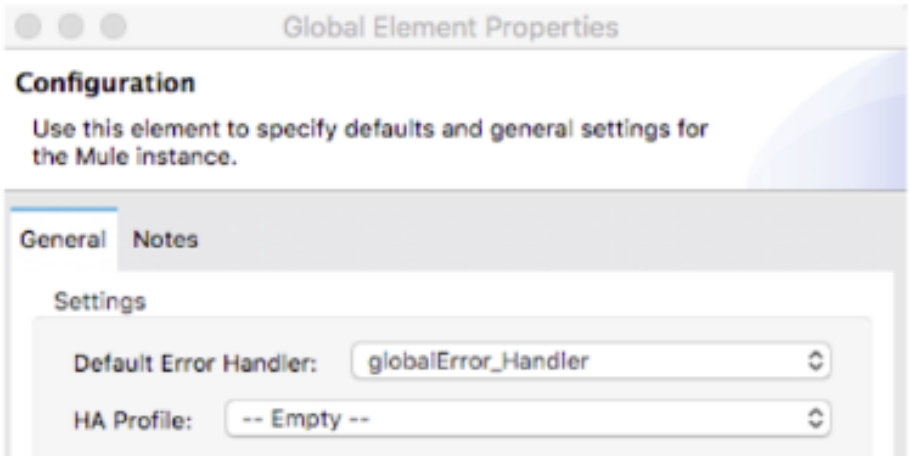
Defining a default error handler for an application

- Add an error handler outside a flow
  - Typically, put it in the global configuration file



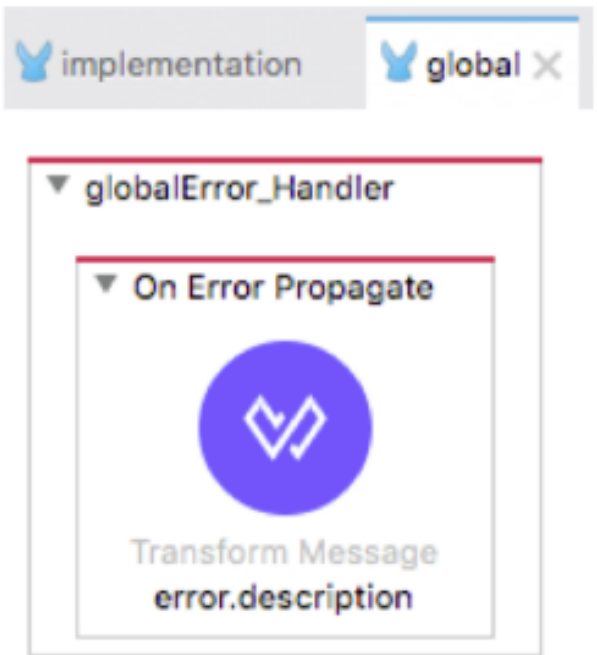
- Specify this handler to be the application's default error handler





Walkthrough 10-2: Handle errors at the application level

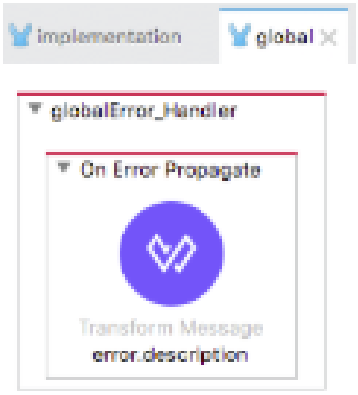
- Create a global error handler in an application
- Configure an application to use a global default error handler
- Explore the differences between the On Error Continue and On Error Propagate scopes
- Modify the default error response settings for an HTTP Listener



Walkthrough 10-2: Handle errors at the application level

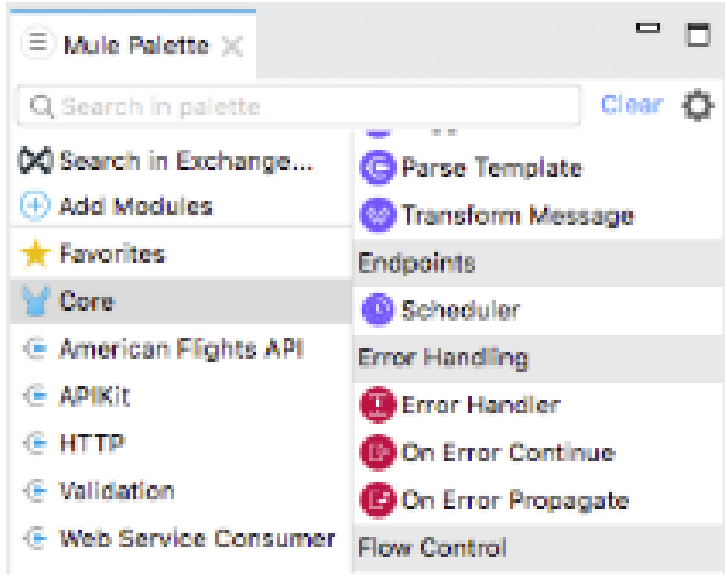
In this walkthrough, you create a default error handler for the application. You will:

- 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

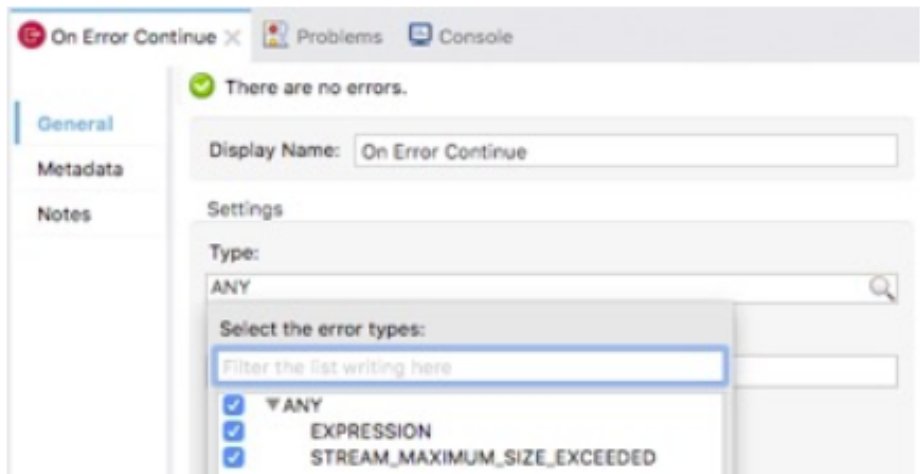
1. Return to global.xml and switch to the Message Flow view.
2. In the Core section of the Mule Palette, locate the Error Handling elements.



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.

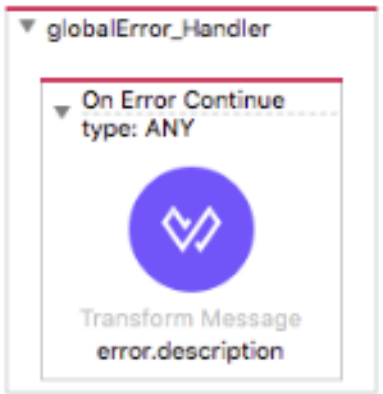
5. In the On Error Continue properties view, click the Search button for type.
6. In the drop-down menu that appears, select ANY.



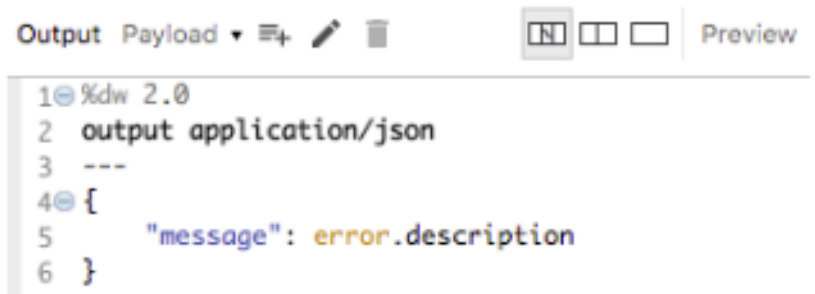
Note: If the drop-down menu does not populate, type ANY into the field.

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.

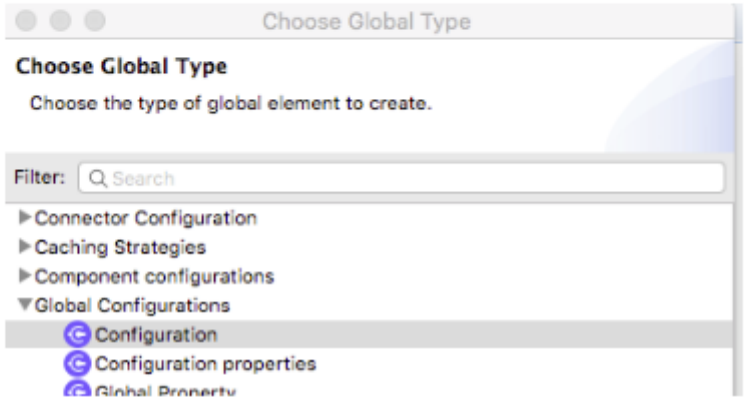


9. In the Transform Message properties view, change the output type to application/json.
10. Add a message property to the output JSON and give it a value of the error.description.

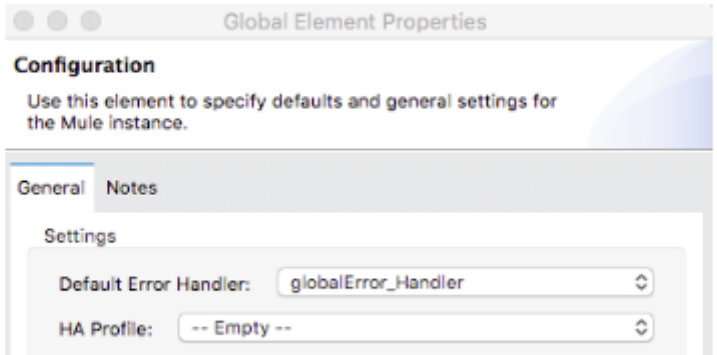


Set a default error handler for the application

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.



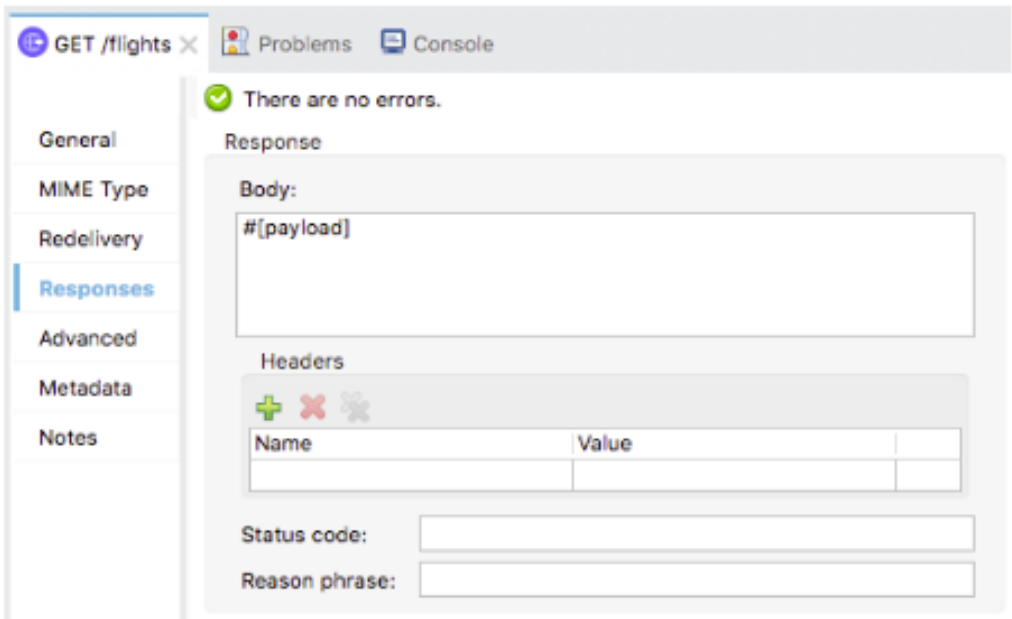
14. In the Global Element Properties dialog box, set the default error handler to globalError_Handler and click OK.



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.

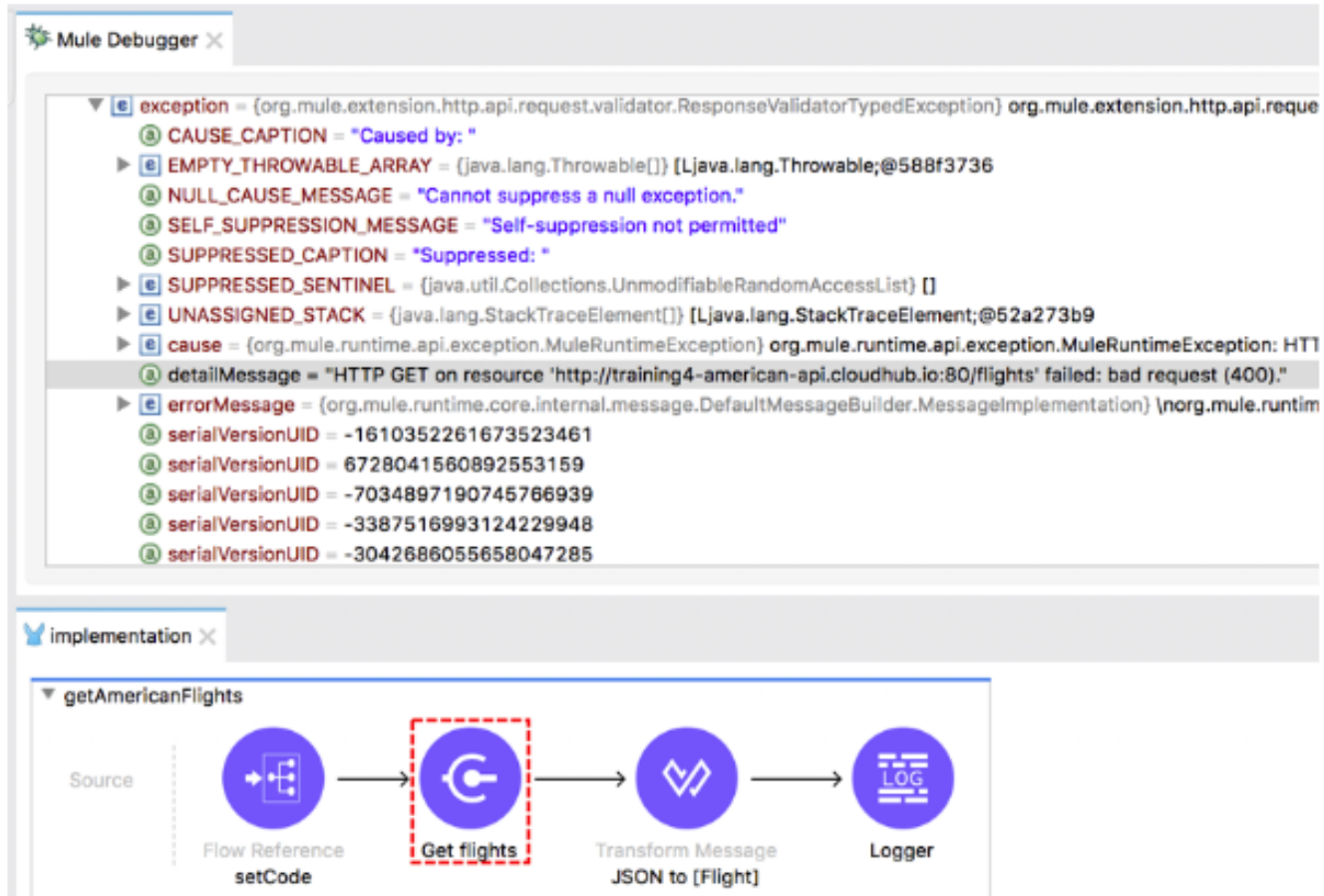
Test the On Error Continue behavior

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.

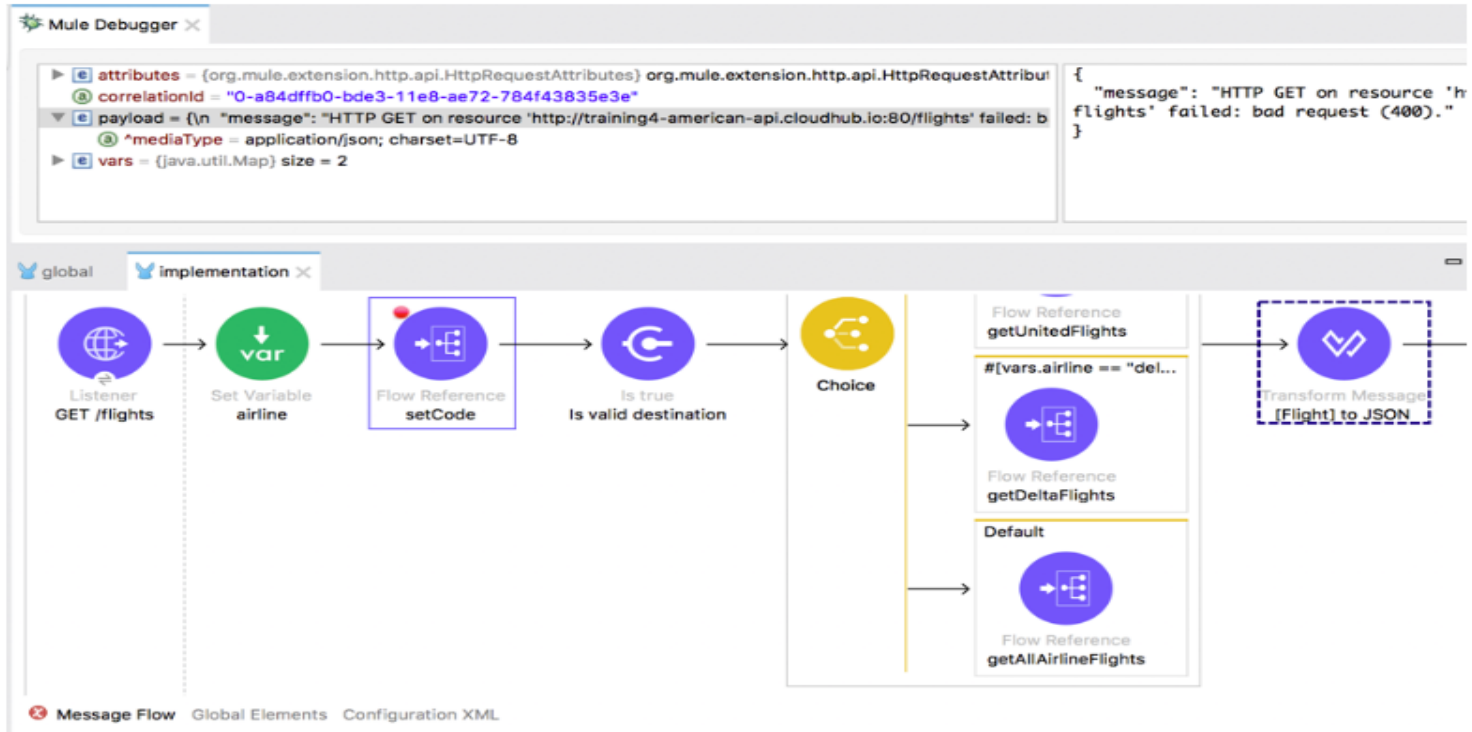
    Note: In Anypoint Studio 7.1.X, focus will switch to global.xml but you will not see the application execution step through an individual error handler.

24. Step again; the event should be passed back to getFlights, which continues to execute after the error occurs.

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



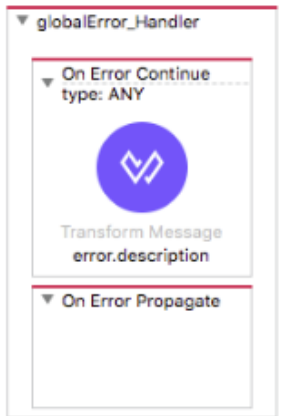
26. In the Mule Debugger, look at the payload and the absence of an exception object in the Mule Debugger.



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

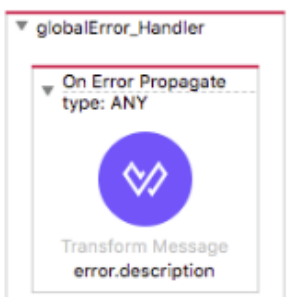
29. Return to Anypoint Studio and switch to the Mule Design perspective.
30. Return to global.xml.
31. Drag an On Error Propagate element from the Mule Palette and drop it to the right or left of the On Error Continue to add it to globalError_Handler.



32. In the On Error Propagate properties view, set the type to ANY.

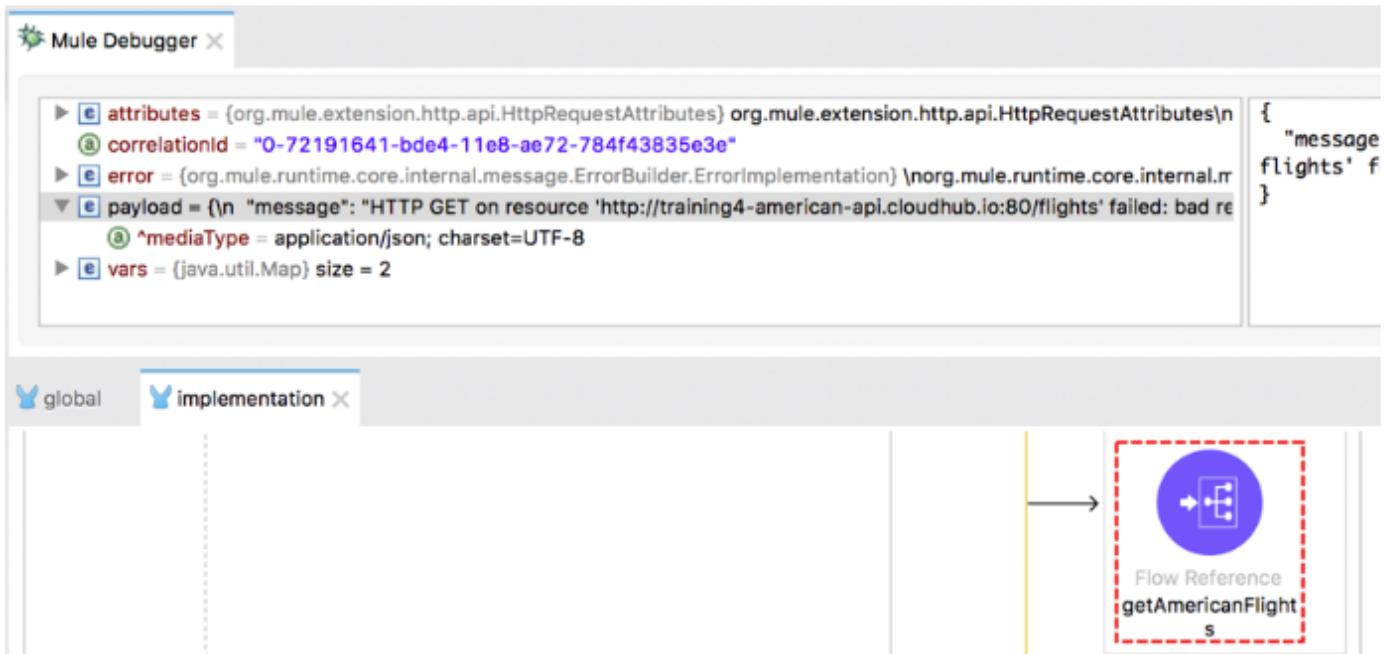
    Note: If the drop-down menu does not populate, type ANY into the field.

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.

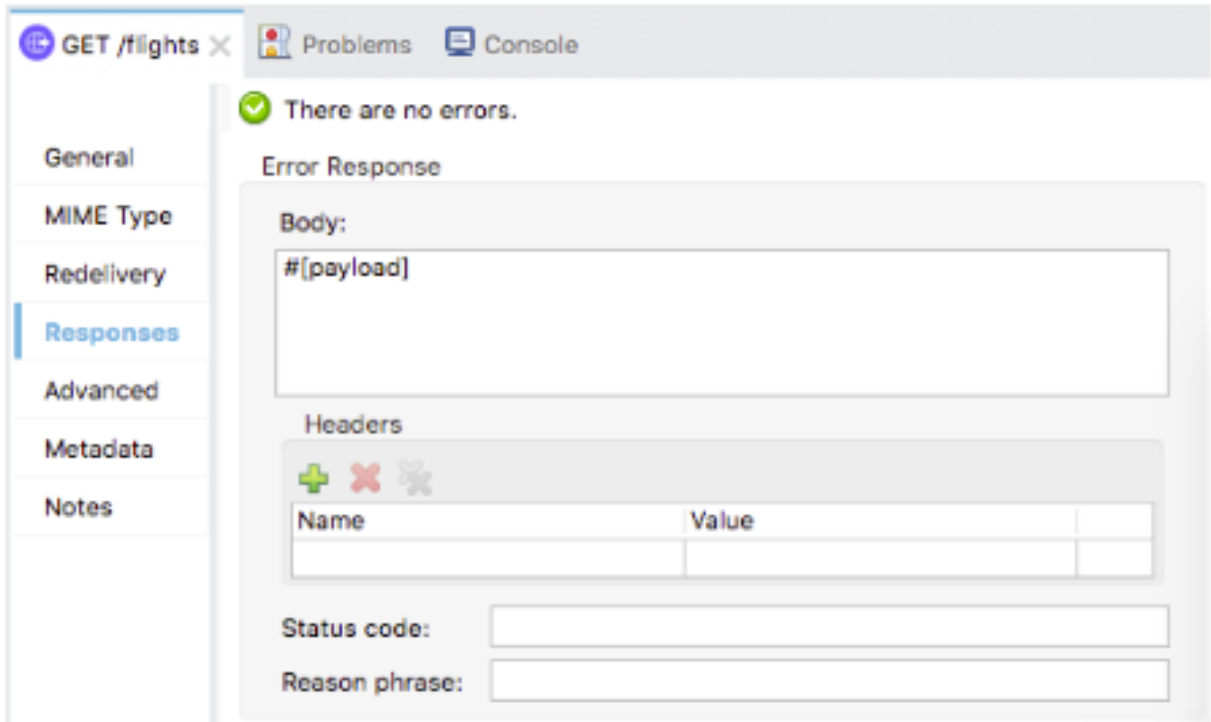
43. Step through the rest of the application.
44. Return to Advanced REST Client; you should see get a 500 Server Error response with the response body equal to the plain text error description – not the payload.

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]
```

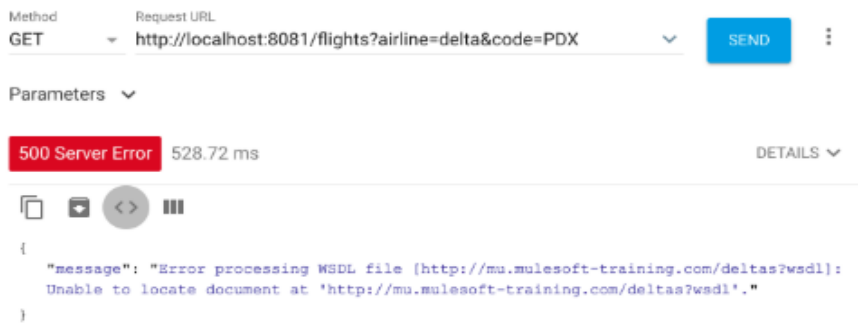


Test the application

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.



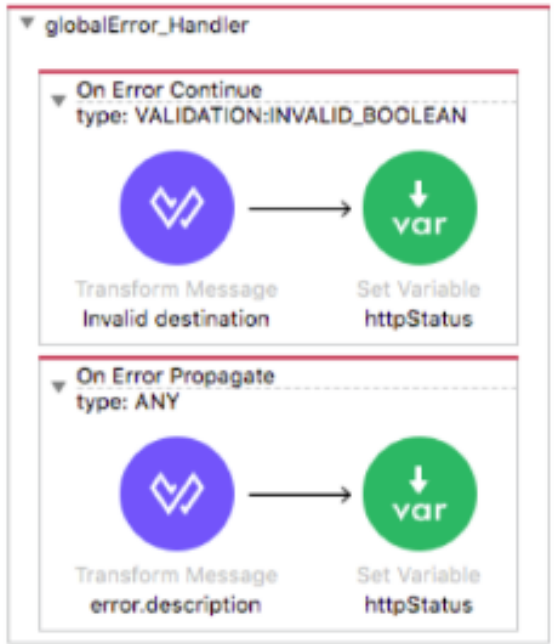
53. Return to Anypoint Studio.

Handling specific types of errors

Adding multiple error handler scopes

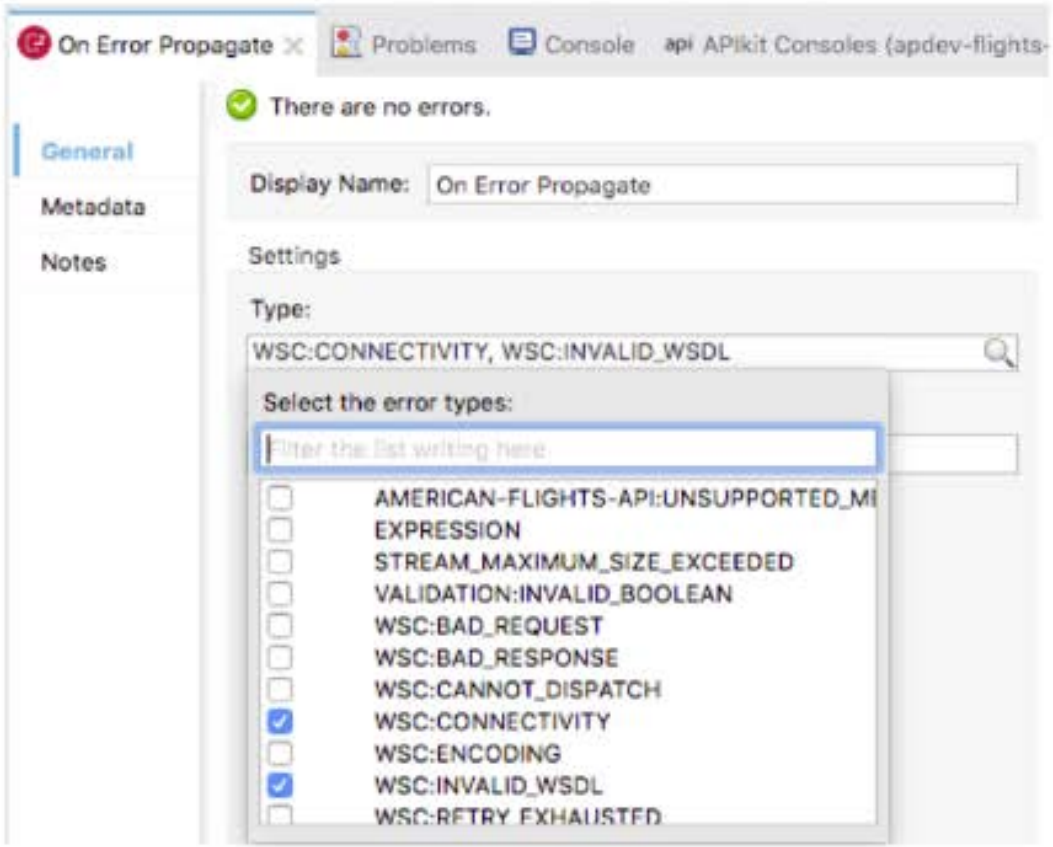
- Each error handler can contain one or more error handler scopes
  - Any number of On Error Continue and/or On Error Propagate

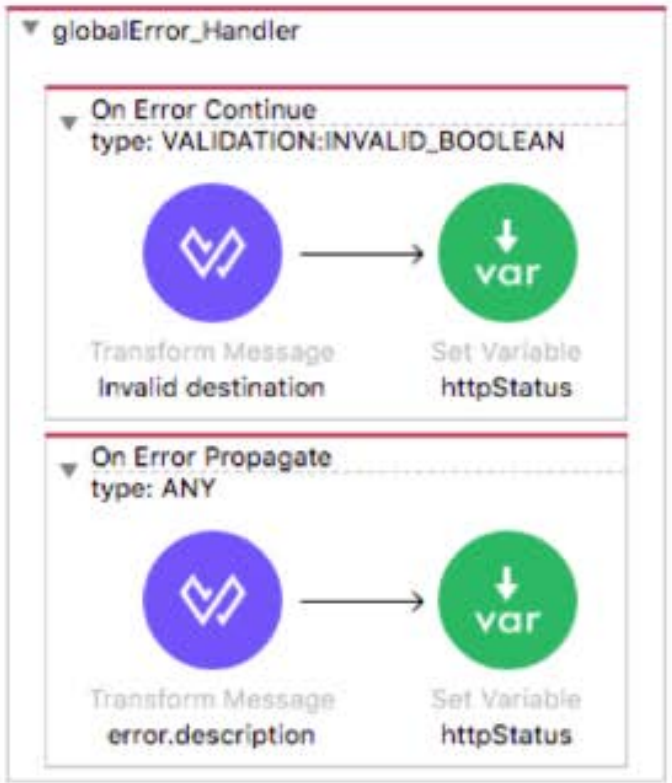
- Each error handler scope specifies when it should be executed
  - The error is handled by the *first* error scope whose condition evaluates to true



Specifying scope execution for specific error types

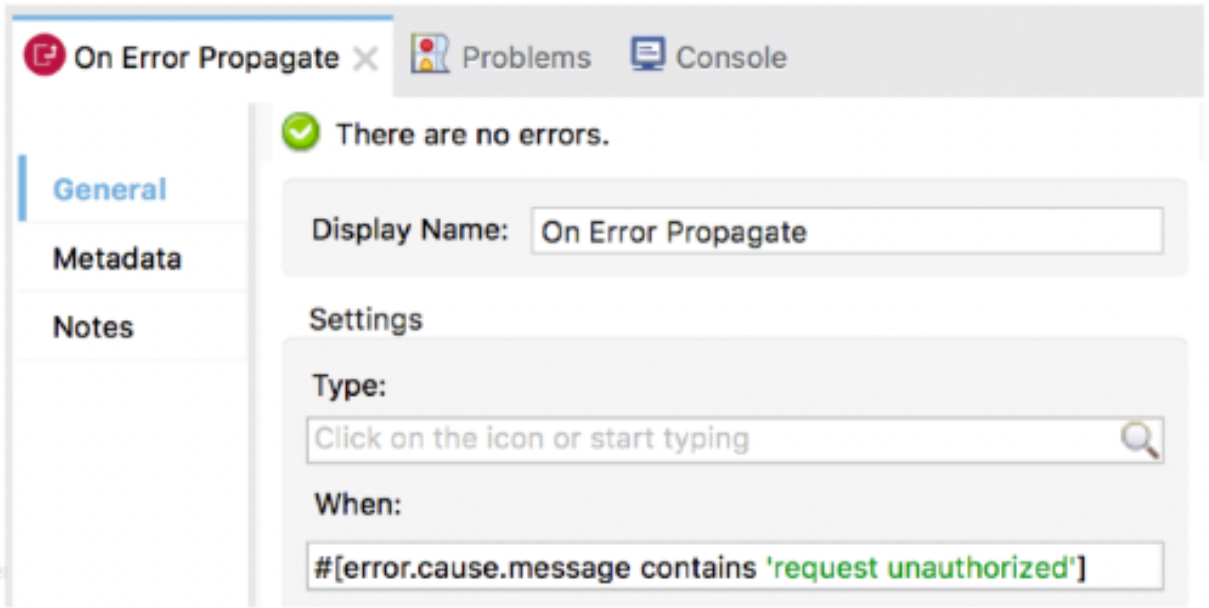
- Set the **type** to ANY (the default) or one or more types or errors





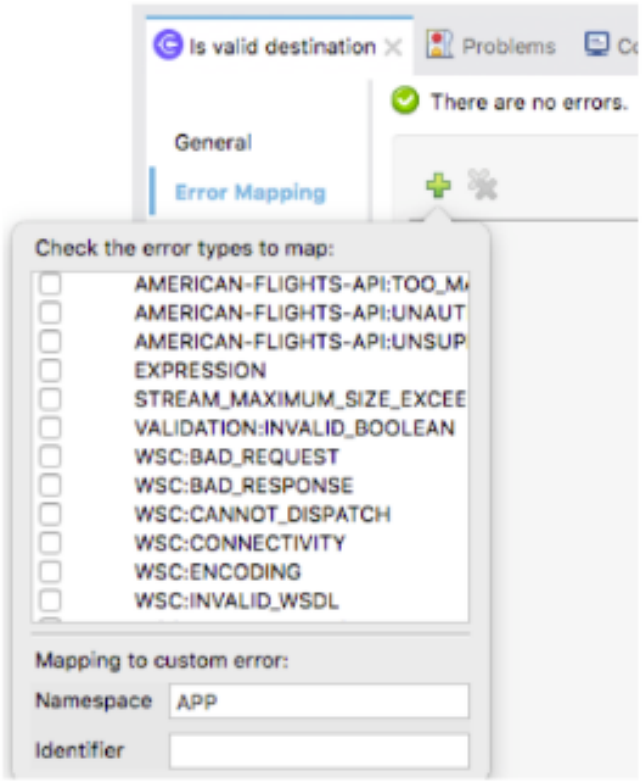
Specifying scope execution upon a specific condition

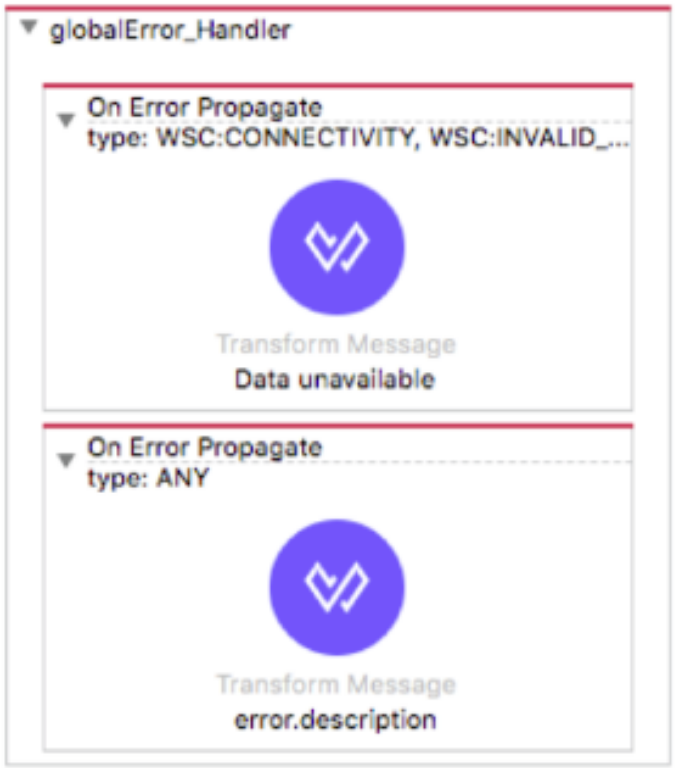
- Set the **when** condition to a Boolean DataWeave expression
  - HTTP:UNAUTHORIZED
  - error.errorType.namespace == 'HTTP'
  - error.errorType.identifier == 'UNAUTHORIZED'
  - error.cause.message contains 'request unauthorized'
  - error.cause.class contains 'http'



Walkthrough 10-3: Handle specific types of errors

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

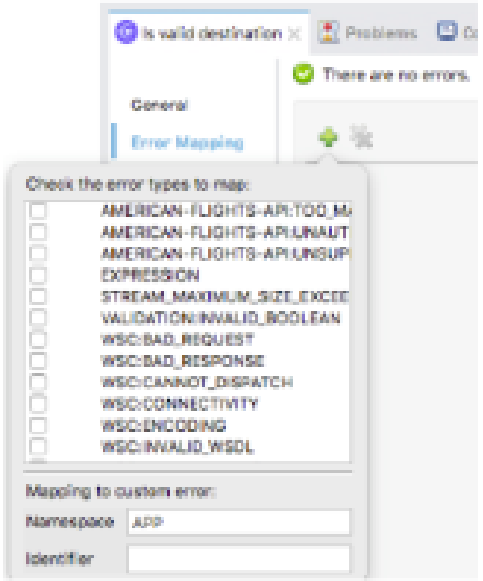


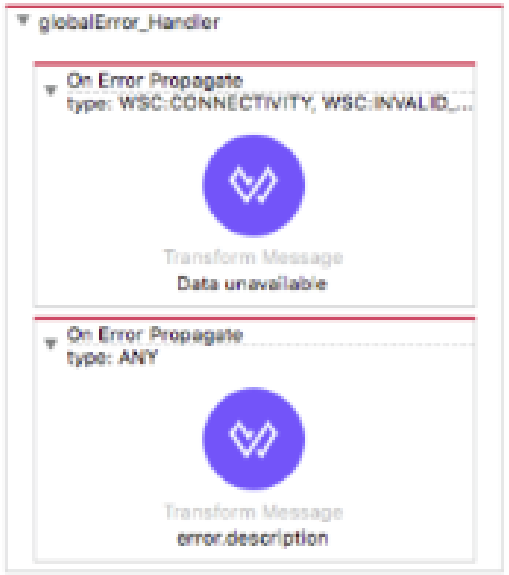


Walkthrough 10-3: Handle specific types of errors

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.



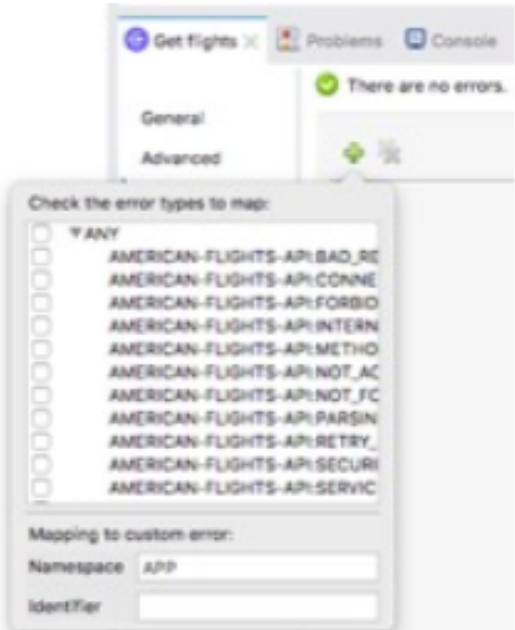


Review the possible types of errors thrown by a Web Service Consumer operation

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.

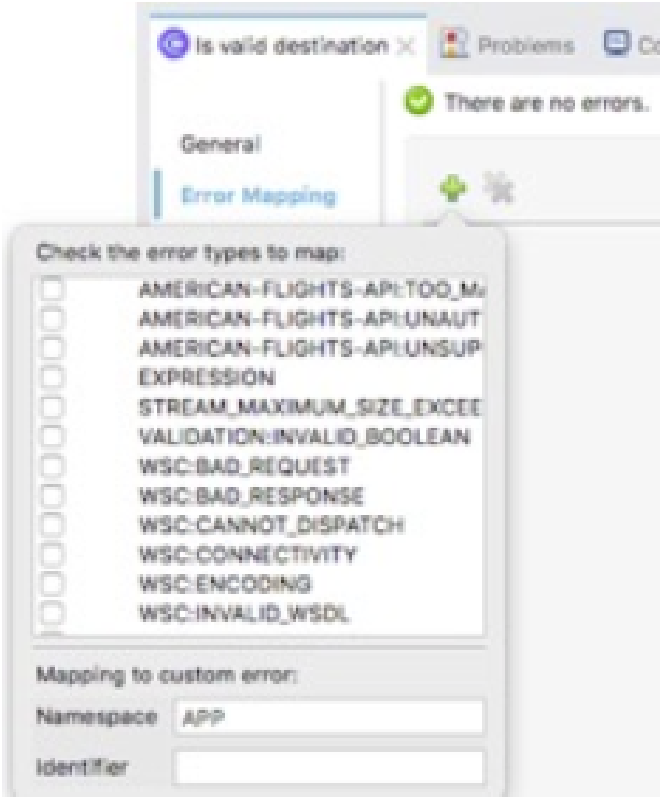
Review the possible types of errors thrown by a REST Connector operator

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.



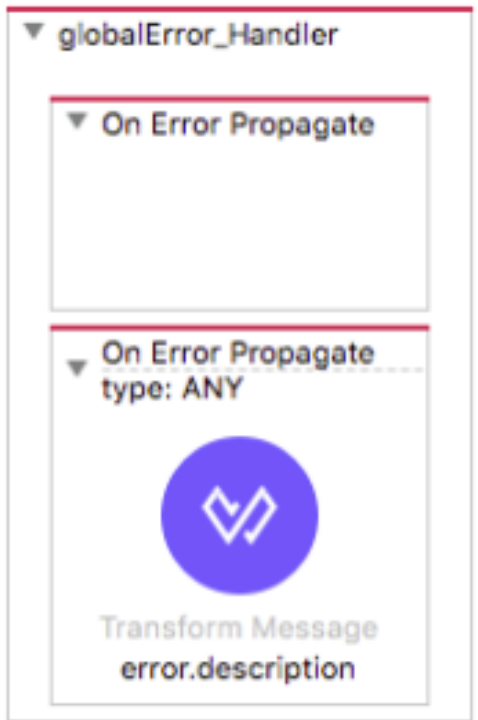
Review the possible types of errors thrown by a Validator operation

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.



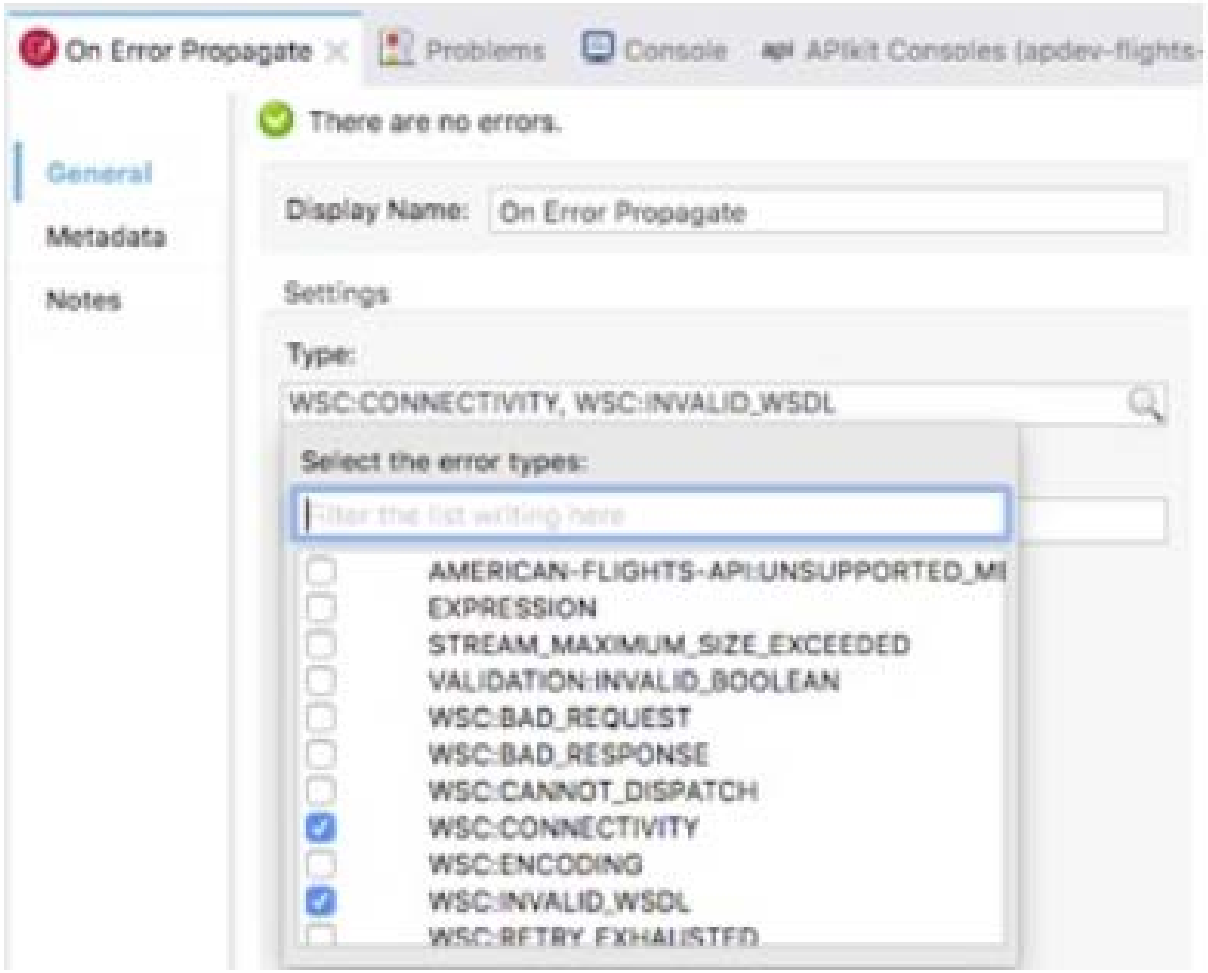
Add a second error handler to catch Web Service Consumer connectivity errors

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.



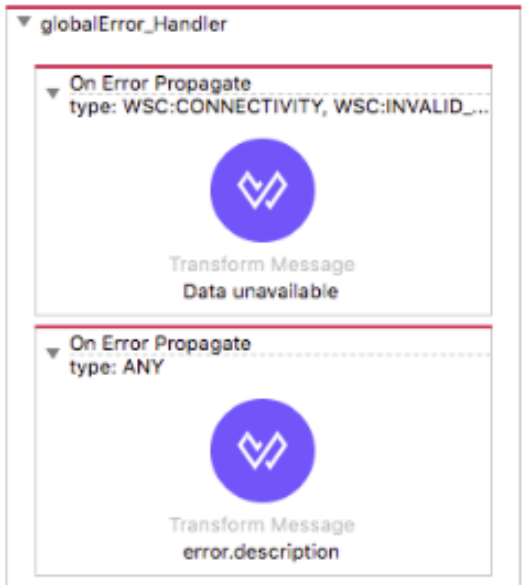
14. In the properties view for the new On Error Propagate, click the Search button.

15. Select WSC:CONNECTIVITY and WSC:INVALID_WSDL.

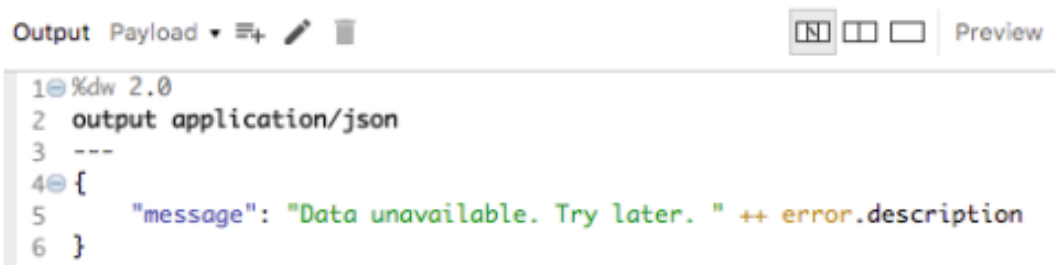


Note: If the drop-down menu does not show up, type this information instead.

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.

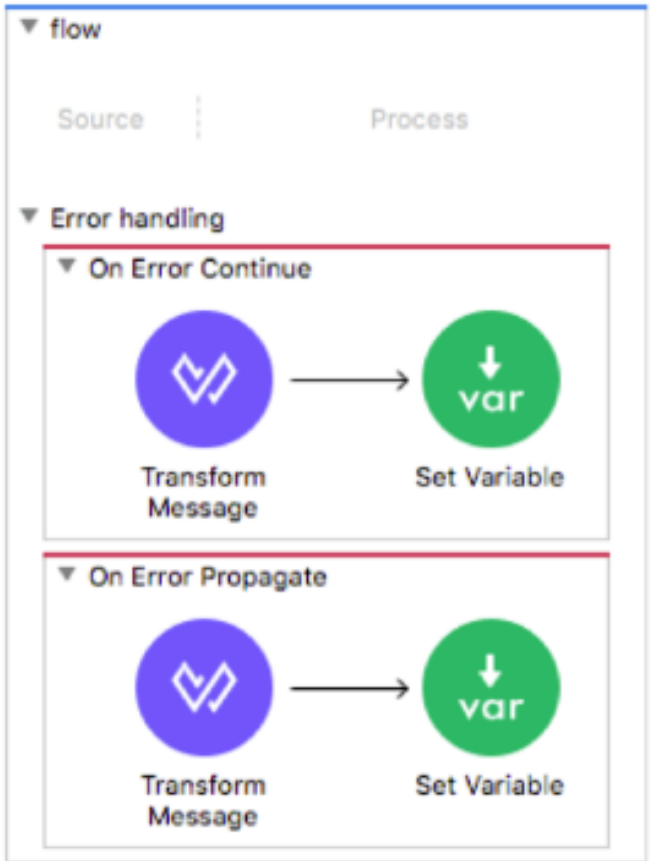
    Note: This should also not be a server error, but either a 4XX bad request or a 200 OK with an appropriate message. You will handle this error differently in the next walkthrough.

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

Handling errors at the flow level

Defining error handlers in flows

- All flows (except subflows) can have their own error handlers
- Any number of error scopes can be added to a flow's error handler

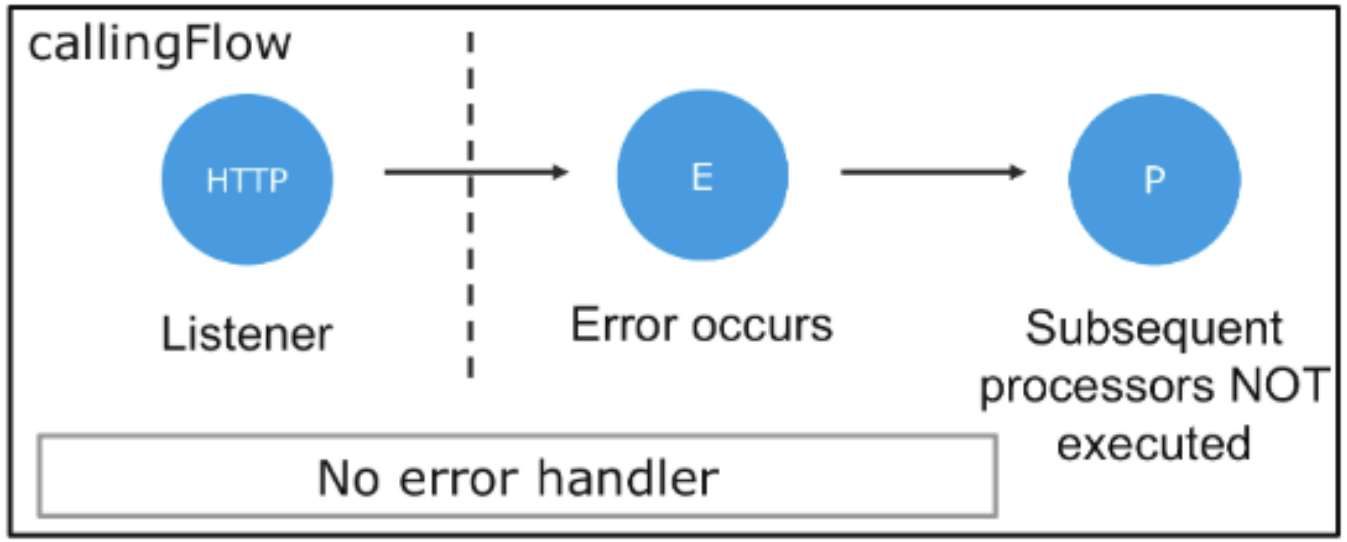


Which error scope handles an error?

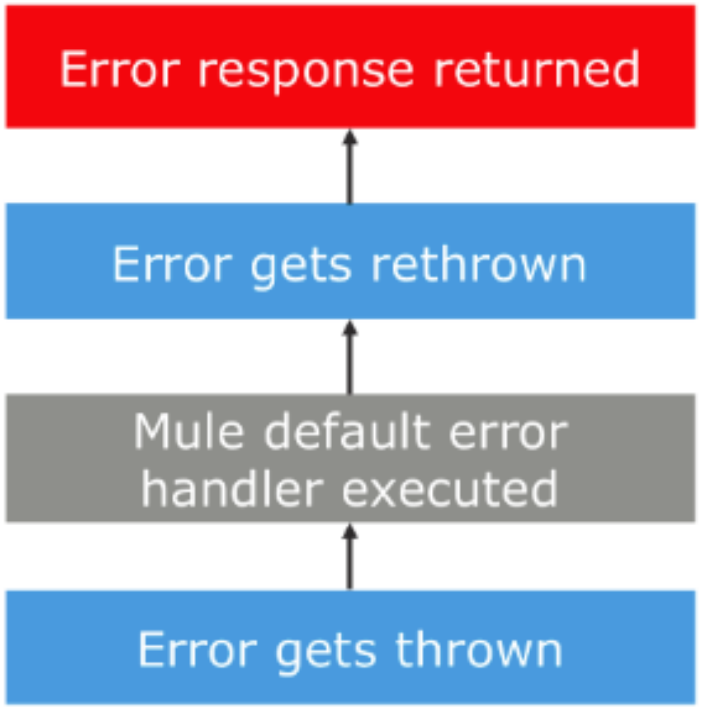
- If a flow *has* an error handler
  - The error is handled by the *first* error scope whose condition evaluates to true
  - **If *no* scope conditions are true,** the error is handled by the **Mule default error handler** NOT any scope in an application's global default handler
    - The Mule default error handler propagates the error up the execution chain where there may or may not be handlers

- If a flow *does not have* an error handler
  - The error is handled by a scope in an **application's default error handler** (the first whose scope condition is true, which may propagate or continue) otherwise it is handled by the Mule default error handler

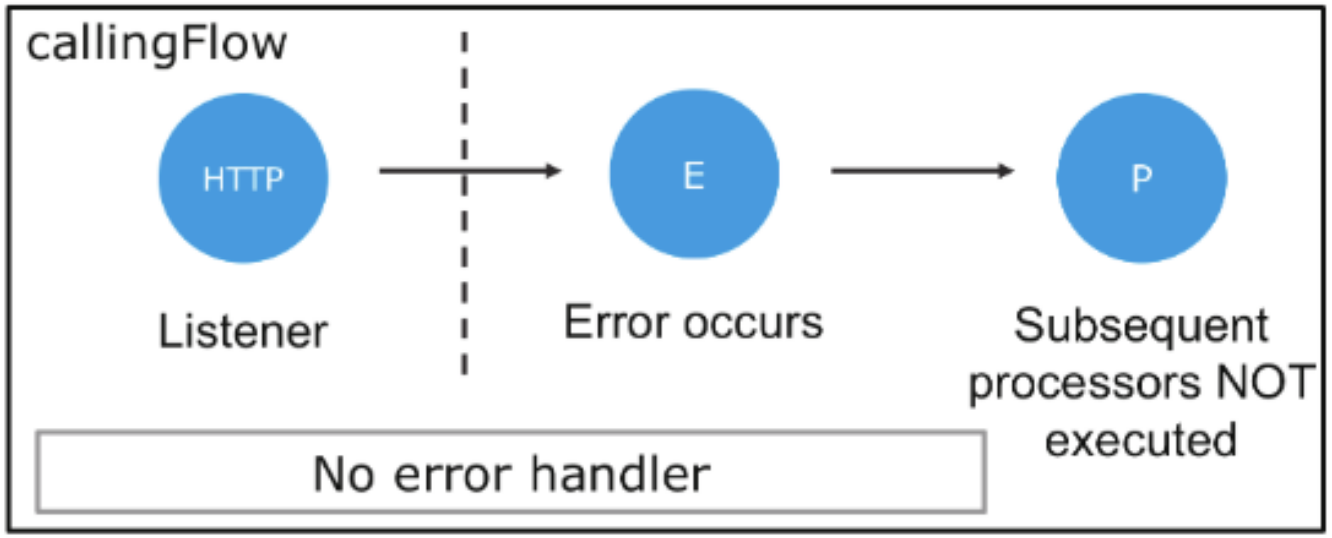
Error handling scenario 1



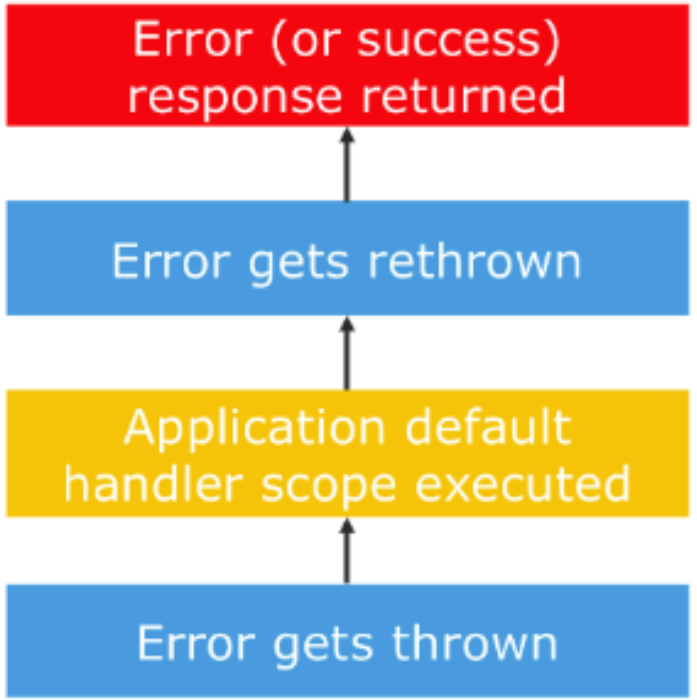
Application default error handler – ***with no matching scope*** *condition*



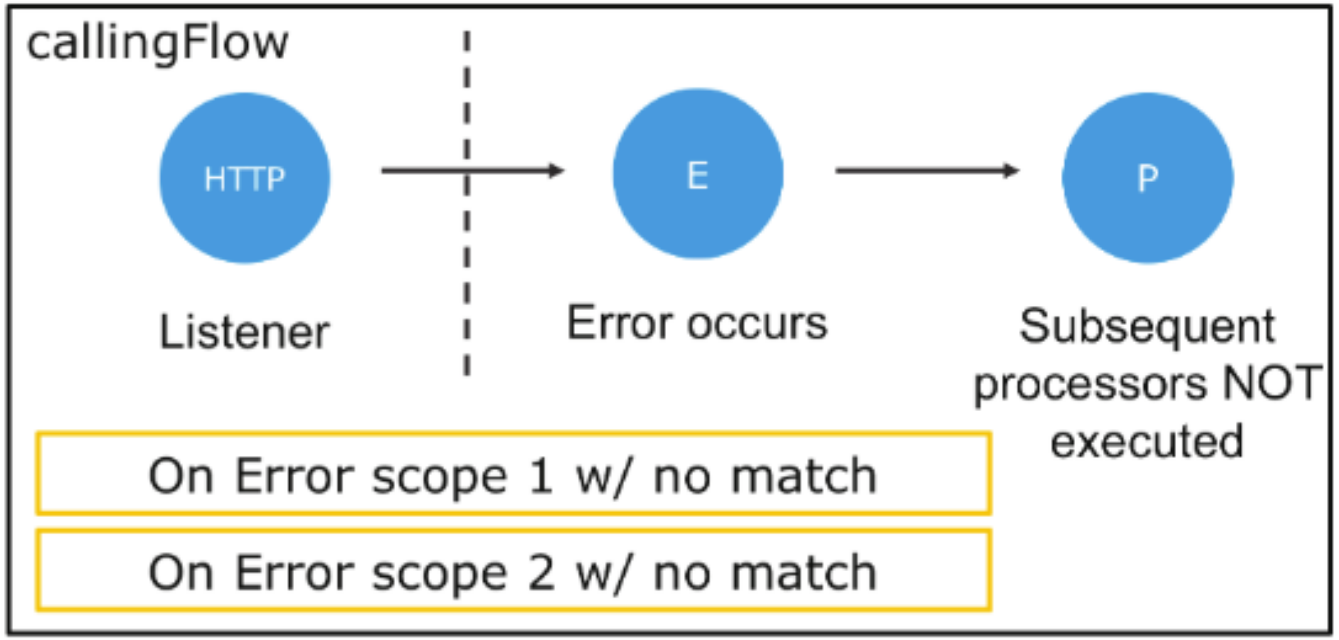
Error handling scenario 2



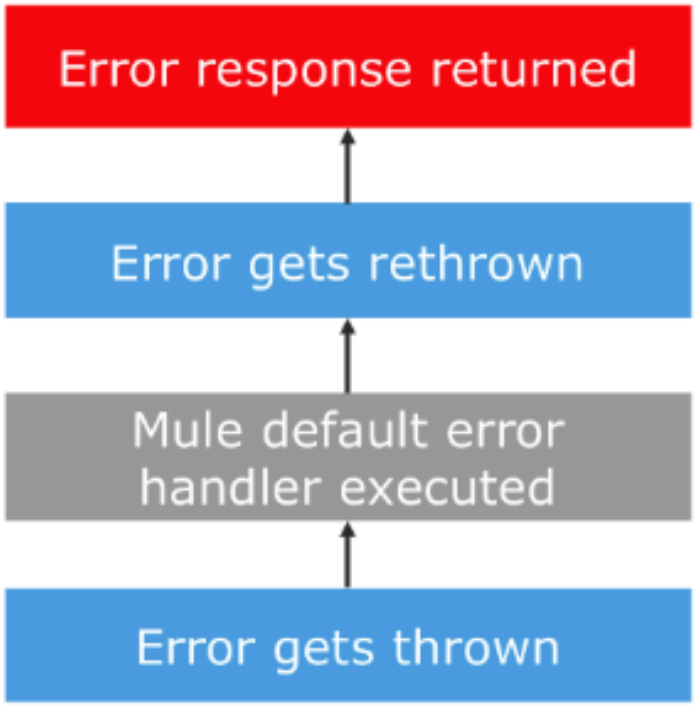
Application default error handler – ***with a matching scope condition***



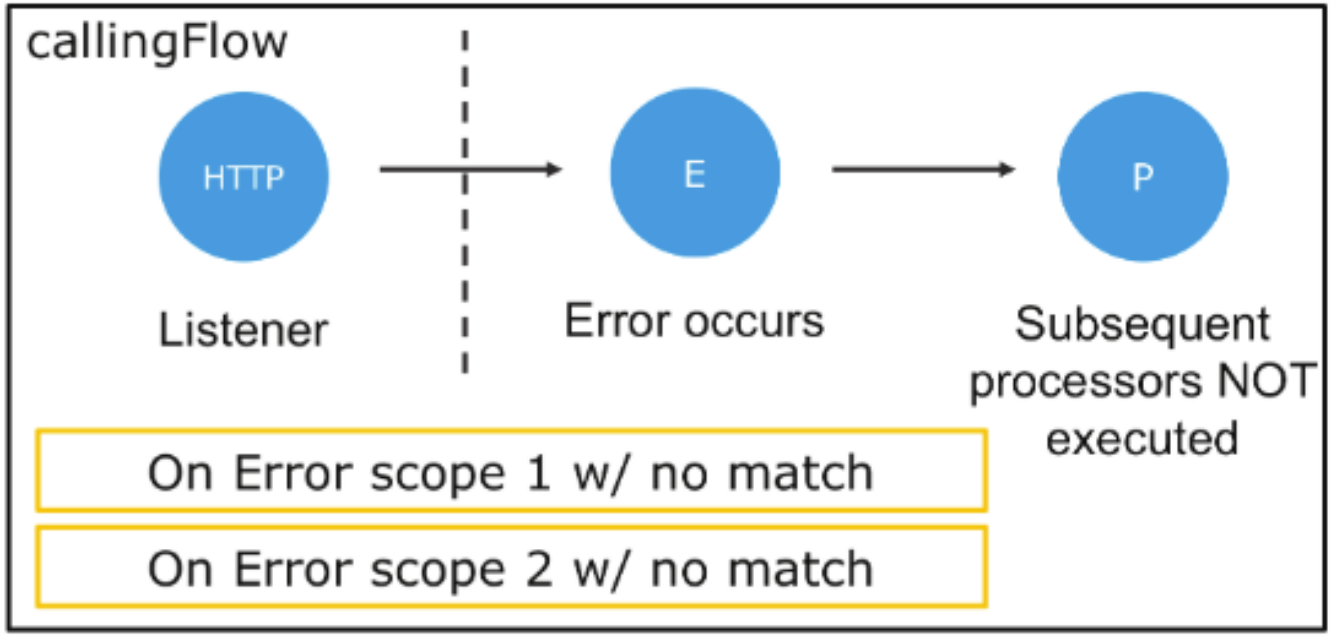
Error handling scenario 3



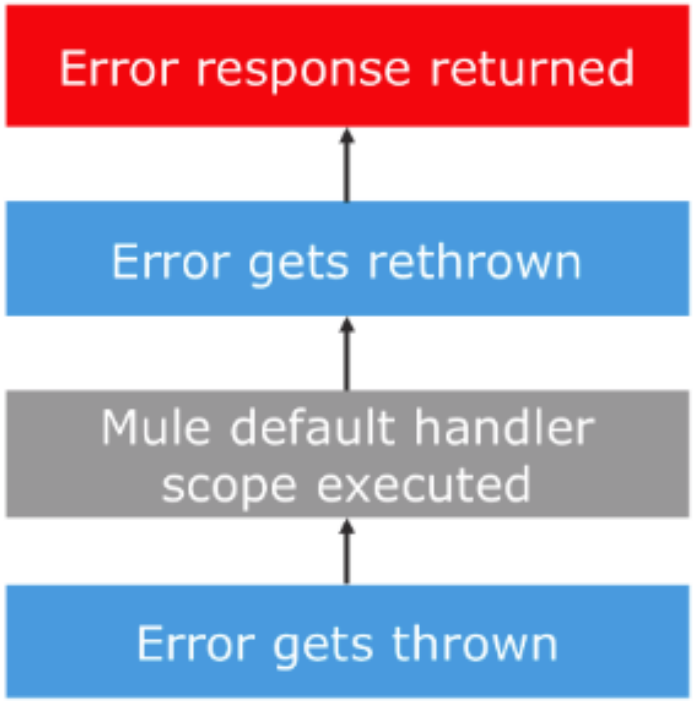
Application default error handler – ***with no matching scope*** *condition*



Error handling scenario 4

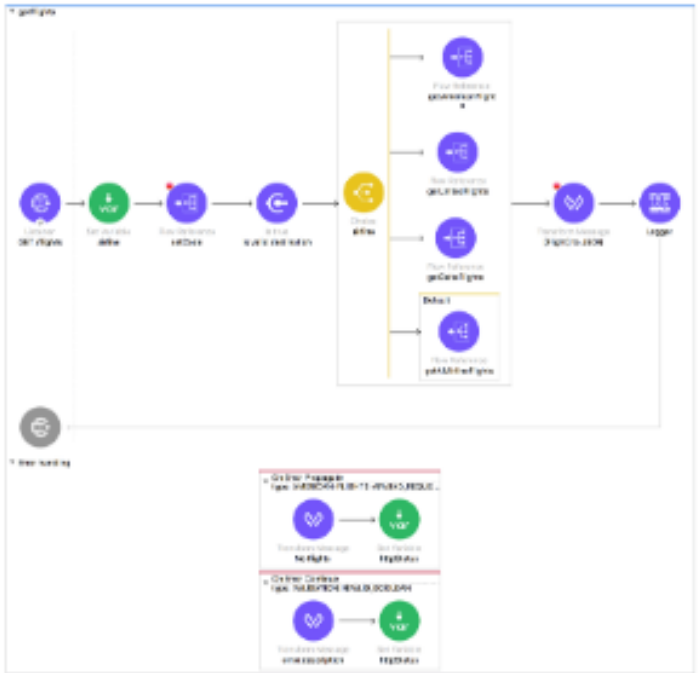


Application default error handler –
with a matching scope condition



Walkthrough 10-4: Handle errors at the flow level

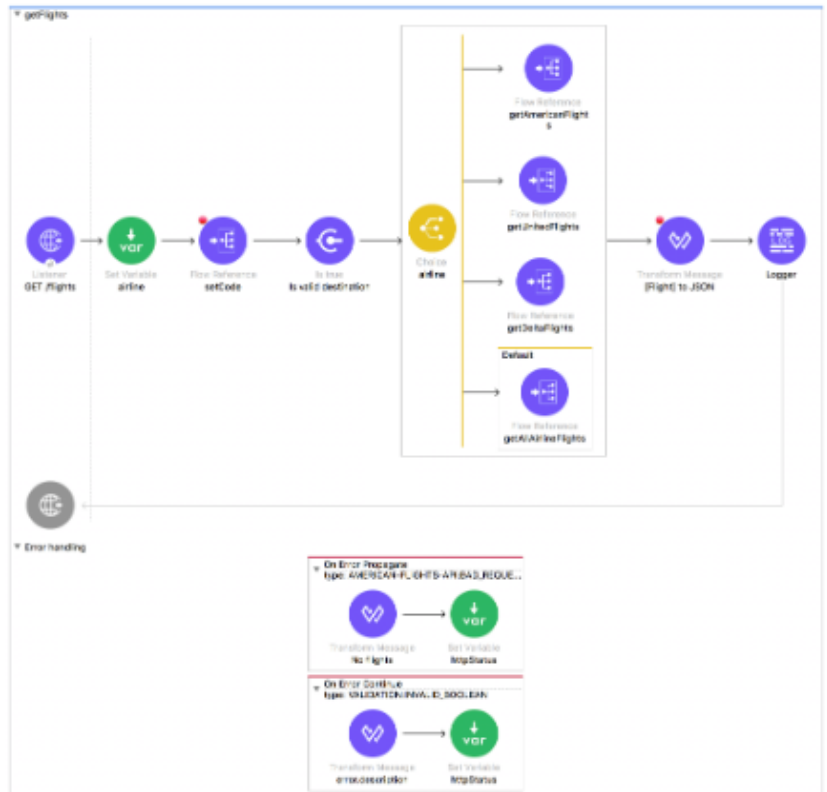
- Add error handlers to a flow
- Test the behavior of errors thrown in a flow and by a child flow
- Compare On Error Propagate and On Error Continue scopes in a flow
- Set an HTTP status code in an error handler and modify an HTTP Listener to return it



Walkthrough 10-4: Handle errors at the flow level

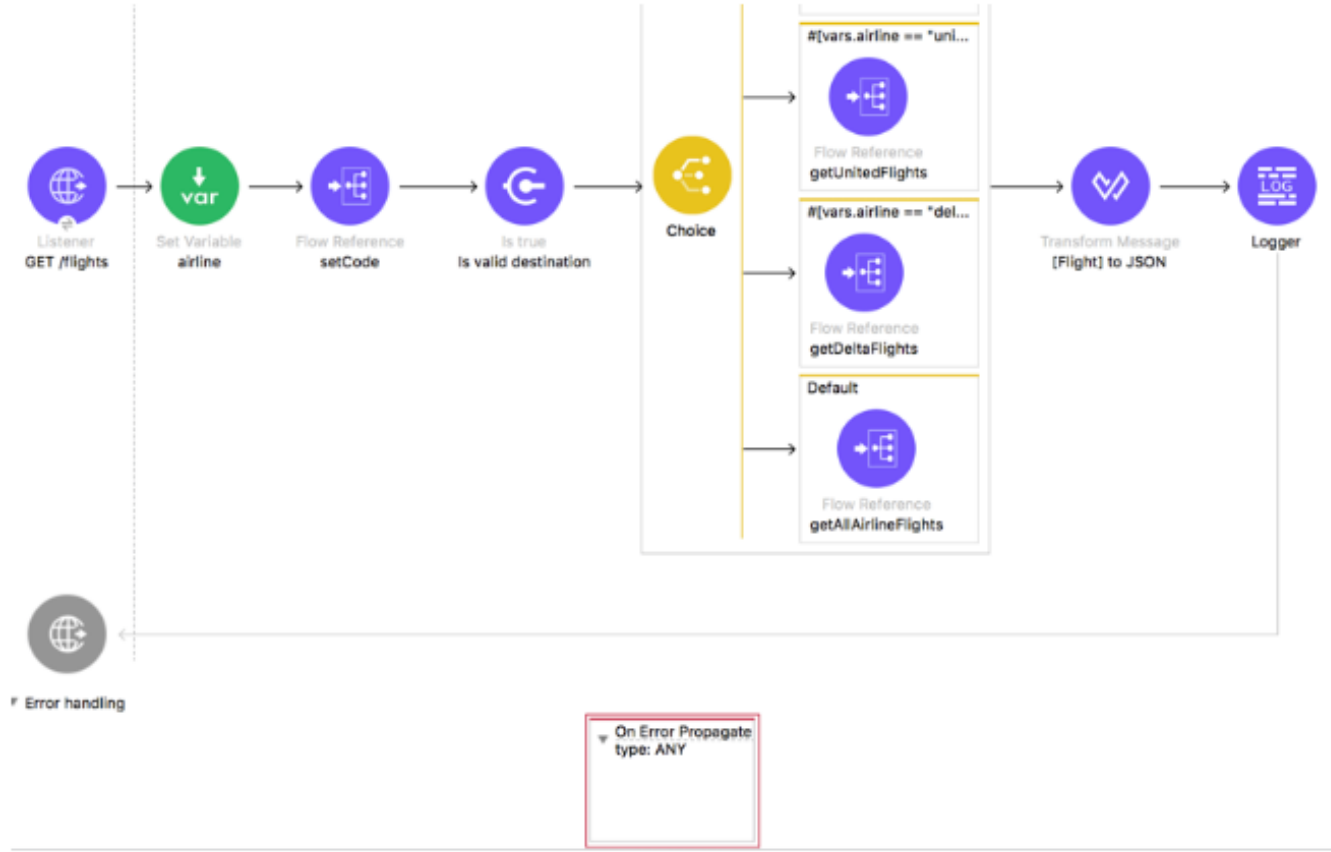
In this walkthrough, you add multiple error handlers to a flow. You will:

- 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.



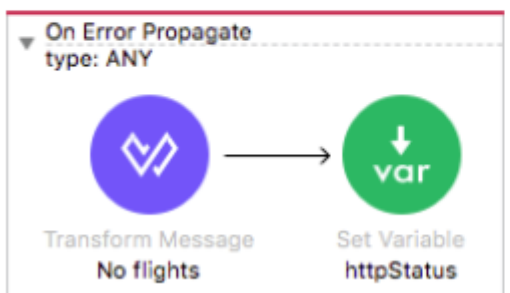
Set the payload in the error handler to the error description

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

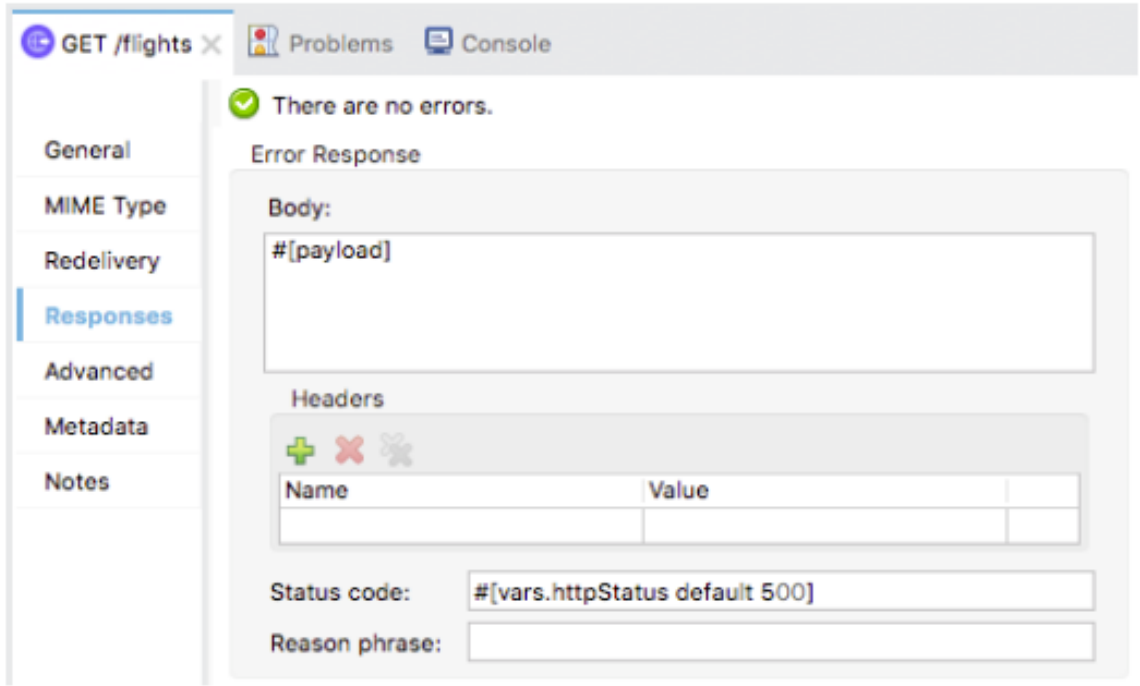
8. Add a Set Variable transformer to the On Error Propagate.
9. In the Set Variable properties view, set the display name and name to httpStatus.
10. Set the value to 200.



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

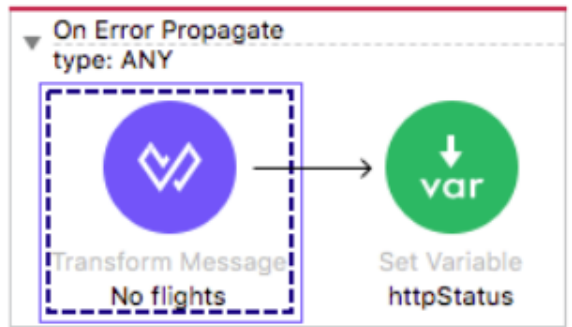
11. Navigate to the Responses tab in the properties view for the GET /flights Listener.
12. Change the error response status code (not the response status code!) to an expression that sets it equal to the value of an httpStatus variable with a default value of 500.

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

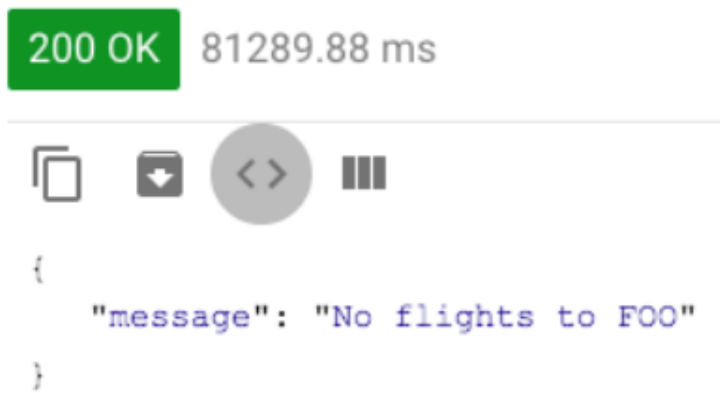


Test the flow's On Error Propagate with an error in the flow

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.

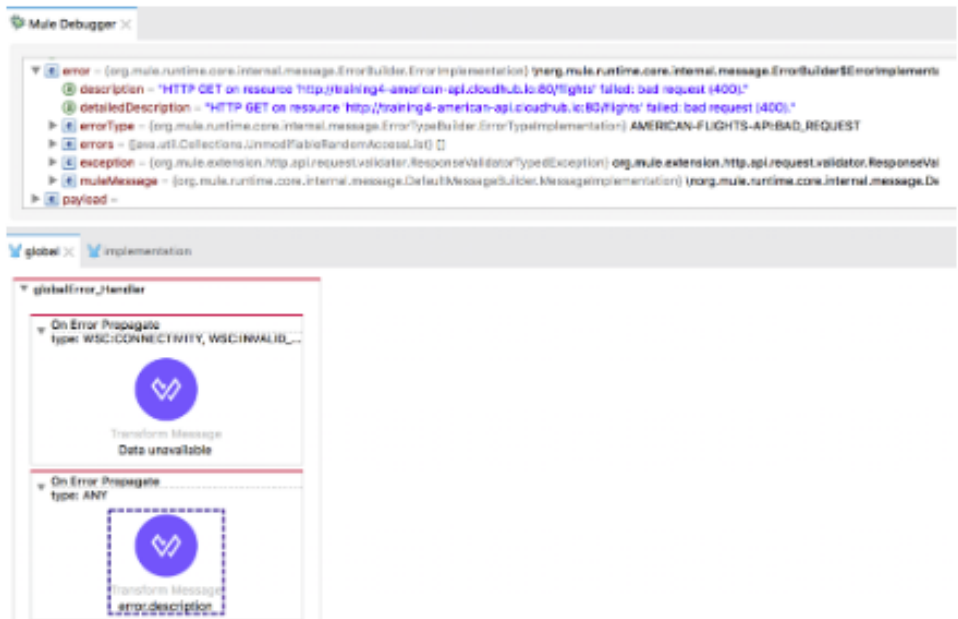


16. Step through the rest of the application.
17. Return to Advanced REST Client; you should get the 200 status code and the JSON message.

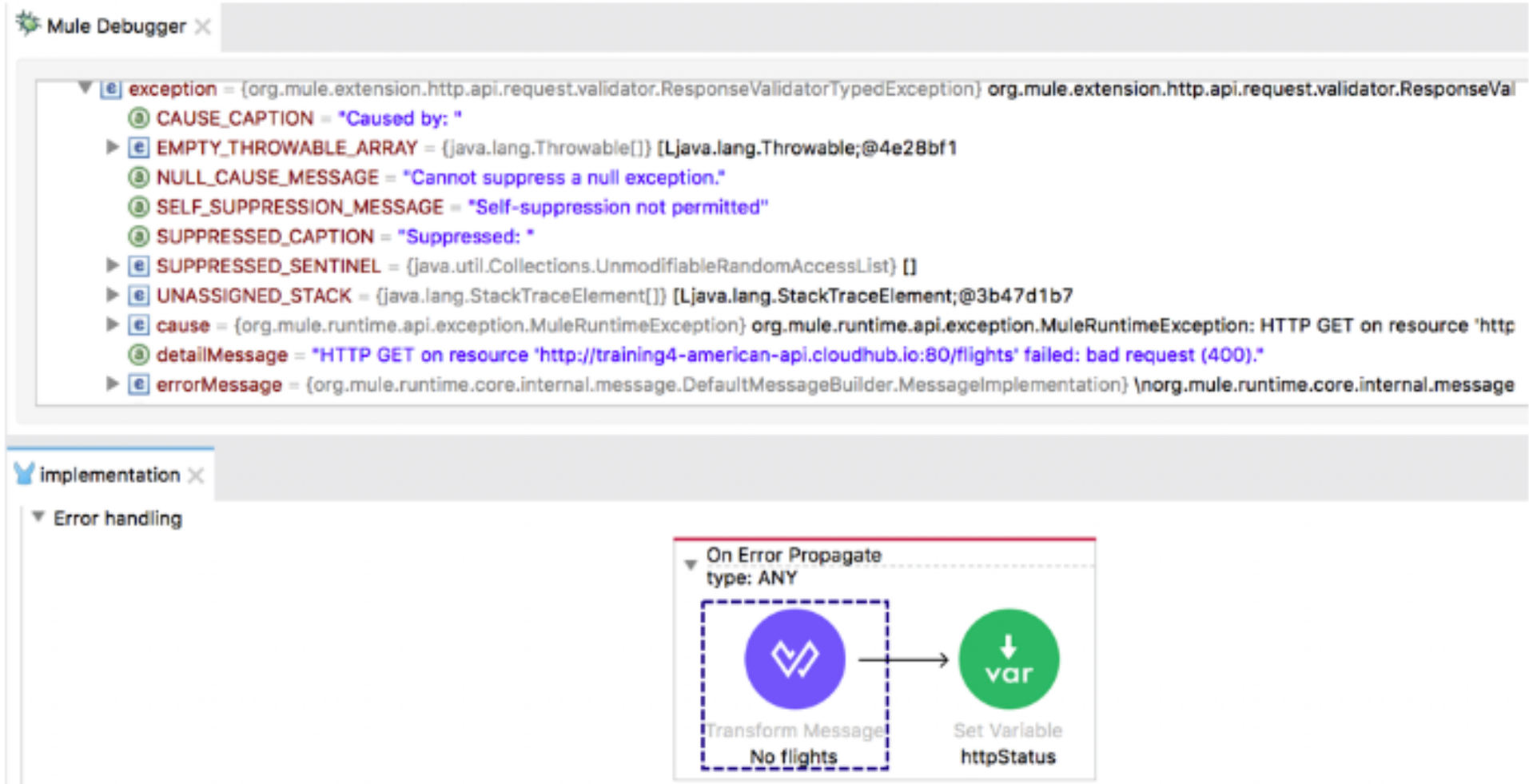


Test the flow's On Error Propagate with an error in a child flow

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.

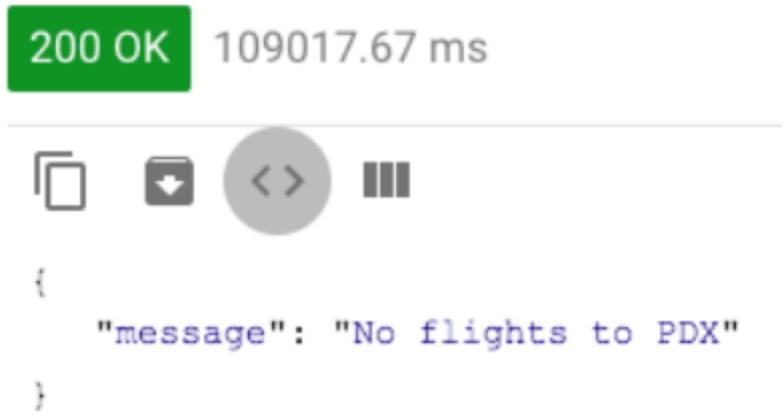


20. Step until the event is passed to the parent flow's error handler.



21. Step through the rest of the application.

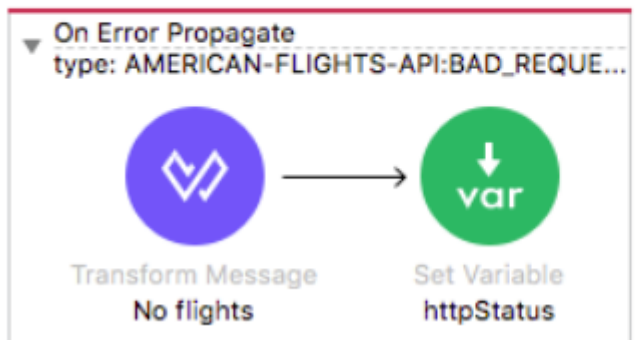
22. Return to Advanced REST Client; you should get the 200 status code and the JSON error message.



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

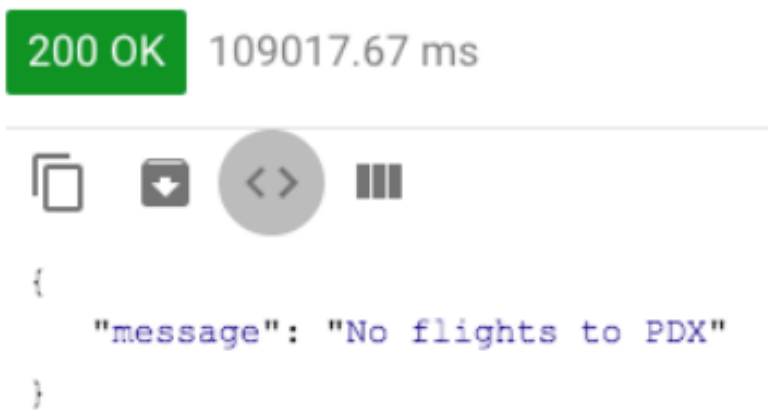
Specify the type of error to be handled by the flow's error handler

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.



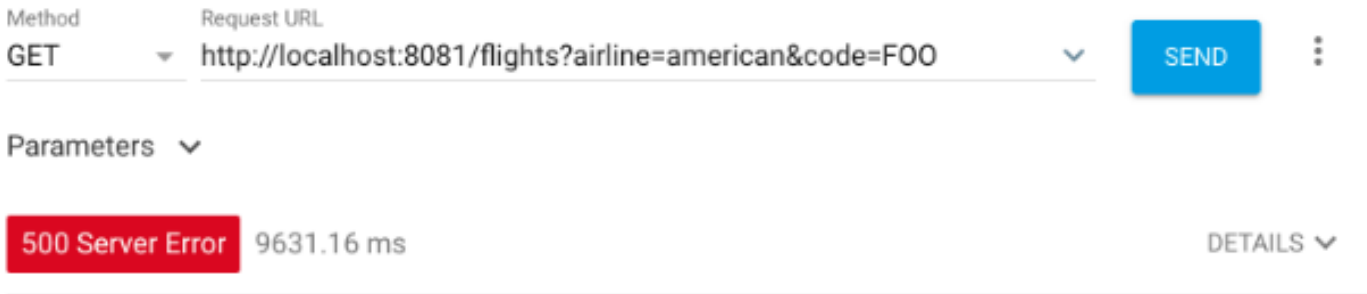
Test the application

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.

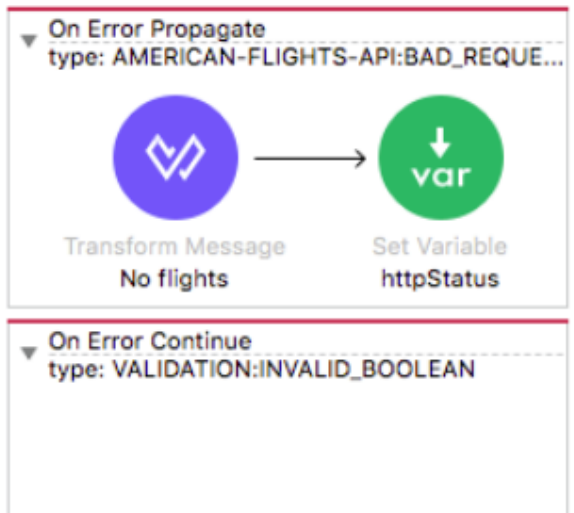
30. In the Mule Debugger, step through the application; you should get a 500 Server Error and no message because the error is not handled by the getFlights flow error handler or by globalError_handler.



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.

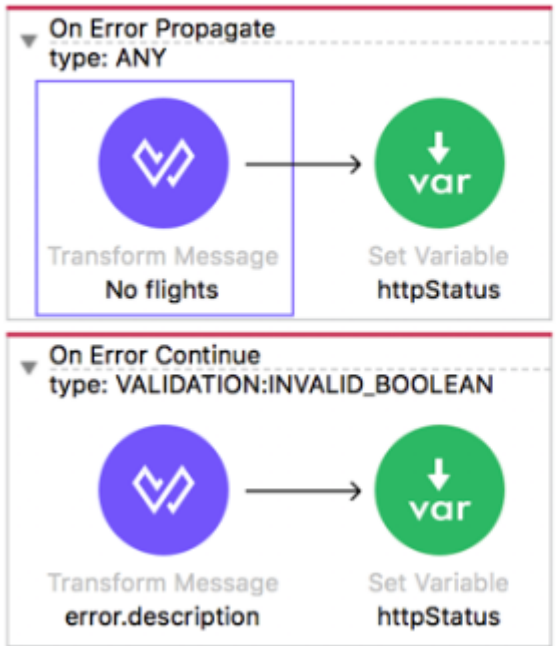


37. Add a Transform Message component to the scope and set the display name to error.description.
38. In the Transform Message properties view, set the payload to a JSON message property equal to the error.description.

```
%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.

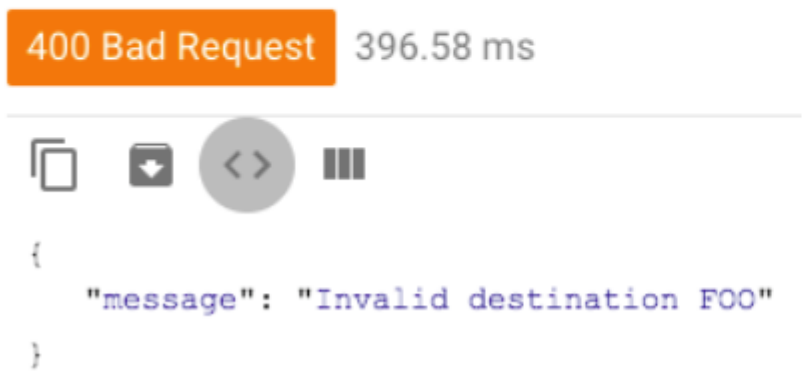
Set the HTTP Listener response status code

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]
```

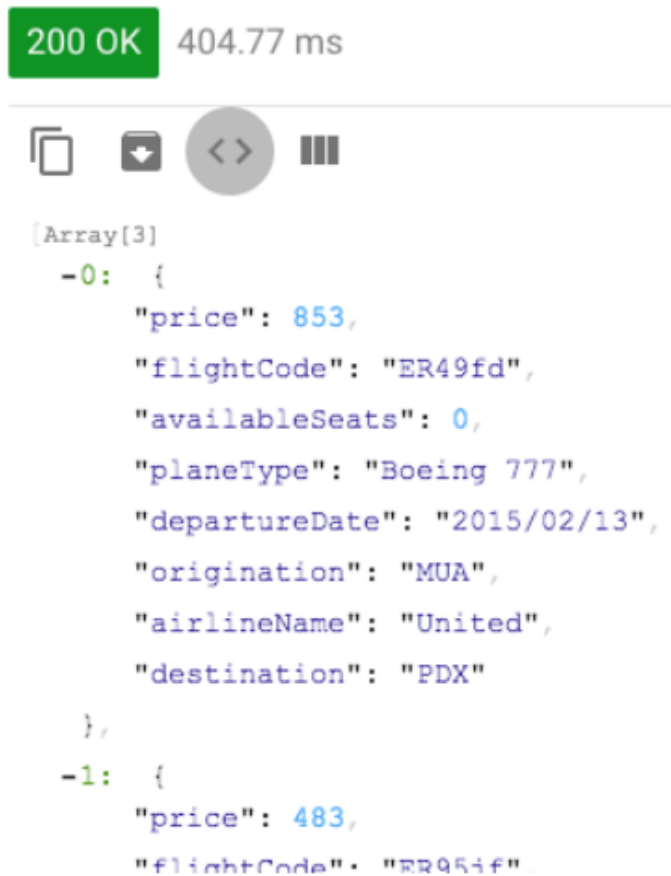
Test the application

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

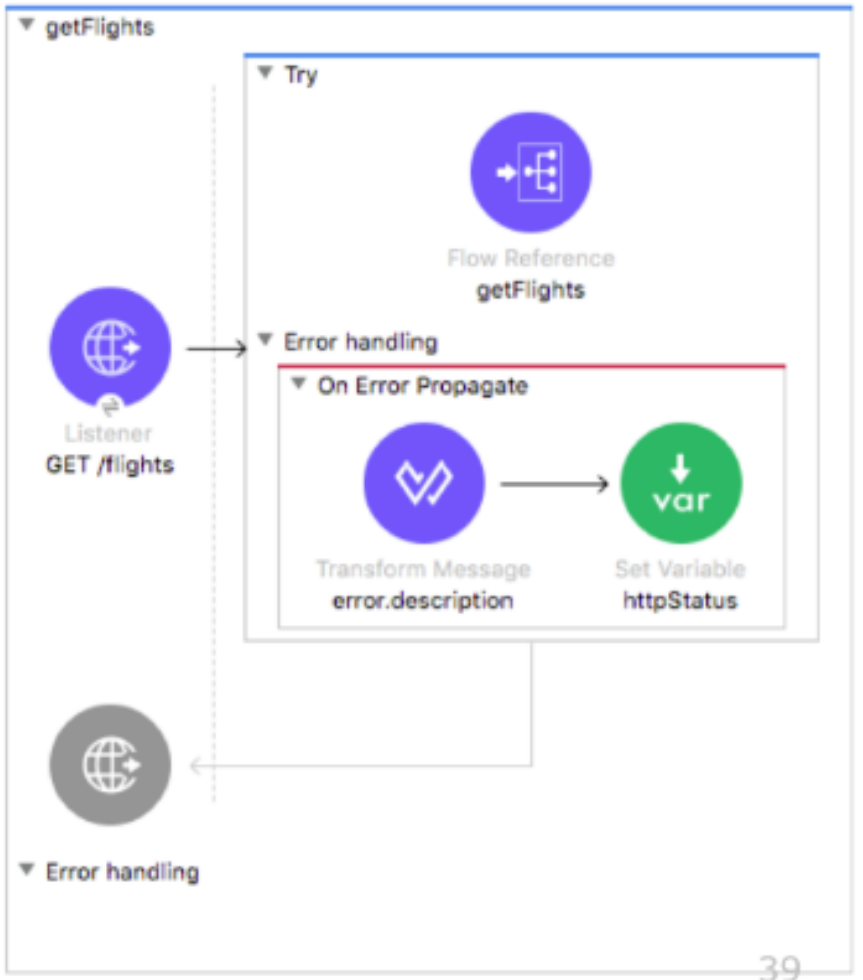
48. In Advanced REST Client, change the code and make a request to http://localhost:8081/flights?airline=american&code=PDX; you should get a 200 response and no flights to PDX.
49. Change the airline to make a request to http://localhost:8081/flights?airline=united&code=PDX; you should get flights to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



Handling errors at the processor level

Handling errors at the processor level

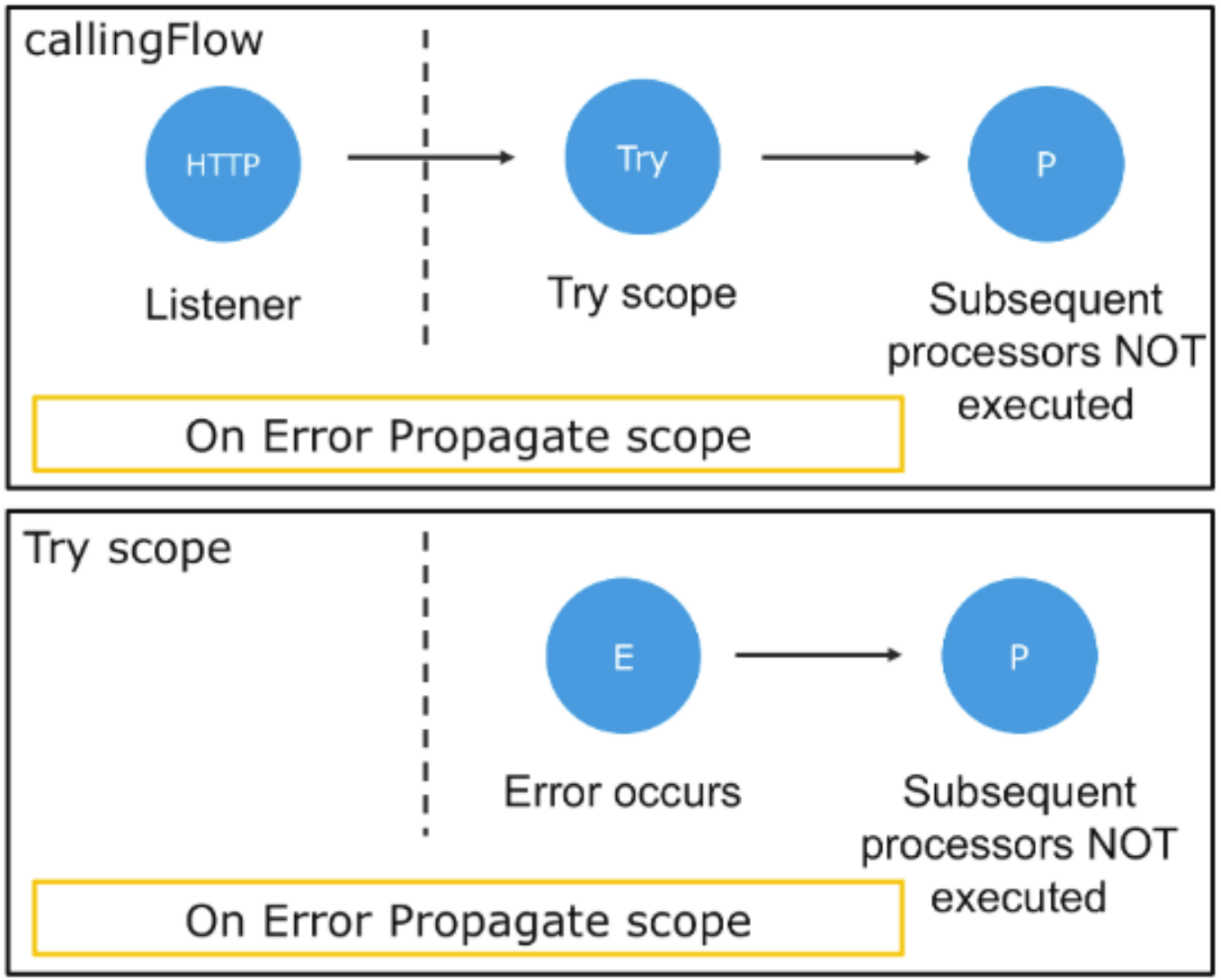
- For more fine grain error handling of elements within a flow, use the Try scope
- Any number of processors can be added to a Try scope
- The Try scope has its own error handling section to which one or or more error scopes can be added

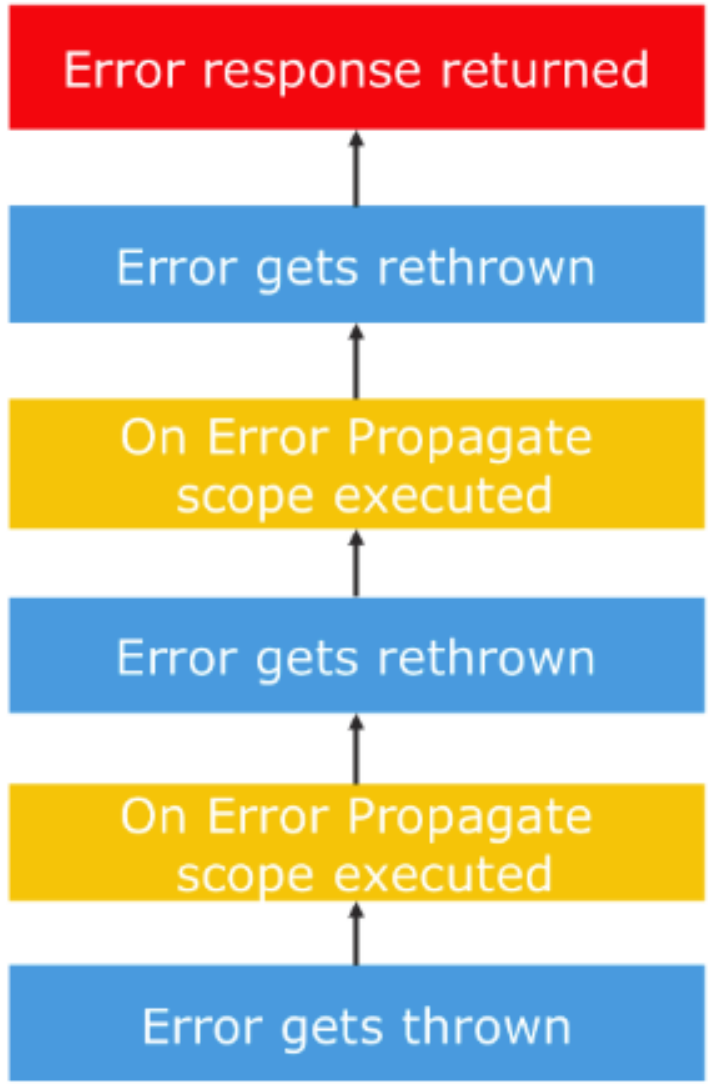


Error handling behavior in the Try scope

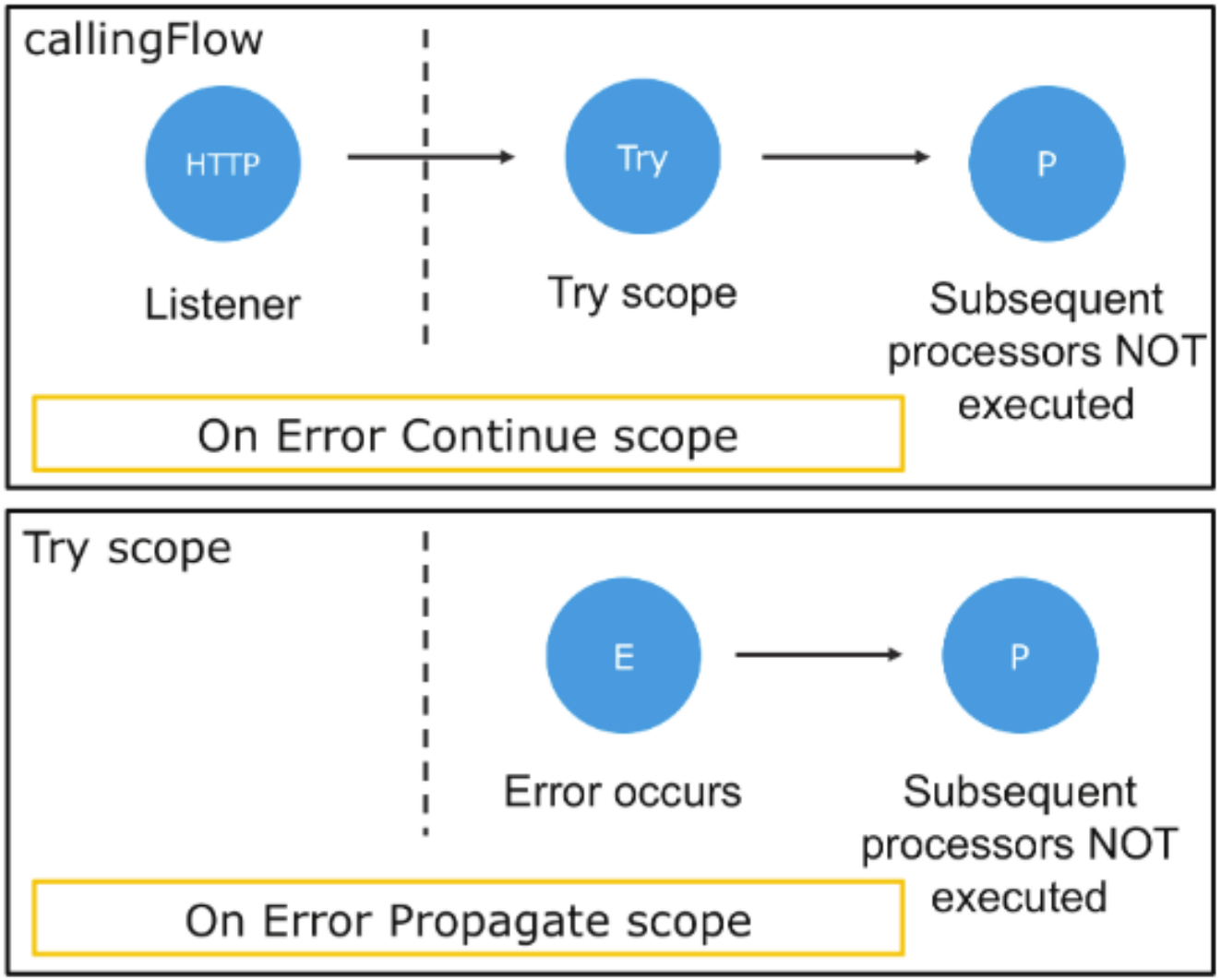
- **On Error Propagate**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the *Try scope* is not executed
    - If a transaction is being handled, it is rolled back
    - The error is rethrown up the execution chain to the parent flow, which handles the error
  - An HTTP Listener returns an *error* response
- **On Error Continue**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the *Try scope* is not executed
    - If a transaction is being handled, it is committed
    - The event is passed up to the parent flow, which continues execution
  - An HTTP Listener returns a *successful* response

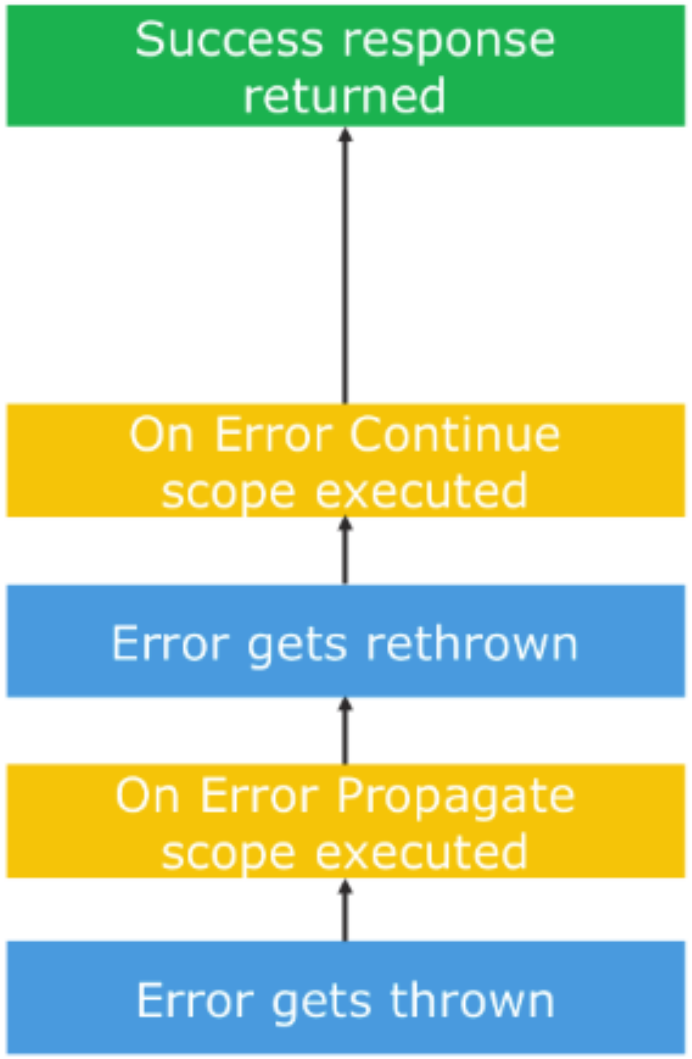
Try scope: Error handling scenario 1



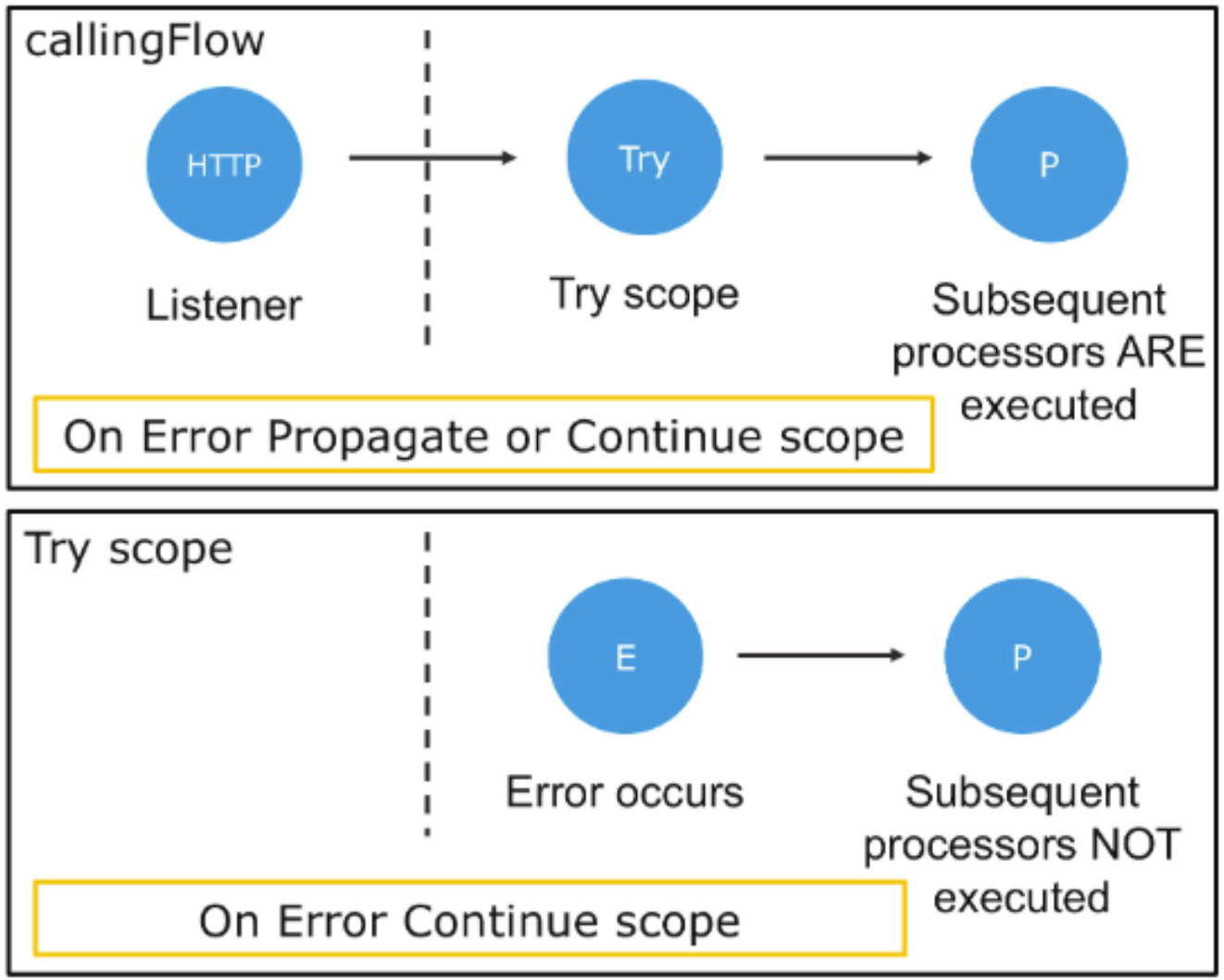


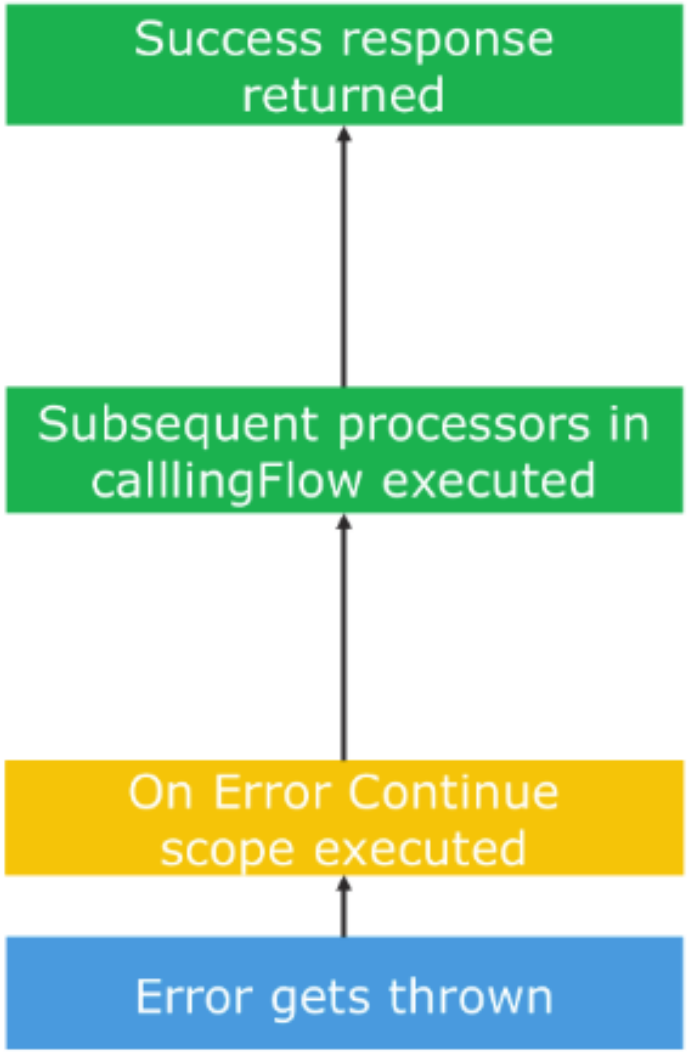
Try scope: Error handling scenario 2





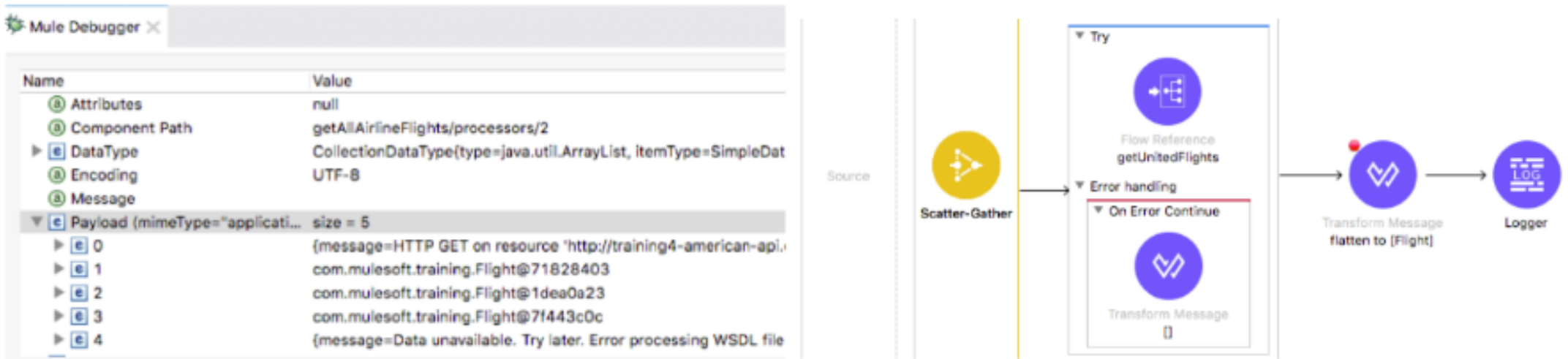
Try scope: Error handling scenario 3





Walkthrough 10-5: Handle errors at the processor level

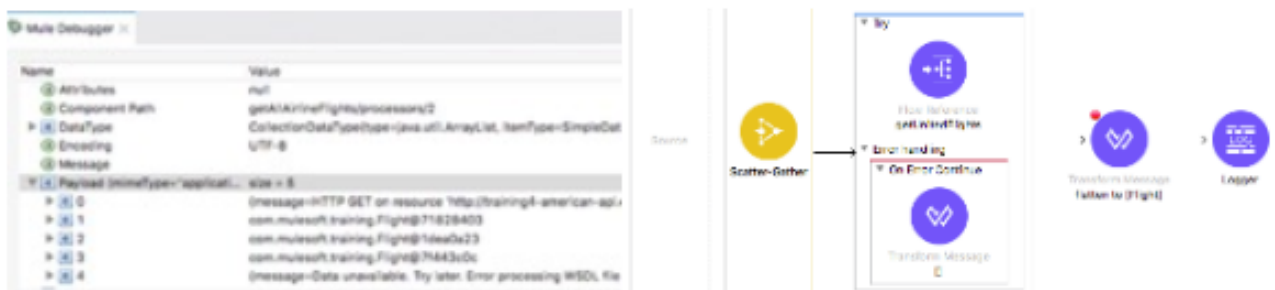
- Wrap the Flow Reference in each branch of a Scatter-Gather in a Try scope
- Use an On Error Continue scope in each Try scope error handler to provide a valid response so flow execution can continue



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

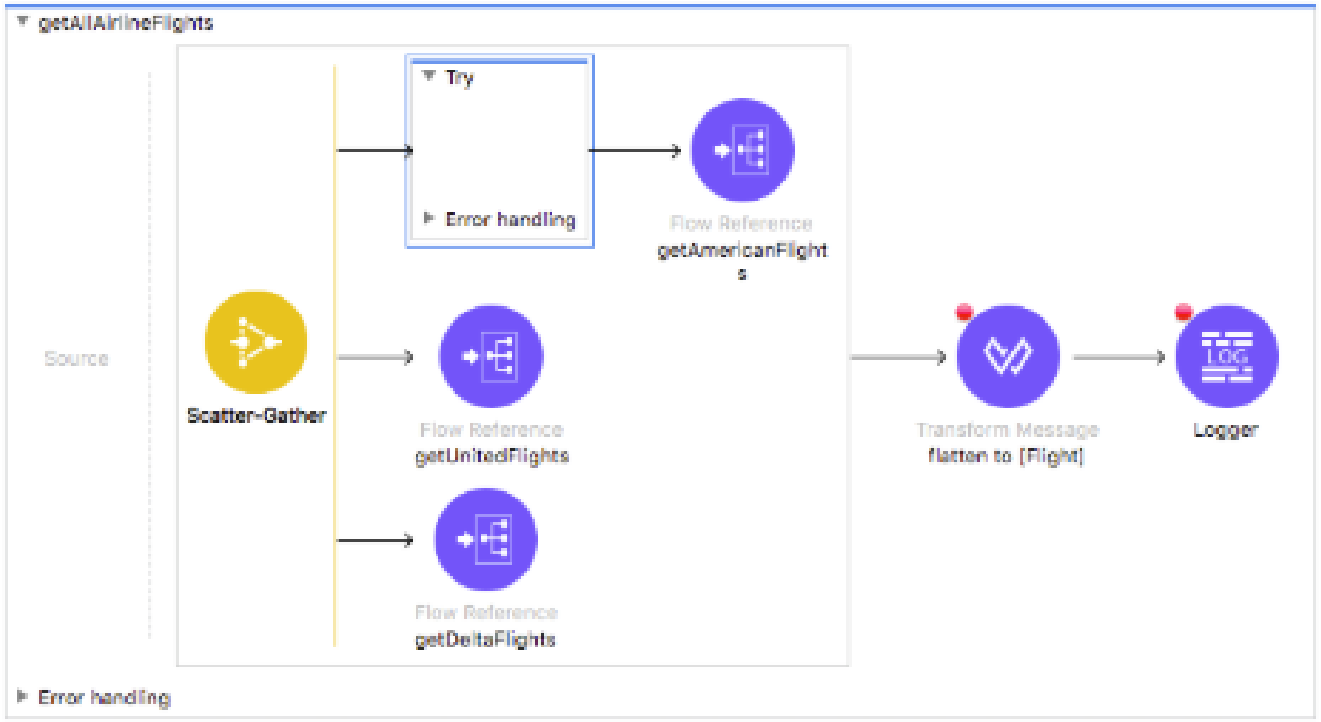
1. Return to implementation.xml in Anypoint Studio.
2. Debug the project.
3. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX.
4. Return to the Mule Debugger and step through the application; you should see a routing exception.

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.

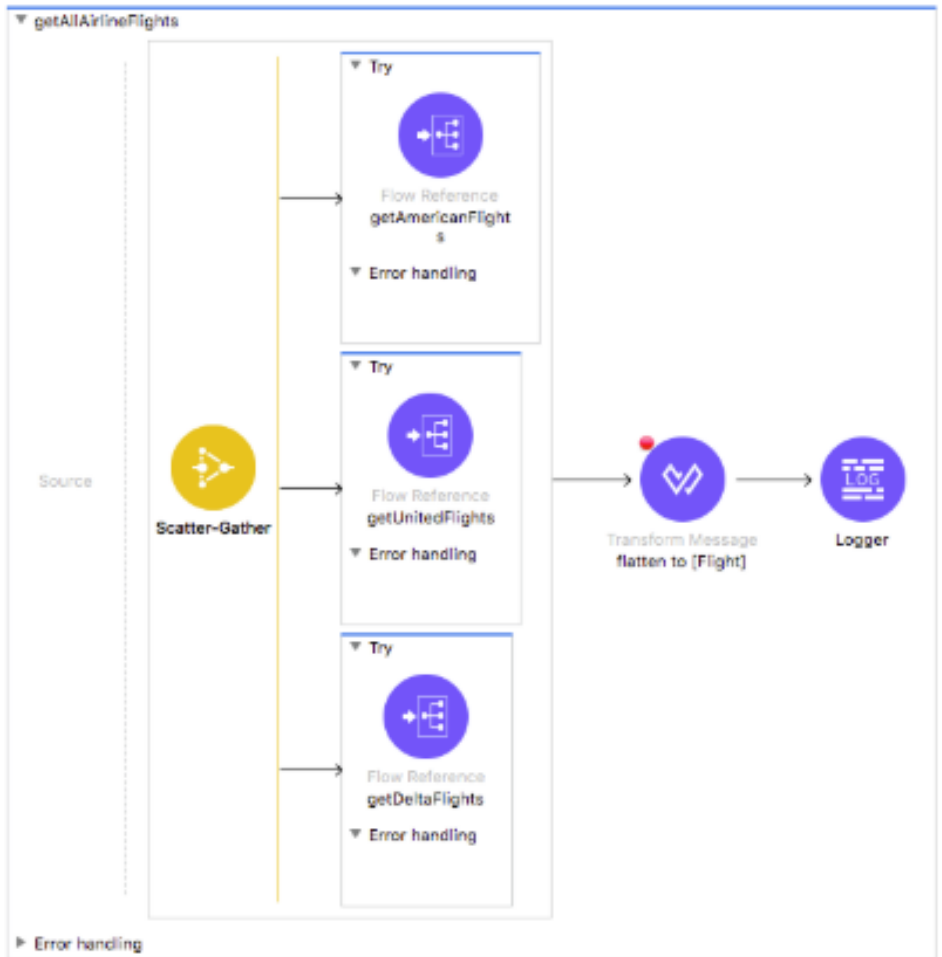
Place each Flow Reference in the Scatter-Gather in a Try scope

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



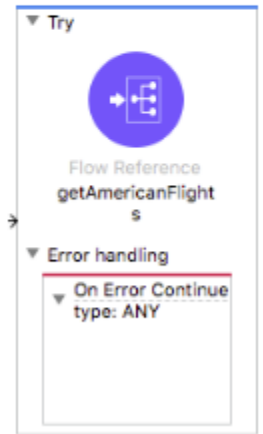
9. Drag the getAmericanFlights Flow Reference into the Try scope.
10. Add two more Try scopes to the Scatter-Gather.

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.



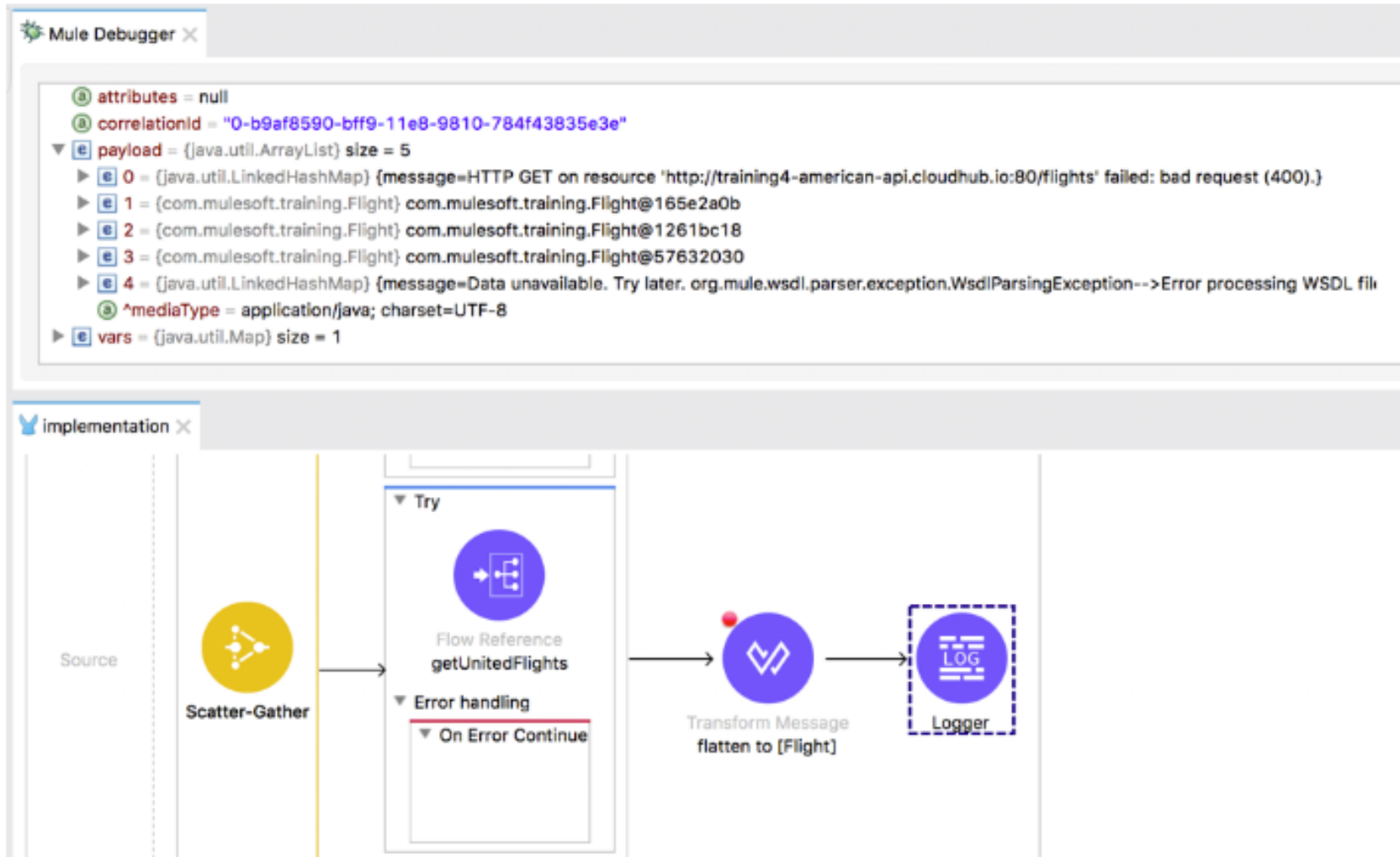
Debug the application

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.

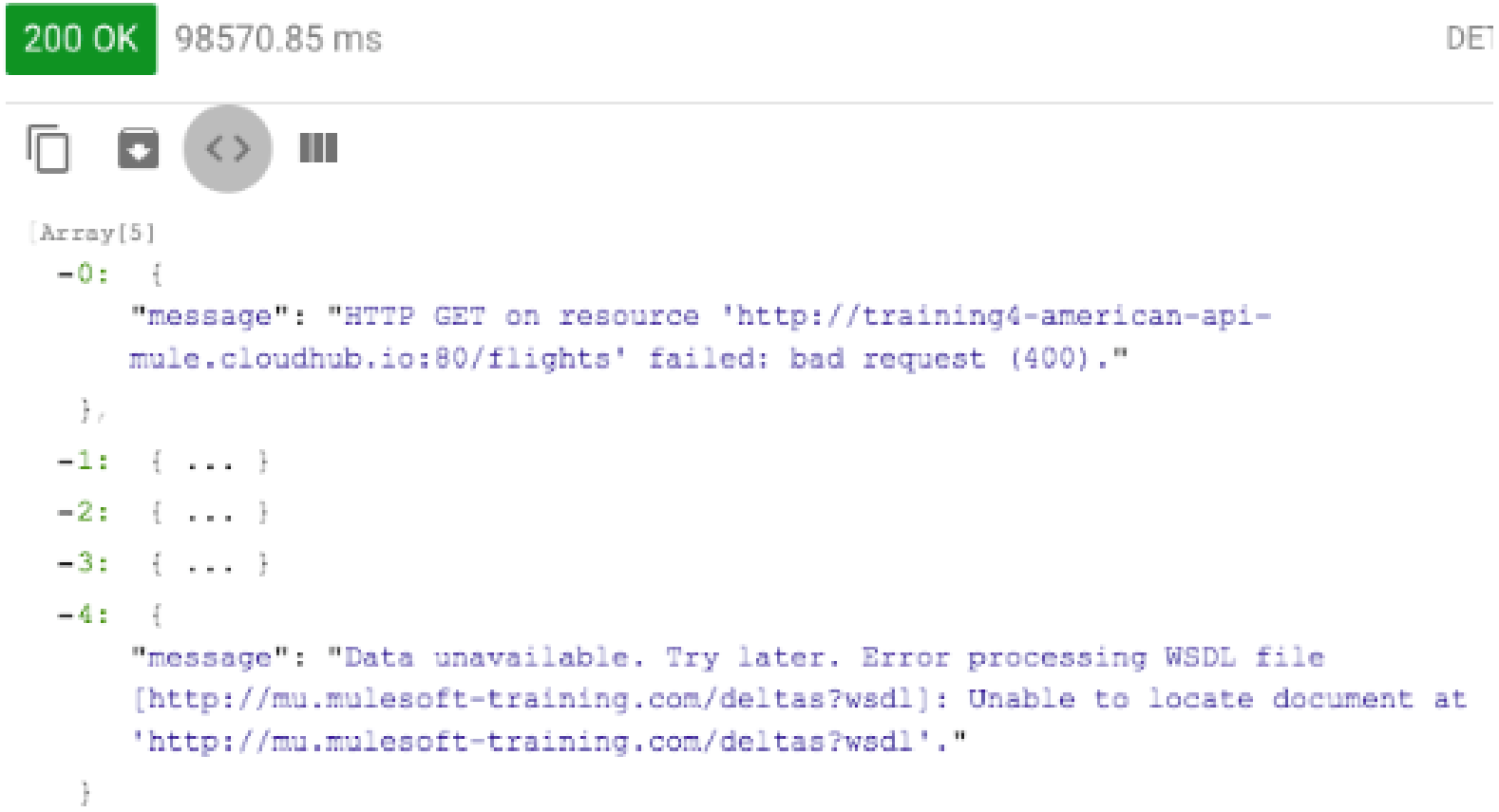
16. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX.
17. In the Mule Debugger, step through the application until you are at the Logger after the Scatter-Gather in getAllAirlineFlights.

18. Expand Payload; you should see three flights (from United) and two error messages.



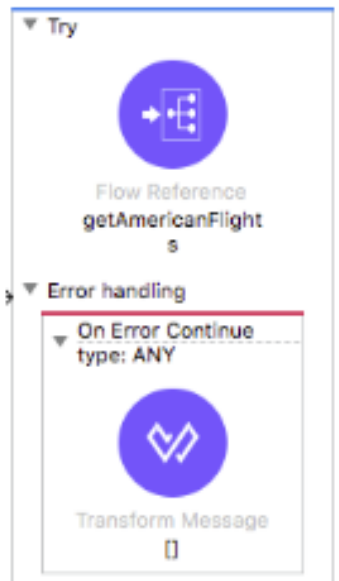
19. Step through the rest of the application.

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

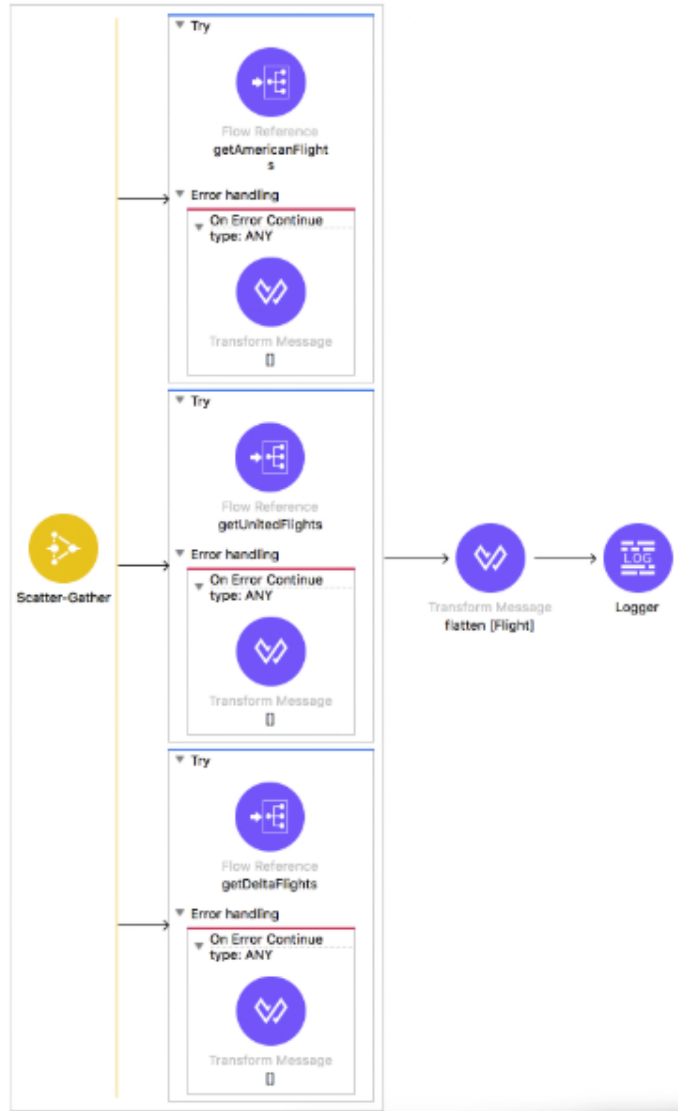
21. Return to Anypoint Studio and switch to the Mule Design perspective.
22. In getAllAirlineFlights, add a Transform Message component to one of the On Error Continue scopes.
23. Set the display name to [].



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

```
1 %dw 2.0
2 output application/java
3 ---
4 []
```

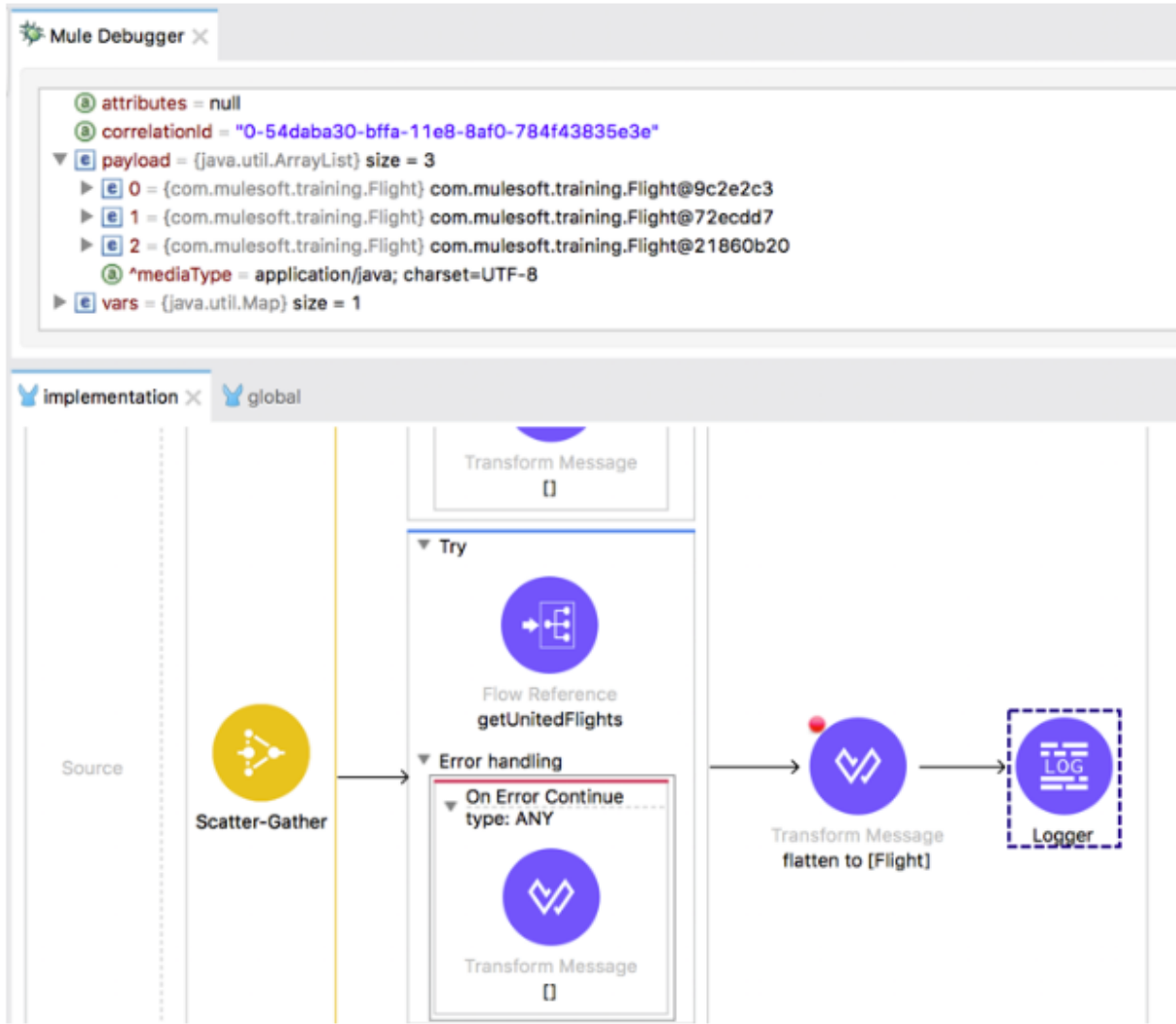
25. Repeat the steps to add the same Transform Message component to the two other error handlers.



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.

Mapping errors to custom error types

Mapping errors for more granular error handling

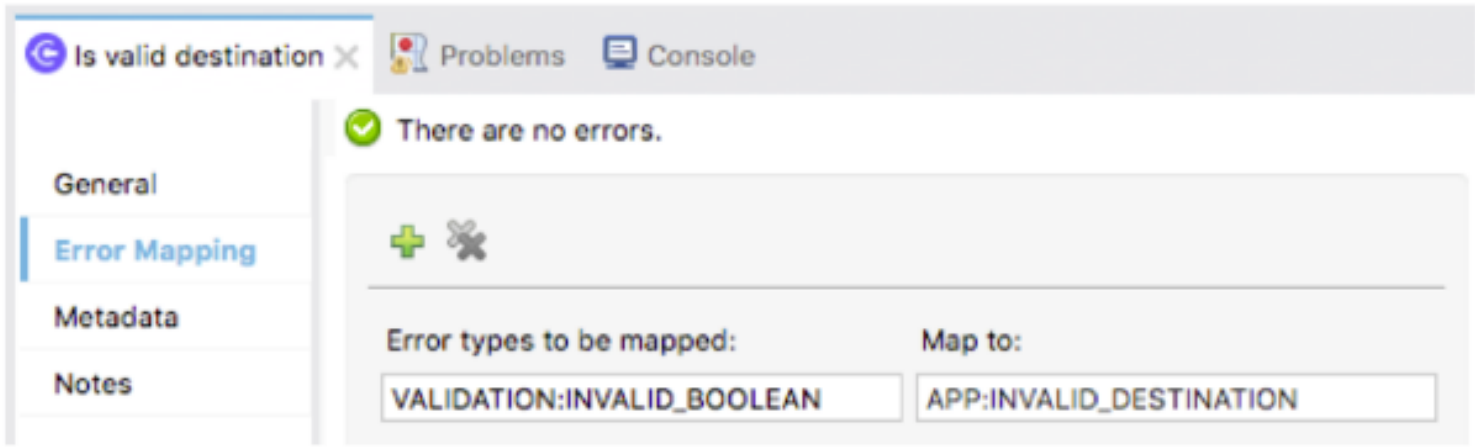
- If an app has two HTTP Request operations that call different REST services, a connectivity failure on either produces the same error
  - Makes if difficult to identify the source of the error in the Mule application logs
- You can map each connectivity error to different custom error types
- These custom error types enable you to differentiate exactly where an error occurred

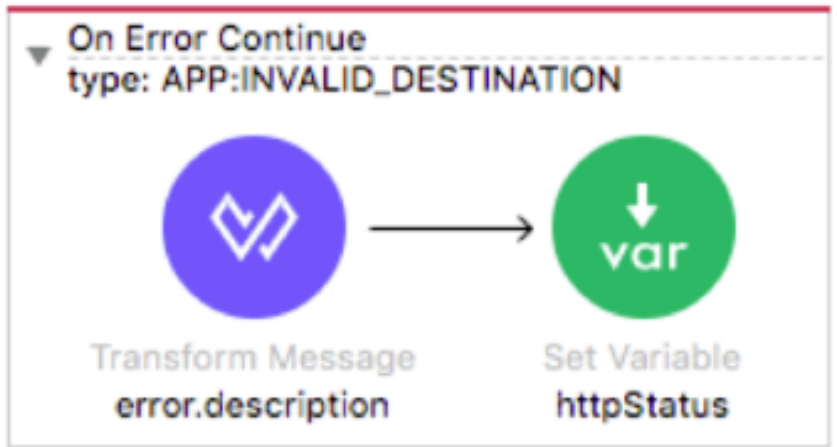
Mapping errors to custom error types

- For each module operation in a flow, each possible error type can be mapped to a custom error type
- You assign a custom namespace and identifier to distinguish them from other existing types within an application
  - Define namespaces related to the particular Mule application name or context
    - CUSTOMER namespace for errors with a customer aggregation API
    - ORDER namespace for errors with an order processing API
  - Do not use existing module namespaces

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

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

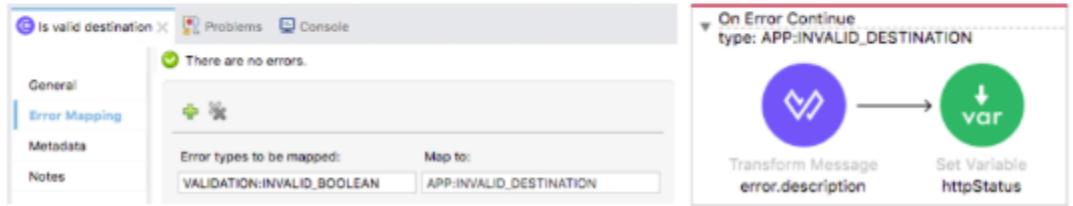




Walkthrough 10-6: Map an error to a custom error type

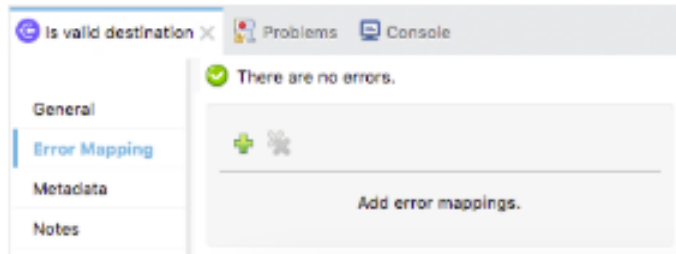
In this walkthrough, you map the Validation error to a custom error type for the application. You will:

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



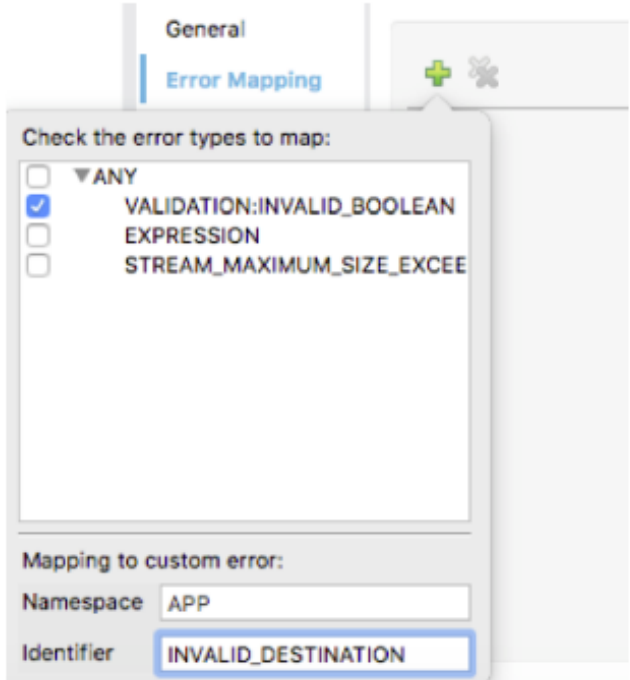
Map a validation module error to a custom error type

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.

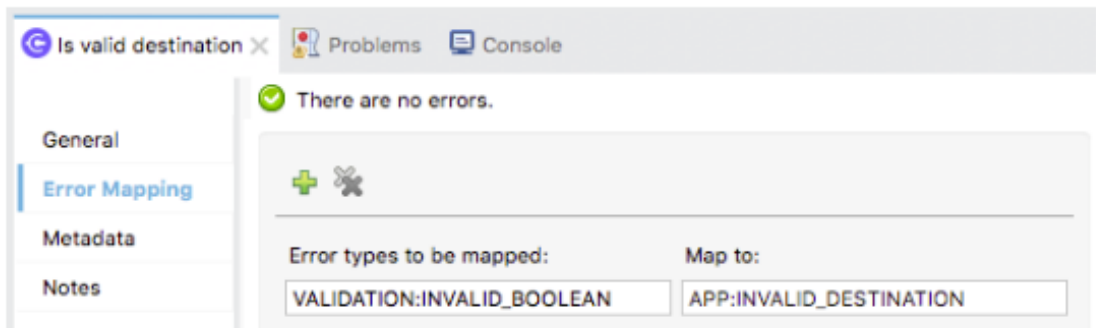


5. Select the VALIDATION:INVALID_BOOLEAN error type.
6. Leave the namespace of the custom error to map to set to APP.

7. Set the identifier to INVALID_DESTINATION.

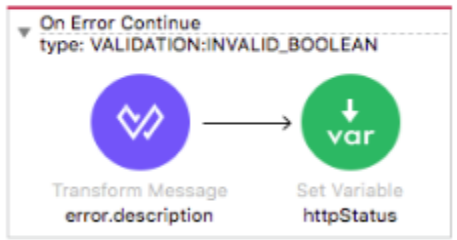


8. Press Enter; you should see your new error mapping.

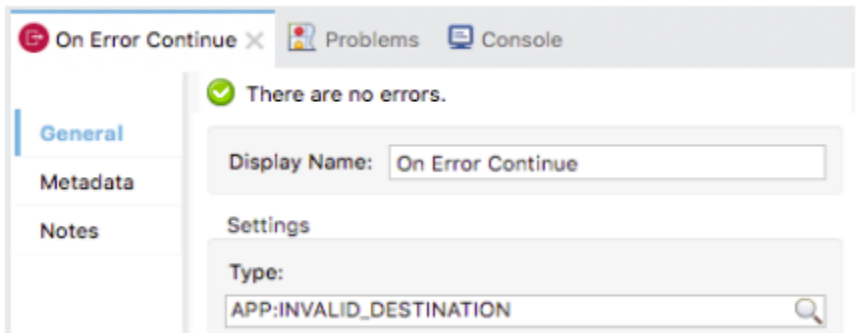


Change the existing validation error handler to catch the new custom type

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

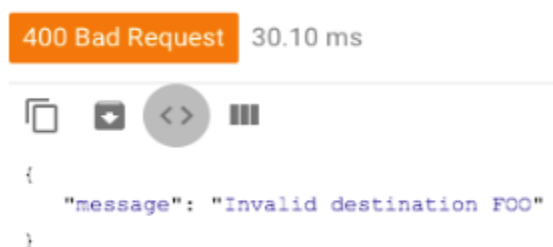


10. Change the type from VALIDATION:INVALID_BOOLEAN to the new APP:INVALID_DESTINATION that should now appear in the type drop-down menu.



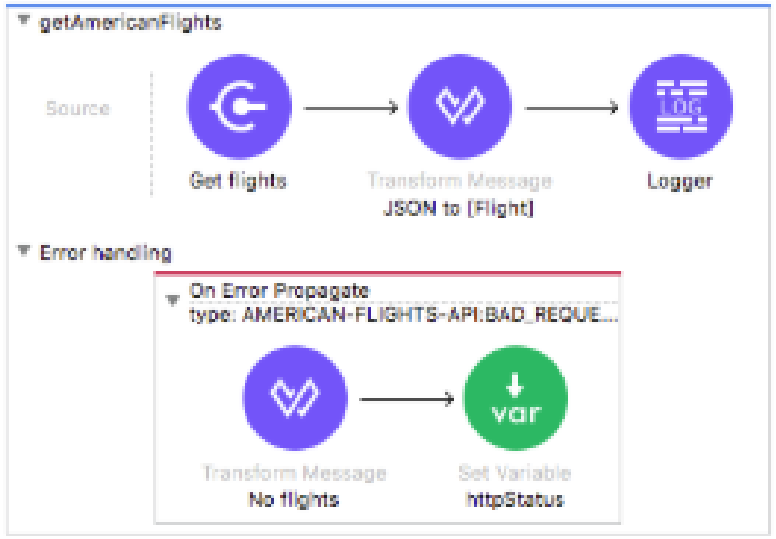
Test the application

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

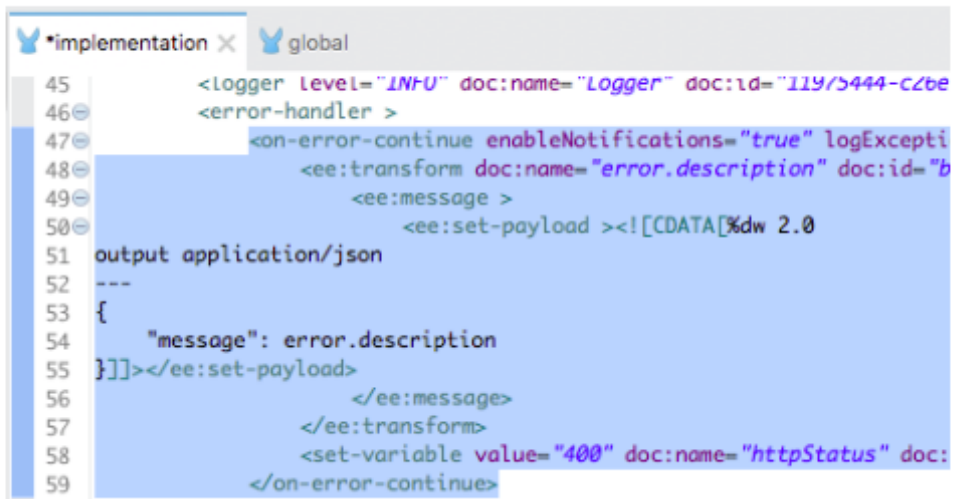
13. Collapse the getAllAirlineFlights and setCode flows.
14. Move the AMERICAN-FLIGHTS-API error scope from getFlights into getAmericanFlights.



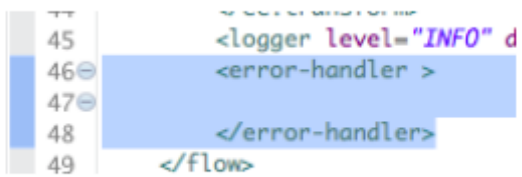
Move the error handler to the global error handler

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.

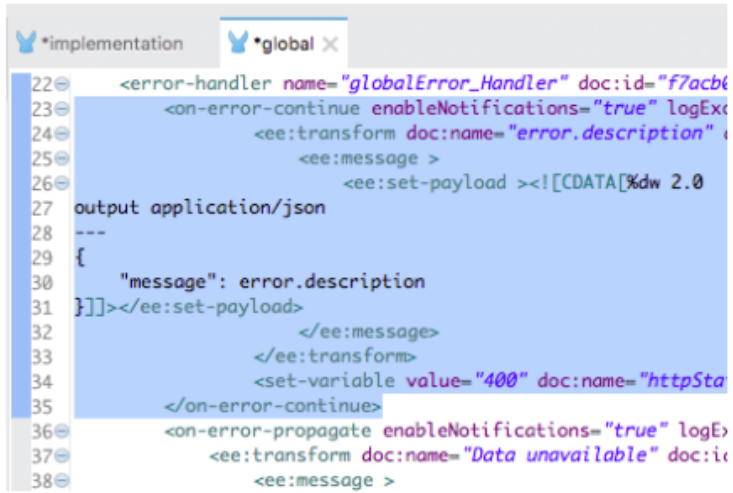


18. Delete the remaining, empty error-handler tag set.

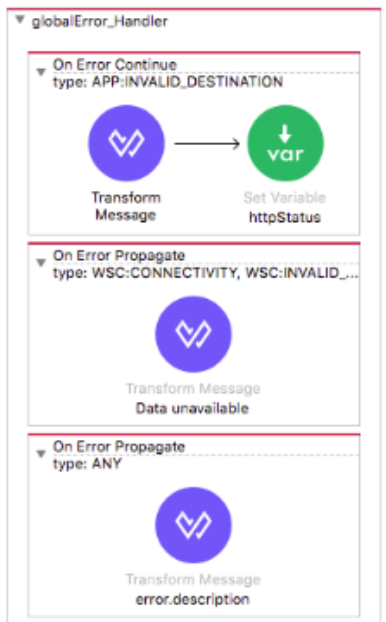


19. Return to global.xml.

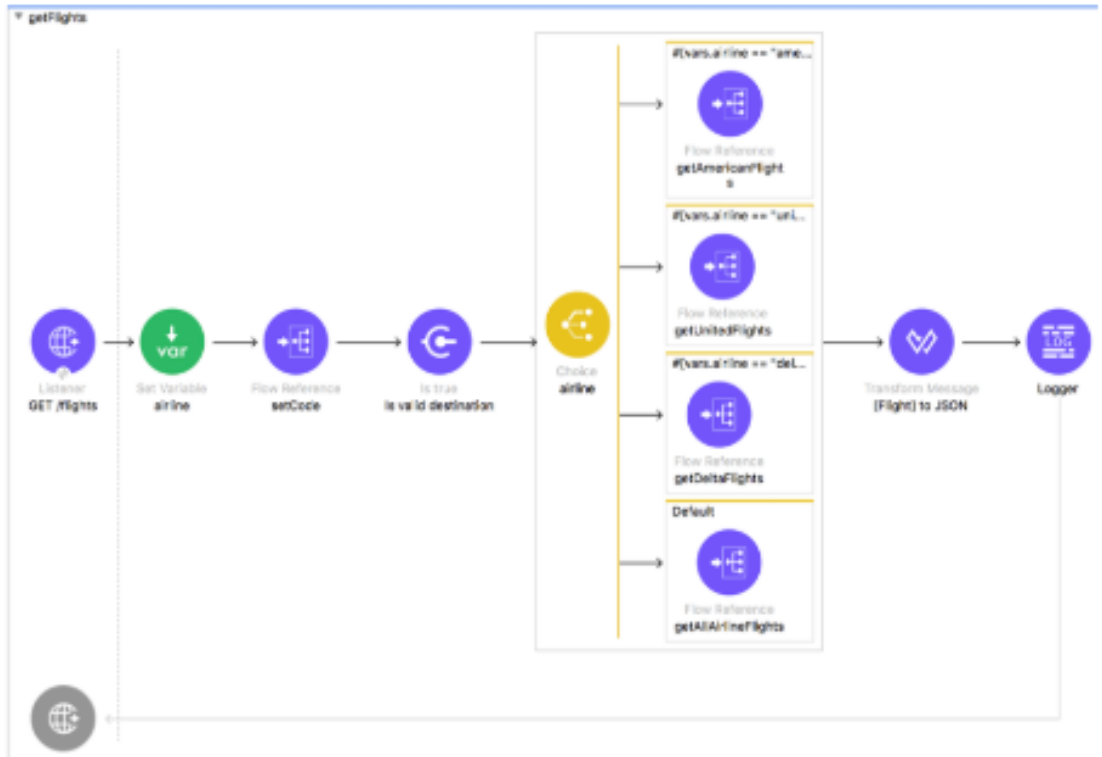
20. Right-click the globalError_Handler and select Go to XML.
21. Place the cursor on a new line inside and at the top of the error-handler.



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



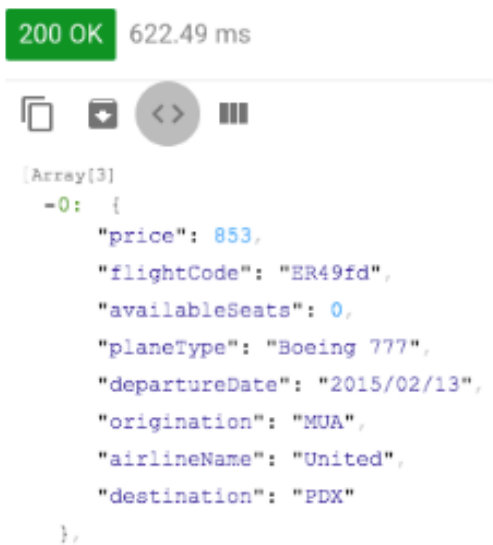
23. Return to implementation.xml and switch to the Message Flow view; you should no longer see an error handler in getFlights.



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.

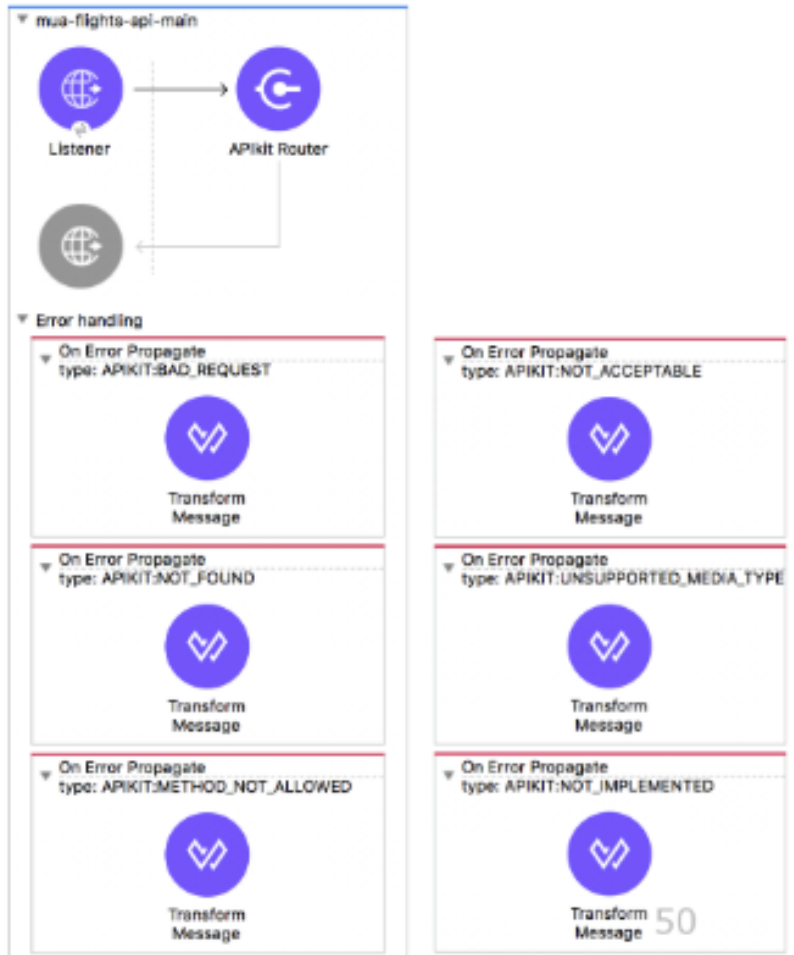


Reviewing and integrating with APIkit error handling

Error handling generated by APIkit

- By default, interfaces created with APIkit have error handlers with multiple On Error Propagate scopes that handle APIkit errors
  - The error scopes set HTTP status codes and response messages

- The main routing flow has six error scopes
  - APIKIT:BAD_REQUEST > 400
  - APIKIT:NOT_FOUND > 404
  - APIKIT:METHOD_NOT_ALLOWED > 405
  - APIKIT:NOT_ACCEPTABLE > 406
  - APIKIT:UNSUPPORTED_MEDIA_TPYE > 415
  - APIKIT:NOT_IMPLEMENTED > 501



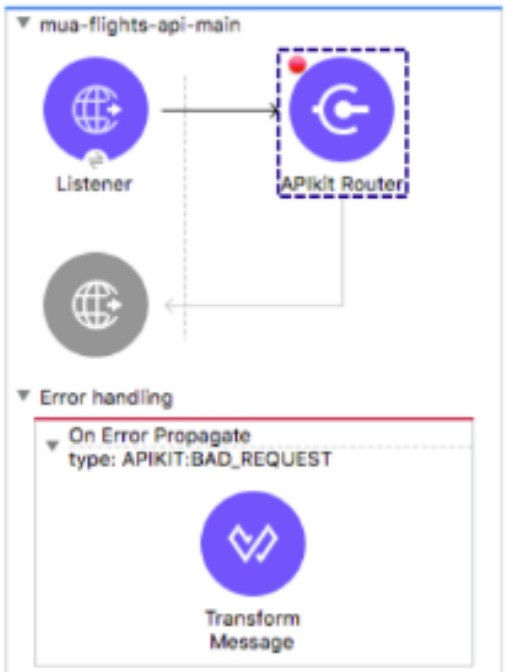
Integrating with APIkit error handling

- You can modify the APIkit error scopes and add additional scopes

- You also need to make sure the error handling in the application works as expected with the new interface router
  - **On Error Continue**
    - Event in implementation is not passed back to main router flow
  - **On Error Propagate**
    - Error is implementation is propagated to main router flow
      - Lose payload and variables

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

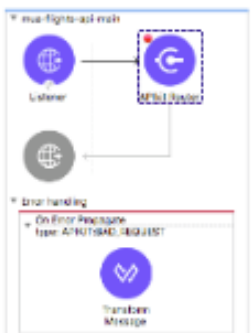
- Review the error handlers generated by APIkit
- Review settings for the APIkit Router and HTTP Listener in the APIkit generated interface
- Connect the implementation to the interface and test the error handler behavior
- Modify implementation error scopes so they work with the APIkit generated interface



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:

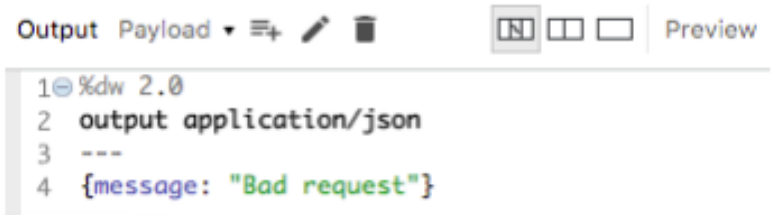
- Review the error handlers generated by APIkit.
- Review settings for the APIkit Router and HTTP Listener in the APIkit generated interface.
- Connect the implementation to the interface and test the error handling behavior.
- Modify implementation error scopes so they work with the APIkit generated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



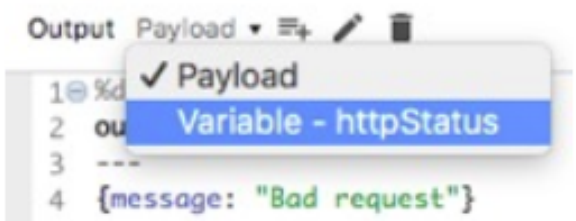
Review APIkit generated error handlers

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.



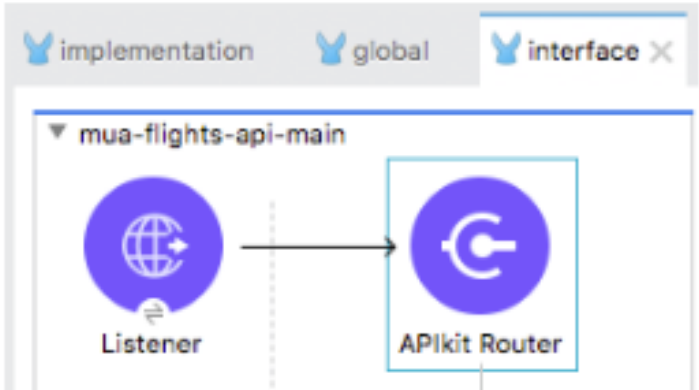
7. Use the output drop-down menu to change the output to Variable – httpStatus.



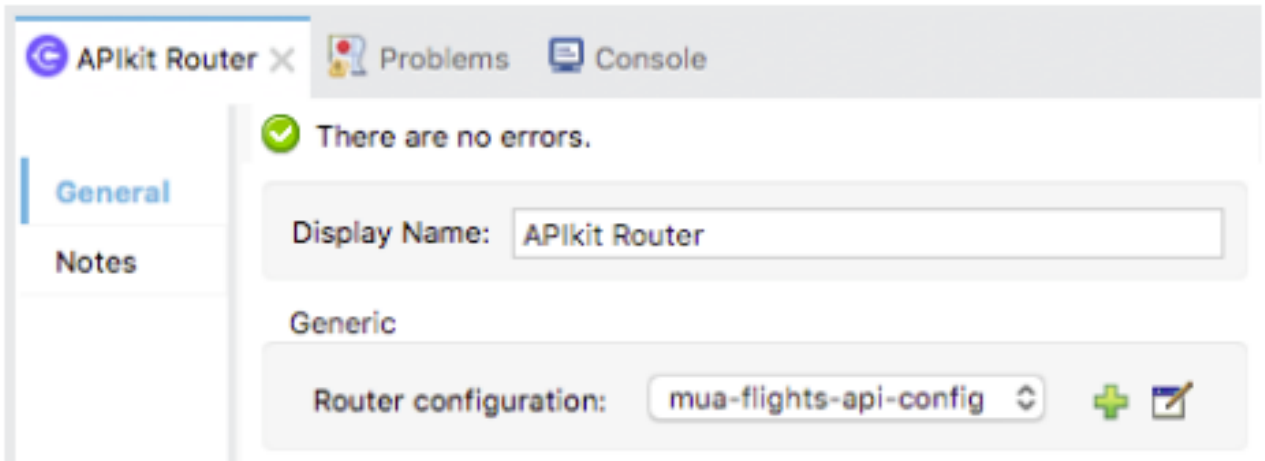
8. Review the expression that sets an httpStatus variable; you should see for a bad request that the httpStatus variable is set to 400.

Review settings for the APIkit Router

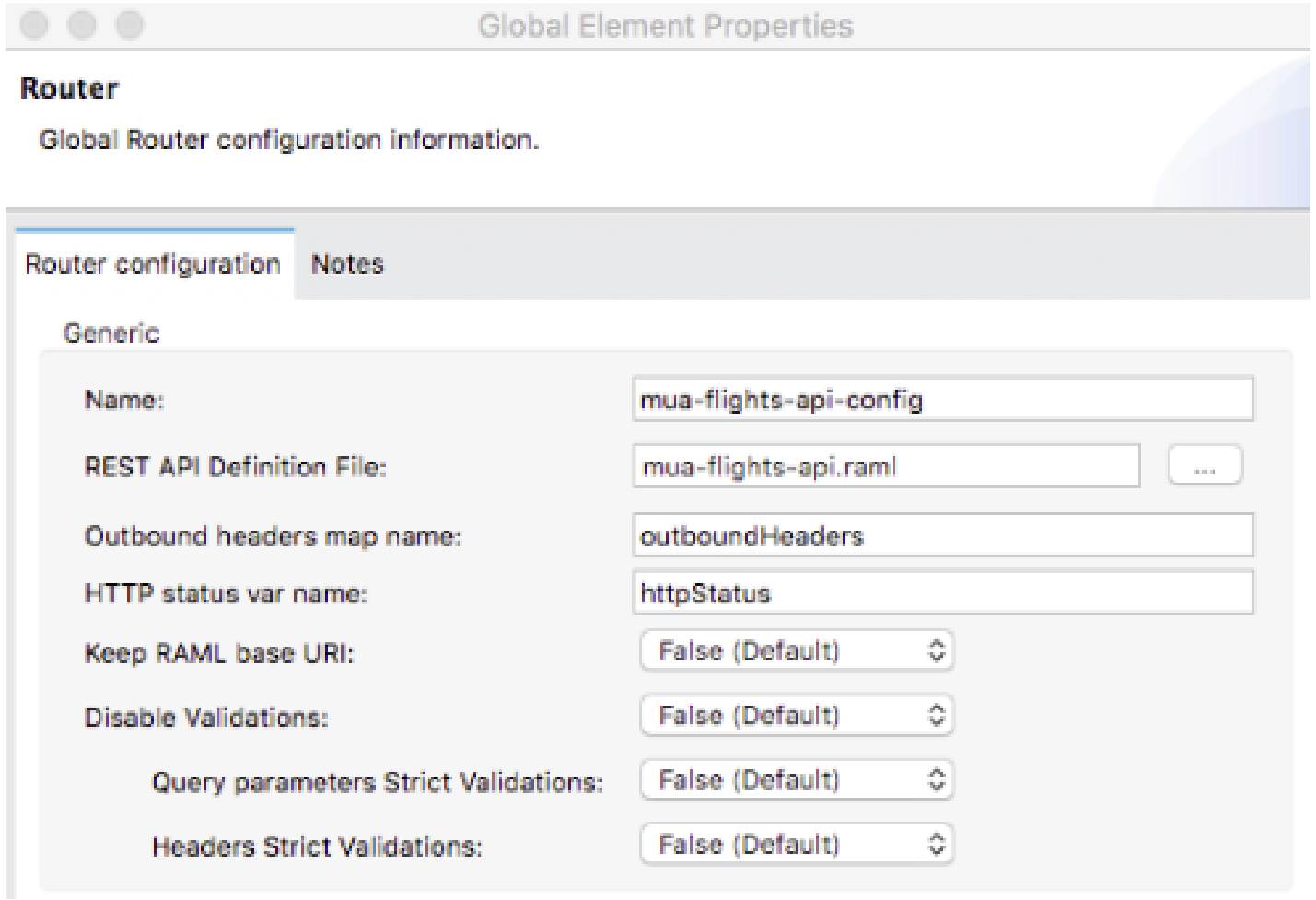
9. Navigate to the properties view for the APIkit Router in mua-flights-api-main.



10. Click the Edit button next to router configuration.



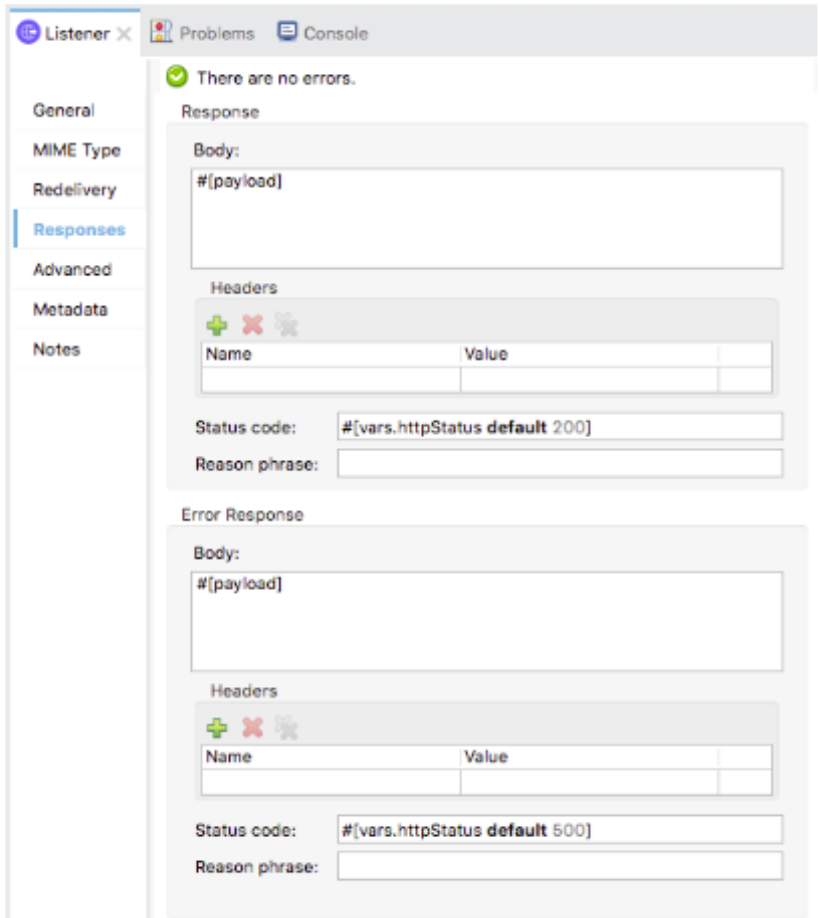
11. In the Global Element Properties dialog box, locate the HTTP status var name setting; you shoul



12. Click OK.

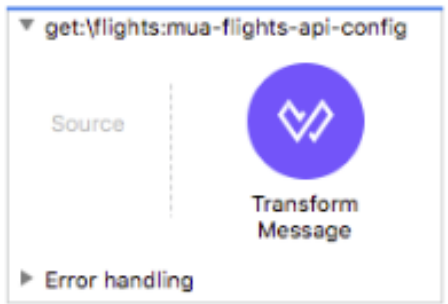
Review settings for the HTTP Listener

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.

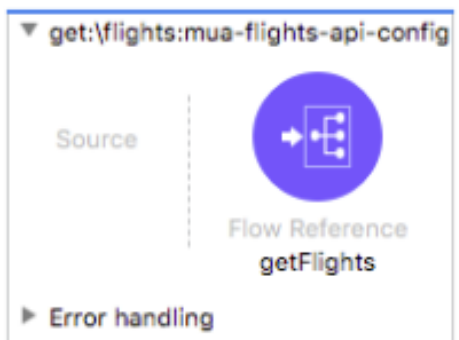


Connect the interface to the implementation

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 and display name to getFlights.



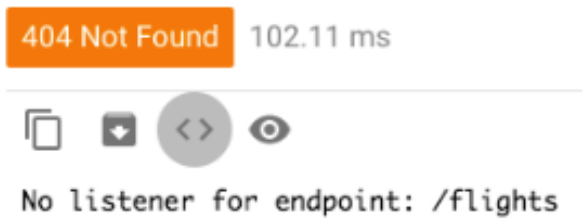
21. Return to implementation.xml.

22. Delete the GET /flights Listener in getFlights.

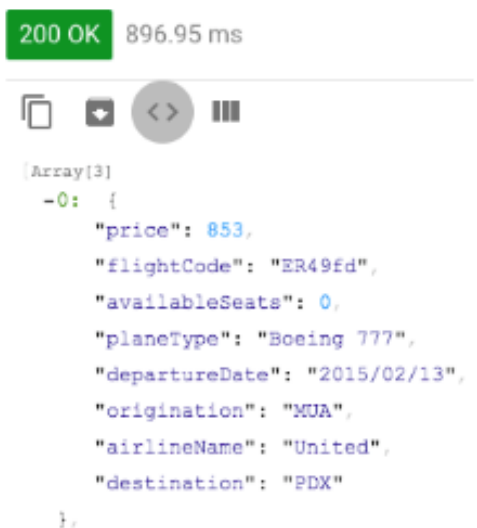
23. Save all the files to redeploy the application.

Test the application

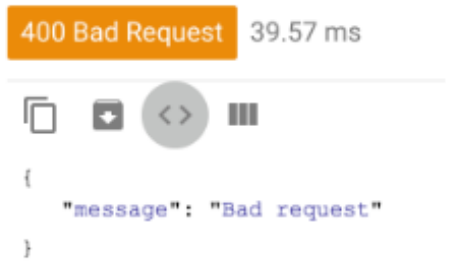
24. In Advanced REST Client, make another request to http://localhost:8081/flights?code=PDX; you should get a 404 response with a no listener message.



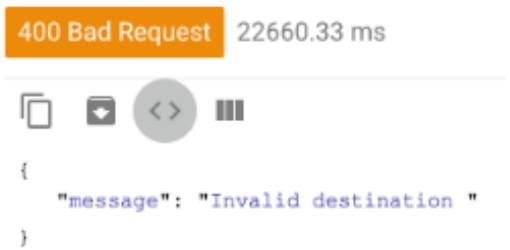
25. Add /api to the URL and make a request to http://localhost:8081/api/flights?code=PDX; you should get flights as before.



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.

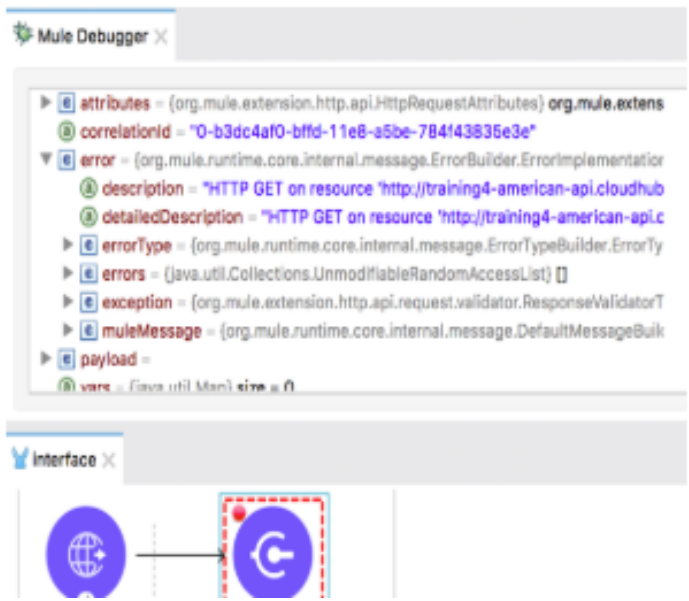
Review the API

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.

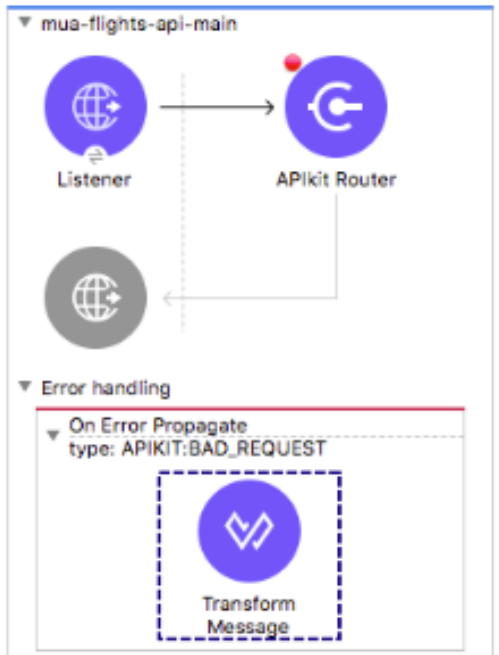
```
queryParameters:
  code:
    displayName: Destination airport code
    required: false
    enum:
      - SFO
      - LAX
      - PDX
      - CLE
      - PDF
```

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.

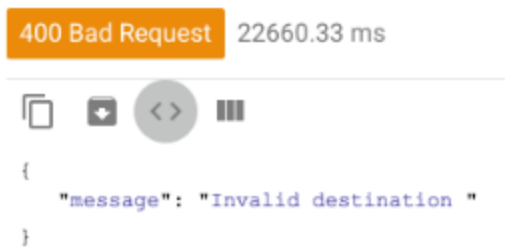


38. In Advanced REST Client, change the code to make a request to http://localhost:8081/api/flights?airline=american&code=FOO.
39. In the Mule Debugger, step through the application; the APIkit router should immediately throw an error.
40. Step again; you should see the error is handled by the that is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

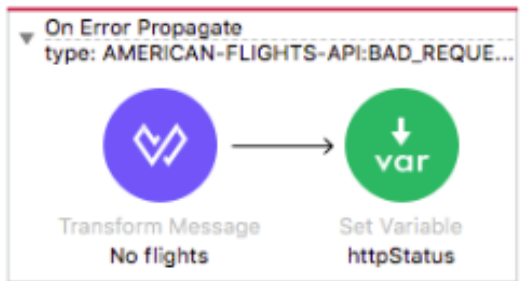
44. Return to Advanced REST Client; you should successfully get a 400 response with the custom message.



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.

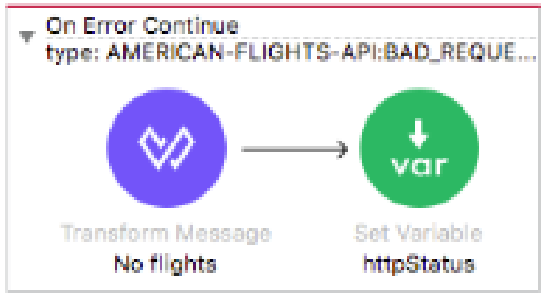


48. Right-click it and select Go To XML.
49. Change the on-error-propagate start and end tags in the getAmericaFlights flow to on-error-continue.

```
        <error-handler >
            <on-error-continue enat
                <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>
```

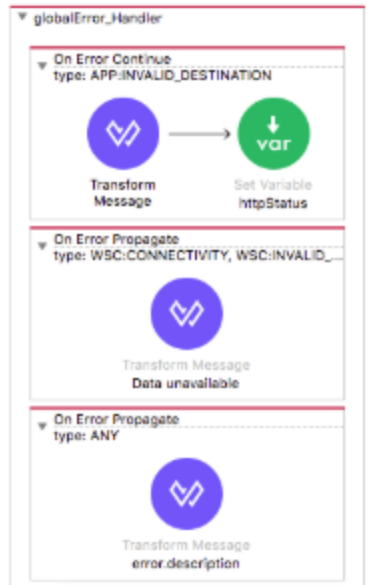
50. Change the doc:name in the start tag to On Error Continue.

51. Switch back to the Message Flow view; you should now see an On Error Continue.



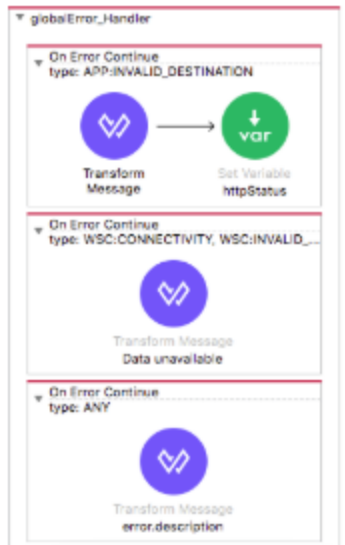
Change the global default error scopes to On Error Continue

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



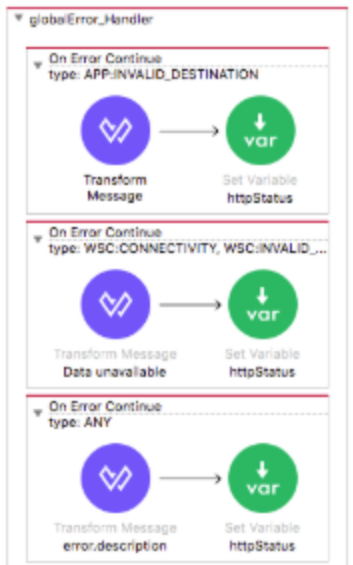
54. Right-click the WSC error handler and select Go To XML.
55. Change the four on-error-propagate start and end tags to on-error-continue.
56. Change the doc:names in both start tags to On Error Continue.

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.



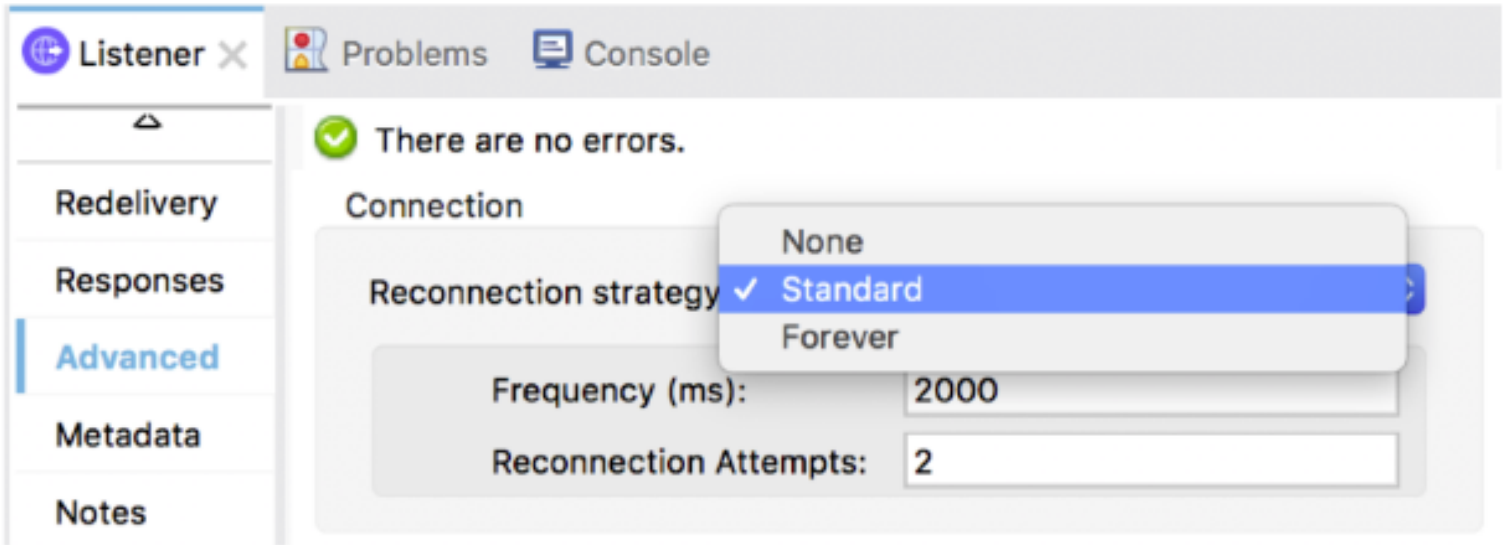
Handling system errors

Applications can have two types of errors

- Messaging errors
  - Thrown within a flow whenever a Mule event is involved
- System errors
  - Thrown at the system-level when *no* Mule event is involved
  - Errors that occur
    - During application start-up
    - When a connection to an external system fails
  - Handled by a system error handling strategy
    - Non configurable
    - Logs the error and for connection failures, executes the reconnection strategy

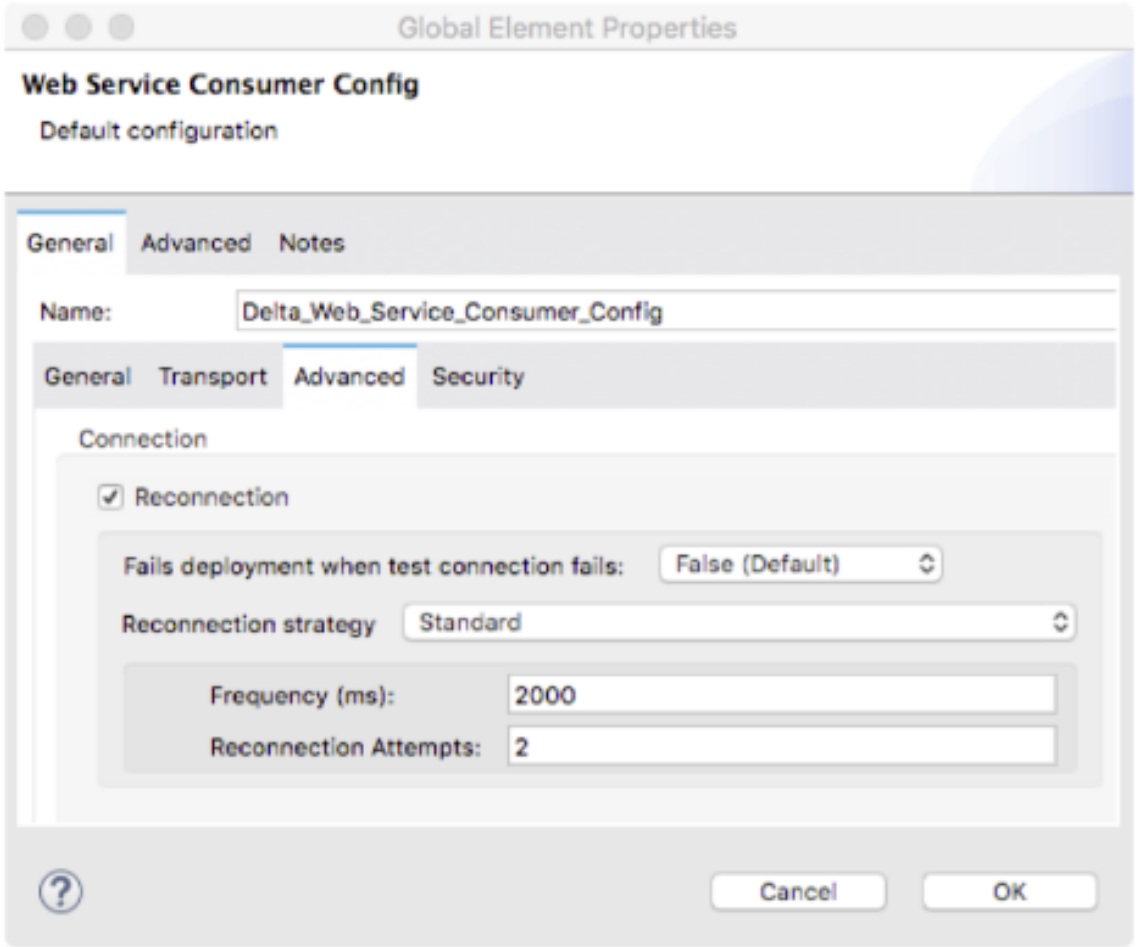
Reconnection strategies

- Set for a connector (in Global Elements Properties) or for a specific connector operation (in Properties view)



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

- Set a reconnection strategy for the Web Service Consumer connector



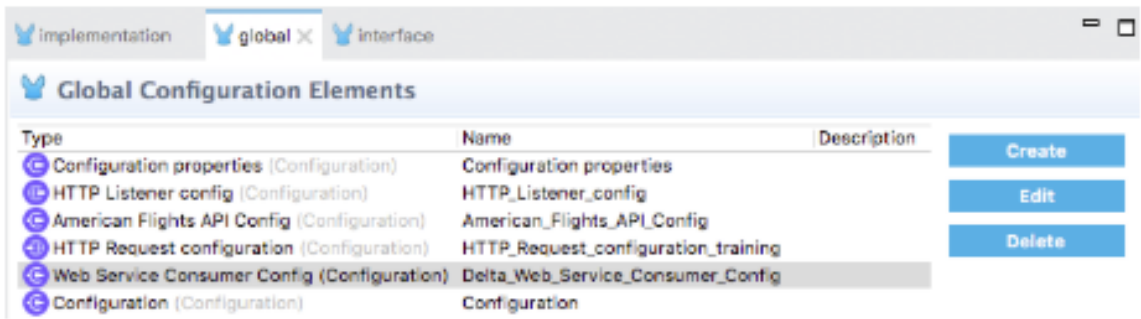
Walkthrough 10-8: Set a reconnection strategy for a connector

In this walkthrough, you will:

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

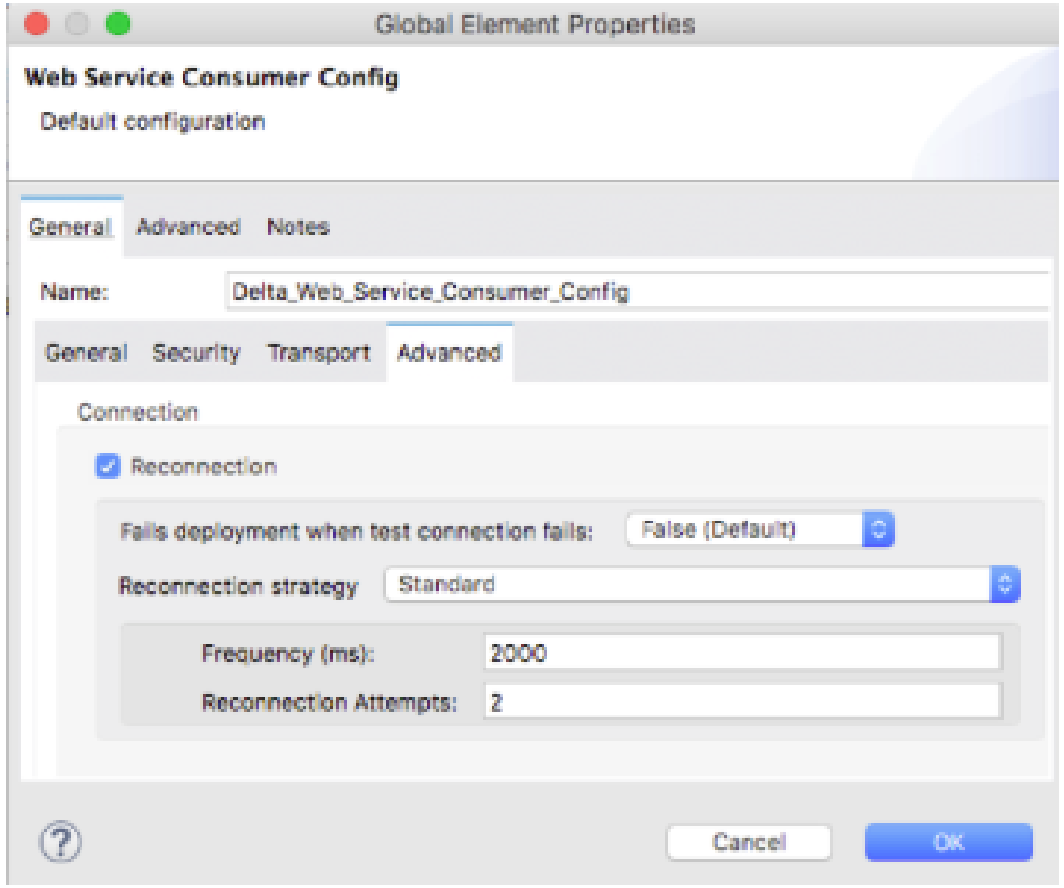
Set a reconnection strategy for a connector

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.

Summary

Summary

- An application can have system or messaging errors
- **System errors** are thrown at the system level and involve no event
  - Occur during application start-up or when a connection to an external system fails
  - Non-configurable, but logs the error and for connections, executes any reconnection strategy
- **Messaging errors** are thrown when a problem occurs within a flow
  - Normal flow execution stops and the event is passed to an error handler (if one is defined)
  - By default, unhandled errors are logged and propagated
  - HTTP Listeners return success or error responses depending upon how the error is handled
  - Subflows cannot have their own error handlers

Summary

- Messaging errors can be handled at various levels
  - For an **application**, by defining an error handler outside any flow and then configuring the application to use it as the default error handler
  - For a **flow**, by adding error scopes to the error handling section
  - For one or more **processors**, by encapsulating them in a Try scope that has its own error handling section
- Each error handler can have one or more error scopes
  - Each specifies for what error type or condition for which it should be executed
- An error is handled by the first error scope with a matching condition
  - **On Error Propagate** rethrows the error up the execution chain
  - **On Error Continue** handles the error and then continues execution of the parent flow

Summary

- Error types for module operations can be mapped to **custom error types**
  - You assign a custom namespace and identifier to distinguish them from other existing types within an application
  - Enables you to differentiate exactly where an error occurred, which is especially useful when examining logs
- By default, interfaces created with **APIkit** have error handlers with multiple On Error Propagate scopes that handle APIkit errors
  - The error scopes set HTTP status codes and response messages
  - You can modify these error scopes and add additional scopes
  - Use On Error Continue in implementation to not pass event back to main router
  - Use On Error Propagate in implementation to propagate error to main router flow

DIY Exercise 10-1: Handle errors

Time estimate: 3 hours

Objectives

In this exercise, you handle errors in a Mule application. You will:

- Add a global default error handler to a Mule application.
- Compare the difference between On Error Propagate and On Error Continue scopes.
- Reference a global error handler from a flow.
- Add a sequence of event processors to a Try scope.
- Set match conditions in error scopes.
- Create custom error mappings.

Scenario

A Mule application retrieves flight data from various flight services. You need to add robust error handling to the Mule application.

Import the starting project

Import /files/module10/flights-mod10-error-handling-starter.jar (in the MUFundamentals_DIYexercises.zip that you can download from the Course Resources) into Anypoint Studio.

Review the flows in the starting project

Review the structure of the gatherdata.xml and standardize.xml files:

- The Mule application has only one endpoint and expects a URI parameter named destination and a query parameter named maxPrice.
- Requests are received by the flights flow in gatherdata.xml.
- This flow uses a Scatter-Gather to concurrently retrieve flight data.
- In the standardize.xml file, getUnited uses a filter to filter invalid payloads returned if the United service is given an invalid destination.
- In the standardize.xml file, getDelta uses DataWeave to filter null payloads returned if the Delta service is given an invalid destination.

Verify the default out-of-the-box error handling behavior

Run the project and make a GET request to http://locahost:8081/flights/MUA?maxprice=1000 and verify that an error is thrown. Debug the Mule application, step through the flows, and note how the error is handled.

Answer the following questions

- What type of error is logged by the default error handler?
- What is the error status code?
- What type of error scope is used by the default error handler (On Error Propagate or On Error Continue)?

Create a global default error handler

In global.xml, add a global error handler with an On Error Propagate scope to catch any error thrown in the Mule application. Inside the handler, log a message that includes the error's namespace and identifier.

Make the same GET request, step through the flows, and note how the error is handled.

Answer the following questions

- How many errors are thrown?
- What error is thrown first?
- Why are there different types of errors thrown?

Add more details to the HTTP Listener error response

In gatherdata.xml, set the error response in the HTTP Listener to the following data structure:

```
#[output application/json ---
 {
  errorObject: error.errorType,
  errorNamespace: error.errorType.namespace,
  errorID: error.errorType.identifier,
  description: error.description
 }
]
```

Add an error handler to a flow

In standardize.xml, add error scopes to the getAmerican flow to handle errors in the following order:

- For HTTP:BAD_REQUEST errors, use an On Error Continue scope and return an empty array.
- For DataWeave transformation errors, use an On Error Propagate scope.
- For all other errors, use an On Error Propagate scope.

Add Loggers to each error scope to identify what error type was thrown. Make the same GET request, step through the flows, and note how the error is handled.

Answer the following questions

- What is the result of the GET request?
- What is the status code returned from the GET request?
- Did the Logger in getAmerican get executed? What is the reason for this behavior?

Create a flow level error handler

Extract the error scopes you defined in the getAmerican flow into a standalone error handler inside standardize.xml. Name the error handler standardize-error-handler.

Add an error handler reference for the getAmerican, getUnited, and getDelta flows to reference the common error handler in standardize.xml.

Note: In Anypoint Studio version 7.1.4, there is no error handler reference in the Mule Palette. You need to directly reference the error handler using the following XML code:

```
<error-handler ref="standardize-error-handler"/>
```

Catch web service errors in a Try scope

In the getDelta flow, add a Try scope after the Build SOAP Request Transform Message component. Move the Web Service Consume operation and the Logger into the Try scope. To intentionally throw an error in the Consume operation, change the SOAP operation from findFlight to findFlights.

Add error scopes to the Try scope in the following order:

- For findFlights operation errors, use an On Error Propagate scope.
- For WSC:BAD_REQUEST, WSC:BAD_RESPONSE, WSC:RETRY_EXHAUSTED, and WSC:TIMEOUT errors, use an On Error Continue scope and return an empty object.
- For all other WSC errors, use an On Error Propagate scope.

Add Loggers to each error scope to identify what error type was thrown. Make the same GET request, step through the flows, and note how the error is handled.

Make a GET request to http://localhost:8081/flights/SFO?maxprice=1000 and step through the flows and observe the behavior.

Catch web service errors in a Try scope

In the getDelta flow, add a Try scope after the Build SOAP Request Transform Message component. Move the Web Service Consume operation and the Logger into the Try scope. To intentionally throw an error in the Consume operation, change the SOAP operation from findFlight to findFlights.

Add error scopes to the Try scope in the following order:

- For findFlights operation errors, use an On Error Propagate scope.
- For WSC:BAD_REQUEST, WSC:BAD_RESPONSE, WSC:RETRY_EXHAUSTED, and WSC:TIMEOUT errors, use an On Error Continue scope and return an empty object.
- For all other WSC errors, use an On Error Propagate scope.

Add Loggers to each error scope to identify what error type was thrown. Make the same GET request, step through the flows, and note how the error is handled.

Make a GET request to http://localhost:8081/flights/SFO?maxprice=1000 and step through the flows and observe the behavior.

Answer the following questions

- What is the result of the GET request?
- What is the status code returned from the GET request?
- Which error handler caught the bad SOAP request?
- What happens to the request if the DELTA:BAD_WSC_REQUEST error scope is changed to an On Error Continue scope?

Verify your solution

Load the solution /files/module10/flights-mod10-error-handling-solution.jar (in the MUFundamentals_DIYexercises.zip that you can download from the Course Resources) and compare your solution.