Anypoint Platform Development: DataWeave 2

Student Manual

October 30, 2020
Mule runtime 4.3

Table of Contents

Module 1: Applying DataWeave Fundamentals


```
readUrl("classpath://examples/flightsJoinedWithAirports.xml",
"application/xml")..*flight
//distinctBy $."airline-name"
filter $."available-seats" as Number > 0 and $.price as Number < 600
//orderBy -$.price orderBy $."airline-name"
//orderBy $."airline-name" orderBy -$.price
orderBy ( $."airline-name" ++ (10000 - $.price ) )
groupBy upper($."airline-name")
```

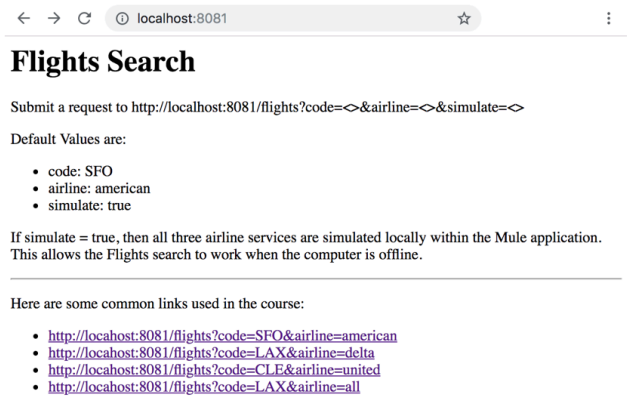
At the end of this module, you should be able to:

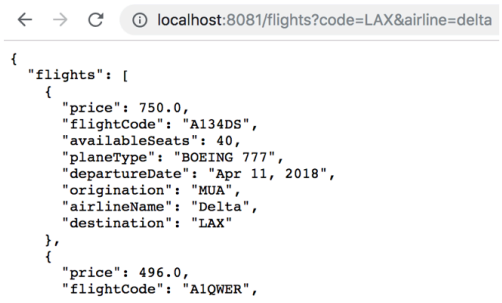
- Apply DataWeave fundamentals as learned in the *Development Fundamentals* course.
- Configure input and output metadata to inform DataSense in Anypoint Studio.
- Set example input to preview DataWeave results in Anypoint Studio.

Walkthrough 1-1: Import a basic DataWeave based Mule project into Anypoint Studio

In this walkthrough, you load a starter Mule project into Anypoint Studio to feed in sample JSON data into a Transform Message component. This template project also defines input and output metadata both in the Transform Message components and in other message processors. You will also create some sample JSON data files you will use in later walkthroughs. You will:

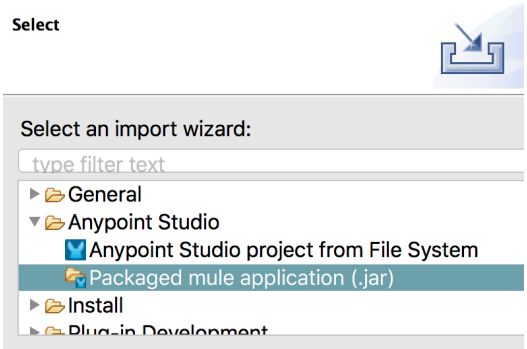
- Verify DataWeave is working correctly in an Anypoint Studio installation.
- Create example JSON files that can be used to mock input to other DataWeave expressions.





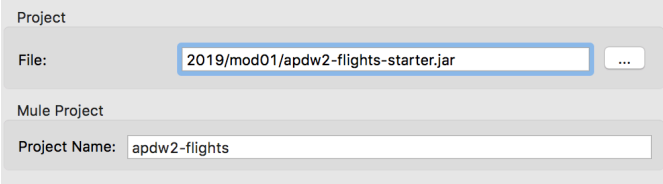
Import a starter Mule project into Anypoint Studio

1. Launch Anypoint Studio.
2. In Anypoint Studio, select File > Import.
3. Select Packaged mule application (.jar).

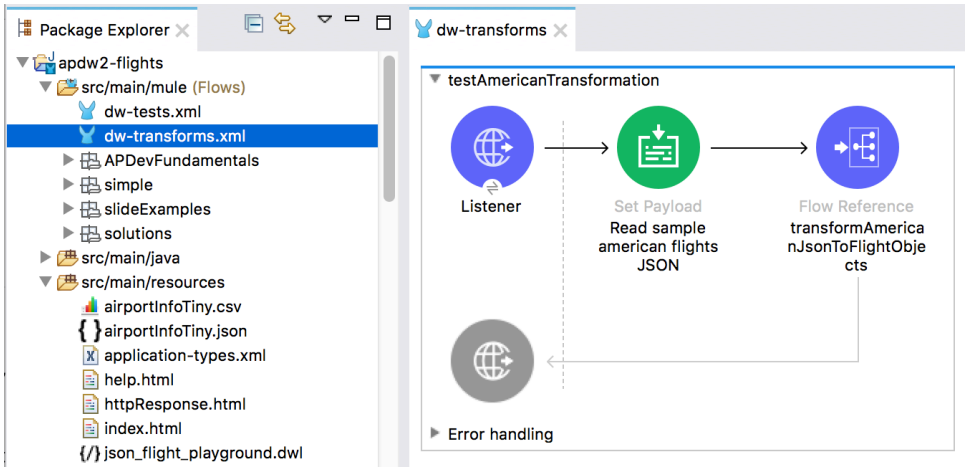


4. Browse to your %studentFiles%\mod01 folder, then select apdw2-flights-starter.jar.

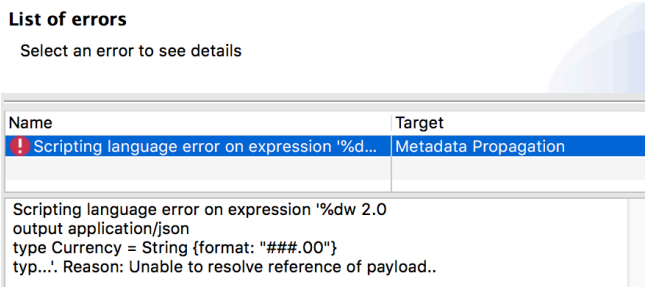
5. Rename the project to apdw2-flights.



6. Verify the project successfully loads into Anypoint Studio.



Note: You may see a Metadata Propagation error reported in the dw-transforms.xml file saying Unable to resolve reference of payload. You can ignore this error for now, and you will fix it in the next walkthrough.



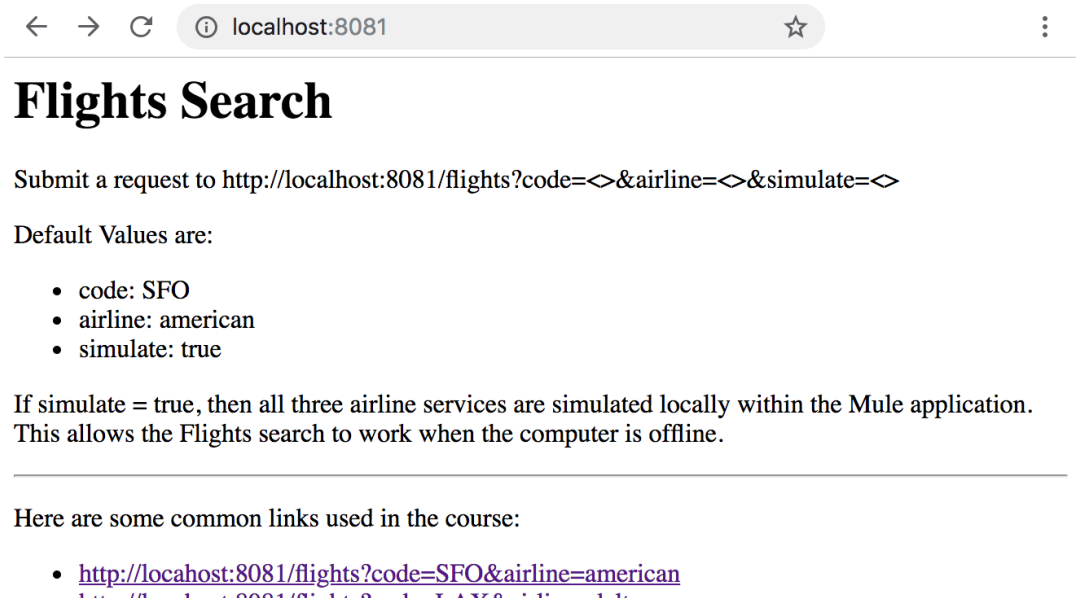
Verify the project runs correctly

7. Run or debug the apdw2-flights project.

Note: You can right-click the canvas for any of the Mule configuration files, then select Debug project apdw-flights.

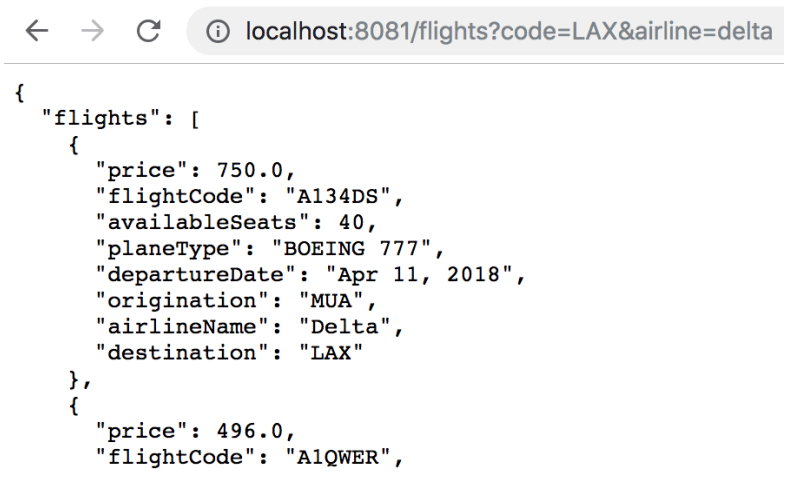
Note: You can ignore any errors about processing the WSDL file and continue to start the debugger. The Mule application has an option to use the same online services used in the Anypoint Platform Development: Fundamentals course, or you can use a local version of the flight services that do not require internet access.

8. In a web client (such as a web browser or the Advanced Rest Client), navigate to http://localhost:8081.
9. Verify a help page is returned to the web client.



Submit some sample requests to the flights service Mule application

10. Select one of the links on the help page.
11. Verify a JSON result is returned to your web client with a flights object containing an array of individual flight objects.



12. Submit a request with no query parameters to the flights service endpoint URL:

    `http://localhost:8081/flights`

13. Verify a response is returned with American Airlines flights to SFO.
14. Submit a request that calls real external web services by setting the simulate query parameter to false in the service endpoint URL:

    `http://localhost:8081/flights?simulate=false&code=LAX&airline=all`

    Note: If you do not have open connectivity to some of the web services, you will get errors in the Anypoint Studio Console view.

15. Submit a request for flights to LAX on all airlines:

    `http://localhost:8081/flights?code=LAX&airline=all`

16. Verify a response is returned with all three airlines to the destination LAX, and that some flights have availableSeats=0.

Create some example flight response JSON files in the Mule project

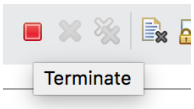
17. Copy the last response JSON to the clipboard.
18. Return to Anypoint Studio.
19. Inside src/main/resources/example paste the copied clipboard text into a new file named flightsAllAirlinesLAX.json.

(Optional) Create similar example JSON response files for the other destination codes

20. Repeat the steps to copy the response from http://localhost:8081/flights?code=SFO&airline=all to a new in src/main/resources/examples named flightsAllAirlinesSFO.json.
21. Repeat the steps to copy the response from http://localhost:8081/flights?code=PDX&airline=all to a new in src/main/resources/examples named flightsAllAirlinesPDX.json.
22. Repeat the steps to copy the response from http://localhost:8081/flights?code=CLE&airline=all to a new in src/main/resources/examples named flightsAllAirlinesCLE.json.

Stop running or debugging the Mule application

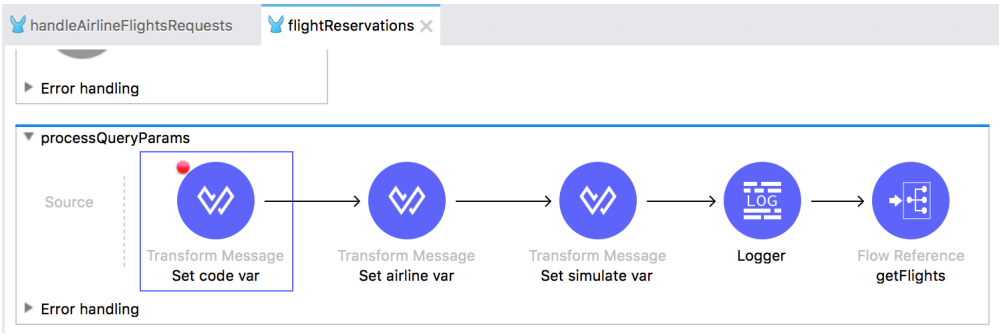
23. In the Console view, click the Terminate button.



Note: You can also right-click in the canvas to stop the Mule application.

(Optional) Review the provided DataWeave code

24. Open src/main/mule/APDevFundamentals/flightReservations.xml.
25. In the processQueryParams flow, set a breakpoint on the first Set code var Transform Message component.



Note: Ignore any invalid input errors that appear in either of the Transform Message components.

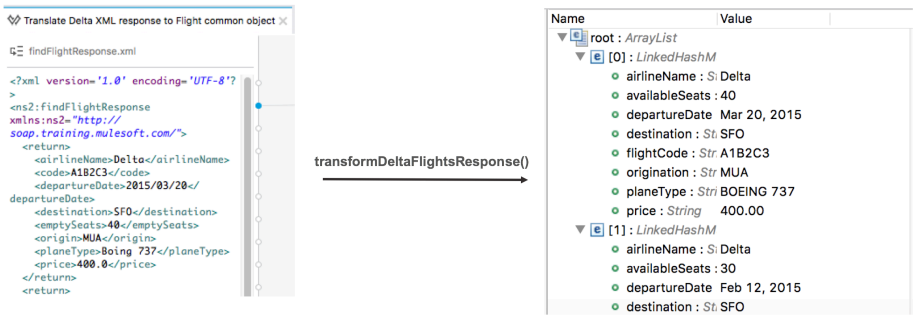
26. View the DataWeave expression for the next Set airline var Transform Message component.
27. Step through the flows in the Mule application and look at the DataWeave code in each Transform Message component.

Note: If you took the Anypoint Platform Development: Fundamentals course, you should be familiar with most of the DataWeave code used in the starter project.

Walkthrough 1-2: Configure metadata and example input for a DataWeave transformation

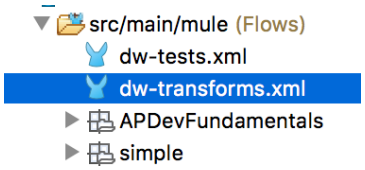
In this walkthrough, you define metadata types and corresponding examples to help you use the Preview pane to write DataWeave expressions. You will:

- Use the Anypoint Studio debugger to capture valid input data at a breakpoint in a flow.
- Assign a metadata type and example data as input to a Transform Message component.
- Read in example data from a sample file into a DataWeave expression.
- Use the Preview pane to view how sample input data is transformed by DataWeave expressions.

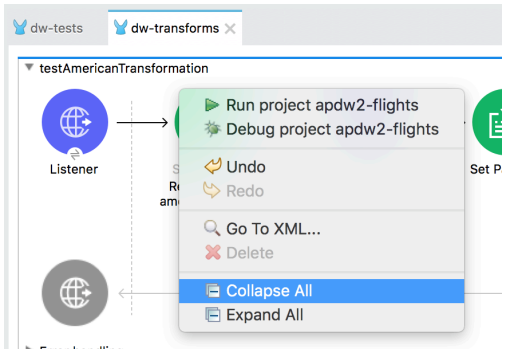


Set breakpoints to capture input data

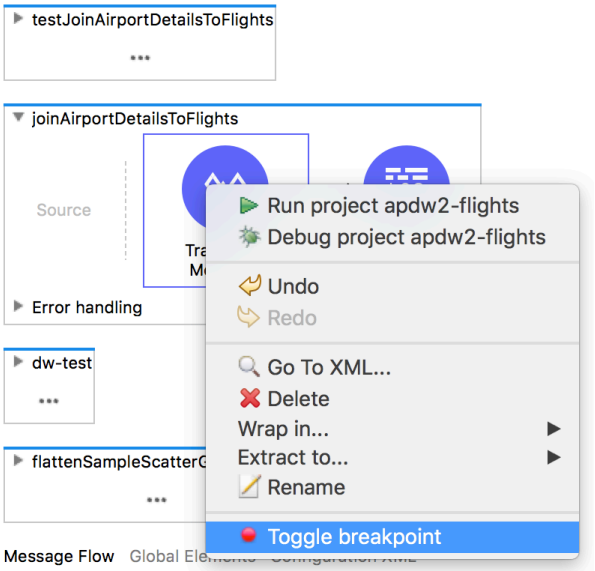
1. Return to Anypoint Studio.
2. In the apdw2-flights project, open src/main/mule/dw-transforms.xml.



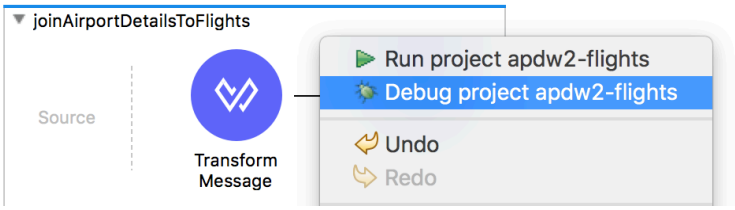
3. Right-click on the canvas and collapse all the flows.



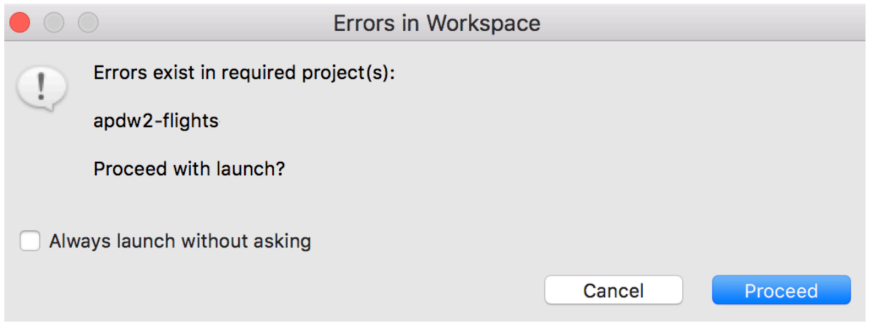
4. Expand the joinAirportDetailsToFlights flow; this the main flow you will be working with to build up the case study for the class.
5. Set a breakpoint on the Transform Message component.



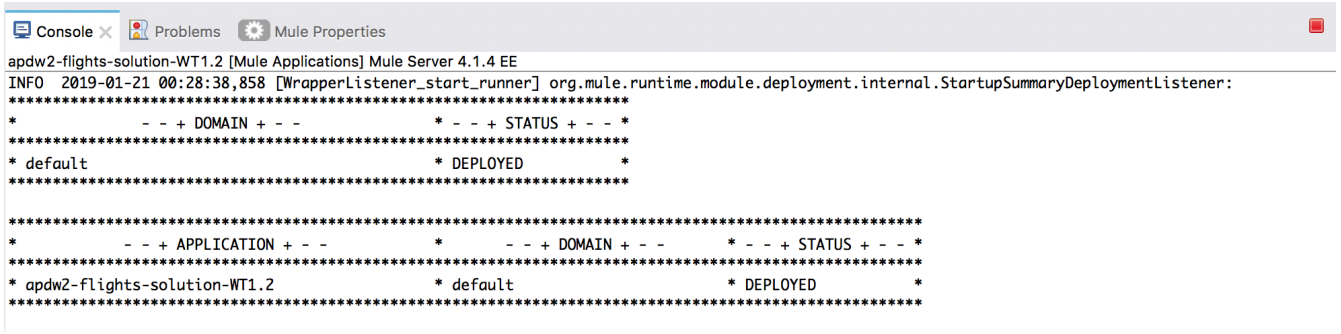
6. Right-click in the canvas and select Debug project apdw2-flights.



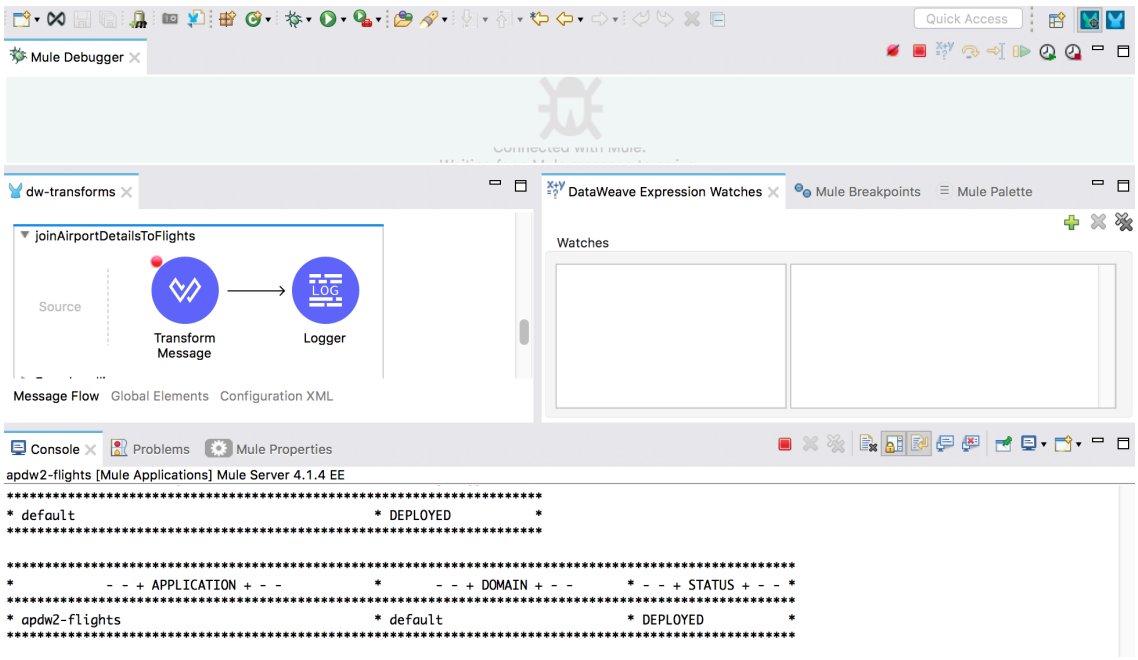
Note: If there is an error reported in the workspace, click Proceed to debug anyway.



7. Select the Console view and wait for the debugger to start successfully.

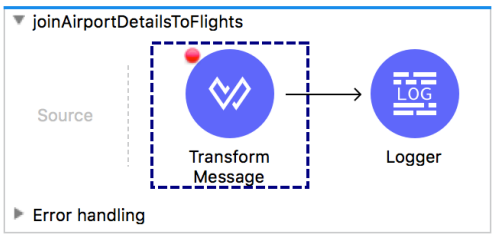


8. Make sure Anypoint Studio switches to the Debug perspective.



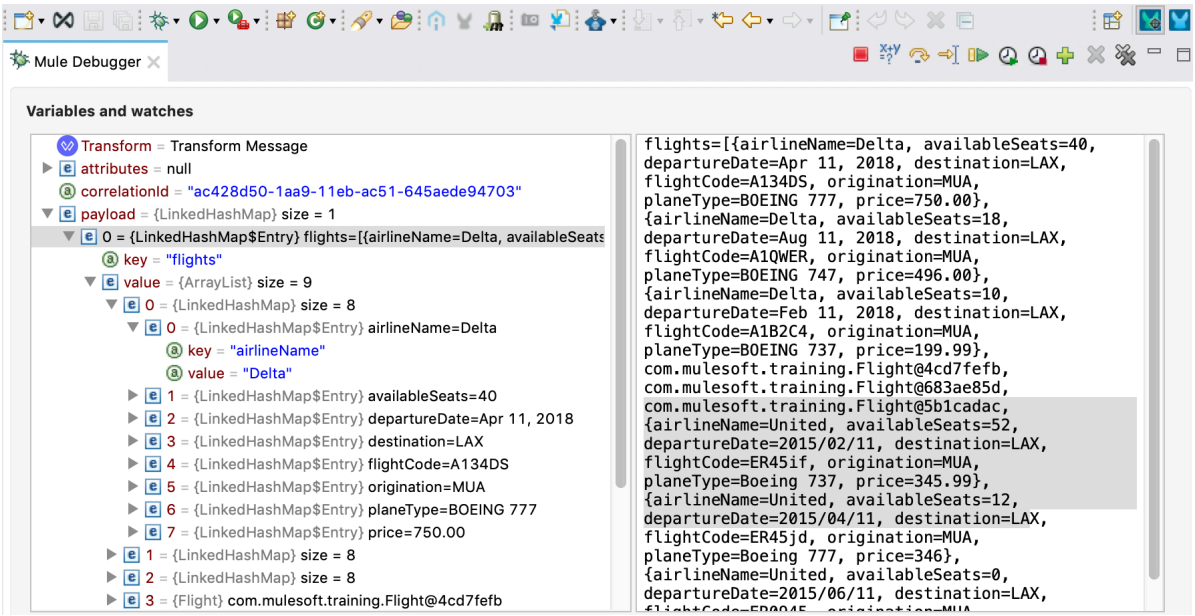
Submit a request to search for flights to all airlines to LAX

9. In a web client, submit a request to http://localhost:8081/flights?code=LAX&airline=all.
10. Verify the debugger stops at the Transform Message component in the joinAirportDetailsToFlights flow.



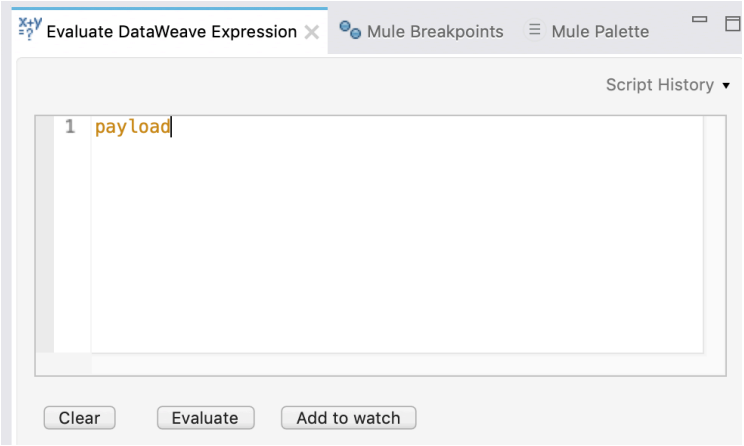
Inspect and manipulate the Mule event at the breakpoint

11. In the Mule Debugger view, select Payload.
12. Expand the flights object and the first child flight object.



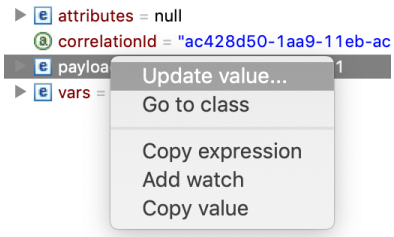
Note: If the Mule Debugger view is empty, step through the flow and submit another request from the web client.

13. In the Evaluate DataWeave Expression view, type the value payload.flights.



14. Click Evaluate; below you should should see nine flight objects returned of type LinkedHashMap.
15. Click Add to watch; the expression should be added to the Variables and watches view in the upper left.
16. Select the new watch expression and expand the flights object and then the first child flight object; the result should be the same as what displays in the Mule Debugger view.

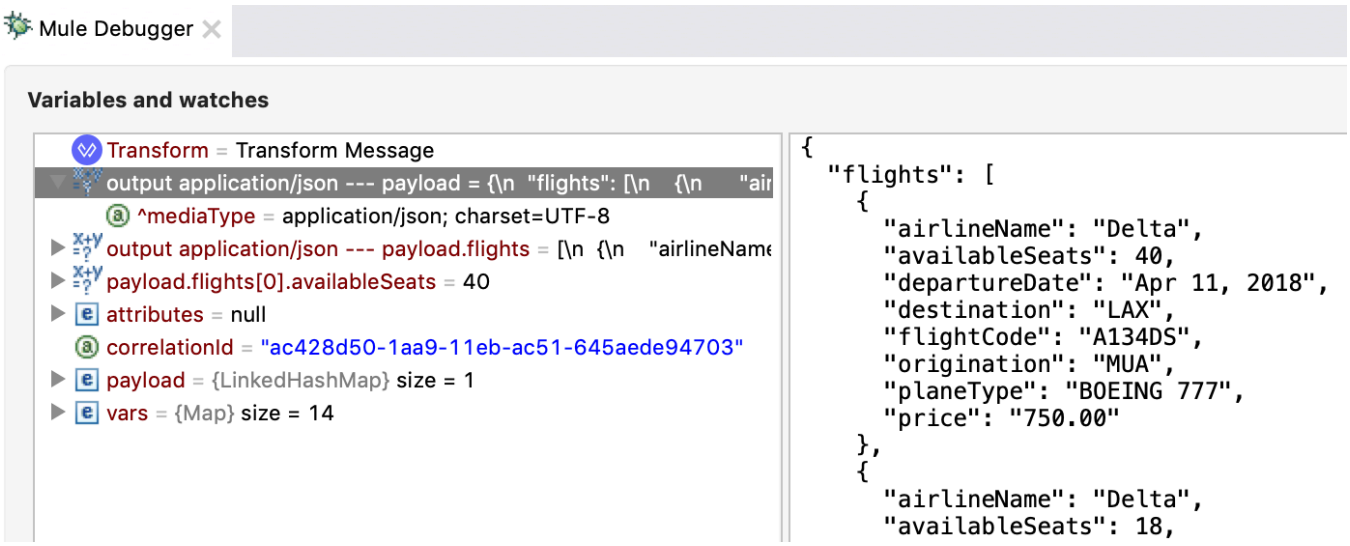
17. Right-click the payload Watch expression and select Update expression.



18. In the Edit expression dialog, change the value to:

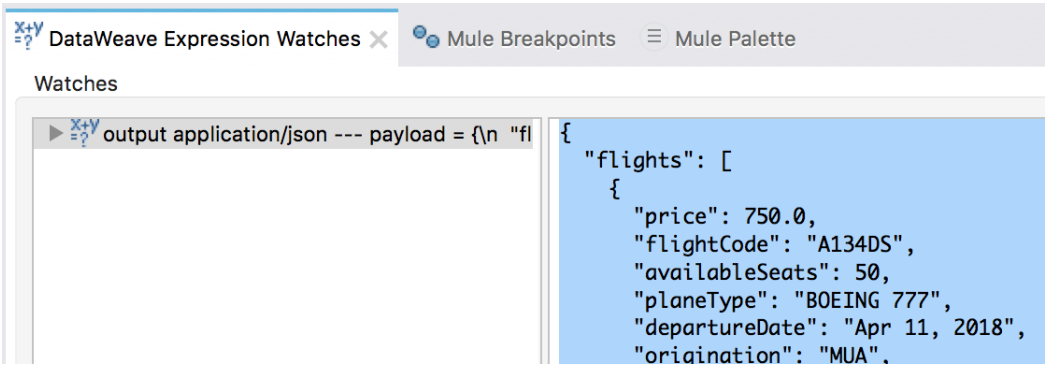
```
output application/json --- payload
```

19. Click OK.
20. Select the new watch expression; in the right side you should see the flights array in JSON format.

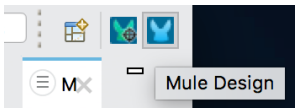


Copy a JSON version of the current payload to a sample file

21. In the Variables and watches output pane, select all the JSON output and copy it to the clipboard.



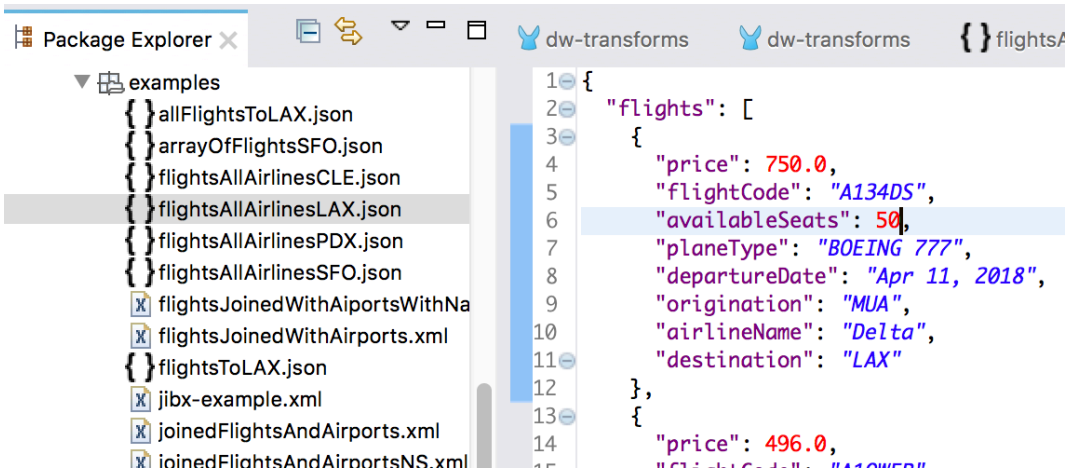
22. In the upper-right, select the Mule Design perspective icon.



23. In src/main/resources/examples, create a new file named flightsToLAX.json.

    Note: A copy of this file is provided in the src/main/resources/examples folder.

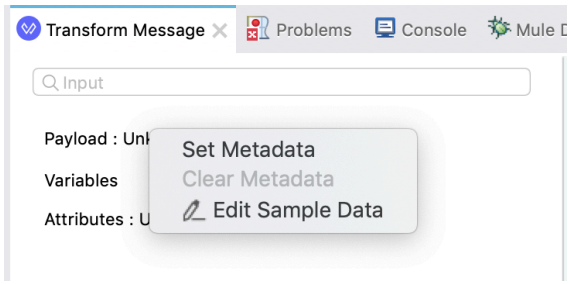
24. In the flightsToLAX.json editor, paste all the text from the clipboard of the JSON formatted payload, then save the file.



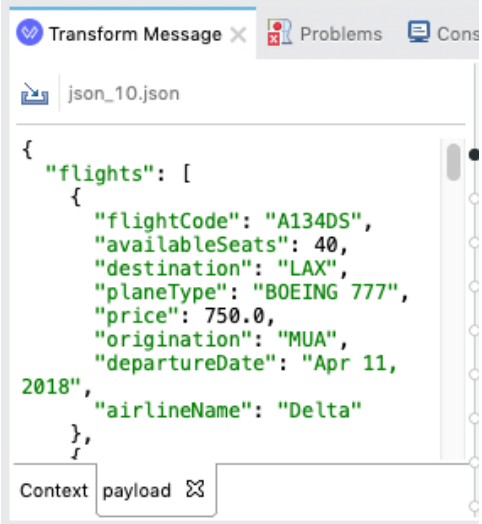
Note: This the same JSON payload that was returned to the web client in the previous walkthrough.

Create a new metadata type from a sample JSON file

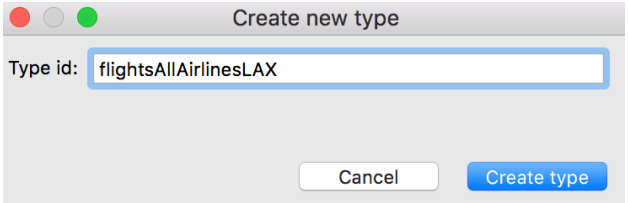
25. Return to the dw-transforms tab and double-click on the Transform Message component of the joinAirportDetailsToFlights flow.In the Input pane, right-click on Payload and select Clear Metadata; if it is grayed out, click outside to close the right-click menu.



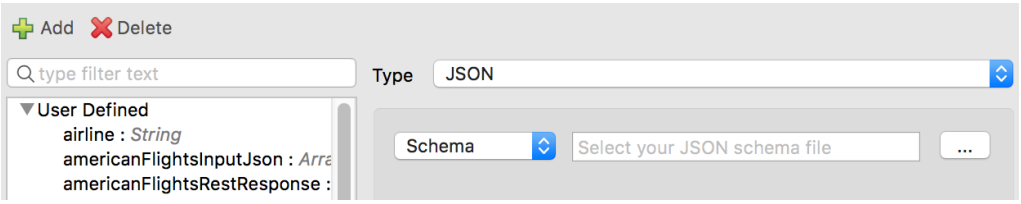
26. If there is a payload tab, click x to delete it.



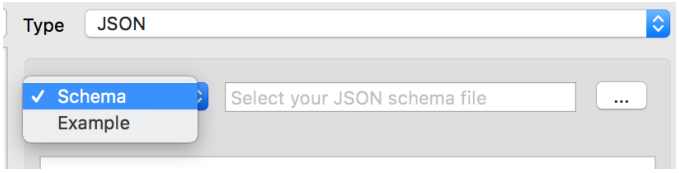
27. To the right of Payload, click Define metadata.
28. In the Select metadata type dialog, click Add.
29. In the Create new type dialog, type flightsAllAirlinesLAX.



30. Click Create type.
31. At the top of the Select metadata type dialog, change the type drop-down list to JSON.

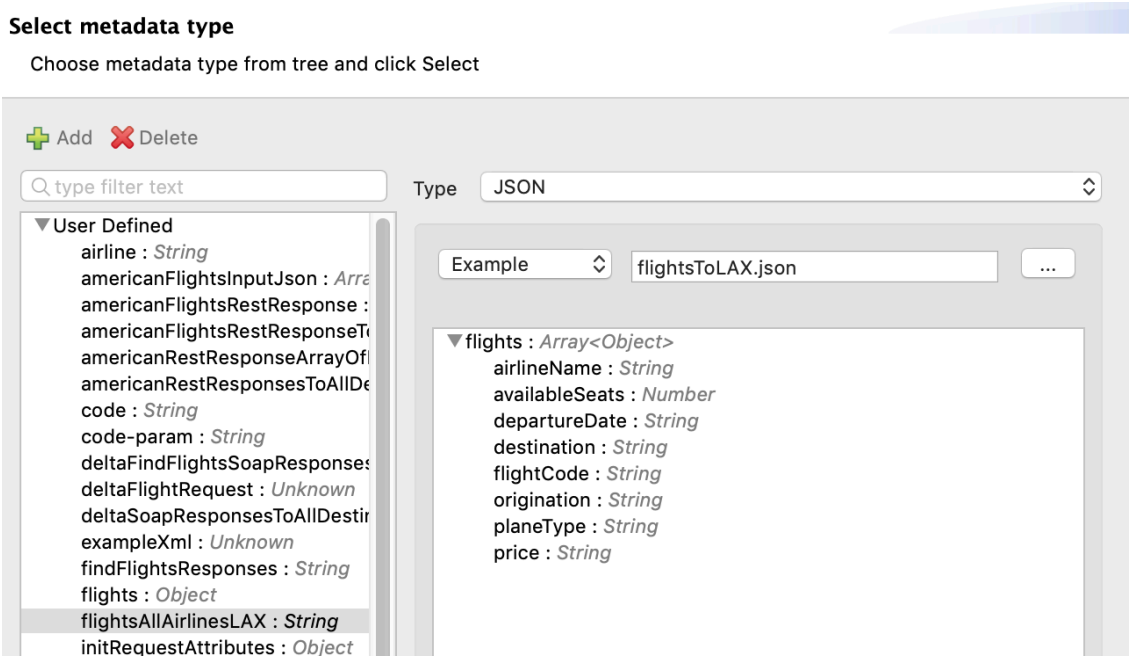


32. In the lower drop-down menu, change the type from Schema to Example.



33. Browse to src/main/resources/examples/flightToLAX.json.
34. Click Open.

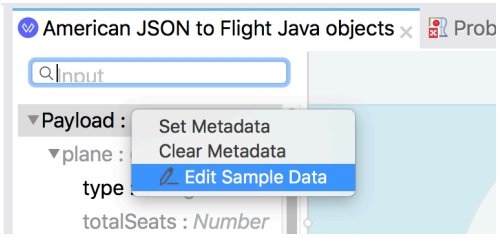
35. Verify a new schema is inferred from this example file and displays on the right side.



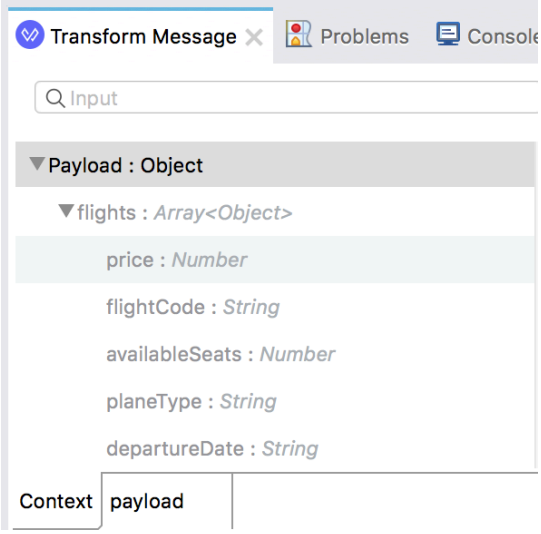
36. Click Select.

Set metadata and example input for the Preview pane

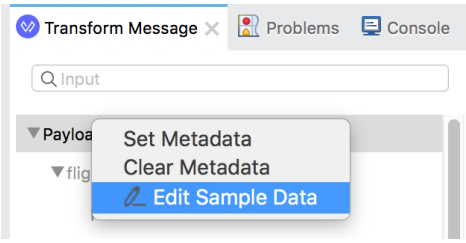
37. In the Input pane, right-click on Payload and select Edit Sample Data.



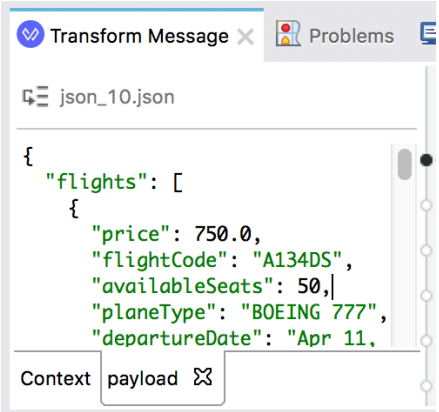
38. Look in the Context tab; you should see the flights object schema.



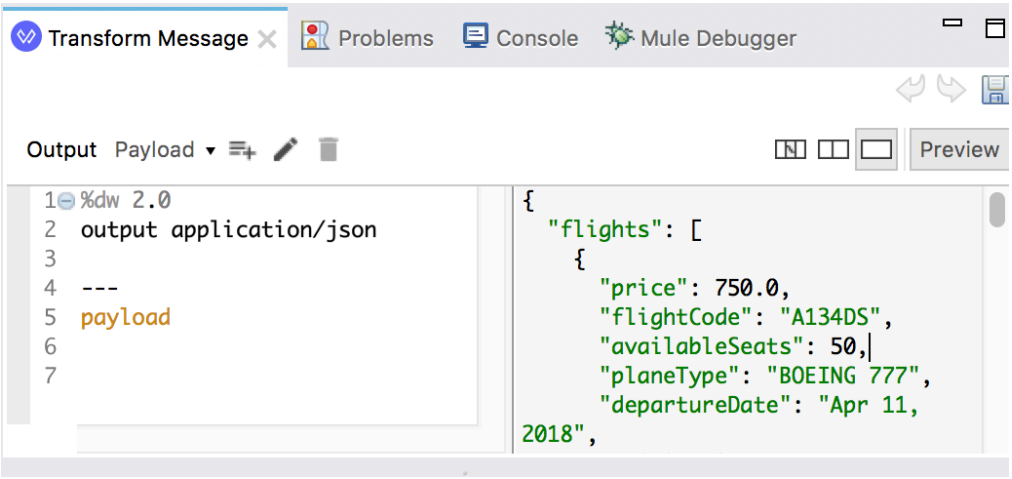
39. Right-click Payload and select Edit Sample Data.



40. In the payload tab, verify you see the flightsToLAX.json data.



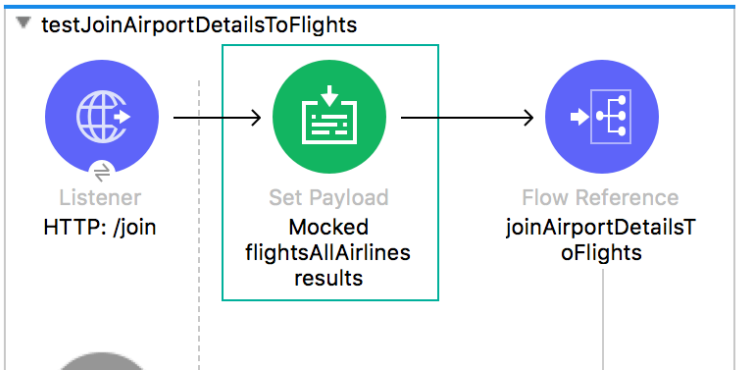
41. Select the Preview pane; you should see the example data captured in the debugger is now used as the payload to the DataWeave expression in the Output pane, and is also displayed in the Preview pane.



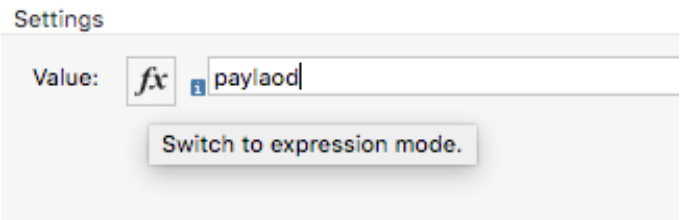
Note: If you don't see the output in the Preview pane change and you see errors related to the Tooling service, select a different Mule component in the Studio canvas than click back to this component.

Access the full DataWeave editor to code a Set Payload transformer

42. Expand the testJoinAirportDetailsToFlights flow and select the Set Payload transformer.



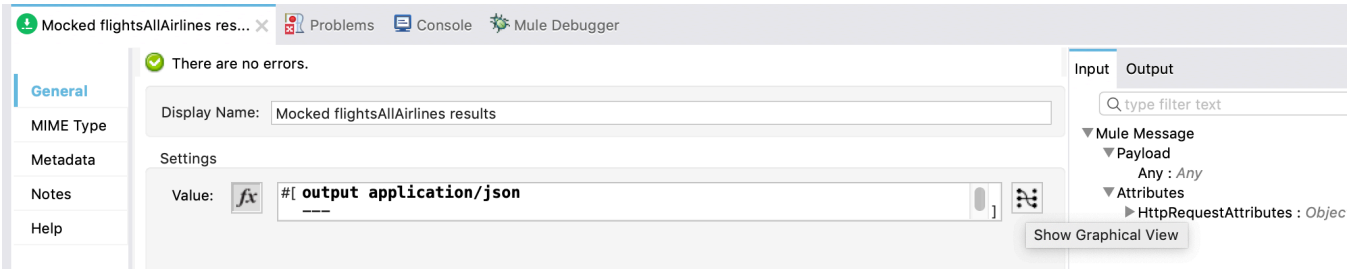
43. In the Configuration view, switch to expression mode.



Note: To switch to expression mode, hover over the mode button to the left of the value textfield, and if the popup message says Switch to expression mode, then click the mode button.

44. Verify the expression mode is selected; the expression mode button should be pressed, and the value in the textfield should begin with #[.

45. On the right side of the value textfield, select the Show Graphical View button; this should open the full DataWeave editor.



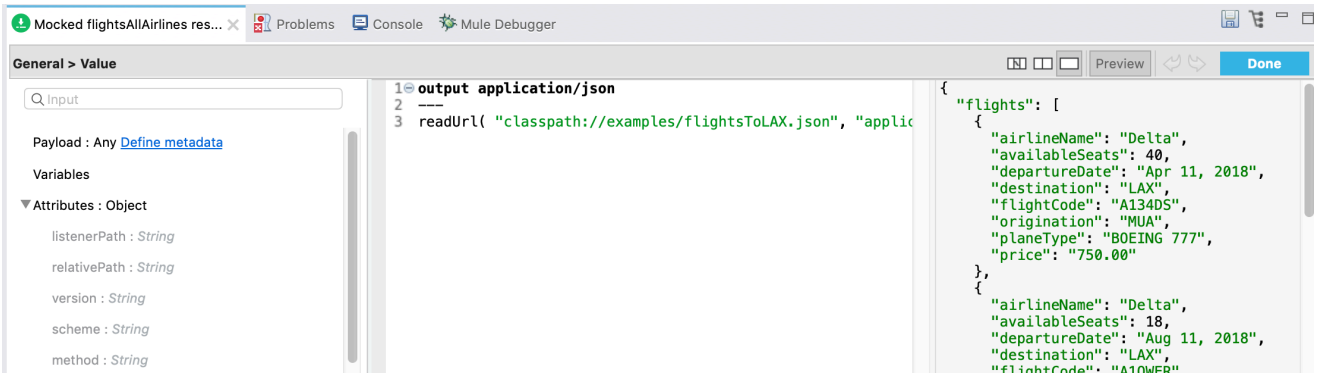
Read in a sample input JSON file and convert it to Java format

46. In the Output pane of the DataWeave editor, write DataWeave code to read in an example JSON file of all flights to LAX from src/main/resources/examples/flightsAllAirlinesLAX.json and change the output directive to application/json.

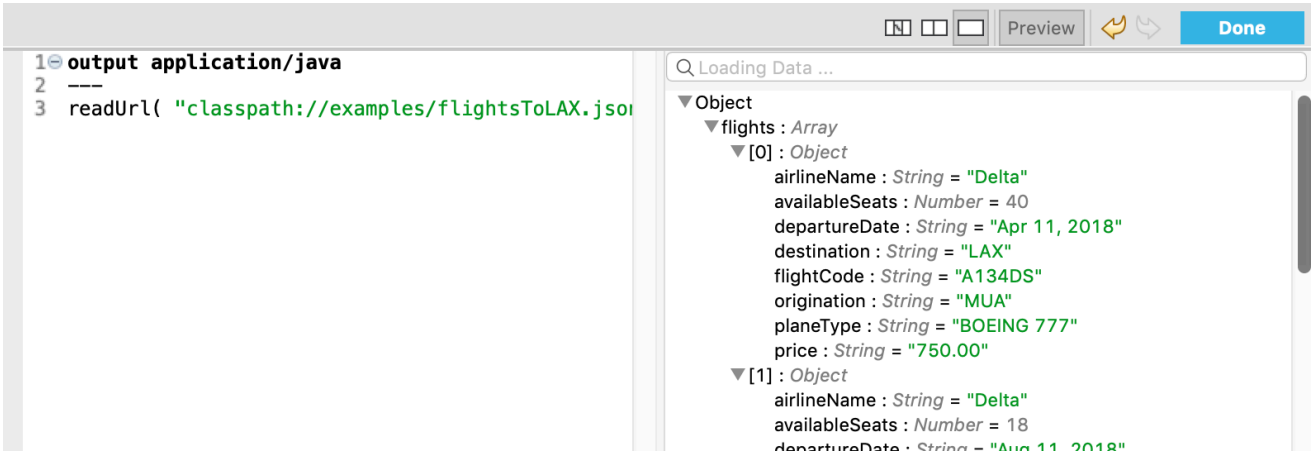
```
output application/json
---
readUrl( "classpath://examples/flightsToLAX.json",
"application/json")
```

Note: The default file format for the readUrl function is application/dw. A copy of this DataWeave code is included in the snippets.txt file in your student files.

47. Look in the Preview pane; you should see the file is read and displays in JSON format.

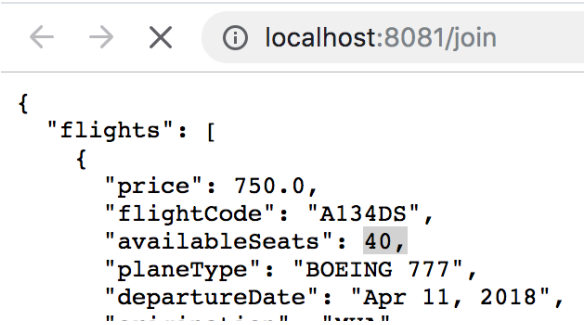


48. Change the output directive to application/java; in the Preview pane you should see the sample JSON read in from flightsAllAirlinesLAX.json is converted to a Java object.



Trigger the test flow

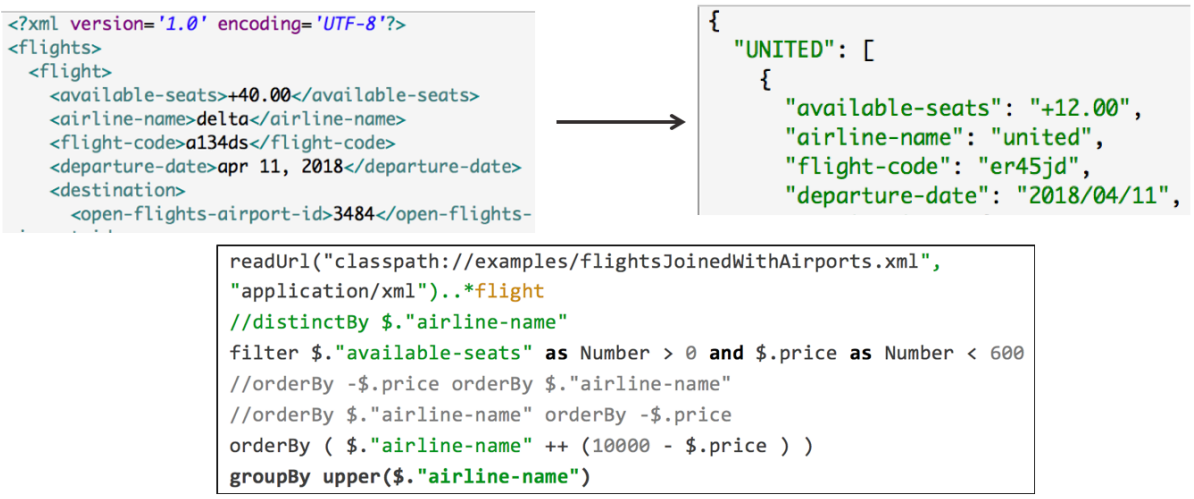
49. In a web client, submit a request to http://localhost:8081/join.
50. Verify a JSON object displays in the web client.



Walkthrough 1-3: Review DataWeave fundamentals

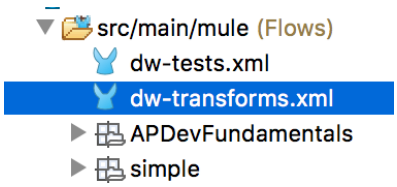
In this walkthrough, you review some simple DataWeave expressions to select parts of an object or array, and to format data. You will:

- Select child and descendant elements of an object or array.
- Test for and assert the existence of keys in a nested data structure.
- Filter, order, and group elements of an object or array.
- Format strings, numbers, and dates.



Import sample data with which to work into a Transform Message component

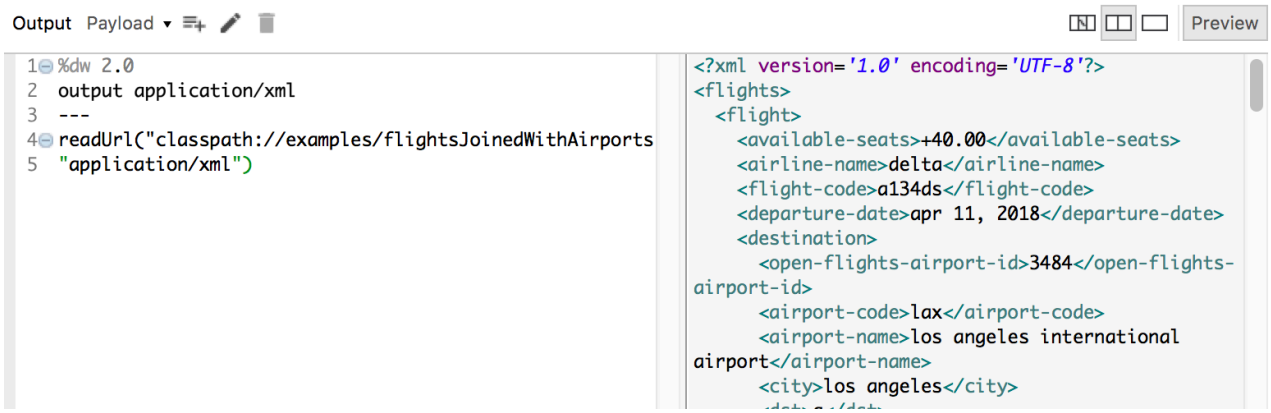
1. Return to Anypoint Studio.
2. Open dw-tests.xml.



3. Drag out a Transform Message component and drop it on the bottom of the canvas to create a new flow.
4. Rename the flow dwReview.
5. Select the Transform Message component of the dw-review flow.
6. In the body expression, load the flightsJoinedWithAirports.xml file by using the readUrl function; the file should be read in and converted to application/java format.

```
readUrl("classpath://examples/flightsjoinedwithairports.xml",
"application/xml")
```

7. Change the output directive to application/xml.
8. Look in the Preview pane; you should see nested XML data displays with airport details as a nested child of each flight object.



Test for object keys nested in an array

9. Assign the current body expression to a variable named data, and set the body expression to data.

```
var data = readUrl("classpath://examples/flightsJoinedWithAirports.xml",
"application/xml")
---
data
```

10. Change the output directive to application/json.
11. Use a multi-value selector to select all the nested flight objects from the read XML file; in the Preview pane you should see the array of flight objects.

```
data..*flight
```

12. Test for the existence of a key named city nested inside any flight object; in the Preview pane ou should see the boolean result true.

```
data..*flight..city?
```

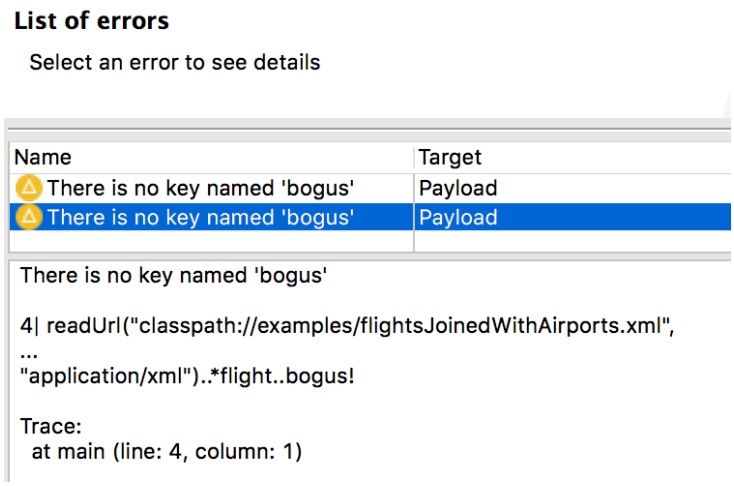
13. Change the key name to bogus; in the Preview pane you should see the result false.

```
data..*flight..bogus?
```

14. Assert, rather than test for, the existence of a key named bogus nested inside any flight object.

```
data..*flight..bogus!
```

15. Look at the generated errors; they should say there is no key named bogus.



16. Assert the existence of a key named "flight-code" nested inside any flight object.

```
data..*flight..*"flight-code"!
```

Note: Make sure to put the dasherized key name "flight-code" in quotes.

17. Look in the Preview pane; you should see the array of flight codes.

```
[
  "a134ds",
  "a1qwer",
  "a1b2c4",
  "ffee0192",
  "rree0001",
  "eefd4511",
  "er45if",
  "er45jd",
  "er0945"
]
```

Note: If the assertion is true, the result is the same as without the assertion character ! at the end of the selector.

Test for object keys nested in an object

18. Select the first flight object using the descendent selector without a *.

```
data..flight
```

19. Look in the Preview pane; you should see an array with just the first flight object.

```
[
  {
    "available-seats": "+40.00",
    "airline-name": "delta",
    "flight-code": "a134ds",
    "departure-date": "apr 11, 2018",
    "destination": {
      "open-flights-airport-id": "3484",
      "airport-code": "lax",
```

20. Use a multi-value selector to still select just the first flight object.

```
data..*flight[0]
```

21. Look in the Preview pane; you should see the same flight object but not inside a parent array.

```
{
  "available-seats": "+40.00",
  "airline-name": "delta",
  "flight-code": "a134ds",
  "departure-date": "apr 11, 2018",
  "destination": {
    "open-flights-airport-id": "3484",
    "airport-code": "lax",
```

22. Test if the outer level flight-code key exists in the first flight object by using the ? key present selector; in the Preview pane you should see the boolean value true

```
data..*flight[0]."flight-code"?
```

23. Test if the nested city key exists in the first flight object by using the ? key present selector; in the Preview pane you should see the boolean value false

```
data..*flight[0]..city?
```

Select an array of child elements with non-repeating airport codes

24. Select the destination object from each flight object of the read XML file; in the Preview pane you should see an array of destination objects.

```
data..*destination
```

```
[
  {
    "open-flights-airport-id": "3484",
    "airport-code": "lax",
    "airport-name": "los angeles international
airport",
    "city": "los angeles",
    "dst": "a",
    "altitude": "125",
    "icao": "klax",
    "longitude": "33.94250107"
  },
  {
    "open-flights-airport-id": "3484",
    "airport-code": "lax",
```

25. Use the distinctBy operator to remove duplicate destination objects with the same airport-code; in the Preview pane you should see an array with only one element.

```
data..*destination
distinctBy $."airport-code"
```


```
[
  {
    "open-flights-airport-id": "3484",
    "airport-code": "lax",
    "airport-name": "los angeles international
airport",
    "city": "los angeles",
    "dst": "a",
    "altitude": "125",
    "icao": "klax",
    "longitude": "33.94250107"
  }
]
```


Note: You must put key names with dashes in quotes.

Select one flight for each airline

26. Change the selector from destination to the parent flight key; you should still see the same array with one object.

```
data..*flight
distinctBy $."airport-code"
```

Note: $."airport-code" is null because it is not an outer key in the flights object.

27. Change the distinctBy key to "bogus"; you should still see the same array with one object.

```
data..*flight
distinctBy $."bogus"
```

Note: If the left-side expression evaluates to a constant value, then every element is considered the same, so distinctBy returns the first element.

28. Change the distinctBy expression to test "airline-name" instead of "airport-code".

```
readUrl("classpath://examples/flightsJoinedWithAirports.xml",
"application/xml")..*flight
distinctBy $."airline-name"
```

29. Look in the Preview pane; you should see an array of three flight objects, with one flight object for each airline (delta, american, and united).


```
[
  {
    "available-seats": "+40.00",
    "airline-name": "delta",
    "flight-code": "a134ds",
    "departure-date": "apr 11, 2018",
    "destination": {
      "open-flights-airport-id": "3484",
      "airport-code": "lax",
      "airport-name": "los angeles international airport",
      "city": "los angeles",
      "dst": "a",
      "altitude": "125",
      "icao": "klax",
      "longitude": "33.94250107"
    },
    "plane-type": null,
    "origination": "mua",
    "price": "+750.00"
  },
  {
    "available-seats": "+0.00",
    "airline-name": "american",
    "flight-code": "ffee0192",
    "departure-date": "2018-01-20t00:00:00",
    "destination": {
      "open-flights-airport-id": "3484",
      "airport-code": "lax",
```


Filter out some flights based on some conditions

30. Remove the distinctBy operator.
31. Add a filter operator to filter out objects where the available-seats value is 50 or less; in the Preview pane you should only see two flight matching this filter condition.

```
data..*flight
//distinctBy $."airline-name"
filter $."available-seats" as Number > 50
```

32. Change the filter condition to filter out flights with available seats = 0 or price > 496.

```
data..*flight
//distinctBy $."airline-name"
filter $."available-seats" as Number > 0 and $.price as Number <= 496
```

33. In the Preview pane; you should see an array of four flight objects where there are available seats and the price is less than or equal to 496


```
[
  {
    "available-seats": "+18.00",
    "airline-name": "delta",
    "flight-code": "a1qwer",
    "departure-date": "aug 11, 2018",
    "destination": {
      "open-flights-airport-id": "3484",
      "airport-code": "lax",
      "airport-name": "los angeles international airport",
      "city": "los angeles",
      "dst": "a",
      "altitude": "125",
      "icao": "klax",
      "longitude": "33.94250107"
    },
    "plane-type": "boeing 747",
    "origination": "mua",
    "price": "+496.00"
  },
  {
    "available-seats": "+12.00",
    "airline-name": "united",
    "flight-code": "er45jd",
```


34. Change the filter expression to use a not operator to specify the condition to remove an element; in the Preview pane you should see the same array of four flight objects.

```
filter not ($."available-seats" as Number == 0 or $.price as Number > 496)
```

Find a flight with the lowest price and a flight with the highest price

35. Find the lowest priced flight from the filtered results, by using the minBy operator; one flight object should be returned with price $199.99.

```
filter not ($."available-seats" as Number == 0 or $.price as Number > 496)
minBy $.price
```

36. Replace the minBy operator with maxBy; one flight object should be returned with price $496.00.

```
filter not ($."available-seats" as Number == 0 or $.price as Number > 496)
maxBy $.price
```

Add evaluation parentheses to make your DataWeave expression easier to read

37. Separate the filter expression boolean operators with DataWeave expressions explicitly evaluated inside parentheses; you should still see the same flight returned with price $496.00.

```
filter ($."available-seats" as Number > 0) and ($.price as Number <= 496)
maxBy $.price
```

Order the array of filtered flights

38. Replace the maxBy operator with the orderBy operator; in the Preview pane you should see the price of each flight in the array is increasing as you look down the array.

```
filter ( $."available-seats" as Number > 0 ) and ( $.price as Number <= 496)
orderBy $.price
```

39. Change the orderBy condition to sort the price in increasing order; in the Preview pane you should see the flights are sorted in decreasing order of price, without regard to the airline-name.

```
filter ( $."available-seats" as Number > 0 ) and ( $.price as Number <= 496)
orderBy -$.price
```

40. Add another orderBy operator to the end of the body expression to further order by the airline-name values; in the Preview pane you should see the flights are now listed first with delta flights in decreasing order of price, followed by united flights in decreasing order of price.

```
filter ( $."available-seats" as Number > 0 ) and ( $.price as Number < 496 )
orderBy -$.price orderBy $."airline-name"
```

41. Swap the -$.price and $."airline-name" tests; in the Preview pane you should see the flights are now in decreasing order of price, but not ordered by airline-name.

```
orderBy $."airline-name" orderBy -$.price
```

Note: This shows order of multiple orderBy operations matters and can produce different results.

42. Use a single orderBy operator to order the filtered flights first by airline-name, then by decreasing price.

Note: To do this, you can concatenate two expressions together that use the airline-name and price so that the flight elements are first sorted by airline-name, then for each airline, by decreasing price

```
readUrl("classpath://examples/flightsJoinedWithAirports.xml",
"application/xml")..*flight
//distinctBy $."airline-name"
filter ( $."available-seats" as Number > 0 ) and ( $.price as Number <= 496 )
//orderBy -$.price orderBy $."airline-name"
//orderBy $."airline-name" orderBy -$.price
orderBy ( $."airline-name" ++ (10000 - $.price ) )
```

Group the ordered result by airline-name, with the airline-name in uppercase

43. Use the groupBy operator to create an object of flights with a key for each airline-name, and the value all flight objects matching that airline-name.

```
readUrl("classpath://examples/flightsJoinedWithAirports.xml",
"application/xml")..*flight
//distinctBy $."airline-name"
filter ( $."available-seats" as Number > 0 ) and ( $.price as Number < 600 )
//orderBy -$.price orderBy $."airline-name"
//orderBy $."airline-name" orderBy -$.price
orderBy ( $."airline-name" ++ (10000 - $.price ) )
groupBy $."airline-name"
```

44. Look in the Preview pane; you should see the result is an object of key/value pairs, with the key the name of an airline, and the value an array of all corresponding flights.


```
{
  "united": [
    {
      "available-seats": "+12.00",
      "airline-name": "united",
      "flight-code": "er45jd",
      "departure-date": "2018/04/11",
      "destination": {
        "open-flights-airport-id":
"3484",
        "airport-code": "lax",
        "airport-name": "los angeles
```


45. Modify the groupBy selector expression to make the airline-name value uppercase.

```
readUrl("classpath://examples/flightsJoinedWithAirports.xml",
"application/xml")..*flight
//distinctBy $."airline-name"
filter ( $."available-seats" as Number > 0 ) and ( $.price as Number < 600 )
//orderBy -$.price orderBy $."airline-name"
//orderBy $."airline-name" orderBy -$.price
orderBy ( $."airline-name" ++ (10000 - $.price ) )
groupBy upper($."airline-name")
```

46. Look in the Preview pane; you should see the keys are now in uppercase.


```
{
  "UNITED": [
    {
      "available-seats": "+12.00",
      "airline-name": "united",
      "flight-code": "er45jd",
      "departure-date": "2018/04/11",
```

Select the available-seats from the current body expression result and format the first available-seat value as an integer

47. Surround the body expression in evaluation parentheses and then add a selector to select the available-seats strings; in the Preview pane you should see an array of seat numbers.

```
)."available-seats"
```

```
[
  "+18.00",
  "+10.00",
  "+12.00",
  "+52.00"
]
```

48. Select the first available-seat value from the current result array and format the string as a Number with zero decimal places; in the Preview pane you should see the value is the string "+18".

```
)."available-seats"[0] as Number as String {format: "+#,###"}
```

Select the first departure-date from the grouped object and change its format

49. Change the last line of the body expression to select the first departure-date.

```
// )."available-seats"[0] as Number as String {format: "+#,###"}
)."departure-date"[0]
```

50. Look in the Preview pane; you should see a date string "2018/04/11".
51. Add formatting expressions to convert the string to a Date object, then change the formatting so the date appears in the format "11 April, 2018".

```
)."departure-date"[0]
as Date {format:"MMM dd, yyyy"}
as Date {format: "dd MMMM, yyyy"}
```

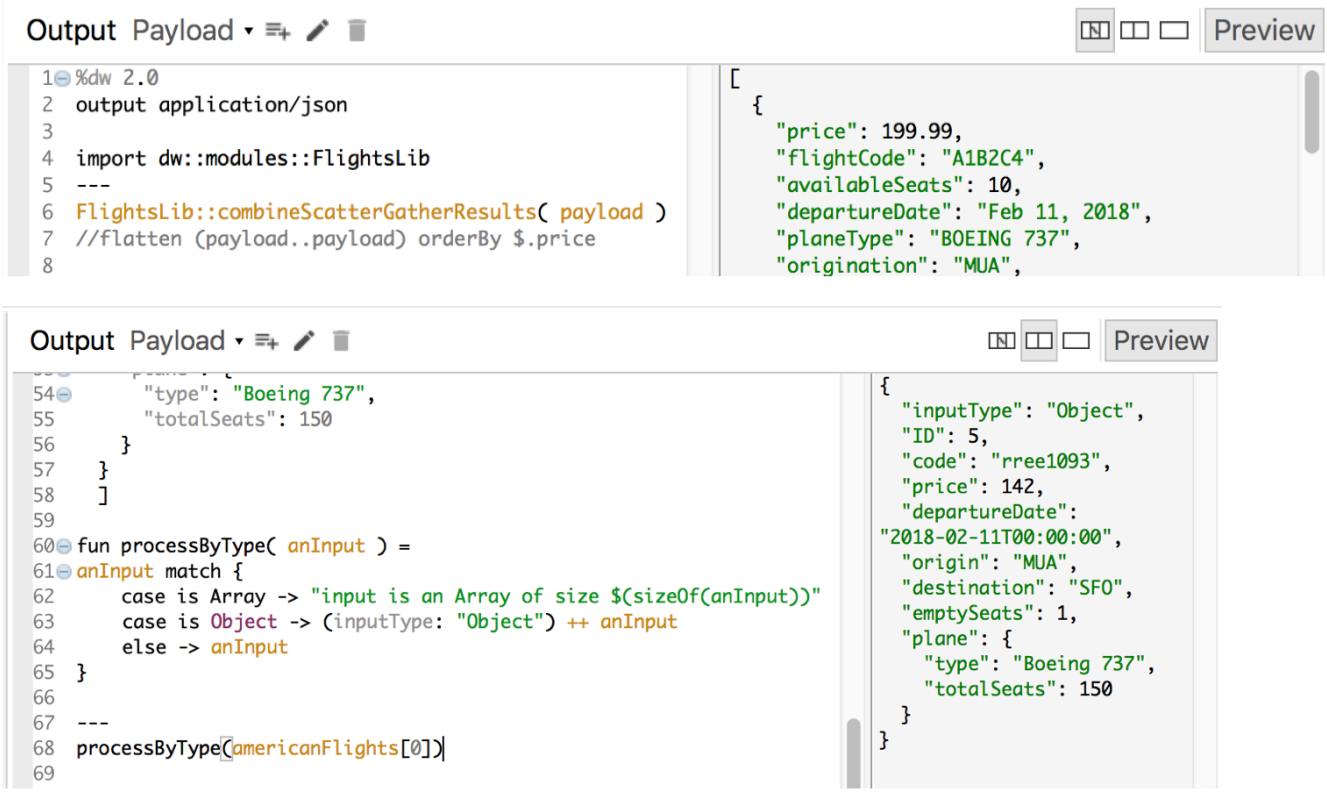
Test the type of the current body expression

52. Pass the body expression into the typeOf function.

```
%dw 2.0
output application/json
---
typeOf
(
  (
        readUrl("classpath://examples/flightsJoinedWithAirports.xml",
             "application/xml")..*flight
        ...
        as Date {format:"MMM dd, yyyy"}
        as Date {format: "dd MMMM, yyyy"}
  )
)
```

53. Look in the Preview pane; you should see the result is the string "Date".

Module 2: Organizing and Reusing DataWeave Code



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

- Organize DataWeave code into variables and functions.
- Pass functions and lambda expressions as arguments to other DataWeave functions.
- Chain DataWeave functions together.
- Create and use reusable DataWeave modules.
- Dynamically evaluate DataWeave scripts.

Walkthrough 2-1: Organize DataWeave code with variables and functions

In this walkthrough, you use variables and functions to split up DataWeave expressions into more organized and sometimes more reusable parts. You will:

- Organize and refactor DataWeave expressions with variables and functions.

Add a variable to a Transform Message component in which to store the airline name

1. In Anypoint Studio, open dw-transforms.xml.
2. In the transformAmericanJsonToJavaObjects flow, select the Transform Message component.

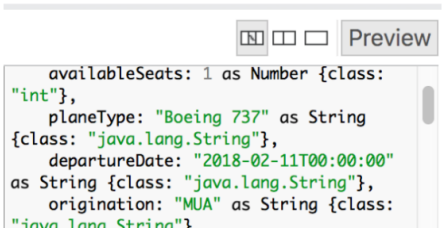
3. In the Output pane, in the header, add a variable named airlineName, and set the value to "American Airlines".

```
var airlineName = "American Airlines"
```

Set a sample payload

4. Open src/main/resources/examples/mockdata/americanFlightsResponse.json.

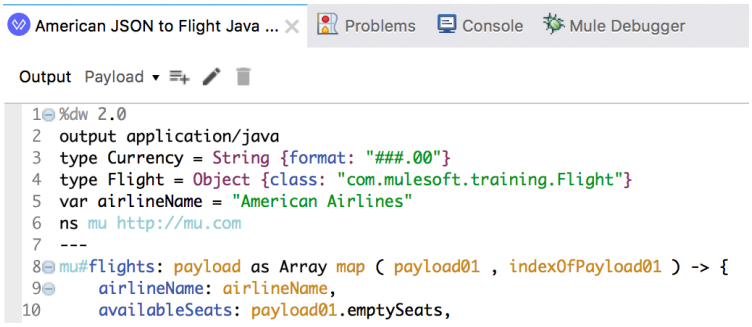
5. Copy the JSON contents to the clipboard.
6. Return to the dw-transform tab.
7. In the Transform Message component, in the Input pane, paste the JSON as payload sample data.
8. Look in the Preview pane; you should see the pasted sample JSON payload is passed to the body expression and evaluates to valid Java.



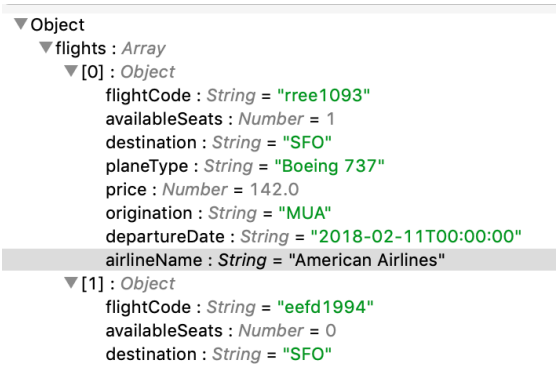
Use the airlineName variable in a DataWeave expression

9. In the body expression, change the mapping expression for the airline key to use the airlineName variable.

```
airlineName: airlineName
```

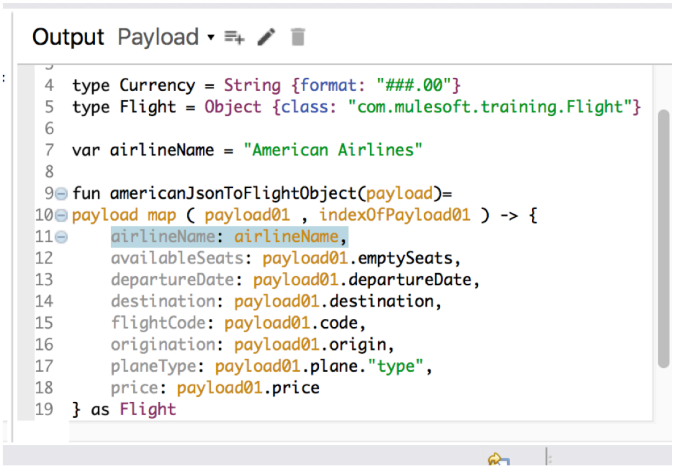


10. Look in the Preview pane; you should see each airlineName key now has the value "American Airlines".



Refactor the body expression into a function

11. Move the entire body expression to a function named americanJsonToFlightObject.



Note: Do not forget the = after the fun declaration.

12. Call this function in the body expression; in the Preview pane you should see the array of Flights Java objects still appears, and verify there are no errors.

```
---
americanJsonToFlightObject( payload )
```

Note: Use the type-ahead feature to complete the variable name.

(Optional) Test the application

13. Run or debug the application.
14. Return to your web client.
15. Submit a request to http://localhost:8081/flights?code=CLE&airline=american

Note: Alternatively, submit a request http://localhost:8081/flights?code=CLE&airline=american&mock=false to use the actual mu.training.mulesoft.com hosted web services.

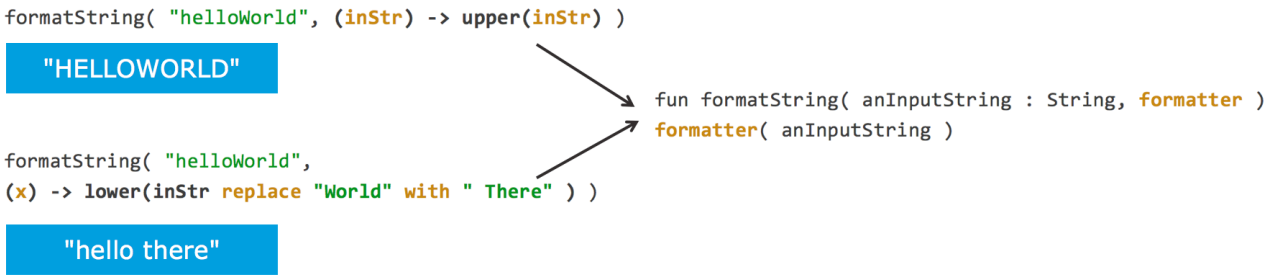
16. Verify the American flights are processed without any errors.

17. Stop the Mule application.

Walkthrough 2-2: Pass functions and lambda expressions as arguments to other functions

In this walkthrough, you write a parent function that accepts a child function argument. The parent function then "abstractly" calls this argument's name inside the parent function implementation. You then call this parent function in the body expression, by setting this argument with different function names and lambda expressions and then explore how the behavior of the called parent function changes with different values for the child function argument. You will:

- Write a function that accepts a function as an input argument.
- Pass a function as an input argument to another function.
- Pass a lambda expression as an input argument to another function.
- Call a two argument function as an operator.



Write a function that accepts a function as an input argument

1. Return to Anypoint Studio and select the dw-tests tab.
2. From the Mule palette, drag out a Transform Message component and drop it into the canvas.
3. Name the new flow passFunctions and then select the Transform Message component.
4. In the header, set the output directive to application/json.
5. In the header, define a new function named formatString that accepts the following two input arguments:
    - The first input argument named anInputString of type String
    - The second input argument named formatter

    ```
    fun formatString( anInputString : String, formatter) =
    ```

Use a function input argument in the function's implementation

6. Apply the formatter function to the anInputString argument.

    ```
    fun formatString( anInputString : String, formatter ) =
    formatter( anInputString )
    ```

Call the formatString function with another string function to convert the input string to uppercase

7. In the body expression, call the formatString function with the upper function and an example string that is all lowercase.

```
---
formatString( "helloWorld", upper )
```

8. Look in the Preview pane; you should see the string is converted to the uppercase string "HELLOWORLD".

Change the input argument to use a lambda expression instead of a function to convert the input string to lowercase

9. Change the body expression, replace the upper function name with a lambda expression.

```
---
formatString( "helloWorld", (inStr) -> upper(inStr) )
```

10. Look in the Preview pane; you should see the result is changed to "helloworld".

Change the lambda expression to a more complex expression

11. In the body expression, change the lambda expression to convert "helloWorld" to "hello there".

```
---
"helloWorld" formatString
(inStr) -> lower( inStr replace "World" with " There" )
```

12. Look in the Preview pane; you should see the result is "hello there".

    Note: You did not have to modify the formatString function to change its behavior.

Strongly type the formatter function

13. Define strong formatting for the formatter function arguments; you might see an error saying the function needs a Number type but was called with a String type.

```
fun formatString( anInputString : String, formatter : (Number) -> String ) =
formatter( anInputString )
```

14. Change the strong formatting to expect String type input.

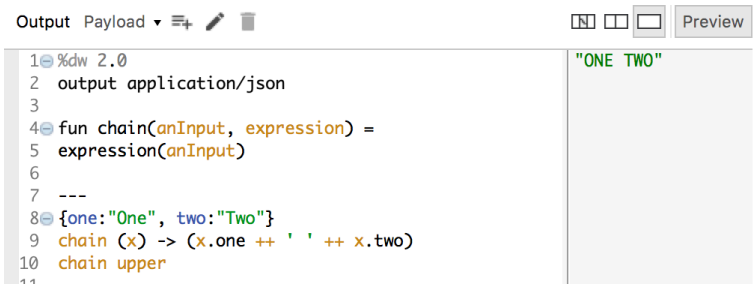
```
fun formatString( anInputString: String, formatter : (String) -> String ) =
formatter( anInputString )
```

15. Look in the Preview pane; the string "hello there" should appear.

Walkthrough 2-3: Chain together DataWeave functions

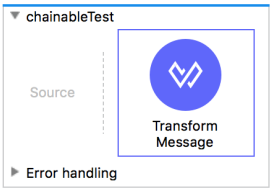
In this walkthrough, you write a chainable DataWeave function that takes two input arguments, where the second argument is a function or lambda expression. Then you chain this function together with other DataWeave functions that also accept lambda expressions. You will:

- Call a two argument function as an operator.
- Call an operator as a two argument function.
- Write a chainable DataWeave function with two input arguments, where the second argument is a function or lambda expression.
- Chain together two single argument DataWeave functions or lambda expressions.



Create an array with which to work

1. Return to the dw-test tab.
2. From the Mule palette, drag out a Transform Message component and drop it into the canvas.
3. Name the new flow chainableTest and select the Transform Message component.



4. Change the output directive to application/json.
5. In the body expression, define an array with numbers that are out of order.

```
[1,5,3,2]
```

6. Look in the Preview pane; you should see the array without any errors.

Chain together two array manipulation functions

7. Chain together the filter and orderBy functions to list all odd numbers in ascending order.

```
[1,5,3,2]
filter ($ mod 2) == 1
orderBy $
```

8. Look in the Preview pane; you should see the result is the array [1,3,5].
9. Chain together the filter and orderBy functions to list all odd numbers in descending order.

```
[1,5,3,2]
filter ($ mod 2) == 1
orderBy -$
```

10. Look in the Preview pane; you should see the result is the array [5,3,1].

Create and use a utility function to chain single argument functions with other functions

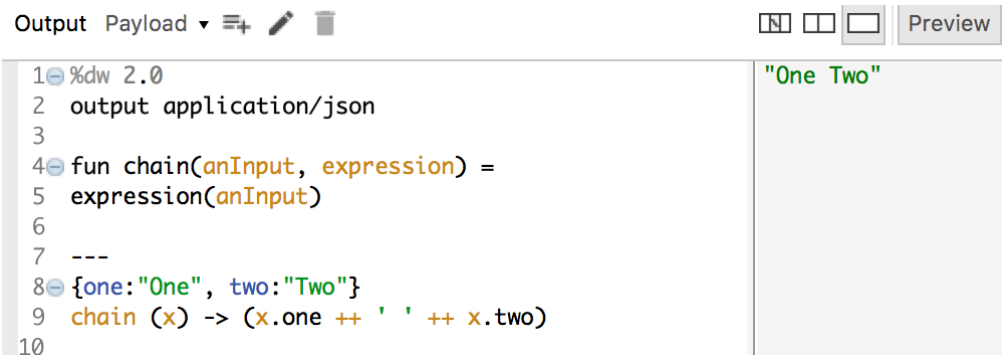
11. In the header, define a function that accepts two input arguments, where the second argument is a function that is applied to the first argument.

```
fun chain(anInput, expression) =
expression(anInput)
```

12. In the body expression, call the chain function with an object and a lambda expression to concatenate the values of the one and two keys, with a space between the two values.

```
---
{one:"One", two:"Two"} chain (x) -> (x.one ++ ' ' ++ x.two)
```

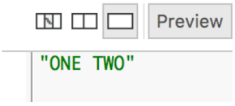
13. Look in the Preview pane; you should see the two object values are concatenated together, with a space added between the two String values.



14. In the body expression, modify the chain function's right-hand expression to pass the result of the concatenated expression to the upper function.

```
upper( {one:"One", two:"Two"} chain (x) -> (x.one ++ ' ' ++ x.two) )
```

15. Look in the Preview pane; you should see the concatenated result is now also formatted as uppercase.

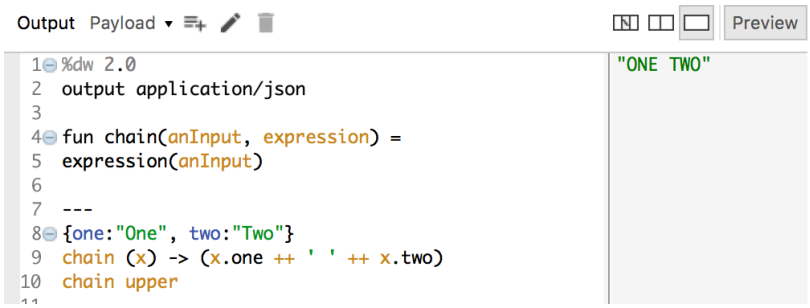


16. Rewrite the body expression to chain the concatenation expression with the upper function.

```
---
{one:"One", two:"Two"}
chain (x) -> (x.one ++ ' ' ++ x.two)
chain upper
```

Note: By chaining functions together, the first concatenation expression does not need to be modified, and the two function calls do not have to be nested inside each other.

17. Look in the Preview pane; you should see the concatenated string is still formatted in uppercase.



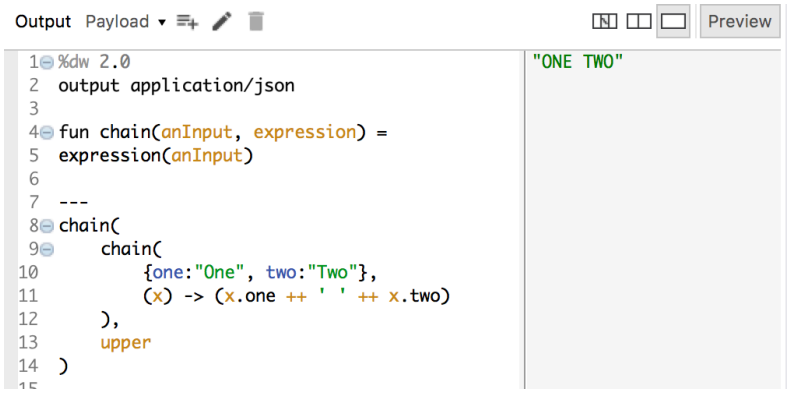
Note: The chain function allows you to chain together single argument functions, instead of having to nest one function inside another function. This lets you pipe together expressions where the previous result is passed down to the next function.

Call an operator as a two argument function

18. Rearrange the chain function calls as nested functions, instead of chained functions.

```
---
chain(
  chain( {one:"One", two:"Two"}, (x) -> (x.one ++ ' ' ++ x.two) ),
  upper
)
```

19. Look in the Preview pane; you should see no errors occur, and the same concatenated string appears, formatted in uppercase.



20. Change the upper function call to instead call the lower function.

```
---
chain(
  chain( {one:"One", two:"Two"},  (x) -> (x.one ++ ' ' ++ x.two) ),
  lower
)
```

21. Look in the Preview pane; you should see the result is the concatenated values formatted in lowercase.

Note: Changing a string to upper or lower is a quick way to verify the Preview pane is still responding to code changes, in order to verify code changes are correct.

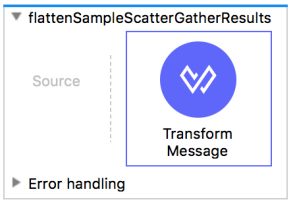
Walkthrough 2-4: Reuse DataWeave code

In this walkthrough, you apply different techniques to reuse DataWeave code. First you store all the DataWeave code for a Transform Message component's set-payload element in an external file using the resource attribute. Then you store DataWeave expressions as functions and variables in a DataWeave module. Then you reuse these variables, functions in other Mule flows. You will:

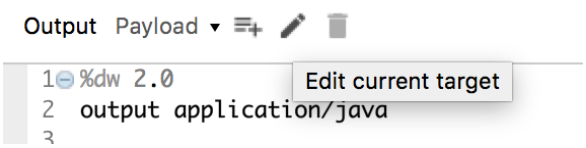
- Reference code for an entire Transform Message component's transformation element from an external file.
- Store DataWeave variables and functions in a DataWeave module file.
- Import and use DataWeave variables and functions from DataWeave modules.

Save the Transform Message component's set-payload transformation target script in a separate file

1. Return to the dw-transforms tab.
2. Scroll down to and expand the flattenSampleScatterGatherResults flow and select the Transform Message component.

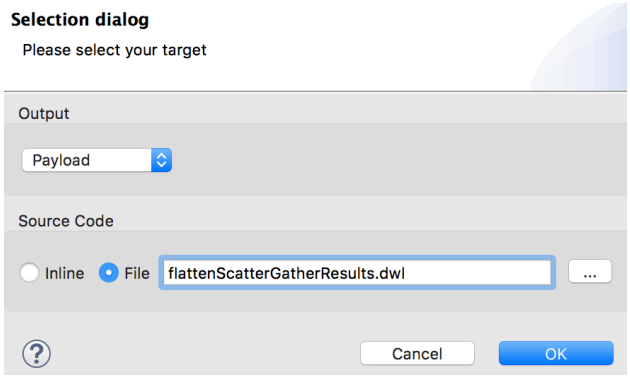


3. In the Output pane, select the Edit current target icon.



4. In the Selection dialog, select file.

5. Type flattenScatterGatherResults.dwl; this the name for the new external file to store the current DataWeave script.



6. Click OK.

Verify the DataWeave transformation is now stored in a separate file

7. At the bottom of the flightReservations canvas, select the Configuration XML tab to view the source XML.

8. Scroll to the bottom of the file and verify the ee:set-payload element has a resources attributed set to flattenScatterGatherResults.dwl.

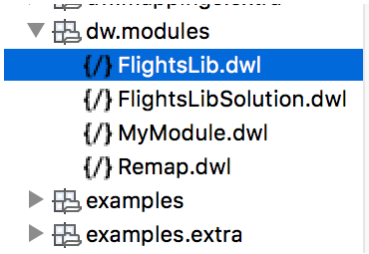
```
    <flow name="flattenSampleScatterGatherResults"  >
        <ee:transform doc:name="Transform Message" >
            <ee:message >
                <ee:set-payload resource="flattenScatterGatherResults.dwl" />
            </ee:message>
        </ee:transform>
    </flow>
</mule>
```

9. In src/main/resources, verify the new flattenScatterGatherResults.dwl file was created.

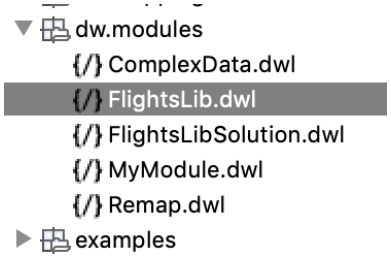
10. Open flattenScatterGatherResults.dwl; it should have the same DataWeave code you typed directly into the Transform Message component, but there is no visual editor.

Edit a custom DataWeave module file

11. In the apdw2-flights project in Anypoint Studio, verify there is a src/main/resources/dw.modules folder.



12. Inside src/main/resources/dw.modules, verify there is a starter file named FlightsLib.dwl.
13. Open FlightsLib.dwl.



Note: It is helpful to drag the FlightLib.dwl view either outside of Anypoint Studio, or on the side of the design canvas, so you can see FlightsLib.dwl and the Mule application flows and their configuration views at the same time.

Define a variable in the external DataWeave file

14. In FlightsLib.dwl, add a DataWeave version directive and define a variable named exchangeRate set to a constant number type value.

```
%dw 2.0
var exchangeRate = 1.35
```

Copy a DataWeave expression to a reusable function in a DataWeave module file

15. Return to the dw-transforms tab and select the Message Flow tab.
16. In the flattenSampleScatterGatherResults flow, select the last Transform Message component and copy the body expression to the clipboard.
17. Return to FlightsLib.dwl and paste the expression as the implementation of a new function named combineScatterGatherResults, but change the payload variable in the pasted expression to an input argument named theInput.

```
fun combineScatterGatherResults( theInput ) =
flights: flatten(  theInput..flights )
```

18. Save your changes to the FlightsLib.dwl file.

Reuse a DataWeave module function in a Transform Message component

19. Return to the dw-transforms tab.
20. In the flattenSampleScatterGatherResults flow, select the Transform Message component.
21. Select the Preview pane and verify the sample payload is flattened.

    Note: If there are errors, set a breakpoint on the Transform Message component and debug the Mule application. Submit a request with query parameter airline=all. Copy the payload at the breakpoint then paste it as example payload to the payload tab of the Transform Message component's Input pane.

22. Change the output to application/json
23. Import the FlightsLib module into the payload transformation's header.

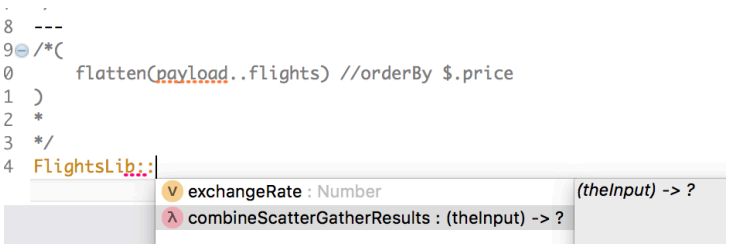
```
%dw 2.0
output application/java
import dw::modules::FlightsLib
var payload = readUrl("classpath://examples/scatterGatherResult.json")
```

Note: Do not include the .dwl suffix.

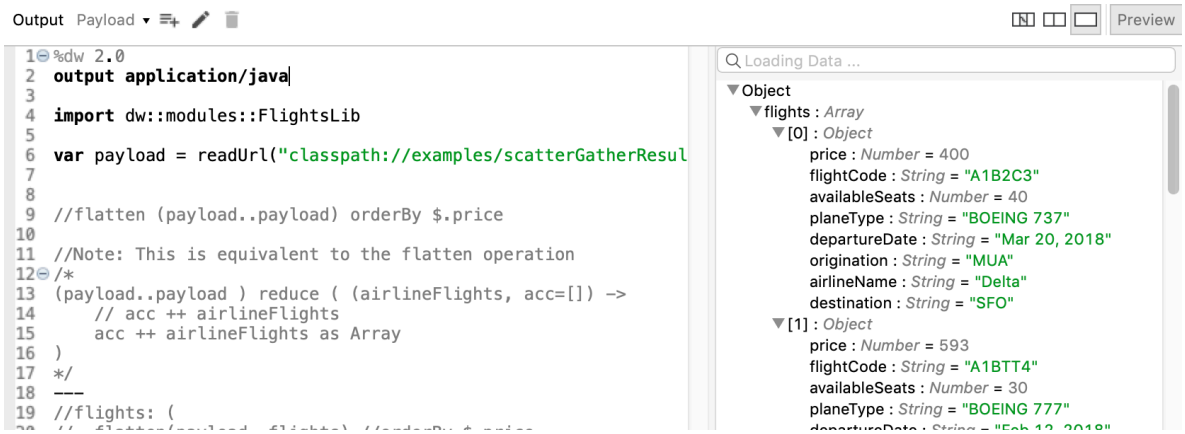
24. Comment out the body expression and replace it with a call to the corresponding function in the FlightsLib module.

```
---
/*
flights: (
       flatten(payload..flights) //orderBy $.price
)
*/
FlightsLib::combineScatterGatherResults( payload )
```

Note: After typing FlightsLib::, press Ctrl+Space to use the type-ahead to see all the functions defined in the imported module.



25. Look in the Preview pane; you should see the same flattened output.

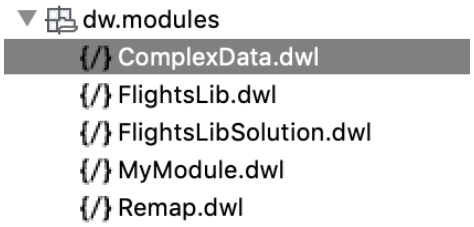


Note: If you get an error, make sure you have saved your changes to the FlightsLib.dwl file, then comment out the body expression and add "hello" as the body expression. After "hello" appears in the Preview pane, replace "hello" with the commented out call to FlightsLib::combineScatterGatherResults(payload).

Note: If you see any Tooling related errors, stop Anypoint Studio, kill any running Java processes, then start Anypoint Studio again.

Reuse functions from another module to shuffle records in an object

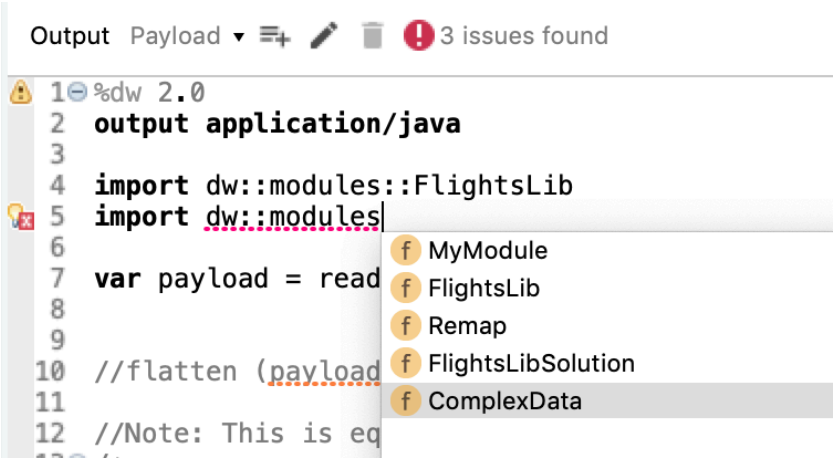
26. In the Package Explorer, verify you have a ComplexData.dwl file in the src/main/resources/dw/modules folder.



27. In the Transform Message component, import the ComplexData module.

Note: Remember that you can press Ctrl+Space to use type-ahead to see all the available module names.

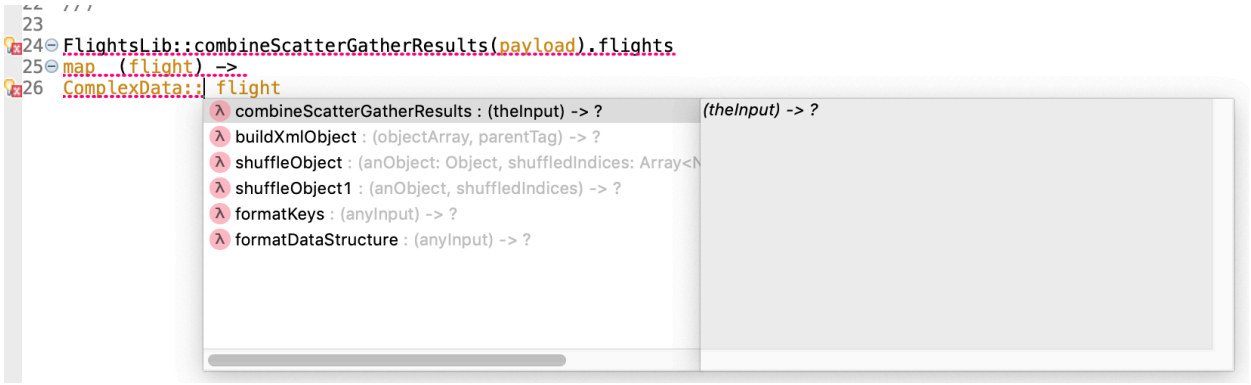
```
import dw::modules::FlightsLib
import dw::modules::ComplexData
```



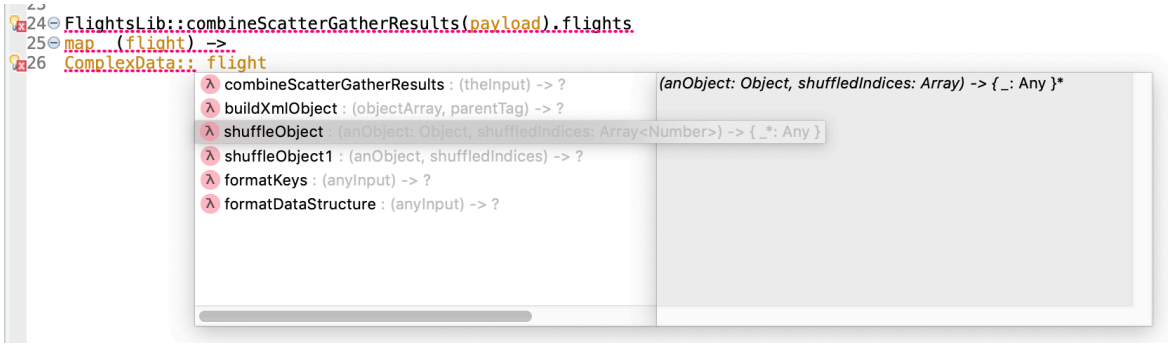
28. In the body expression, use a map operator to iterate over the result of the current body expression; in the Preview pane verify no errors are introduced.

```
---
FlightsLib::combineScatterGatherResults(payload).flights
map  (flight) ->
flight
```

29. At the end of the current body expression, press Ctrl+Space to view the available functions for the ComplexData module.

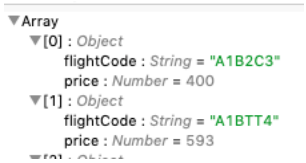


30. Select the shuffleObject function and verify it takes two parameters, an input object and a shuffledIndices array of numbers.



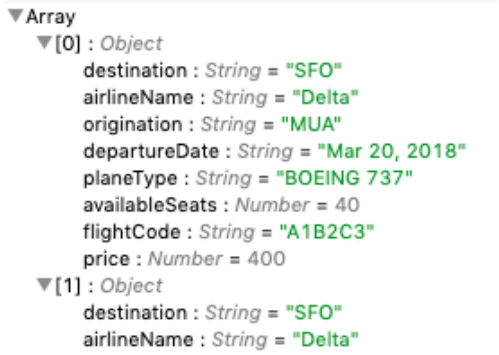
31. Type Enter to add the shuffleObject function.
32. Move flight to the first argument of the shuffleObject function, then add the array [0,1,1] for the second argument; in the Preview pane you should see an array of objects, where each object has one flightCode element followed by the price element duplicated twice.

```
FlightsLib::combineScatterGatherResults(payload).flights
map  (flight) ->
ComplexData::shuffleObject(flight, [1,0,0])
```



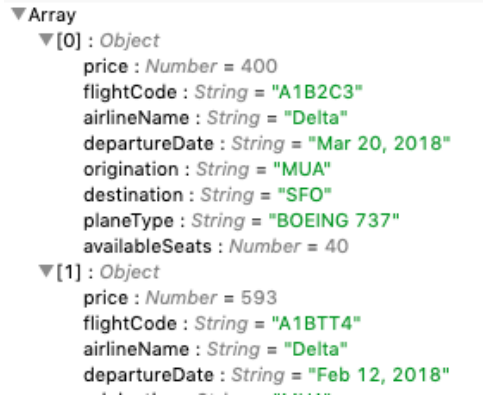
33. Change the second shuffleObject argument to the indices of the flight object in reverse order; all eight flight elements should display in reverse order.

```
FlightsLib::combineScatterGatherResults(payload).flights
map  (flight) ->
ComplexData::shuffleObject(flight, (sizeOf(flight) - 1) to 0 )
```



34. Change the shuffleIndex array to display flight object elements in the order price, flightCode, airlineName, departureDate, origination, destination, planeType, availableSeats.

```
FlightsLib::combineScatterGatherResults(payload).flights
map  (flight) ->
ComplexData::shuffleObject( flight, [0,1,6,4,5,7,3,2] )
```



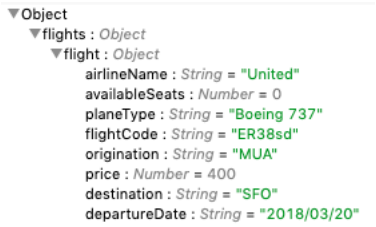
Reuse a module function to build a valid XML data structure from an array of objects

35. Define the scope of the map operator by adding evaluation parentheses around the map operator's lambda expression.

```
FlightsLib::combineScatterGatherResults(payload).flights
map  ( (flight) ->
//ComplexData::shuffleObject(flight, (sizeOf(flight) - 1) to 0 )
ComplexData::shuffleObject(flight, [0,1,6,4,5,7,3,2])
)
```

36. Call the ComplexData::buildXmlObject function with current body expression and the parent key string "flight"; you should see the array of objects converted to an object with a single flights root element.

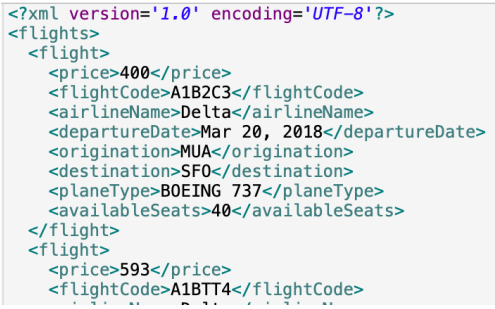
```
FlightsLib::combineScatterGatherResults(payload).flights
map  ((flight) ->
//ComplexData::shuffleObject(flight, (sizeOf(flight) - 1) to 0 )
ComplexData::shuffleObject(flight, [0,1,6,4,5,7,3,2])
)
ComplexData::buildXmlObject("flight")
```



Note: You will code this function in a later walkthrough.

37. Change the output directive to application/xml; you should see valid XML display without any errors.

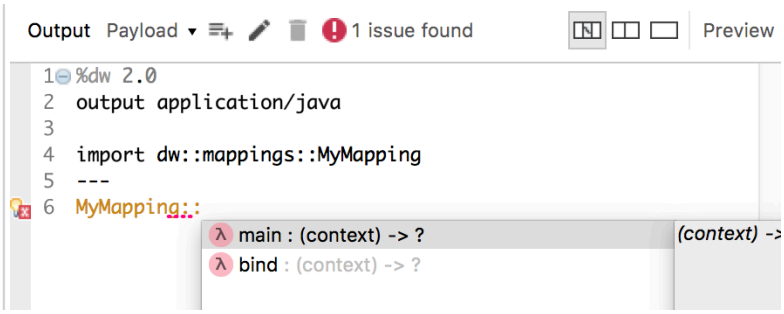
```
output application/xml
```



(Optional) Walkthrough 2-5: Create and reuse custom DataWeave mapping files

In this walkthrough, you extend the techniques from the last walkthrough to reuse DataWeave code. Instead of creating new functions, you store the complete DataWeave body expression and its associated variables and functions in a custom mapping file. Then you reuse this mapping file by calling it as a main() function call from another flow component. To properly carry out this main() function call, you also supply any required input arguments. You will:

- Create and reuse custom mapping files.
- Supply proper input arguments to a main function call to a mapping file.



Define a private function in a custom mapping file

1. In the src/main/resources/dw.mappings folder of your Mule project, create a new file named MyMapping.dwl.

   Note: If it does not already exist, create a folder named mappings in src/main/resources/dw.

2. Add a header input directive for a variable named someObject of type application/json.

   ```
   %dw 2.0
   input someElement application/json
   ```

   Note: The input directive is only used by custom mapping files to allow input parameters to be passed into the custom mapping file's body expression. It's never used by DataWeave code within a flow.

3. Add another input directive for a variable named formatter of type application/json.

   ```
   %dw 2.0
   input someElement application/json
   input formatter application/json
   ```

4. Import the capitalize function from the dw::core::Strings module.

   ```
   import capitalize from dw::core::Strings
   ```

5. Define a function named formatKey that capitalizes an input string and appends the string Key.

```
fun formatKey(aString: String) =
capitalize(aString) ++ "Key"
```

6. Define a function named createObjectFromArray that takes an Array argument named someElement and a String to String lambda expression argument named formatter.

```
fun createObjectFromArray(
       someElement: Array , formatter : (String) -> String
) = "???"
```

7. Implement this function to build a formatted key/value pair, where

   - The key name is created by passing the result of formatKey on the first element of the someElement array to the formatter function.
   - The value is the second element of the someElement array.

```
fun createObjectFromArray( someElement: Array ,
formatter : (String) -> String) =
{
      ( formatter( formatKey ( someElement[0] as String ) ) ): someElement[1]
}
```

   Note: The formatter function is not currently defined. It is named in the input directive. Remember that in DataWeave functions and variables are the same thing, so the input directive for the variable named formatter can be used here as a function call. There are not separate directives for variables vs. functions vs. lambda expressions.

Use private functions of the custom mapping in the same custom mapping file's body expression

8. Call the createObjectFromArray function using the someElement and formatter input arguments

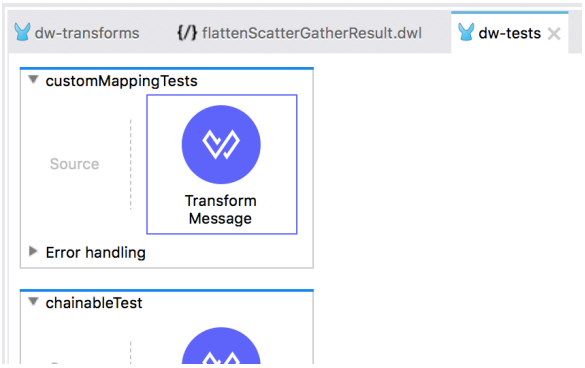
```
---
createObjectFromArray( someElement , formatter)
```

   Note: The arguments are declared in this custom mapping file as input variables of type application/json, but the value is not declared in this file's header. Instead, these arguments are passed in when this custom mapping file is used by another flow component. Each argument is also strongly typed in the function definition in the custom mapping file. The formatter argument is strongly typed as a (String) -> String lambda, which is an anonymous way of typing the allowed functions that can be set for this argument.

9. Save your changes to the MyMapping.dwl file.

Access the custom mapping file from another Mule component anywhere in the Mule application

10. Return to the dw-tests tab.
11. Drag out a Transform Message component and drop it in the canvas to create a new flow.
12. Rename the flow customMappingTest.
13. Select the Transform Message component.

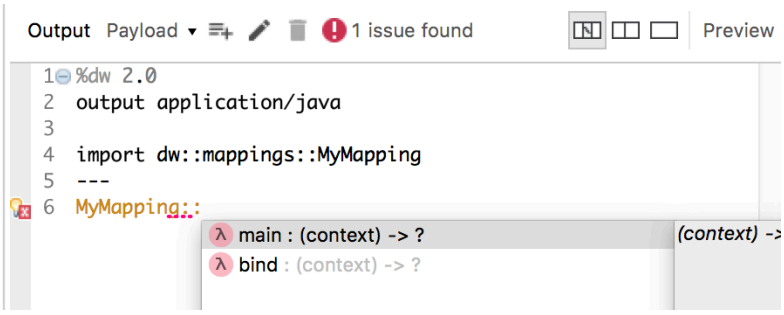


14. In the header, change the output to application/json and import the MyMapping.dwl file.

```
%dw 2.0
output application/json
import dw::mappings::MyMapping
```

In the body expression, call the MyMapping file's main method

15. In the body expression, type MyMapping::.
16. Use the content assist feature (type Ctrl+Space or right-click and select Content assist) to select the main method.



17. Verify you do not see the private functions formatKey nor createObjectFromArray.

    Note: Custom mapping files only expose their body expression via the main function. Any variables and functions declared in the custom mapping file's header are private and not available.

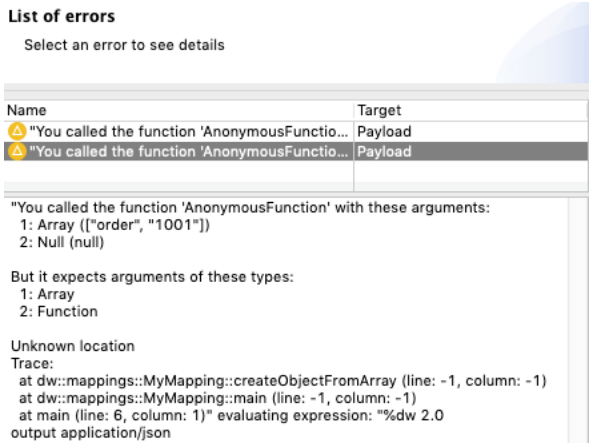
Pass in required input variables to the custom mapping file's main function

18. Assign the someElement variable in the main() function call by adding a key/value pair for the someElement key inside the main() function's Object argument and set the value to the array ["order","1001"]; an error should be reported.

    ```
    MyMapping::main( {someElement: ["order", "1001"]} )
    ```

    Note: The someElement and formatter input arguments must be declared using input directives in the custom mapping file.

19. Look at the error reported; it should show a second argument is missing in the main() function call, but main is called 'AnonymousFunction'.



20. Assign the formatter function in the main() function call by adding a key/value pair for the formatter key inside the main() function's Object argument, and set the formatter value as a simple lambda expression.

    ```
    MyMapping::main( {someElement: ["order", "1001"], formatter: (k)->k } )
    ```

    Note: Remember that the someElement and formatter input arguments must be declared using input directives in the custom mapping file. These names are case sensitive.

    Note: Because main accepts input variables in an object format, the order the key names appear does not matter.

21. Look in the Preview pane; you should see the order key is reformatted by the private formatKey function and transformed into a key/value pair {"OrderKey": "1001"}.

```
{
  "OrderKey": "1001"
}
```

22. Change the formatter value to call the lower function.

```
MyMapping::main( {someElement: ["order", "1001"], formatter: lower } )
```

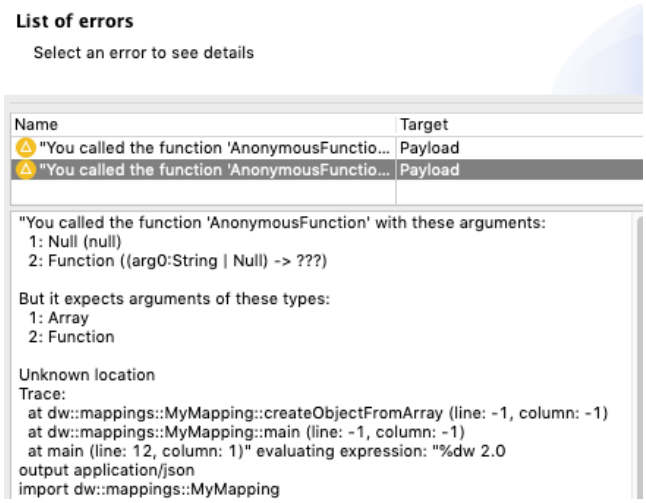
23. Look in the Preview pane; you should see the order key is now formatted in all lowercase.

```
{
  "orderkey": "1001"
}
```

24. In the body expression, change the MyMapping::main() function's input argument from someObject to incorrect.

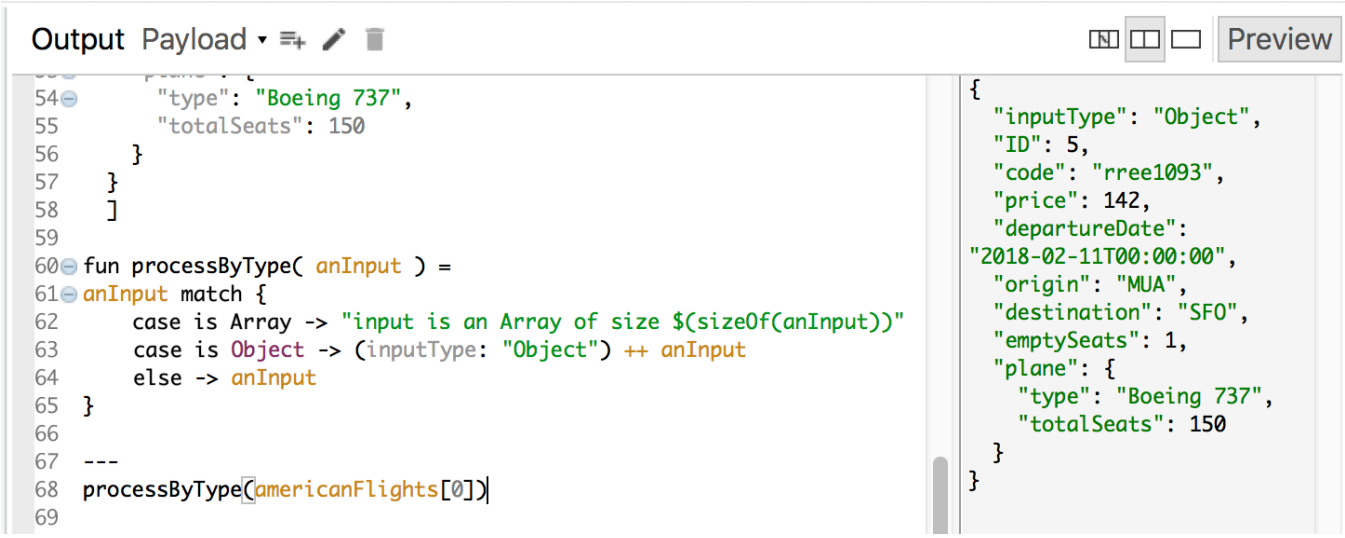
```
MyMapping::main( { incorrect: ["order", "1001"], formatter: lower } )
```

25. Verify an error indicates the someElement value in the createObjectFromArray function call was null.



26. Change the input argument named incorrect back to someElement.
27. Verify the error is fixed and the correct result appears in the Preview pane.

Module 3: Writing Defensive and Robust DataWeave



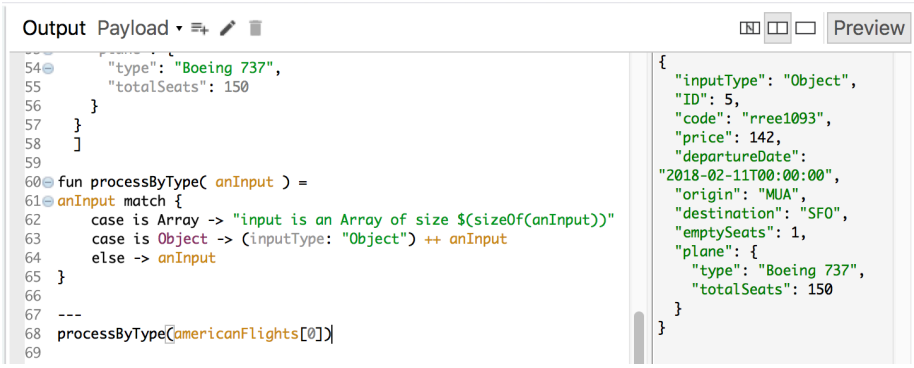
At the end of this module, you should be able to:

- Write more defensive DataWeave expressions that filter and route data based upon conditions.
- Write more robust functions using a match statement to test for data types.
- Handle errors.
- Log from inside DataWeave expressions.

Walkthrough 3-1: Match DataWeave types and conditions

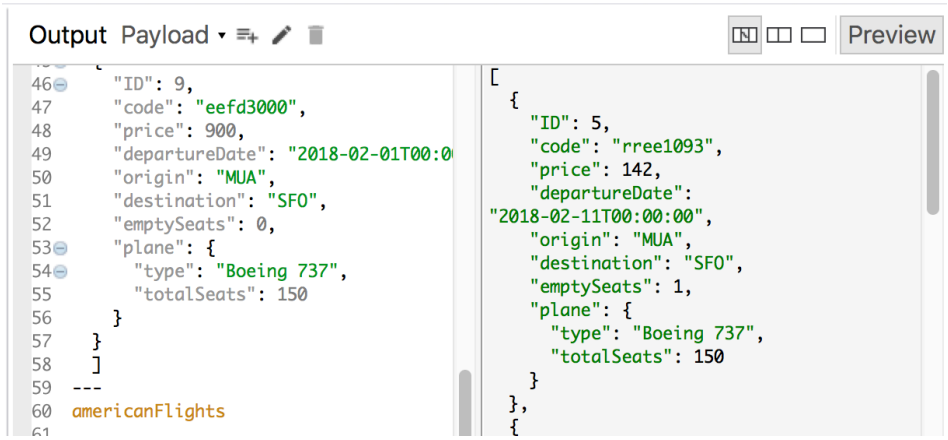
In this walkthrough, you investigate the various data types you can use in DataWeave with a match statement. This allows you to write one function for multiple types of input. You will:

- Write a DataWeave function that takes conditional action based on the input data type.
- Create various match statement conditions.



Overload a function to test for different types of input

1. Return to the dw-tests tab.
2. In the matchOperatorTest flow, select the Transform Message component.
3. In the Output pane, select Preview.
4. Verify the output is an array of flight objects.



5. Create a new function called overloaded with a single argument named anInput, and implement this function with strong typing to only accept array type input.

```
fun overloaded ( anInput : Array) =
"input is an Array of size $(sizeOf(anInput))"
```

6. In the body expression, call overloaded with a sample array.

```
overloaded( [1,2])
```

7. Look in the Preview pane; you should see the function prints the response message:

```
"input is an Array of size 2"
```

8. Add another function also named overloaded with a single argument named anInput typed as Object , and implement this function with strong typing to only accept object type input.

```
fun overloaded ( anInput : Object) =
"input is an Object of size $(sizeOf(anInput))"
```

9. In the body expression, call overloaded with a sample object.

```
overloaded( {one: "One", two: "Two", three: "Three"} )
```

10. Look in the Preview pane; you should see the function prints the response message:

```
"input is an Object of size 3"
```

Add an overloaded function to handle null inputs

11. Create another function called overloaded with a single argument named anInput typed as Null, and implement this function with strong typing to only accept Null type input.

```
fun overloaded ( anInput : Null) =
"input is Null"
```

12. In the body expression, call overloaded with a sample array.

```
overloaded( null )
```

13. Look in the Preview pane; you should see the function prints the response message:

```
"input is Null"
```

Start using a single match statement instead of overloaded functions

14. Create a new function named processByType with a single argument named anInput, and implement this function with a match statement with just a default else expression.

```
fun processByType( anInput ) =
anInput match {
  else -> anInput
}
```

15. Change the body expression to call this function with the americanFlights variable.

```
---
processByType( americanFlights )
```

16. Verify there are no errors.
17. Look in the Preview pane; you should see the same payload output.

    Note: So far, the additional match statement code has not changed the payload at all.

Return a different result based on the input data type

18. Add a selector to match if the input is an array, with a transformation expression that returns a string message that includes the size of the array.

```
fun processByType( anInput ) =
anInput match {
    case a if( a is Array ) -> "input is an Array of size $(sizeOf(a))"
    else -> anInput
}
```

19. Change the selector to a simpler case expression that does not name the input.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    else -> anInput
}
```

20. Look in the Preview pane; you should see the string "input is an Array of size 4".

    Note: This shows you the payload is being recognized as type Array.

21. Add another selector to match if the input is an object, with a transformation expression to just return a string indicating the input is an object.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> "input is an Object"
    else -> anInput
}
```

22. Change the body expression to:

```
processByType( americanFlights[0] )
```

23. Look in the Preview pane; you should see the output "input is an Object".

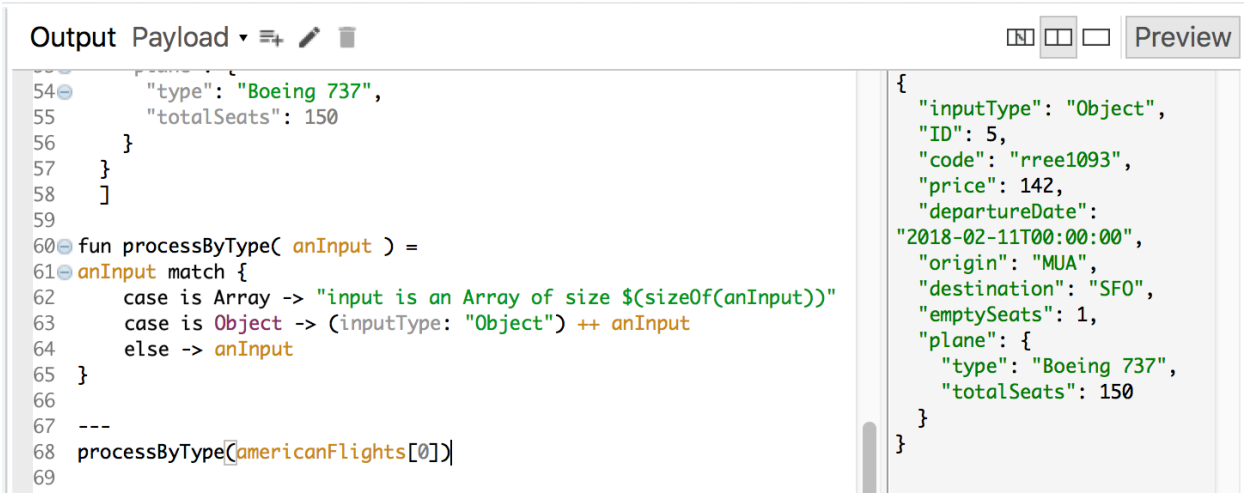
    Note: The match statement correctly distinguished between the Array type of americanFlights and the Object type of americanFlights[0]. Notice how this one function can test for different input types, rather than having to overload the function as you did with the overloaded function.

If the input is of type Object, concatenate a new key/value to the beginning of the input

24. In the processByType function, change the object type transformation to also concatenate a new key/value to the top of the input object.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    else -> anInput
}
```

25. Look in the Preview pane; you should see the first flight object from the payload prints out with a new inputType key/value concatenated to the beginning of the object.



26. Change the else transformation to print out the input's type.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    else -> typeOf( anInput )
}
```

27. Change the body expression to a number value:

```
processByType(americanFlights[0].ID)
```

28. Look in the Preview pane; you should see the string "Number", indicating the type that was matched in the match statement's else expression.
29. Modify the match statement's else expression to concatenate the input and the input type into a response message string.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    else -> "input " ++ anInput ++ " is of type " ++ typeOf( anInput )
}
```

30. Look in the Preview pane; you should see the output "input 5 is of type Number".

Add a condition to format numbers

31. Add a case statement with a selector to match if the input is of type Number and the transformation to format numbers to 2 decimal places.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    case is Number -> $ as String {format: "#,###.00"} as Number
    else -> "input " ++ anInput ++ " is of type " ++ typeOf( anInput )
}
```

32. Look in the Preview pane; you should see the formatted number 5.00.
33. Modify the Number case transformation expression to multiply the Number value by the exchangeRate var from the dw::modules::FlightsLib module.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    case is Number -> ($ * dw::modules::FlightsLib::exchangeRate)
                    as String {format: "#,###.00"} as Number
    else -> "input " ++ anInput ++ " is of type " ++ typeOf( anInput )
}
```

34. Look in the Preview pane; you should see the output changed from 5.00 to 6.75.

Add a case to transform the specific literal value "SFO" to "San Francisco"

35. Add a condition to test for the string literal "SFO" and replace it with "San Francisco"; make sure to write the case in the right location in the match statement, before other strings are matched.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    case "SFO" -> "San Francisco"
    case is Number -> $ as String {format: "#,###.00"} as Number
    else -> "input " ++ anInput ++ " is of type " ++ typeOf( anInput )
}
```

36. Change the body expression to call the processByType function with the input string "SFO".

```
processByType("SFO")
```

37. Look in the Preview pane; you should see the output is "San Francisco".

Verify the "SFO" case is in the right location in the match statement

38. Comment out the "SFO" case from the match statement.

```
fun processByType( anInput ) =
anInput match {
    case is Array -> "input is an Array of size $(sizeOf(anInput))"
    case is Object -> (inputType: "Object") ++ anInput
    //case "SFO" -> "San Francisco"
    case is Number -> $ as String {format: "#,###.00"} as Number
    else -> "input " ++ anInput ++ " is of type " ++ typeOf( anInput )
}
```

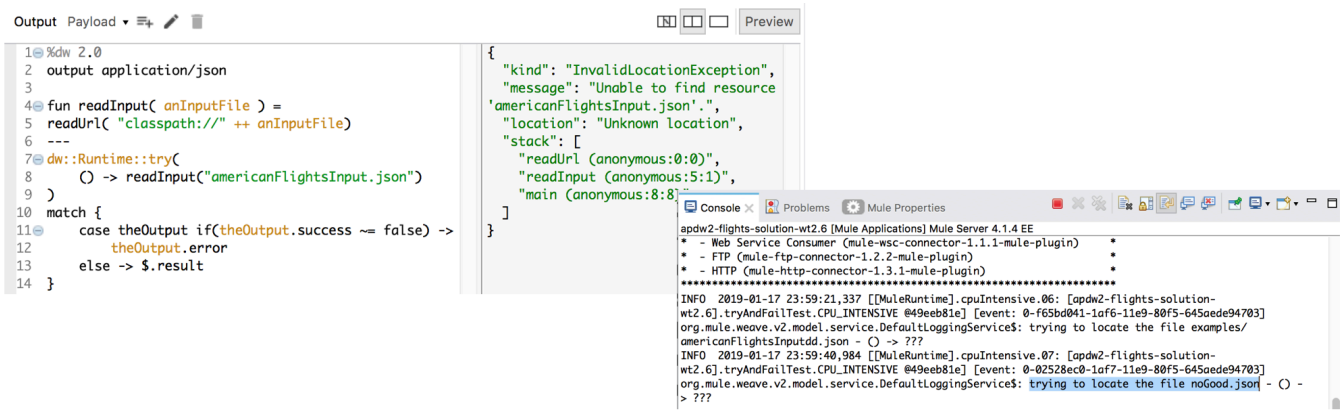
39. Look in the Preview pane; you should see the output changes to the else statement's transformation.

```
"input SFO is of type String"
```

Walkthrough 3-2: Make DataWeave expressions more defensive and robust

In this walkthrough, you wrap DataWeave expressions in a try function, and then match the result to handle and recover from various error conditions. You also learn how to throw exceptions when an error condition cannot be fixed, and how to log messages inside DataWeave expressions. You will:

- Test the result of an expression by using the try function.
- Try to fix a failed DataWeave expression by using an orElseTry function.
- Add final error handling by using an orElse function.
- Throw errors using the fail function.
- Log messages from inside DataWeave expressions.



Write a function to read in a file

1. Return to the dw-tests tab.
2. Drag out a new Transform Message component and drop it at the bottom of the canvas.
3. Name the new flow tryAndFailTest.
4. Select the Transform Message component and change the output directive to application/json.
5. Add a function named readInputFile that calls the readUrl function on an input file path.

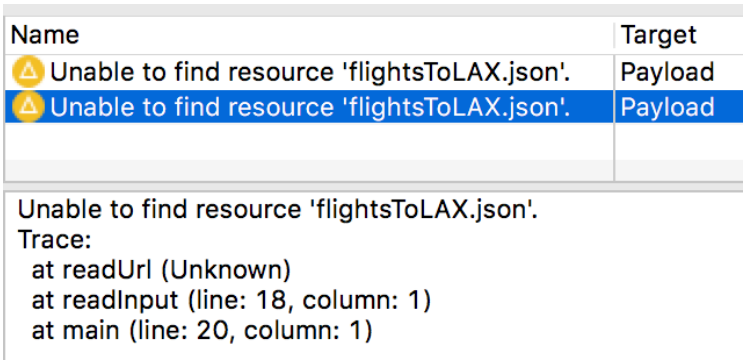
```
fun readInput( anInputFile ) =
readUrl("classpath://" ++ anInputFile)
```

Handle the case where this function is called with an invalid file path

6. In the Package Explorer view, verify the src/main/resources/examples folder contains the flightsToLAX.json; otherwise copy it from the examples/mockdata folder to the examples folder.
7. Call this function in the body expression.

```
readInput("flightsToLAX.json")
```

8. Verify an error is thrown indicating the file cannot be found.



9. Wrap the body expression in a call to the dw::Runtime::try() function.

```
dw::Runtime::try (
	readInput("flightsToLAX.json")
)
```

10. Verify the same error is reported, and the Preview pane does not show any results.
11. Modify the expression inside the try() function to be a lambda expression.

```
dw::Runtime::try (
	() -> readInput("flightsToLAX.json")
)
```

12. Verify the error is resolved but now appears in the TryResult object's error element.

Try another time if the first try fails

13. Add an import declaration to import the entire dw::Runtime module functions.

```
import * from dw::Runtime
```

14. Add an orElseTry function call to look for the file in the examples sub-folder, and also shorten the original dw::Runtime::try call to just try.

```
try (
	() -> readInput("flightsToLAX.json")
)
orElseTry (
	() -> readInput("examples/flightsToLAX.json")
)
```

Move the try function into the readInput function implementation

15. In the header, add an import statement to import all of the dw::Runtime module, so the functions and variables do not need to be prepended with the Runtime:: module prefix.

```
import * from dw::Runtime
```

16. Change the body expression to just call readInput without the try or orElseTry functions.

```
---
readInput("examples/flightsToLAX.json")
```

17. In the readInput function, wrap the readUrl function call in a try function call.

```
fun readInput( anInputFile ) =
try(
	readUrl("classpath://" ++ anInputFile)
)
```

18. Also move the orElseTry function call

```
fun readInput(anInputFile) =
try (
	() -> readUrl("classpath://" ++ anInputFile )
)
orElseTry (
	() -> readUrl("classpath://examples/" ++ anInputFile )
)
```

19. Look in the Preview pane; you should the result is successful.

```
{
  "success": true,
  "result": [
    {
      "price": 750.0,
      "flightCode": "A134DS",
      "availableSeats": 40,
      "planeType": "BOEING 777",
      "departureDate": "Apr 11, 2018",
```

Handle the case where the file cannot be found in an examples subfolder

20. Add an orElse function to call the fail function to notify the file cannot be found.

```
fun readInput(anInputFile) =
try (
      () -> readUrl("classpath://" ++ anInputFile )
)
orElseTry(
      () -> readUrl("classpath://examples/" ++ anInputFile )
)
orElse(
    fail("Can't find the file: " ++ anInputFile ++
    " nor /examples/" ++ anInputFile)
)
```

21. Look in the Preview pane; you should see the result value displayed.

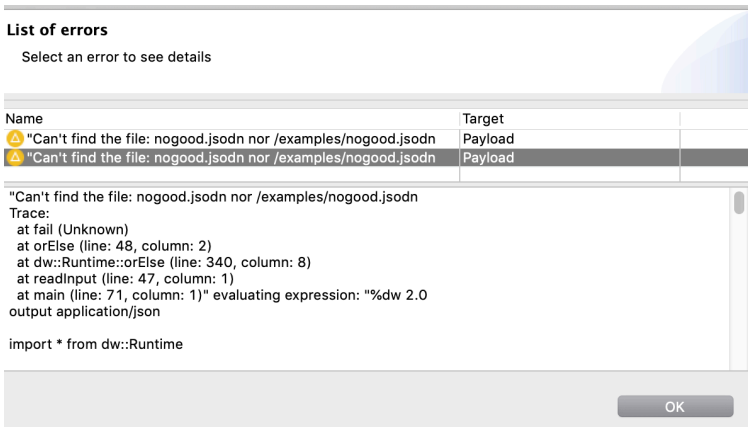
```
{
  "flights": [
    {
      "airlineName": "Delta",
      "availableSeats": 40,
      "departureDate": "Apr 11, 2018",
      "destination": "LAX",
      "flightCode": "A134DS",
      "origination": "MUA",
      "planeType": "BOEING 777",
      "price": "750.00"
    },
    {
      "airlineName": "Delta",
      "availableSeats": 18,
      "departureDate": "Aug 11, 2018",
      "destination": "LAX",
```

Note: This shows you that the orElse function masks the try and orElseTry mechanism for handled errors.

22. Change the body expression to an invalid file name; the Preview pane should still show the previous result and not refresh due to an error.

```
readInput("noGood.json")
```

23. View the error message; you should see your custom fail function error message.



Note: Just as with the successfully handled error, the try and orElseTry results are hidden by the orElse function. In this case the error is thrown by the Mule component and handled by the flow's error handler(s).

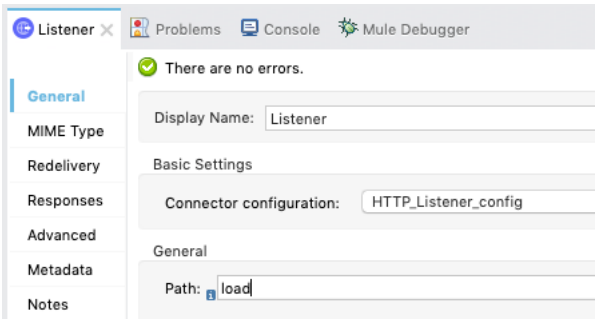
Add logging to the readInput function

24. Wrap the first case expression of the first match statement in a log function, and create a log message that indicates the input file could not be found but the function will try again inside the examples folder.

```
fun readInput(anInputFile) =
try (
    log( "Could not read in the file, will try in the examples sub-folder",
       () -> readUrl("classpath://" ++ anInputFile )
    )
)
orElseTry(
       () -> readUrl("classpath://examples/" ++ anInputFile )
)
orElse(
    fail("Can't find the file: " ++ anInputFile ++
    " nor /examples/" ++ anInputFile)
)
```

Verify how thrown errors propagate to the flow's error handler

25. Add an HTTP Listener to the tryAndFailTest flow.
26. Configure the HTTP Listener's Connectore configuration to HTTP_Listener_config and set the Path to /load.



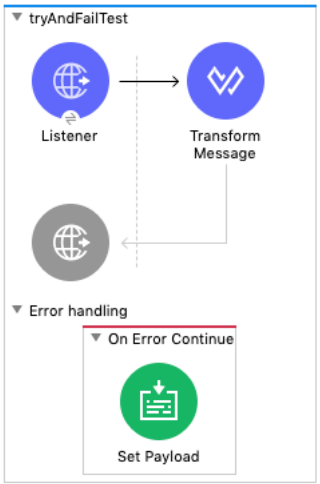
27. Change the body expression to look for the file in the filename query parameter.

```
readInput(attributes.queryParms.filename)
```

28. Change the body expression of the Transform Message component to call readInput with the filename query parameter, and provide a default filename value.

```
readInput(attributes.queryParams.filename default "flightsToLAX.json")
```

29. Add an On Error Continue scope to the flow.



30. In the On Error Continue scope, add a Set Payload component to set the payload to the received error object.

```
output application/json --- myError: error
```

Test the Mule application

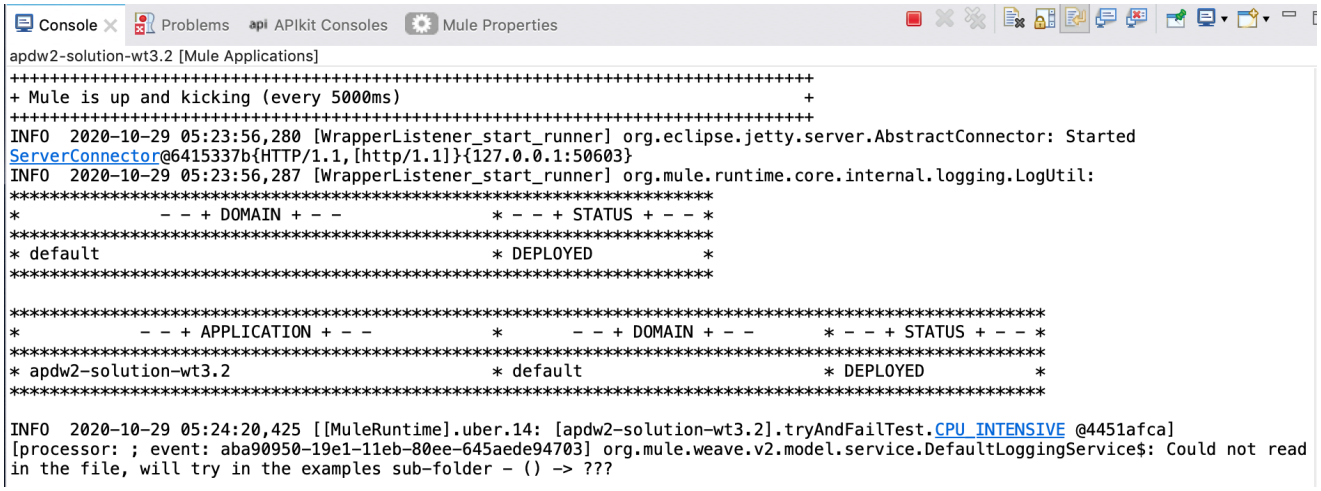
31. Debug the Mule application; ignore any warnings that errors exist in the project.
32. In a web client, submit a get request to:

    http://localhost:8081/load?filename=flightsToLAX.json

33. Verify the file is located in the examples/ folder, and the contents are returned to the web client.

View log messages in the Studio Console view

34. Look at the last log message in the Console view; it should be the log message from the tryAndFailTest flow.



Trace error messages from DataWeave to the parent Mule flow

35. Open a REST client such as the Advanced Rest Client.

    Note: You can also use a web browser but you'll have to use the developer tools to view the HTTP status code returned from the request.

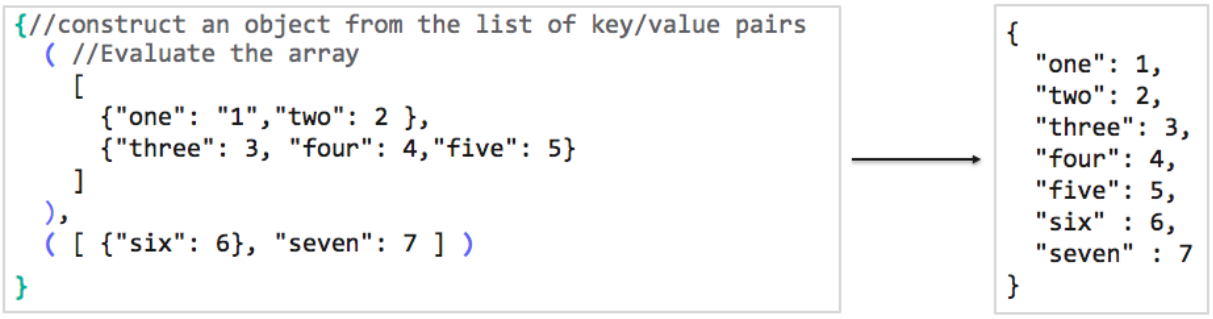
36. Change the filename query parameter value to noGood.json.

    http://localhost:8081/load?filename=noGood.json

37. Verify an error message generated in the On Error Continue scope is returned; you should see the root element myError you coded in the Set Payload component of the On Error Continue scope, and inside the error message you should see the message from the fail() function call inside the Transform Message component.

38. Verify the response status code is 200, because the error was handled by an On Error Continue scope.
39. Stop the debugger.

Module 4: Constructing Arrays and Objects



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

- Add components to and remove elements from arrays and objects.
- Construct objects from lists of DataWeave expressions by using object constructor curly braces { }.
- Troubleshoot common issues when using object constructor curly braces { }.

Walkthrough 4-1: Add and remove data from objects and arrays

In this walkthrough, you use the ++, +, -- and – operators to combine and remove objects and arrays from DataWeave expression results. You will:

- Combine objects and arrays by using the ++ and + operators.
- Remove parts of an object or array by using the -- and - operators.

Create a variable with an example array value and a variable with an example object value

1. Return to the dw-tests tab.
2. Drag out a Transform Message component and drop it in the canvas.
3. Rename the new flow addAndConcat.
4. In the Output pane, in the header, change the output to application/json.
5. In the header, define a variable named array1 containing a Number type value, a String type value, and a Boolean type value.

```
var array1 = [1, "a", true]
```

6. Define a variable named object1 with two key/value pairs.

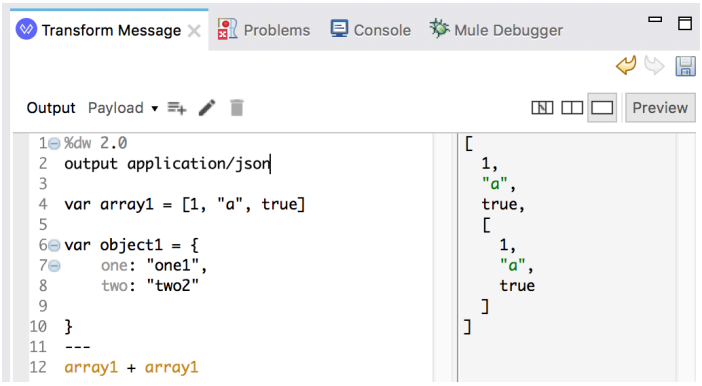
```
var object1 = {
    one: "one1",
    two: "two2"
}
```

Identify differences between adding vs. concatenating arrays together

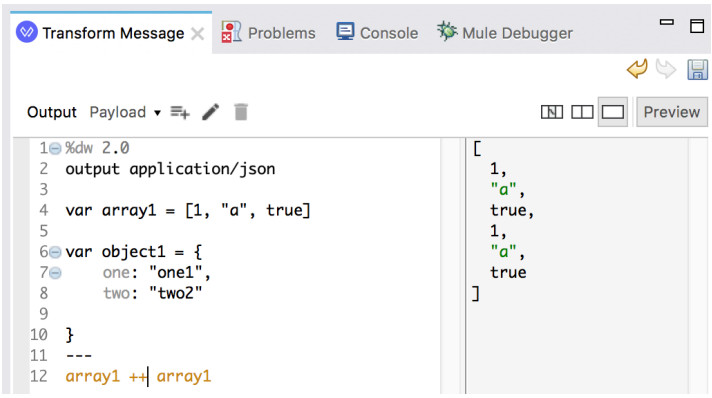
7. Change the body expression to add array1 with itself.

```
array1 + array1
```

8. Look in the Preview pane; you should see array1 is added as a forth element of array1, so there are four elements in the output array, with the last element being all of array1.



9. Change the + operator to the ++ concatenation operator and view the change in the Preview pane.

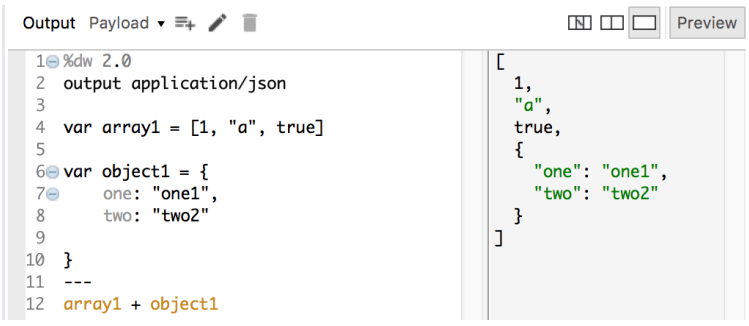


Identify what happens when an array and object are added or concatenated together

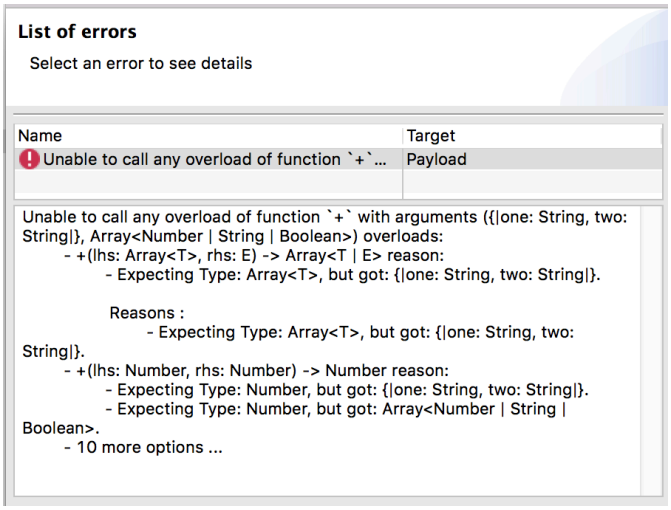
10. Change the body expression to add object1 to array1.

```
array1 + object1
```

11. Look in the Preview pane; you should see object1 is added as the fourth element of the output array.



12. Reverse the order of array1 and object1.
13. Verify an error appears that indicates the + operator cannot be overloaded with the provided argument types.



14. Change the body expression to:

```
array1 ++ object1
```

15. Verify this generates some errors.

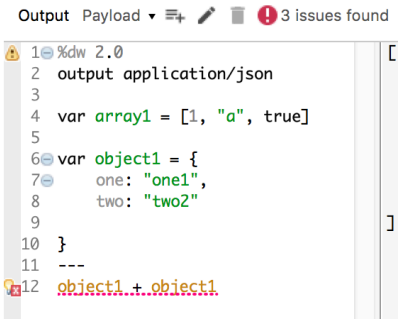
16. Inspect the errors and verify they report the ++ operator cannot be overloaded with these types of arguments.

Compare adding vs. concatenating together two objects

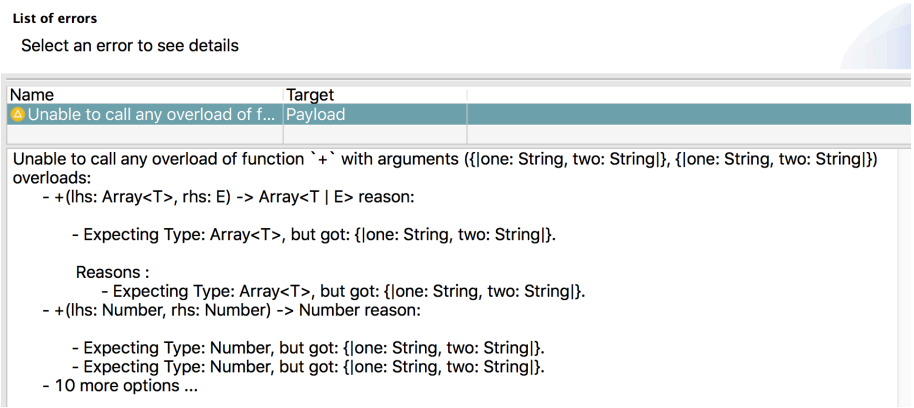
17. Attempt to add two objects together.

```
object1 + object1
```

18. Verify this expression generates some errors.



19. Inspect the errors and verify they indicate the + operator cannot be overloaded with Objects for each of the two arguments.



20. Change the body expression to:

```
object1 ++ array1
```

21. Verify this generates errors.
22. Inspect the errors and verify they report the ++ operator cannot be overloaded with these types of arguments.

    Note: the + operator adds object1 as a child, but the ++ operator combines the key/values into a single object.

23. In the body expression, change the + add operator to the ++ concatenation operator.

```
object1 ++ object1
```

24. Verify the two objects are merged together into one object with four key/value pairs.

Remove a key from an object

25. Remove the key "one" from the current body expression.

```
object1 ++ object1 - "one"
```

26. Look in the Preview pane; you should see only the last key/value pair is removed.

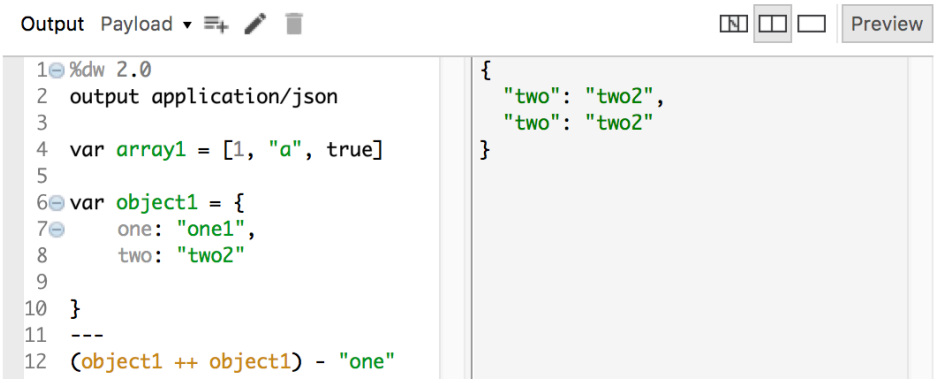
27. Add evaluation parentheses () around the - operator expression.

```
object1 ++ ( object1 - "one" )
```

28. Look in the Preview pane; you should see the result is unchanged.
29. Move the evaluation parentheses () to surround the first concatenation expression.

```
( object1 ++ object1 ) - "one"
```

30. Verify both key/value pairs with the key "one" are removed.

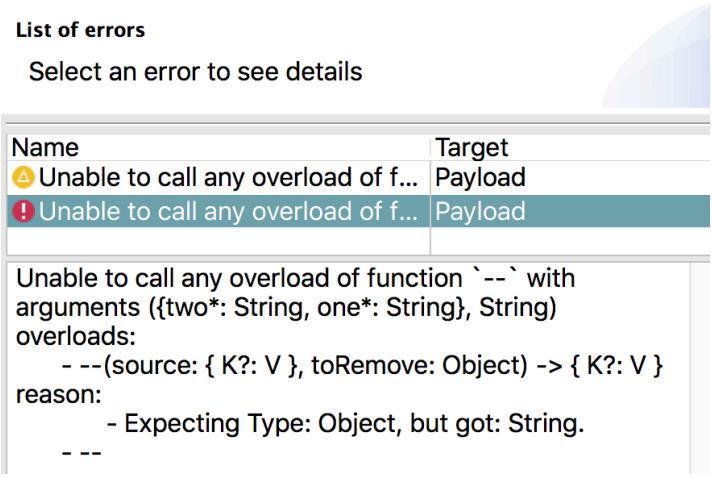


Remove key/values using the minus minus -- operator

31. Attempt to remove the key "one" using the minus minus -- operator.

```
( object1 ++ object1 ) -- "one"
```

32. Verify this generates errors indicating you cannot overload the -- operator with a String argument.



33. Change the string "one" to a key/value pair.

```
( object1 ++ object1 ) --  one:"one1"
```

34. Verify both copies of the "one" key/value pair are removed.

35. Change the removed value to "two2".

```
( object1 ++ object1 ) --  one:"two2"
```

36. Verify no key/value pairs are removed.
37. In the body expression, replace the key/value pair with object1.

```
( object1 ++ object1 ) --  object1
```

38. Look in the Preview pane; you should see the result is an empty Object.

Walkthrough 4-2: Zip, unzip, and flatten arrays

In this walkthrough, you zip, unzip, and flatten arrays. You also combine these functions to see their behavior together. You will:

- Zip and unzip two arrays.
- Flatten inner array elements of an array.

Zip two arrays together

1. Return to the dw-test tab.
2. From the Mule palette, drag out a Transform Message component and drop it in the canvas to create a new flow.
3. Name the new flow zipTests.
4. Change the output directive to application/json.
5. In the header, define two arrays with different elements and different sizes (number of elements).

```
var array1 = ["A","B","C","D"]
var array2 = [1,2,3]
```

6. In the body expression, zip the two arrays together.

```
array1 zip array2
```

7. Look in the Preview pane; you should see the result is an array of arrays, with the child arrays gathering corresponding array indexes.

8. Verify the fourth element D of array1 is skipped when the two arrays are zipped together.

Rewrite the zip operator as a function with two arguments

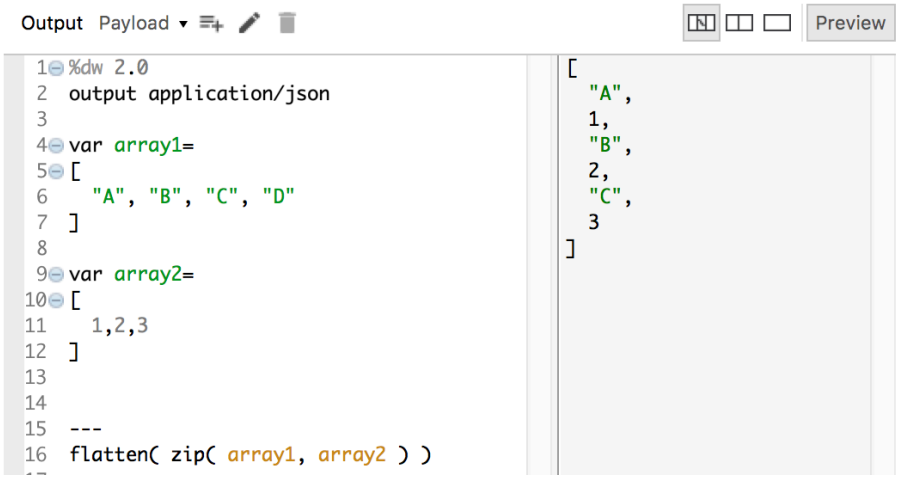
9. Change the body expression to:

```
zip( array1, array2 )
```

10. Look in the Preview pane; you should see the result is unchanged.
11. Flatten the zipped array.

```
flatten( zip( array1, array2 ) )
```

12. Look in the Preview pane; you should see the first three elements of array1 ["A", "B", "C"] is zipped together with array2 [1,2,3] into one flat array [“A”, 1, “B”, 2, “C”, 3].

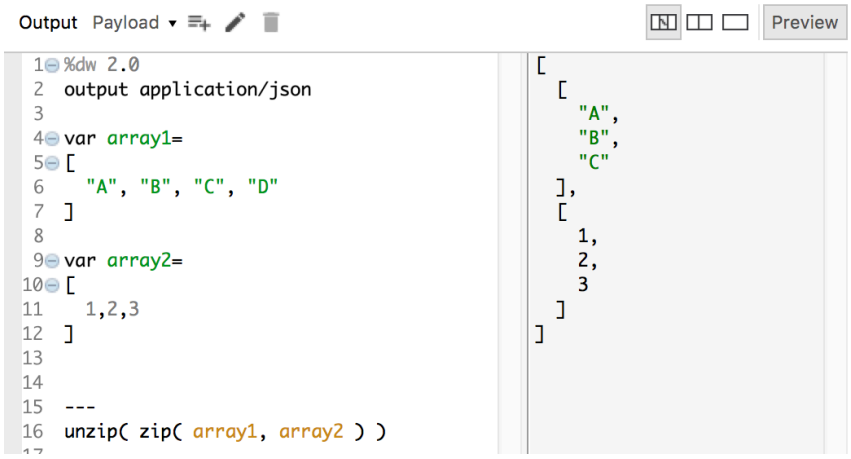


Use the unzip function on some zipped arrays

13. Change the body expression to unzip the two zipped arrays.

```
unzip( zip ( array1, array2 ) )
```

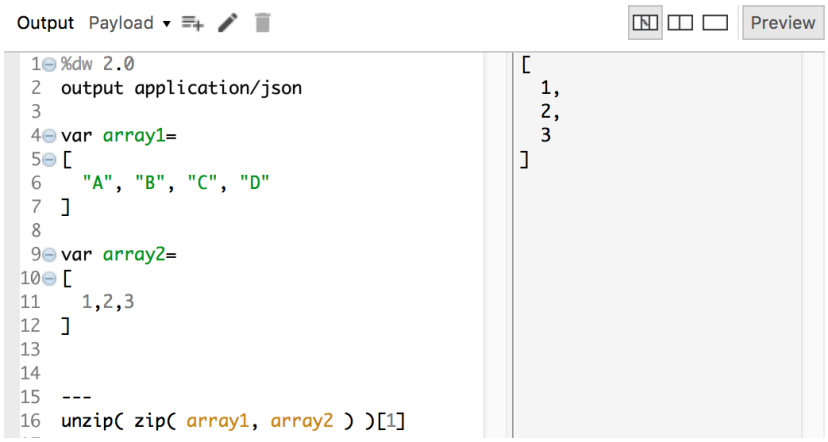
14. Verify the fourth element "D" of array1 is not recovered by the unzip operation, and does not appear in the first child array of the result.



15. Write a body expression to recover the smaller array2 from the unzipped arrays.

```
unzip( zip ( array1, array2 ) )[1]
```

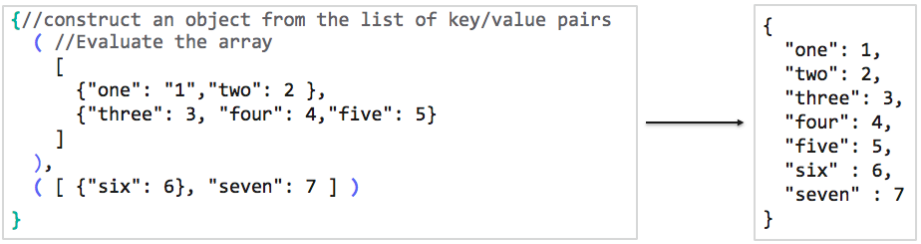
16. Look in the Preview pane; you should see array2.



Walkthrough 4-3: Construct objects from lists of expressions by using object constructor curly braces

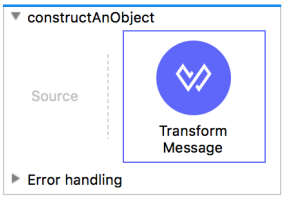
In this walkthrough, you use object constructor curly braces { } to combine lists of elements into a new Object. You will:

- Apply object constructor curly braces to a list of elements.
- Flatten an array of objects to a single object by using object constructor curly braces.



Construct an object from other objects and key/value pairs

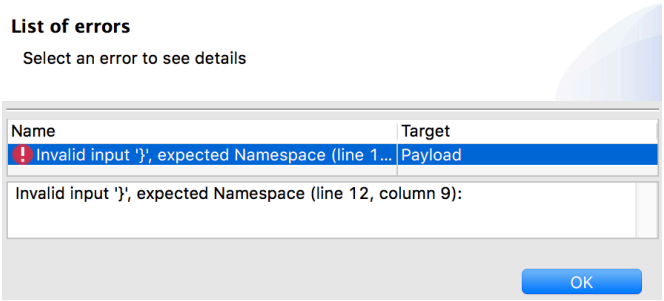
1. Return to the dw-tests tab.
2. In the canvas, copy/paste the addAndConcat flow to a new flow named constructAnObject.



3. Change the body expression to apply object constructor curly braces around object1.

```
---
{ object1 }
```

4. Verify the errors indicate a problem with the closing object constructor curly brace '}' surrounding object1.



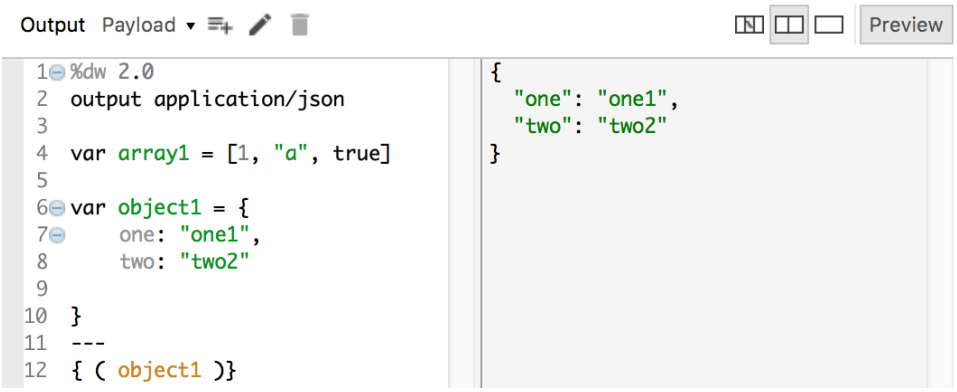
5. Surround the expression inside the object constructor curly braces with evaluation parentheses.

```
{ ( object1 ) }
```

6. Verify the errors are fixed.

Note: The evaluation parentheses had to be added so that the object constructor curly braces can extract all the key/value pairs from object1.

7. Look in the Preview pane; you should see the original object1 key/value pairs appear unchanged.



Note: Each key/value pair is extracted from object1, then reassembled into a new object with the same key/value pairs.

8. Add a key/value pair to the object constructor curly braces list of elements.

```
{ ( object1 ) , three: "three3" }
```

9. Look in the Preview pane; you should see all the key/value pairs have been combined into one object.

Note: The object constructor curly braces are applied one at a time to each of its elements, each time extracting the key/value pairs in each element, then all the key/value pairs are combined into one object.

Properly use object constructor curly braces around an array of objects

10. Define a new variable named array2 that contains two objects constructed with object1 as the value of each key/value.

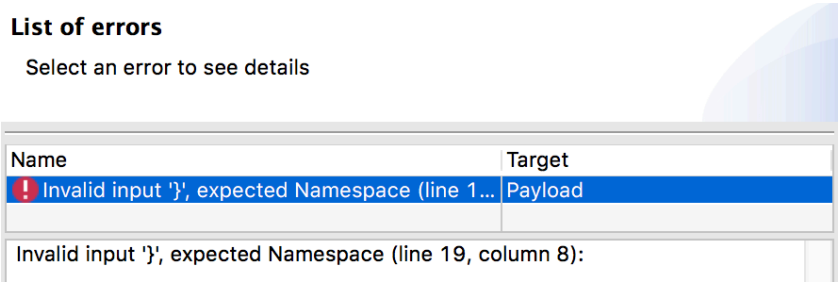
```
var array2  = [
       objOne: object1,
       objTwo: object1
]
```

11. Change the body expression to array2.
12. Look in the Preview pane; you should see the array of objects appears, with each object's value object1.

13. Select the Transform Message component.
14. Apply object constructor curly braces around array2.

```
{ array2 }
```

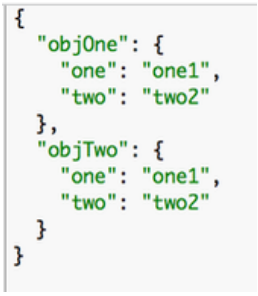
15. Verify a similar error is reported as before, indicating a problem with the closing object constructor curly brace '}' around array2.



16. In the body expression, surround the body expression inside the object constructor curly braces with evaluation parentheses.

```
{ ( array2 ) }
```

17. Verify the errors are fixed and the Preview pane shows an object of objects.



```
{
  "objOne": {
    "one": "one1",
    "two": "two2"
  },
  "objTwo": {
    "one": "one1",
    "two": "two2"
  }
}
```

Compare how object constructor curly braces change arrays of objects, objects, and key/values

18. In the body expression, create an object with the current body expression used as a value for a key named extracted, and array2 as the value to a key named original.

```
{
    extracted: {(array2)},
    original: array2
}
```

19. Verify the original key with value array2 is an array of two objects, but the extracted key with value {(array2)} is a flattened object of the two array2 objects.

20. Apply the object constructor curly braces to a list of elements, where the elements are array2, object1, and a new key/value pair; remember to enclose the Array and Object in evaluation parentheses.

```
{ (array2), (object1), four:"four4" }
```

21. Look in the Preview pane; you should see a new flattened object is constructed from the key/value pairs of all the elements in the comma separated list.

Module 5: Operating upon Arrays and Objects

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

- Conditionally test, count, and sum up elements of an array by using the dw::core::Arrays module.
- Divide an array into smaller child arrays using the divide function.
- Extract out key names, key namespaces, key attributes, and values of an object by using the dw::core::Objects module.
- Select key names, key namespaces, key attributes, and values of descendants of an object by using selectors.

Walkthrough 5-1: Operate upon array elements by using the dw::core::Arrays module

In this walkthrough, you use functions from the dw::core::Arrays module to operate on elements of an array. You will:

- Test if some or all array elements match a condition.
- Count how many array elements match a condition.
- Apply a function to each element of an array then sum the results together.
- Divide an array into smaller sub-arrays.

```
%dw 2.0
output application/json
//import * from dw::core::Arrays
import * from dw::core::Arrays

var flights =
readUrl("classpath://examples/americanFlightsResponse.json")

---
{
  smallerPlanes: flights..totalSeats countBy ($ < 200),
  totalNumberPlanes: sizeOf( flights )
}
```

```
{
  "smallerPlanes": 4,
  "totalNumberPlanes": 5
}
```

Read in an example data file with which to work

1. Return to the dw-tests tab.
2. From the Mule palette, drag out a Transform Message component and drop it in the canvas.
3. Rename the new flow testArraysModule and change the output directive to application/json.
4. In the Transform Message component, read in the examples/extra/americanFlightsResponse.json file and assign it to a variable named flights.

```
%dw 2.0
output application/json
var flights = readUrl("classpath://examples/extra/americanFlightsResponse.json")
---
flights
```

5. Look in the Preview pane; you should see the American flights are displayed.

```
[
  {
    "ID": 9,
    "code": "eefd3000",
    "price": 900,
    "departureDate": "2018-02-01T00:00:00",
    "origin": "MUA",
    "destination": "SFO",
    "emptySeats": 0,
    "plane": {
      "type": "Boeing 737",
      "totalSeats": 150
    }
  },
  {
    "ID": 7,
    "code": "eefd1994",
    "price": 676,
    "departureDate": "2018-01-01T00:00:00",
```

Use the some function to test if any array elements match a condition

6. In the header, import the some function from the dw::core::Arrays module.

```
import some from dw::core::Arrays
```

7. Test if any objects have totalSeats with a value over 200.

```
flights..totalSeats some ($ > 200)
```

8. Look in the Preview pane; you should see the result is true.
9. Change the import expression to require using the Arrays namespace.

```
%dw 2.0
output application/json
//import some from dw::core::Arrays
import dw::core::Arrays
```

10. Change the body expression to use a lambda expression to test if there are more than 400 totalSeats.

```
var flights = readUrl("classpath://examples/flightsAllAirlinesLAX.json ")
---
//flights..totalSeats some ($ > 200)
Arrays::some(flights..totalSeats, (ts) -> (ts > 400))
```

11. Look in the Preview pane; you should see the result is false

*Note: This means **no** child flight objects in the flights array have more than 400 total seats available.*

Use the every function to test if every array element matches a condition

12. Change the import directive to:

```
import * from dw::core::Arrays
```

13. Change the body expression to test if every object has totalSeats with a value less than 400.

```
flights..totalSeats every ($ < 400)
```

14. Look in the Preview pane; you should see the result is the boolean value true.

Count the number of flights with less than 200 total seats

15. Change the body expression use the countBy function to count the number of flight objects with a totalSeats < 200, and to also print out the number of flight objects.

```
{
  smallerPlanes: flights..totalSeats countBy ($ < 200),
  totalNumberPlanes: sizeOf( flights )
}
```

16. Look in the Preview pane; you should see the count is less than the size of the flights array.

Use the sumBy function with various conditions

17. Change the body expression to use the sumBy function to add the totalSeats values from all the flights.

```
{
    totalSeatsAllPlanes: flights..totalSeats sumBy $
}
```

18. Look in the Preview pane; you should see the number 900 is displayed.
19. Change the body expression to use the sumBy function to add up the totalSeats values from each flight that matches the condition totalSeats < 200.

```
{
    smallerPlanes: flights..totalSeats sumBy ( if($<200) $ else 0 )
}
```

20. Look in the Preview pane; you should see the number 600 is displayed.
21. Change the body expression to use the sumBy function to add up the totalSeats values from each flight that matches the condition totalSeats < 200.

```
{
    smallerPlanes: flights..totalSeats filter ($ < 200 ) sumBy $
}
```

22. Look in the Preview pane; you should see the same number 600 is displayed.
23. Change the body expression to use the sumBy function to add up the sum of squares of the totalSeats values from each flight that matches the condition totalSeats < 200.

```
{
    smallerPlanes: flights..totalSeats sumBy ( if($<200) $*$ else 0 )
}
```

24. Look in the Preview pane; you should see the number 90000 is displayed.

Split up an array into equal sized child arrays by using the divideBy operator

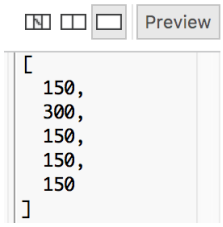
25. Change the import directive to require the Arrays:: namespace to be prepended when referencing variables and functions from the module, such as Arrays::sumBy.

```
import dw::core::Arrays
```

26. Change the body expression back to the array of totalSeats values.

```
flights..totalSeats
```

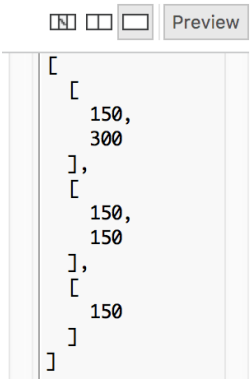
27. Look in the Preview pane; you should see the result is an array of totalSeats values.



28. Use the Arrays module's divideBy function to divide the array of totalSeats values into child arrays of size 2.

```
flights..totalSeats Arrays::divideBy(2)
```

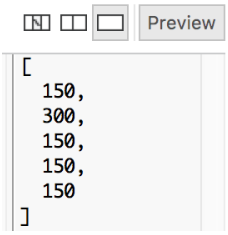
29. Look in the Preview pane; you should see the total seats array is divided up into child arrays of size 2, except for the last child array that is size 1, because there were no longer enough array elements remaining.



30. Use the flatten function to extract all the child array elements into a single fattened array.

```
flatten( flights..totalSeats Arrays::divideBy 2 )
```

31. Look in the Preview pane; you should see the result is the same as flights..totalSeats.



Walkthrough 5-2: Select and merge parts of objects by using the dw::core::Objects module

In this walkthrough, you use functions from the dw::core::Objects module to extract parts of an object. You will:

- Separate out the key, value, and attributes of each element of an object by using dw::core::Objects module functions.
- Merge one object with another object by using the mergeWith function.

Create an object with namespaces and attributes

1. Return to the dw-tests tab.
2. From the Mule palette, drag out a Transform Message component and drop it in the canvas.
3. Rename the new flow to testObjectsModule.
4. Change the output to application/dw.
5. In the header, add a namespace directive with a new namespace named myns; the namespace URL can be anything.

   ```
   ns myns http://mydomain.com
   ```

6. Define a variable that uses the myns namespace, and add some attributes to the elements.

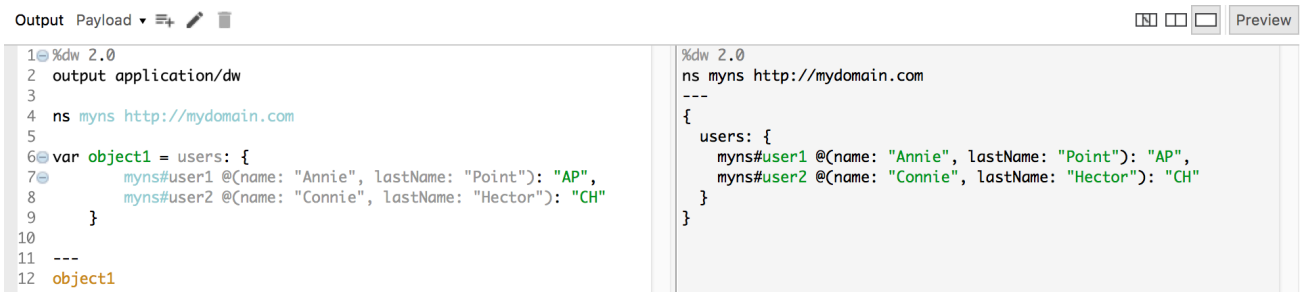
   ```
   var object1 =
   {
      users:
      {
         myns#user1 @(name: "Annie", lastname: "Point"): "AP",
         myns#user2 @(name: "Connie", lastname: "Hector"): "CH"
      }
   }
   ```

View how attributes and namespaces are represented in different output formats

7. Change the body expression to:

```
---
object1
```

8. Look in the Preview pane; you should see the namespace and attributes appear.



9. Change the output directive to application/xml.
10. Look in the Preview pane; you should see valid XML displays.

```
<?xml version='1.0' encoding='UTF-8'?>
<users>
  <myns:user1 xmlns:myns="http://mydomain.com" name="Annie" lastName="Point">AP</myns:user1>
  <myns:user2 xmlns:myns="http://mydomain.com" name="Connie" lastName="Hector">CH</myns:user2>
</users>
```

11. Change the output directive to application/json.
12. Look in the Preview pane; you should see the namespace and attributes are skipped.

```
{
  "users": {
    "user1": "AP",
    "user2": "CH"
  }
}
```

Extract key names and values from an object

13. Change the body expression to use the namesOf function on object1.users; the result should be an array of user key names.

```
namesOf( object1.users )
```

```
[
  "user1",
  "user2"
]
```

14. Change the body expression to use the valuesOf function to display the values of each user element.

```
valuesOf( object1.users )
```

```
[
  "AP",
  "CH"
]
```

15. Change the body expression to use the keysOf function to display the key sets of each user element; the result should be the same as the result of the namesOf function.

```
keysOf( object1.users )
```

```
[
  "user1",
  "user2"
]
```

Note: Unlike the namesOf function result, which only contains the key name, the keysOf function returns a key set contains all the components of a key - the namespace, the key name, and any attributes.

Extract the namespace and attributes for each user element's key set

16. Change the body expression to use a map operator and the # selector to extract the namespace for each key set; the result should be the namespace URL for each key.

```
keysOf(object1.users) map ( user ) -> user.#
```

```
[
  "http://mydomain.com",
  "http://mydomain.com"
]
```

Find out the type of Array created by the keysOf function

17. Change the map expression to print the type of each user element of the Array produced by the keysOf function.

```
keysOf(object1.users) map ( user ) ->  "index$$": typeOf( user )
```

```
[
  {
    "index0": "Key"
  },
  {
    "index1": "Key"
  }
]
```

Extract the entry set from each user element

18. Change the keysOf function to the entriesOf function; you should see the function outputs an Object for each of the object1 users.

```
entriesOf(object1.users) map ( user ) ->  "index$$": typeOf( user )
```

```
[
  {
    "index0": "Object"
  },
  {
    "index1": "Object"
  }
]
```

19. Change the body expression to just apply entriesOf to object1.users.

```
entriesOf (object1.users)
```

```
[
  {
    "key": "user1",
    "value": "AP",
    "attributes": {
      "name": "Annie",
      "lastName": "Point"
    }
  },
  {
    "key": "user2",
    "value": "CH",
    "attributes": {
      "name": "Connie",
      "lastName": "Hector"
    }
  }
]
```

Note: The entriesOf function result contains the key, value, and any attributes for each element of the object.

Extract the namespace, attributes, and value for each user element from the entry set

20. Change the body expression to use a map operator and the # selector to extract the namespace for each key set returned by the entrySet function; you should see an array of namespace URLs.

```
entriesOf (object1.users) map ( user ) -> ( user.key.# )
```

```
[
  "http://mydomain.com",
  "http://mydomain.com"
]
```

21. Change the map operator lambda expression to list the attributes for each user element returned by the entrySet function; you should see an array of objects, with two attributes each.

```
entriesOf (object1.users) map ( user ) -> ( user.attributes )
```

```
[
  {
    "name": "Annie",
    "lastName": "Point"
  },
  {
    "name": "Connie",
    "lastName": "Hector"
  }
]
```

Note: If Preview fails to display the attributes, add an HTTP Listener to the flow and run the debugger. The result should be returned to a web client request to this flow.

22. Change the map operator lambda expression to list the value for each user element returned by the entrySet function; you should see an array of two values.

```
entriesOf (object1.users) map (user) -> ( user.value )
```

```
[
  "AP",
  "CH"
]
```

Merge one object with another object by using the merge function

23. Import the Objects module.

```
import dw::core::Objects
```

24. Change the body expression to merge an object {one:"Number1} into an object {one:"ONE", one: "ONE2", two:"TWO"}.

```
{one:"ONE", one: "One2", two:"TWO"} Objects::mergeWith {one:"Number1"}
```

25. Look in the Preview pane; all the one keys from the first object should be replace with the key/value from the second object.

```
{
    "two": "TWO",
    "one": "Number1"
}
```

Walkthrough 5-3: Select values and parts of keys from descendant objects

In this walkthrough, you write DataWeave expressions for a particular key name to extract one or all values or key/values in a data structure matching the key. You also write DataWeave expressions to extract namespaces and attributes of the keys of descendant objects. You will:

- Select one or multiple values or key/values from descendant objects that match a key name.
- Select key namespaces or key attributes from descendant objects.

Import some complex data with which to work

1. Return to the dw-tests tab.
2. In the Mule palette, drag out a Transform Message component and drop it into the canvas to create a new flow.
3. Name the flow testSelectors.
4. Select the Transform Message component.
5. Change the output directive to application/xml.
6. Declare a variable named payload and assign it to the XML content of the examples/joinedFlightsAndAirportsNS.xml file.

   ```
   var payload =
   readUrl("classpath://examples//joinedFlightsAndAirportsNS.xml",
   "application/xml")
   ```

7. Set the body expression to payload.

8. Look in the Preview pane; you should see a complex object with several XML namespaces and attributes.

```
<?xml version='1.0' encoding='UTF-8'?>
<flights xmlns="http://acme.com" company="MUA" BU="travel">
  <fl:flight airline="american" xmlns:fl="http://flights.com" xmlns:ap="http://airports.com">
    <fl:available-seats>+40.00</fl:available-seats>
    <fl:airline-name>american</fl:airline-name>
    <fl:flight-code>AA103</fl:flight-code>
    <fl:departure-date>apr 11, 2018</fl:departure-date>
```

Explore the structure of the complex XML object

9. Verify the flights key contains two attributes named company and BU.
10. Verify each child flight key contains an airline attribute.
11. Verify the flights key uses the default namespace, but each flight key and its outer child elements uses the fl namespace, and each nested descendant destination object's keys use the ap namespace.

```
    <fl:price>+750.00</fl:price>
    <fl:destination airportCode="klax">
      <ap:airport-name xmlns:ap="http://airports.com">los angeles international airport</ap:airport-name>
      <ap:city xmlns:ap="http://airports.com">los angeles</ap:city>
      <ap:altitude xmlns:ap="http://airports.com">125</ap:altitude>
      <ap:icao xmlns:ap="http://airports.com">klax</ap:icao>
      <ap:longitude xmlns:ap="http://airports.com">33.94250107</ap:longitude>
    </fl:destination>
  </fl:flight>
  <fl:flight xmlns:fl="http://flights.com" airline="delta">
    <fl:available-seats>+18.00</fl:available-seats>
```

Select the value or the key/value of the root key of the payload object

12. In the body expression, select the flights key.

```
payload.flights
```

13. Look in the Preview pane; you should see an error reports the result cannot be converted to valid XML because there is not a unique root element.
14. Change the output directive to application/json.

15. Look in the Preview pane; you should see an object of flight objects appears, with the namespaces and attributes ignored.


```
{
  "flight": {
    "available-seats": "+40.00",
    "airline-name": "american",
    "flight-code": "AA103",
    "departure-date": "apr 11, 2018",
    "plane-type": null,
    "origination": "mua",
    "price": "+750.00",
    "destination": {
      "airport-name": "los angeles international airport",
      "city": "los angeles",
      "altitude": "125",
      "icao": "klax",
      "longitude": "33.94250107"
    }
  },
  "flight": {
    "available-seats": "+18.00",
    "airline-name": "delta",
```


16. Change the body expression to use the & key/value selector to include the flights key in the selected result.

```
payload.&flights
```

17. Look in the Preview pane; you should see the result is a single key/value pair with outer key name "flights", and the flights element's value is the same as the result of the payload..flights selector.

Select the namespace or attributes of the flights key

18. Change the body expression to select the namespace of the flights key.

```
payload.flights.#
```

19. Look in the Preview pane; you should see the default namespace http://acme.com is returned as a string.

20. Change the body expression to select the namespace of the flights key.

```
payload.flights.@
```

21. Look in the Preview pane; you should see both attributes are listed as an object, not as an array.

Select every descendent element of the payload and calculate the number of elements returned by the descendent selector

22. Change the body expression to select every descendent element of the payload.

```
payload..
```

23. Look in the Preview pane; you should see an array of objects is returned with individual flight elements, and their child elements.

```
[
  {
    "flight": {
      "available-seats": "+40.00",
      "airline-name": "american",
      "flight-code": "AA103",
```

24. Pass the current body expression into the sizeOf() function.

```
sizeOf(payload..)
```

25. Look in the Preview pane; you should see the result 43.

Verify the outer root key/value is not included in the result of the descendent selector

26. Delete sizeOf but keep the descendent selector expression surrounded by evaluation parentheses.

```
(payload..)
```

27. Look in the Preview pane; you should see a large array of all descendants of the payload.
28. Attempt to select the flights element's value from the array of all descendants.

```
(payload..).flights
```

29. Look in the Preview pane; you should see the result is null.

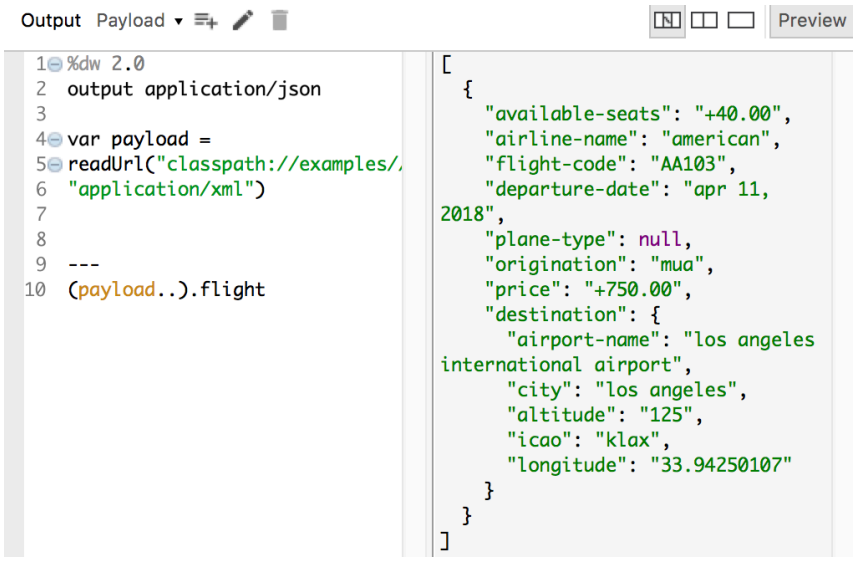
Note: the payload is an object with outer root key flights. The descendants selector .. selects all descendants of the flights key's value, which is an object that does not include the flights key.

Select the first flight object from the result of the descendent selector

30. Change the body expression to select the value of the first flight element.

```
(payload..).flight
```

31. Look in the Preview pane; you should see the array changes to an array of a single object, where the object is the key/value pairs of the first flight object's value, without the flight key.

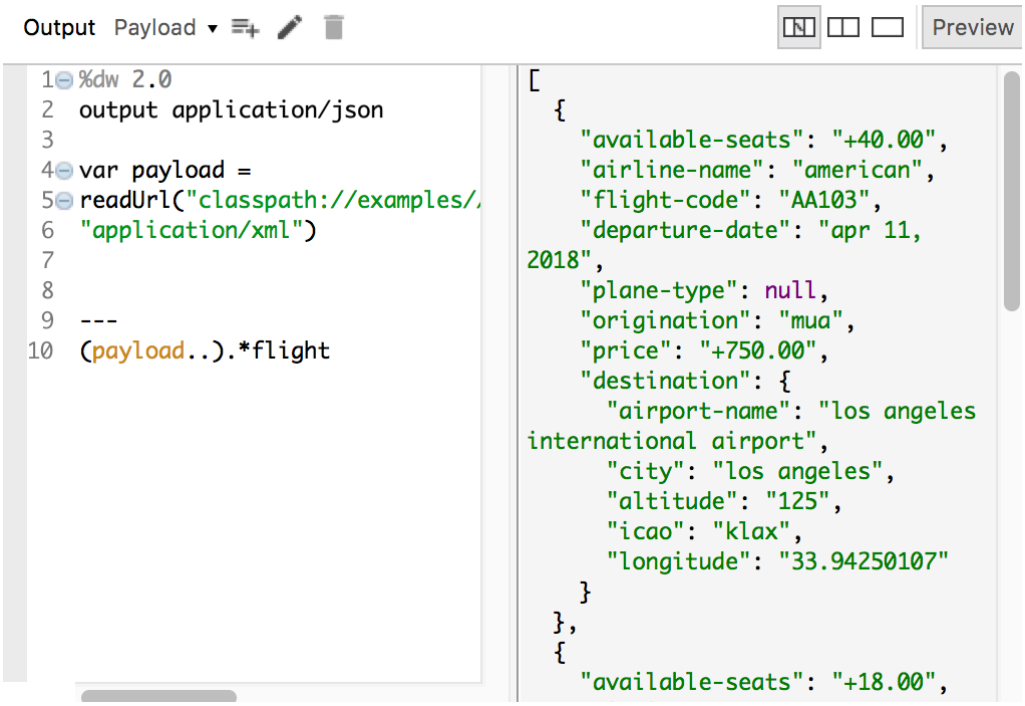


Select the attributes of each flight object from the result of the descendent selector

32. Change the body expression to select the value of every flight element.

```
(payload..).*flight
```

33. Look in the Preview pane; you should see the array changes to an array of a multiple objects, where each object is the key/value pairs of the matched flight object, without the flight key.



34. Change the body expression to still select all descendent flight values, but without using the intermediate evaluation parentheses.

```
payload..*flight
```

35. Verify the result is still an array of flight object key/values without the flight key.
36. Change the body expression to use the @ attribute selector to select the attributes from each result of the previous expression.

```
payload..*flight.@
```

37. Look in the Preview pane; you should see the result is an array of three objects, with each object containing a key/value pair for the corresponding airline attribute.

```
%dw 2.0
output application/json

var payload =
readUrl("classpath://examples//flightsJoinedWithAiportsWi
"application/xml")

---
payload..*flight.@
```

```
[
  {
    "airline": "american"
  },
  {
    "airline": "delta"
  },
  {
    "airline": "united"
  }
]
```

Select the airline attribute for the delta flight element

38. Change the body expression to apply the @ attribute selector to the second element of the payload..*flight result.

```
payload..*flight[1].@
```

Note: If the Preview pane does not show the result, use a do statement with a variable to break up the expression into the selector of the delta flight object, followed by the .@ attribute selector.

```
do{
  var deltaFlight = payload..*flight[1]
  ---
  deltaFlight.@
}
```

39. Look in the Preview pane; you should see the result is an object with the single "airline":"delta" key/value pair.

```
%dw 2.0
output application/json

var payload =
readUrl("classpath://examples//flightsJoinedWithAiportsWi
"application/xml")

---
payload..*flight[1].@
```

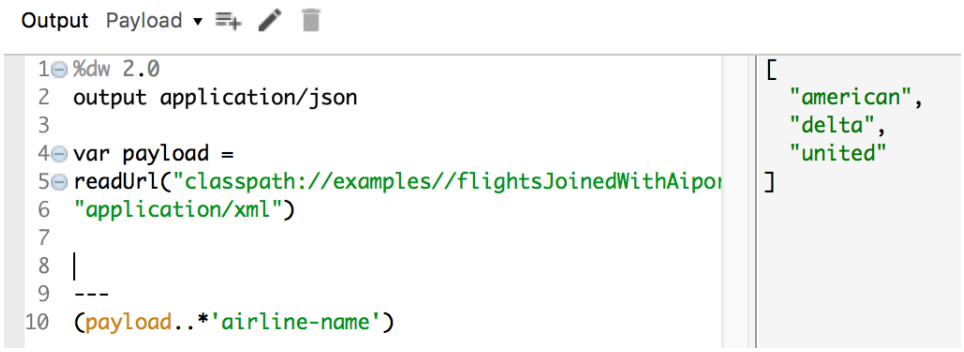
```
{
  "airline": "delta"
}
```

Select the value of each airline-name descendent key/value

40. Change the body expression to select the value of any descendent element with the key name airline-name; be sure to put the dasherized key name in single our double quotes.

```
payload..*"airline-name"
```

41. Look in the Preview pane; you should see the result is an array of the three airline names.



Select the prefix and uri of a descendant key's namespace

42. Change the body expression to select the namespace of the array of airline-name keys.

```
payload..*"airline-name".#
```

43. Look in the Preview pane; you should see the result is null.

    Note: The resulting array does not have a namespace. Only its child elements have namespaces.

44. Change the body expression to select the namespace of the delta flight's airline-name element.

```
payload..*"airline-name"[1].#
```

45. Look in the Preview pane; you should see the result is the uri "http://flights.com".
46. Change the output directive to application/dw.
47. Look in the Preview pane; you should see the result changes to an object with the complete namespace of the airline-name key, with both a prefix and uri.

48. Change the body expression to select the prefix of the delta flight object's airline-name element.

```
payload..*"airline-name"[1].#.prefix
```

49. Look in the Preview pane; you should see the result is the namespace prefix string "fl".
50. Change the output directive to application/json.

51. Look in the Preview pane; you should see the namespace prefix "fl".

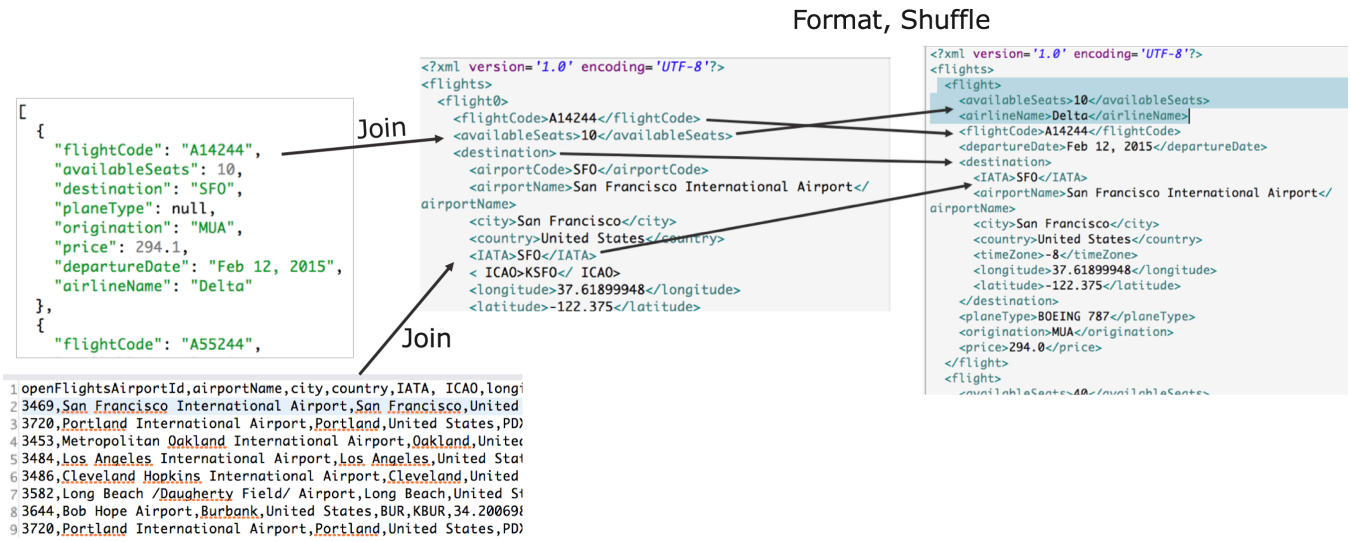
    Even though the default for application/json output is to just show the uri namespace value, the prefix value is still accessible with application/json output.

52. Change the body expression to select the prefix of the delta flight object's city element.

    ```
    (payload..*city)[1].#.prefix
    ```

53. Look in the Preview pane; you should see the namespace prefix "ap".

Module 6: Iteratively and Recursively Mapping Data



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

- Join together objects and arrays into nested data structures by using the map and mapObject operators.
- Extract an array of keys and/or values from an object by using the pluck operator.
- Format and transform complex nested data structures by using recursive functions.

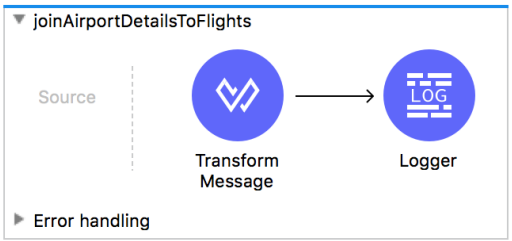
Walkthrough 6-1: Join reference data with message data by using map operators

In this walkthrough, you join input data with reference data. You can use this same technique to join results from a Scatter/Gatherer Flow Control component, or any other input data such as flowVars, inbound parameters, or other data. You will:

- Store reference data in a DataWeave variable.
- Join input schema with reference schema by using a map operator.

Read in airport details reference data from a CSV file

1. Return to the apdw2-flights project in Anypoint Studio.
2. In the dw-transforms.xml file, in the joinAirportDetailsToFlights flow, select the Transform Message component.



3. Verify the output directive is application/json.

```
%dw 2.0
output application/json
```

4. Add a header to set a variable named airportInfo from a JSON file named airportinfoTiny.json.

```
var airportDetails =
readUrl("classpath://airportInfoTiny.csv","application/csv")
```

Note: The airportInfoTiny.csv file is already stored in the starter Mule application's Classpath, in the src/main/resources folder. In a real project you might read in this data from an external system using a connector.

5. Change the body expression to the airportDetails variable.

```
---
airportDetails
```

6. Look in the Preview pane; you should see the airport details data displays in JSON format.

Group the airportInfo objects by the IATA code

7. Change the airportDetails expression to group the airportInfo JSON by the IATA code.

```
var airportDetails =
readUrl("classpath://airportInfoTiny.csv","application/csv")
groupBy $.IATA
```

Note: The IATA code is the three-letter code used in the payload's destination key (such as SFO, LAX, PDX, and CLE). You will use this value to join the payload's destination with the details from this array.

8. Look in the Preview pane; you should see the airlineInfo variable is transformed to an object with key values set to the airport IATA codes.


```
{
  "LTN": [
    {
      "openFlightsAirportId": "492",
      "airportName": "London Luton Airport",
      "city": "London",
      "country": "United Kingdom",
      "IATA": "LTN",
      "ICAO": "EGGW",
      "longitude": "51.87469864",
      "latitude": "-0.368333012",
      "altitude": "526",
      "timeZone": "0",
      "DST": "E",
      "timeZone": "Europe/London",
      "type": "airport",
      "source": "OurAirports"
    }
  ],
  "JER": [
    {
      "openFlightsAirportId": "499",
      "airportName": "Jersey Airport",
      "city": "Jersey",
```


Note: If you have any trouble reading in the file, there are copies of the airportInfo CSV and JSON files in the %studentFiles%\mod02 folder.

Access the airport details for SFO

9. Change the body expression to the array value of the airportDetails object's "SFO" key.

```
airportDetails["SFO"]
```

10. Look in the Preview pane; you should see you see the airport details for SFO as an array containing one object.


```
[
  {
    "openFlightsAirportId": "3469",
    "airportName": "San Francisco International Airport",
    "city": "San Francisco",
    "country": "United States",
    "IATA": "SFO",
    " ICAO": "KSFO",
    "longitude": "37.61899948",
    "latitude": "-122.375",
    "altitude": "13",
    "timeZone": "-8",
    "DST": "A",
    "timeZone": "America/Los_Angeles",
    "type": "airport",
    "source": "OurAirports"
  }
]
```

11. Change the body expression to return the first (and only) object element of the current array of objects.

```
airportDetails["SFO"][0]
```


```
{
  "openFlightsAirportId": "3469",
  "airportName": "San Francisco International Airport",
  "city": "San Francisco",
  "country": "United States",
  "IATA": "SFO",
  "ICAO": "KSFO",
  "longitude": "37.61899948",
  "latitude": "-122.375",
  "altitude": "13",
  "timeZone": "-8",
  "DST": "A",
  "timeZone": "America/Los_Angeles",
  "type": "airport",
  "source": "OurAirports"
}
```


Note: You can also use this expression to achieve the same result:

```
{( airportDetails["SFO"] )}
```

Concatenate airportDetails corresponding to the current flight destination value

12. Change the body expression to map each flight object of the payload's flights key.

```
---
payload.flights map (flight) -> flight
```

Note; This expression gives you access to each flight object element of the payload Array through the flight argument.

13. Look in the Preview pane; you should see the payload, unchanged.

Note: If you see errors indicating the map operator cannot be overloaded, coerce the payload to an Array.

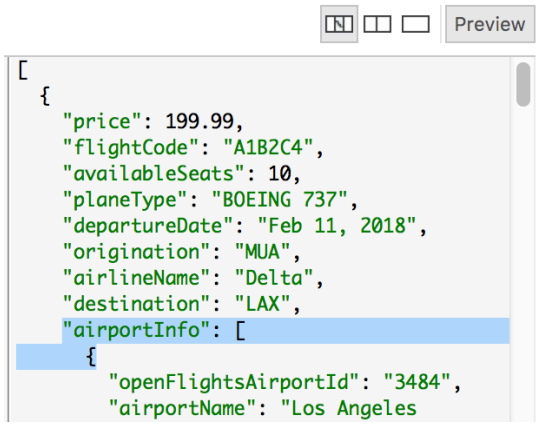
```
---
(payload.flights as Array) map (flight) -> flight
```

14. Concatenate a key/value pair to add in the airportInfo corresponding to the flight.destination element.

```
---
payload.flights as Array map ( (flight) ->
   flight ++
   {
      airportInfo:
      airportDetails[flight.destination]
   }
)
```

Note: Be sure to surround the map operator's lambda expression in evaluation parentheses, otherwise the flight argument will not be found.

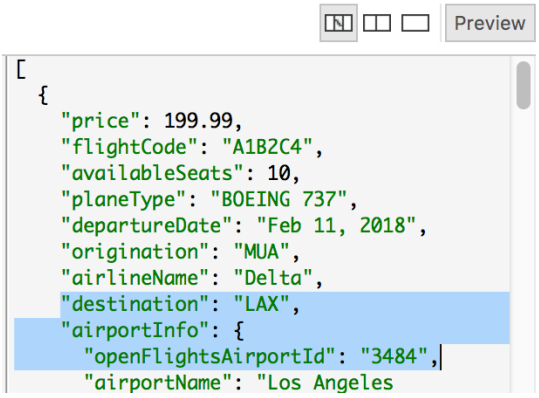
15. Look in the Preview pane; you should see the airportInfo key/value is added with value an array of objects.



16. Verify the joined airportInfo key has a value that is an array of a single object.
17. Change the body expression to extract the first object element of the airportInfo[flight.destination] array.

```
---
payload.flights map ( (flight) ->
   flight ++
   {
     airportInfo:
     airportDetails[flight.destination][0]
   }
)
```

18. Look in the Preview pane; you should see the airport details now appear as an object instead of as an array.



Move the body expression to a function named flightsJoinedWithAirportInfo

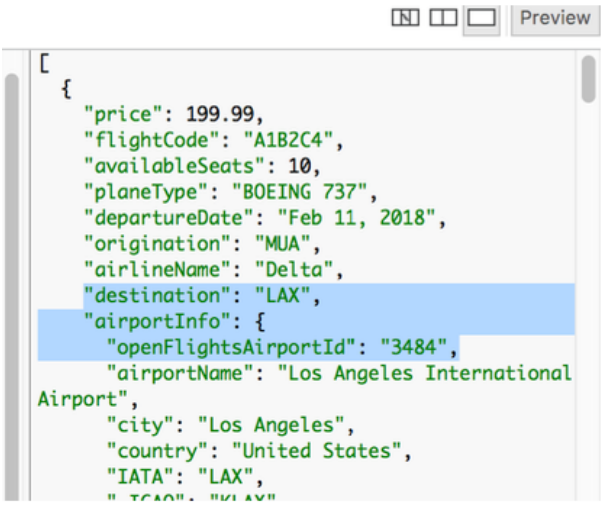
19. Replace the current body divider `---` with a function directive for a function named flightsJoinedWithAirportInfo that takes the payload as an input argument.

```
fun flightsJoinedWithAirportInfo(payload) =
```

20. At the very bottom of the DataWeave expression, below the new function implementation, add a body divider `---` and change the body expression to call the new flightsJoinedWithAirportInfo function with the current payload.

```
flightsJoinedWithAirportInfo(payload)
```

21. Look in the Preview pane; you should see the same joined JSON array.



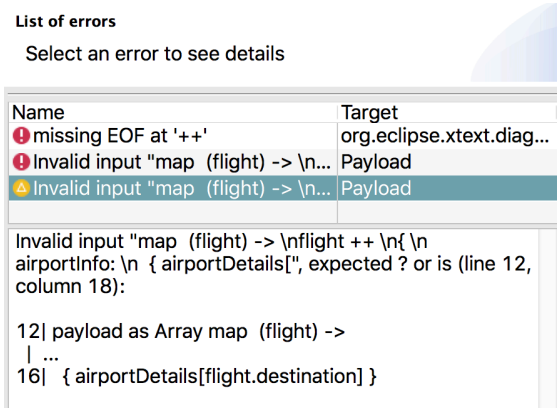
Join the airport details as an object, not as an array

22. In the flightsJoinedWithAirportInfo implementation, surround the airport details array in object constructor curly braces.

```
payload.flights map ( (flight) ->
   flight ++
   {
      airportInfo:
      { airportDetails[flight.destination] }
   }
)
```

Note: You might want to back up your current function as flightsJoinedWithAirportInfoBak.

23. View the error that is generated and verify it relates to invalid input at the airportDetails expression.



24. To fix the error, put evaluation parentheses inside the object constructor curly braces.

```
payload.flights map ( (flight) ->
   flight ++
   {
      airportInfo:
      { ( airportDetails[flight.destination] ) }
   }
)
```

25. Verify the error is fixed.

Note: You could also just write:

```
airportDetails[flight.destination][0]
```

to extract the single object from the array, but using an object constructor will also work for to extract all the objects from an array with several object elements.

26. Look in the Preview pane; you should see the same output appears, with the airport details joined as an object, not an array of an object.

27. Write another DataWeave expression to extracts the airport details object from its parent array by accessing the first element of the array.

```
fun flightsJoinedWithAirportInfo(payload) =
payload.flights map ( (flight) ->
   flight ++
   {
      airportInfo:
       airportDetails[flight.destination][0]
   }
)
```

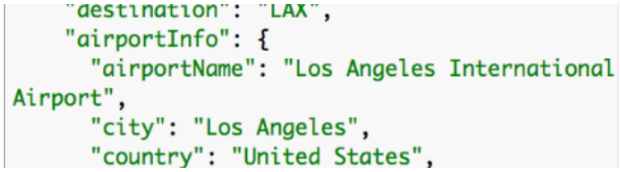
28. Look in the Preview pane; the airport details should still appear as a child object of the airportInfo key.

Delete unneeded keys from the airport details

29. Modify the flightsJoinedWithAirportInfo variable to add a new key named airportCode with the flight.destination value, and delete the keys "openFlightsAirportId", "type", and "source".

```
fun flightsJoinedWithAirportInfo(payload) =
payload.flights map ( (flight) ->
   flight ++
   {
      airportInfo:
       airportDetails[flight.destination][0]
      - "openFlightsAirportId" - "type" - "source"
   }
)
---
flightsJoinedWithAirportInfo( payload )
```

30. Look in the Preview pane; you should see the three key/value pairs are removed from each flight object.



Delete unneeded keys all at once using an array of key names

31. Replace the separate - operations with one -- operation.

```
fun flightsJoinedWithAirportInfo(payload) =
payload.flights map ( (flight) ->
   flight ++
   {
      airportInfo:
      airportDetails[flight.destination][0]
      --["openFlightsAirportId", "type", "source"]
   }
)
---
flightsJoinedWithAirportInfo( payload )
```

32. Look in the Preview pane; you should see the three key/values are still removed.

Use the pluck operator to get an array of keys, values, or indices from an input object

33. In the header, declare a function named listKeys that accepts an object argument.

```
fun listKeys(anObject : Object) = anObject
```

34. Call the listKeys function in the body expression

```
---
//flightsJoinedWithAirportInfo( payload )
listKeys( airportDetails["SFO"][0] )
```

35. In the preview pane you should see the SFO airport details key/values.

```
{
  "openFlightsAirportId": "3469",
  "airportName": "San Francisco International Airport",
  "city": "San Francisco",
  "country": "United States",
  "IATA": "SFO",
  "ICAO": "KSFO",
  "longitude": "37.61899948",
  "latitude": "-122.375",
  "altitude": "13",
  "timeZone": "-8",
  "DST": "A",
  "timeZone": "America/Los_Angeles",
  "type": "airport",
  "source": "OurAirports"
}
```

36. Modify the listKeys function implementation to use pluck to extract an array of values of the input object; in the Preview pane you should see the result is an array.

```
fun listKeys(anObject : Object) =
anObject pluck $
```

```
[
  "openFlightsAirportId",
  "airportName",
  "city",
  "country",
  "IATA",
  "ICAO",
  "longitude",
  "latitude",
  "altitude",
  "timeZone",
  "DST",
  "timeZone",
  "type",
  "source"
]
```

37. Modify the listKeys function implementation to use pluck to create an object with the key set to the index in the input object and value is the key name.

```
fun listKeys(anObject : Object) =
anObject pluck ($$$): $
```

```
[
  {
    "0": "openFlightsAirportId"
  },
  {
    "1": "airportName"
  },
  {
    "2": "city"
  },
  {
    "3": "country"
  },
  {
    "4": "IATA"
```

38. Modify the listKeys function implementation to use pluck to create an array of just the key names.

```
fun listKeys(anObject : Object) =
anObject pluck $$
```

```
[
  "openFlightsAirportId",
  "airportName",
  "city",
  "country",
  "IATA",
  "ICAO",
  "longitude",
  "latitude",
  "altitude",
  "timeZone",
  "DST",
  "timeZone",
  "type",
  "source"
]
```

39. Copy the result in the Preview pane to a new variable named keysToRemove.
40. Delete strings from keysToRemove array so you only remove the keys named openFlightsAirportId, type, and source.

```
vars keysToRemove = ["openFlightsAirportId", "type", "source"]
```

Modify the flightsJoinedWithAirportInfo function to delete all key names in the keysToRemove array

41. Change the comments in the body expression to again call the flightsJoinedWithAirportInfo function.

```
---
//listKeys( airportDetails[0] )
flightsJoinedWithAirportInfo( payload )
```

42. Modify the implementation of flightsJoinedWithAirportInfo to remove all the airportDetails elements with key names contained in the keysToRemove array; the airportInfo value should now be just an empty object.

```
payload.flights map ( (flight) ->
   flight ++
   {
      airportInfo:
      airportDetails[flight.destination][0]
      --  keysToRemove
   }
)
---
flightsJoinedWithAirportInfo( payload )
```

43. In the Preview pane, verify you again see the truncated list of airport details key/values for the airportInfo value.


```
[
  {
    "airlineName": "Delta",
    "availableSeats": 40,
    "departureDate": "Apr 11, 2018",
    "destination": "LAX",
    "flightCode": "A134DS",
    "origination": "MUA",
    "planeType": "BOEING 777",
    "price": "750.00",
    "airportInfo": {
      "airportName": "Los Angeles International Airport",
      "city": "Los Angeles",
      "country": "United States",
      "IATA": "LAX",
      "ICAO": "KLAX",
      "longitude": "33.94250107",
      "latitude": "-118.4079971",
      "altitude": "125",
      "timeZone": "-8",
      "DST": "A",
      "timeZone": "America/Los_Angeles"
    }
  },
  {
    "airlineName": "Delta",
    "availableSeats": 18,
```

Walkthrough 6-2: Build an object from another object by using the mapObject operator

In this walkthrough, you refactor the previous flightsJoinedWithAirportInfo transformation to do the same data transformation using the mapObject operator instead of the map operator and object constructor curly braces. You will carry out this walkthrough is steps to help you to continue to "think in DataWeave" to decompose complex transformation problems into smaller and more reusable functions. You will:

- Transform each outer key/value of an object into a new object by using a mapObject operator.
- Join input schema with reference schema by using a mapObject operator.
- Selectively remove keys from an object by matching key names in a lookup list array.

Create another version of the flightsJoinedWithAirportInfo function

1. Return to the dw-transforms tab.
2. In the joinAirportDetailsToFlights flow, select the Transform Message component.

   Note: If you had trouble with the last walkthrough, you can also load the starter project into a new project with a different name, and start from there.

3. Copy and paste the first two lines of the flightsJoinedWithAirportInfo function directive to a new function directive named flightsJoinedWithAirportInfoV2, then add a third line to just return the current flight object.

   ```
   fun flightsJoinedWithAirportInfoV2(payload) =
   payload.flights map (flight) ->
   flight
   ```

4. Comment out the current body expression's function call, and underneath call the new V2 function.

   ```
   ---
   // flightsJoinedWithAirportInfo (payload)
   flightsJoinedWithAirportInfoV2(payload)
   ```

5. Look in the Preview pane; you should see the original input payload without any joined airport details.

For each flight object in the map operator, use a nested mapObject operator to iterate over each element of the current flight object

6. In the flightsJoinedWithAirportInfoV2 function implementation, add a mapObject operator to iterate over each element of the current flight object.

```
fun flightsJoinedWithAirportInfoV2(payload) =
payload.flights map (flight) ->
flight mapObject ( value,key ) ->
  { (key) : value }
```

7. Look in the Preview pane; you should see the same original payload.
8. Convert each key to uppercase.

```
fun flightsJoinedWithAirportInfoV2(payload) =
payload.flights map (flight, index) ->
  flight mapObject (value,key) ->
    { ( upper( key ) ) : value }
```

9. Look in the Preview pane; you should see all the keys appear in uppercase.

Note: This shows you that the mapObject operator iterates over each element inside each flight object.

Label each flight object with a unique key

10. Use the index parameter to add a unique key to each flight object.

```
fun flightsJoinedWithAirportInfoV2(payload) =
payload.flights as Array map (flight, index) ->
"flight$(index)": flight mapObject (value,key) ->
    { (upper( key ) ) : value }
```

11. Look in the Preview pane; you should see the parent flight# key is added to each flight object.

```
[
  {
    "flight0": {
      "PRICE": 200.0,
      "FLIGHTCODE": "rree1000",
      "AVAILABLESEATS": 5,
      "PLANETYPE": "Boeing 737",
      "DEPARTUREDATE": "2016-01-20T00:00:00",
      "ORIGINATION": "MUA",
      "AIRLINENAME": "American Airlines",
      "DESTINATION": "CLE"
    }
  },
  {
    "flight1": {
      "PRICE": 245.0,
      "FLIGHTCODE": "ER3kfd",
      "AVAILABLESEATS": 13,
```

Use a match statement to apply a different transformation to the destination key

12. Add a match statement to test each flight object key then pass through the original key/value pair.

```
fun flightsJoinedWithAirportInfoV2 (payload) =
payload.flights as Array map  (flight, index) ->
"flight$(index)": flight mapObject (value,key) ->
    key match {
        else -> { (key): value }
    }
```

13. Verify there are no errors and that the payload still appears in the Preview pane, but the keys are no longer in uppercase.


```
[
  {
    "flight0": {
      "price": 200.0,
      "flightCode": "rree1000",
      "availableSeats": 5,
      "planeType": "Boeing 737",
      "departureDate": "2016-01-20T00:00:00",
      "origination": "MUA",
      "airlineName": "American Airlines",
      "destination": "CLE"
    }
  },
  {
    "flight1": {
      "price": 245.0,
      "flightCode": "ER3kfd",
      "availableSeats": 13,
```


Move the transformation logic for the "destination" case to another function

14. Add a case to test if the key is "destination", and if so, to call a new function named joinAirportDetails to join airport details with the current flight data.

```
fun flightsJoinedWithAirportInfoV2(payload) =
payload.flights as Array map (flight, index) ->
"flight$(index)": flight mapObject (value,key) ->
    key match {
       case "destination" -> joinAirportDetails(flight)
      else -> { (key): value }
    }
```

Note: You can pass the parent map operator's parameter flight to the inner mapObject function.

15. Hover over the joinAirportDetails(flight) error, then click on Create Function joinAirportDetails.

16. Verify the new joinAirportDetails function is declared above the flightsJoinedWithAirportInfoV2 function.

```
fun joinAirportDetails(param0)= ???
```

17. Change param0 to flight remove the inferred strong typing, then add the implementation expression to return a key/value pair with the key named "destination" and the value the flight.destination.

```
fun joinAirportDetails(flight)=
destination: flight.destination
```

18. Look in the Preview pane; no errors should occur and the payload should appear with the destination value the code "CLE" from the payload.

Change the destination key's value to include the corresponding airport details

19. In the joinAirportDetails implementation, add the flight.destination to a key named airportCode, then concatenate this key/value with the airportDeatils for the corresponding flight.destination value.

```
fun joinAirportDetails(flight) =
destination:
   (
      {airportCode: flight.destination }
      ++ airportDetails[flight.destination][0]
   )
```

20. Look in the Preview pane; you should see there is no duplicate destination key, and the new airportCode key is added to the child destination details object.


```
[
  {
    flight0: {
      price: 200.0,
      flightCode: "rree1000",
      availableSeats: 5,
      planeType: "Boeing 737",
      departureDate: "2016-01-20T00:00:00",
      origination: "MUA",
      airlineName: "American Airlines",
      destination: {
        airportCode: "CLE",
        openFlightsAirportId: "3486",
        airportName: "Cleveland Hopkins International Airport",
        city: "Cleveland",
        country: "United States",
        IATA: "CLE",
        " ICAO": "KCLE",
        longitude: "41.4117012",
```


Filter out unneeded flight key/values using a lookup list array of key names

21. In the joinAirportDetails function, wrap the airportDetails line in a call to a new function named filterDetailsKeys that also takes keysToRemove as a second argument.

```
fun joinAirportDetails(flight) =
destination: (
   {airportCode: flight.destination}
   ++
   (
      filterDetailsKeys(
         airportDetails[flight.destination][0], keysToRemove
      )
   )
)
```

22. Implement the filterDetailsKeys function.

```
fun filterDetailsKeys(detailsObject, keysArray) =
detailsObject
```

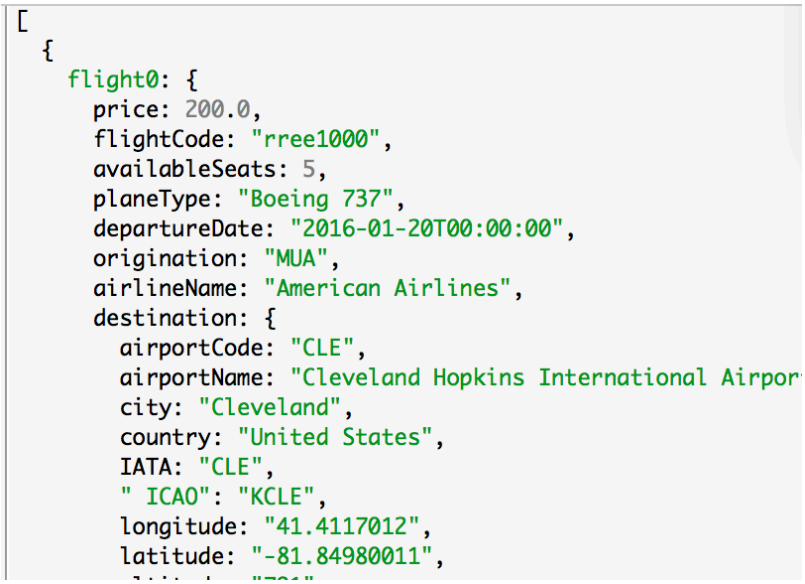
23. Verify no errors occur.
24. Add a mapObject operator to iterate over the destination details elements.

```
fun filterDetailsKeys( detailsObject, keysArray ) =
detailsObject mapObject ( (destValue, destKey) ->
  { (destKey) : destValue }
)
```

25. Add a test to skip destKey if it is in the keysArray lookup list.

```
fun filterDetailsKeys( detailsObject, keysArray ) =
detailsObject mapObject ( (destValue, destKey) ->
    if ( keysArray contains (destKey as String) ) {}
    else { (destKey): destValue }
)
```

26. Look in the Preview pane; you should see the keys listed in the lookup list array are removed.



Note: You might want to experiment with different if/else conditions and see the results in the destination value. You could also have skipped this function call and just used the minus minus -- operator directly with the keysToRemove array.

Add and apply a function to convert any array of objects into an XML object

27. Import the dw::ComplexData module.
28. Wrap the current body expression in the ComplexData::buildXmlObject() function, and add the second root element argument "flight".

```
ComplexData::buildXmlObject(
    flightsJoinedWithAirportInfoV2( payload ), "flight"
)
```

29. Look in the Preview pane; you should see the single root "flights" is added to the object of flight elements.

Remove the extra flight# key and output as application/xml

30. In the flightsJoinedWithAirportInfoV2 function, comment out or remove the part of the line with the code "flight$(index)".

```
fun flightsJoinedWithAirportInfoV2(payload) =
payload.flights as Array map (flight, index) ->
//"flight$(index)":
flight mapObject (value,key) ->
    key match {
       case "destination" -> joinAirportDetails(flight)
      else -> { (key): value }
    }
```

31. Look in the Preview pane; you should see the extra flight# key is removed.

32. Change the output directive to application/xml.

33. Look in the Preview pane; you should see valid XML.

```
<?xml version='1.0' encoding='UTF-8'?>
<flights>
  <flight>
    <airlineName>Delta</airlineName>
    <availableSeats>40</availableSeats>
    <departureDate>Apr 11, 2018</departureDate>
    <destination>
      <airportCode>LAX</airportCode>
      <airportName>Los Angeles International Airport</airportName>
      <city>Los Angeles</city>
      <country>United States</country>
      <IATA>LAX</IATA>
      <ICAO>KLAX</ICAO>
      <longitude>33.94250107</longitude>
      <latitude>-118.4079971</latitude>
      <altitude>125</altitude>
      <timeZone>-8</timeZone>
      <DST>A</DST>
      <timeZone>America/Los_Angeles</timeZone>
    </destination>
    <flightCode>A134DS</flightCode>
    <origination>MUA</origination>
    <planeType>BOEING 777</planeType>
```

Save the enriched flight XML output to a file in the Mule project

34. In the preview pane, copy all the XML text to the clipboard.
35. In the Package Explorer right-click on src/main/resources and select New > File.
36. Name the new file flightsWithDetails.xml.
37. In the XML file editor, select the Source tab.
38. Paste the XML from the clipboard to the file, then save the file.

Note: A copy of this file is provided in the examples folder from the starter project.

(Optional) Test the application and verify flights are joined with airport details

39. Run or debug the Mule application.
40. Submit a request to http://localhost:8081/flights?code=LAX&airline=all.
41. Verify you get a response with flights to LAX, with the joined LAX airport details.
42. Stop the Mule application.

Build a valid JSON object with unique key names from an object array

43. Write a similar function named buildJsonObject to add unique parent key names to each flight value of the objectArray object.

```
fun buildJsonObject( objectArray, parentTag) =
{(
   objectArray map (element, index) -> {
      ( parentTag ++ index ): element
   }
)}
```

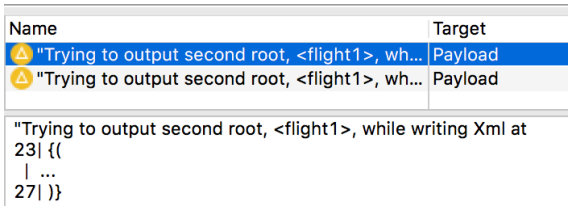
44. Strongly type the two input arguments.

```
fun buildJsonObject( objectArray: Array<Object>, parentTag: String ) =
```

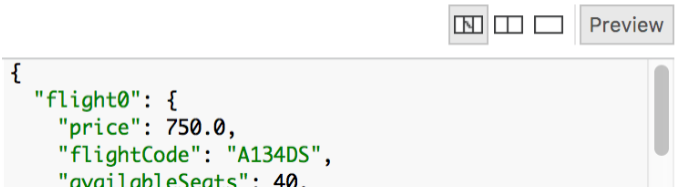
45. Return to the dw-transforms tab.
46. Change the body expression to call buildJsonObject instead of buildXmlObject.

```
buildJsonObject( flightsJoinedWithAirportInfoV2( payload ), "flight" )
```

47. Look in the Preview pane; you should see an error is reported that the result cannot be formatted as XML.



48. In the header, change the output directive from application/xml to application/json.
49. Look in the Preview pane; you should see the error is fixed and JSON formatted data appears.
50. Verify each outer key has a unique name like flight0 and flight1.



Save the enriched flight JSON output to a file in the Mule project

51. In the preview pane, copy all the XML text to the clipboard.
52. In the Package Explorer right-click on src/main/resources and select New > File.
53. Name the new file flightsWithDetails.xml.
54. In the XML file editor, select the Source tab.
55. Paste the XML from the clipboard to the file, then save the file.

Add the XML and JSON formatting functions to a DataWeave module

56. Copy the buildJsonObject function to src/main/resources/dw/modules/ComplexData.dwl

{/} *ComplexData.dwl

```
%dw 2.0

var exchangeRate = 1.35

fun combineScatterGatherResults( theInput ) =
flatten ( theInput..payload ) orderBy $.price

fun buildXmlObject( objectArray, parentTag ) =
{
  (dw::core::Strings::pluralize(parentTag))  : {(
    objectArray map (element) -> {
      (parentTag): element
    }
  )}
}


fun buildJsonObject( objectArray: Array<Object>, parentTag: String ) =
{(
   objectArray map (element, index) -> {
      ( parentTag ++ index ): element
   }
)}
```

Walkthrough 6-3: Join complex schema by using the join function

In this walkthrough, you use the join function to join flight and destination airport details with passenger flight reservation records. You will:

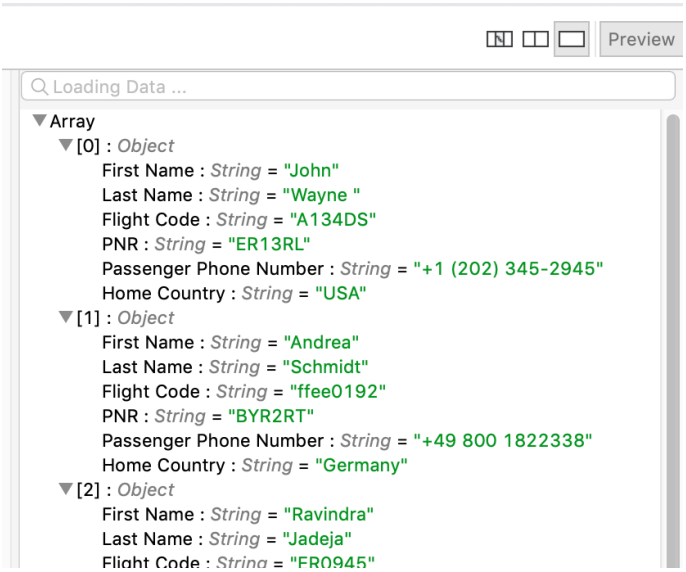
- Use the join function to join together two complex objects read from different input MIME types.
- Add cases to a join function to selectively update certain elements in a complex data structure.
- Add conditions to cases in a join function to conditionally modify branches of a complex data structure.

Import mock data with which to work

1. Return to the dw-transforms tab.
2. Drag out a Transform Message component and drop it at the bottom to create a new flow and name the flow enrichPassengerRecords.
3. In the Output pane, add a var directive to import the passengerRecords.csv file to a var named passengerRecords.

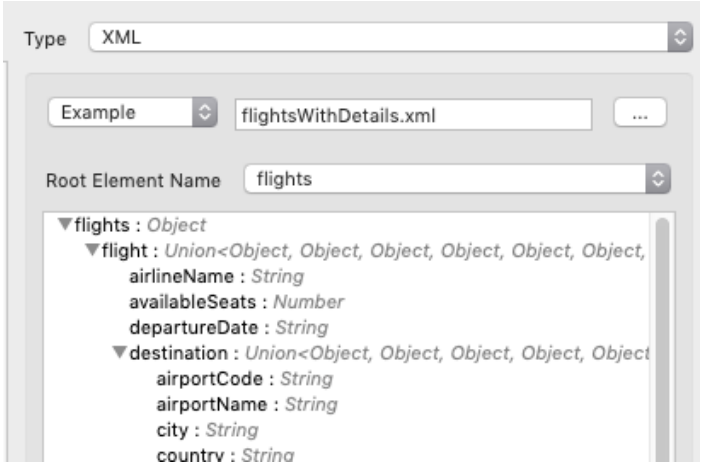
```
%dw 2.0
output application/java
var passengerRecords = readUrl("classpath://passengerRecords.csv", "application/csv")
---
```

4. Set the body expression to passengerRecords; you should see the records converted from CSV to Java.



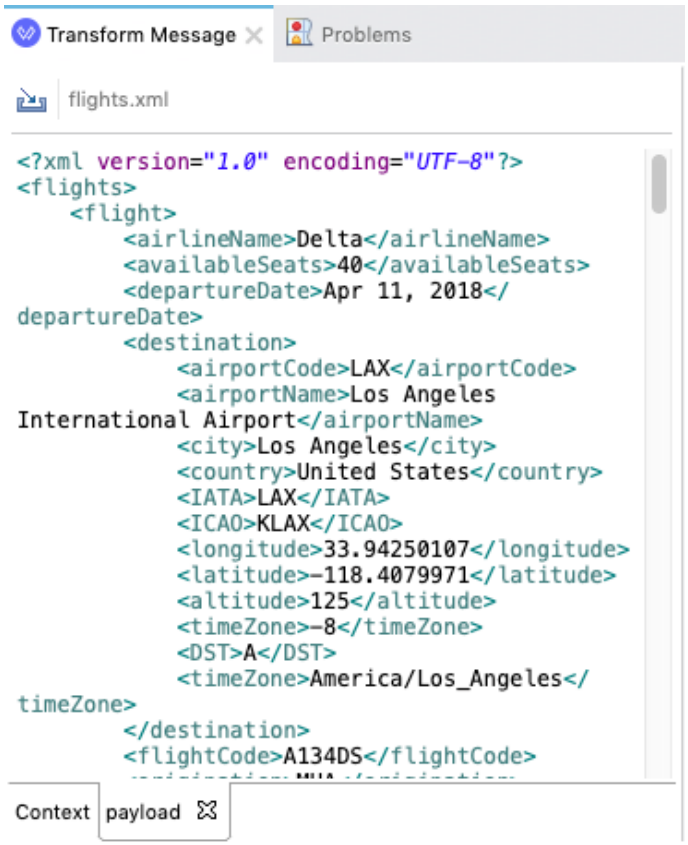
Set example input matching the payload of the joinAirportDetailsToFlights result

5. In the input pane, next to Payload click the Define metadata link.
6. In the Select metadata type dialog, click Add.
7. In the Create new type dialog set the Type id to flightsWithDetails.
8. Click Create type.
9. Set the Type to XML.
10. Change the Schema drop-down list to Example.
11. Browse in the Mule project for src/main/resources/flightsWithDetails.xml.



12. Click Select.

13. Right-click Payload and select Edit Sample Data; the XML file should appear in a new payload tab.



14. Change the body expression to payload; in the Preview pane you should see the flights object display as a Java object.
15. Set a variable named flightsWithDetails and assign it to payload.

```
var flightsWithDetails = payload
```

Note: If the payload does not display properly in the Preview pane, assign the flightsWithDetails var the value readUrl("classpath://flightsWithDetails.xml", "application/xml").

Locate the keys in each data structure on which to perform the join

16. Change the output directive to application/dw.

17. Change the body expression to list the array of flight objects inside the flightsWithDetails variable; you should see the flightCode key in the outer level of each object and the first value is "A134DS".

```
flightsWithDetails..*flights
```


```
[
  {
    flight: {
      price: "750",
      flightCode: "A134DS",
      availableSeats: "50",
      planeType: "BOEING 777",
      departureDate: "Apr 11, 2018",
      origination: "MUA",
      airlineName: "Delta",
      destination: {
        airportCode: "LAX",
        openFlightsAirportId: "3484",
```


18. Comment out this line and replace it with passengerRecords to view the array of passenger records read in from the passengers.csv file; in the Preview pane you should see the "Flight Code" key in the outer level of each passenger object and the first element should have the same value " A134DS" as the flightCode key.

```
//flightsWithDetails..*flights
passengerRecords
```


```
[
  {
    "First Name": "John",
    "Last Name": "Wayne ",
    "Flight Code": "A134DS",
    PNR: "ER13RL",
    "Passenger Phone Number": "+1 (202) 345-2945",
    "Home Country": "USA"
  },
  {
    "First Name": "Andrea",
    "Last Name": "Schmidt",
```


Join the passenger records with corresponding flight details records

19. Call the dw::core::Arrays::join function with the arguments flightsWithDetails..*flights and passengerRecords, and the simple lambda expressions the evaluate to the same string "SAME".

```
dw::core::Arrays::join(
    passengerRecords,
    flightsWithDetails.*flights,
    (pass) -> "SAME",
    (flight) -> "SAME"
)
```

20. If you see an error indicating join can't be called with some of the argument types, coerce the first and second arguments as Array<Object> types; this should remove the error

```
dw::core::Arrays::join(
       passengerRecords as Array<Object>,
       flightsWithDetails..*flight as Array<Object>,
       (pass) -> "SAME",
       (flight) -> "SAME"
)
```

Note: These errors should only appear from when editing DataWeave in Studio. They should not prevent the DataWeave code from executing properly at runtime.

21. Verify this creates a cross product matching each passenger left element with each element of the entire flightsWithDetails.*flights array; there should be four outer objects, and each "r" element should have all the flights combined into one object.


```
[
  {
    l: {
      "First Name": "John",
      "Last Name": "Wayne ",
      "Flight Code": "A134DS",
      PNR: "ER13RL",
      "Passenger Phone Number": "+1 (202) 345-2945",
      "Home Country": "USA"
    },
    r: {
      flight: {
        price: "750",
        flightCode: "A134DS",
        availableSeats: "50",
        planeType: "BOEING 777",
        departureDate: "Apr 11, 2018",
        origination: "MUA",
```


22. Modify the passenger key lambda expression to return the "Flight Code" string value; the Preview pane should change to an empty array because none of the passenger flight codes.

```
dw::core::Arrays::join(
    passengerRecords,
    flightsWithDetails.*flights,
    (pass) -> pass."Flight Code",
    (flight) -> "SAME"
)
```

23. Change the passenger lambda to a conditional expression to hardcode the string "A134DS" if the pass."Flight Code" equals "A134DS", and otherwise to hardcode the string "NONE"; the Preview pane should show the first "Flight Code" is not "A134DS".

```
(pass) -> if( pass."Flight Code" == "A134DS") pass."Flight Code" else ("SAME")
```


```
[
  {
    l: {
      "First Name": "Andrea",
      "Last Name": "Schmidt",
      "Flight Code": "ffee0192",
      PNR: "BYR2RT",
      "Passenger Phone Number": "+49 800 1822338",
      "Home Country": "Germany"
    },
    r: {
      flight: {
        price: "750",
        flightCode: "A134DS",
```


24. Scroll down and verify none of the joined left passenger objects have the "Flight Code" value "A134DS".
25. Change the flight lambda from hardcoding the string "SAME" to hardcode the first flight record "A134DS"; the Preview pane should change to only left passenger records with "Flight Code" equal to "A134DS".


```
[
  {
    l: {
      "First Name": "John",
      "Last Name": "Wayne ",
      "Flight Code": "A134DS",
      PNR: "ER13RL",
      "Passenger Phone Number": "+1 (202) 345-2945",
      "Home Country": "USA"
    },
    r: {
      flight: {
        price: "750",
        flightCode: "A134DS",
        availableSeats: "50",
```


26. Scroll down and verify all of the joined left passenger objects have the "Flight Code" value "A134DS".
27. Change the flights lambda expression to the current flight object's flightCode string value.

```
(flight) -> flight..flightCode[0]
```

Note: This changes the hardcoded value "A134DS" to the flightCode value of each flight object.

28. Verify you get an error that some of the lambda expression results are null.
29. Modify the flight lambda expression to account for possible null values.

```
(flight) -> flight..flightCode[0] default "NULL"
```

30. Use a selector to verify only two passenger records are joined, and each has the "Flight Code" equal to "A134DS".

```
dw::core::Arrays::join(
    passengerRecords,
    flightsWithDetails.*flights,
    (pass) -> if( pass."Flight Code" == "A134DS") pass."Flight Code" else ("SAME"),
    (flight) -> flight..flightCode[0] default "NULL"
)
.1
```

31. Change the selector to .r and verify all flight records are matching

    *Note: This is because the flightsWithDetails.*flights selector returns a one element array of one object containing all the flights. Each right "r" key's value is the input argument of the entire set of all flights.*

Modify the join function's arguments to properly match the correct corresponding individual flight objects

32. Modify the second right join argument of the join function to join individual flight records rather than the entire flights key's value; now each left passenger object should match with a single right flight object, and the left and right objects should share the same flight code value.

```
dw::core::Arrays::join(
    passengerRecords,
    flightsWithDetails..*flight ,
    (pass) ->  pass."Flight Code",
    (flight) -> flight.flightCode
)
```


```
[
  {
    l: {
      "First Name": "John",
      "Last Name": "Wayne ",
      "Flight Code": "A134DS",
      PNR: "ER13RL",
      "Passenger Phone Number": "+1 (202) 345-2945",
      "Home Country": "USA"
    },
    r: {
      price: "750",
      flightCode: "A134DS",
      availableSeats: "50",
```


33. Scroll down and verify every left and right pair have matching flight code values; this verifies the join is now coded correctly.

Explore differences between the join, leftJoin, and outerJoin functions

34. Change the join function to leftJoin.
35. In the Preview pane, scroll to the bottom of the results and verify in addition to the same join function results; you should also see an additional left passenger record that does not match any of the flight ids.


```
        longitude: "33.94250107",
        latitude: "-118.4079971",
        altitude: "125",
        timeZone: "-8",
        DST: "A",
        timeZone: "America/Los_Angeles"
      }
    }
  },
  {
    l: {
      "First Name": "Kawabata",
      "Last Name": "Makoto",
      "Flight Code": "NOTFOUND",
      PNR: "MMLTTY",
      "Passenger Phone Number": "+81 52-249-7757",
      "Home Country": "Japan"
    }
  }
]
```


36. Change the leftJoin function to outerJoin.
37. In the Preview pane, scroll to the bottom of the results and verify you still see all the leftJoin results, followed by any unmatched right flight records.


```
        DST: "A",
        timeZone: "America/Los_Angeles"
      }
    }
  },
  {
    l: {
      "First Name": "Kawabata",
      "Last Name": "Makoto",
      "Flight Code": "NOTFOUND",
      PNR: "MMLTTY",
      "Passenger Phone Number": "+81 52-249-7757",
      "Home Country": "Japan"
    }
  },
  {
    r: {
      price: "496",
      flightCode: "A1QWER",
      availableSeats: "18",
      planeType: "BOEING 747",
      departureDate: "Aug 11, 2018",
      origination: "MUA",
      airlineName: "Delta",
      destination: {
        airportCode: "LAX",
        airportName: "Los Angeles International Airport",
        city: "Los Angeles",
        country: "United States",
        IATA: "LAX",
        ICAO: "KLAX",
        longitude: "33.94250107",
        latitude: "-118.4079971",
        altitude: "125",
        timeZone: "-8",
        DST: "A",
        timeZone: "America/Los_Angeles"
      }
    }
  },
  {
    r: {
```

38. Compare the flightCode of the last right only object with the earlier join results and verify this flightCode does not appear in any of the joined records.
39. Change the outerJoin function back to join; in the Preview pane you should now only see objects with both left and right child objects.

    Note: It is critical you change the outerJoin function back to join before moving on to the next walkthrough steps.

Process the joined flight result array into a structured object

40. Move the current body expression to a variable named joinedPassengerAndFlights.

```
var joinedPassengerAndFlights =
dw::core::Arrays::join(
       passengerRecords,
       flightsWithDetails..*flight,
       (pass) ->  pass."Flight Code",
       (flight) -> flight.flightCode
)
```

41. Create a function named combineJoinResult that is passed a joinResult argument and initially just returns the joinedResult argument.

```
fun combineJoinResult(joinResult) =
joinResult
```

42. Change the body expression to call the combineJoinResult function with the joinedPassengerAndFlights variable.

```
---
combineJoinResult(joinedPassengerAndFlights)
```

43. In the combineJoinResult function, pass joinResult to a map operator to combine the joined left and right records; this should combine the passenger and flight elements in a flat manner.

```
fun combineJoinResult(joinResult) =
joinResult map (e,i) -> e.l ++ flight: e.r
```


```
[
  {
    "First Name": "John",
    "Last Name": "Wayne ",
    "Flight Code": "A134DS",
    PNR: "ER13RL",
    "Passenger Phone Number": "+1 (202) 345-2945",
    "Home Country": "USA",
    price: "750",
    flightCode: "A134DS",
    availableSeats: "50",
    planeType: "BOEING 777",
```

44. Add some additional structure to place the flight details in a child key named flight, and move the passenger "Flight Code" element and the matching flightCode element to an attribute of the flight key named code.

```
fun combineJoinResult(joinResult) =
joinResult map (e,i) ->
e.l - "Flight Code" ++ {flight @(code: e.r.flightCode):( e.r )}
```

Reuse a FlightsLib module function to build a valid XML data structure to store the joined records as travel reservation records

45. Add an import directive to import the dw::modules::ComplexData module.

```
import dw::modules::ComplexData
```

46. Wrap the current body expression in the ComplexData::buildXmlObject function, and set the second argument to "reservation".

```
ComplexData::buildXmlObject(
      combineJoinResult(joinedPassengerAndFlights),
      "reservation"
)
```

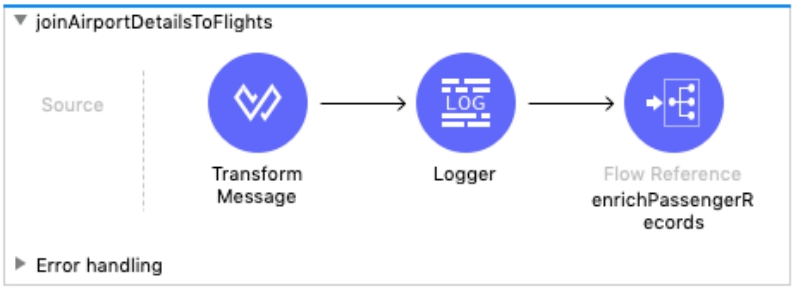
47. Change the output directive to application/xml and verify valid XML displays in the Preview pane.

```
<?xml version='1.0' encoding='UTF-8'?>
<reservations>
  <reservation>
    <First Name>John</First Name>
    <Last Name>Wayne </Last Name>
    <PNR>ER13RL</PNR>
    <Passenger Phone Number>+1 (202) 345-2945</Passenger Phone Number>
    <Home Country>USA</Home Country>
    <flight flightCode="A134DS">
      <price>750</price>
      <availableSeats>50</availableSeats>
      <planeType>BOEING 777</planeType>
      <departureDate>Apr 11, 2018</departureDate>
      <origination>MUA</origination>
      <airlineName>Delta</airlineName>
      <destination>
        <airportCode>LAX</airportCode>
        <airportName>Los Angeles International Airport</airportName>
        <city>Los Angeles</city>
```

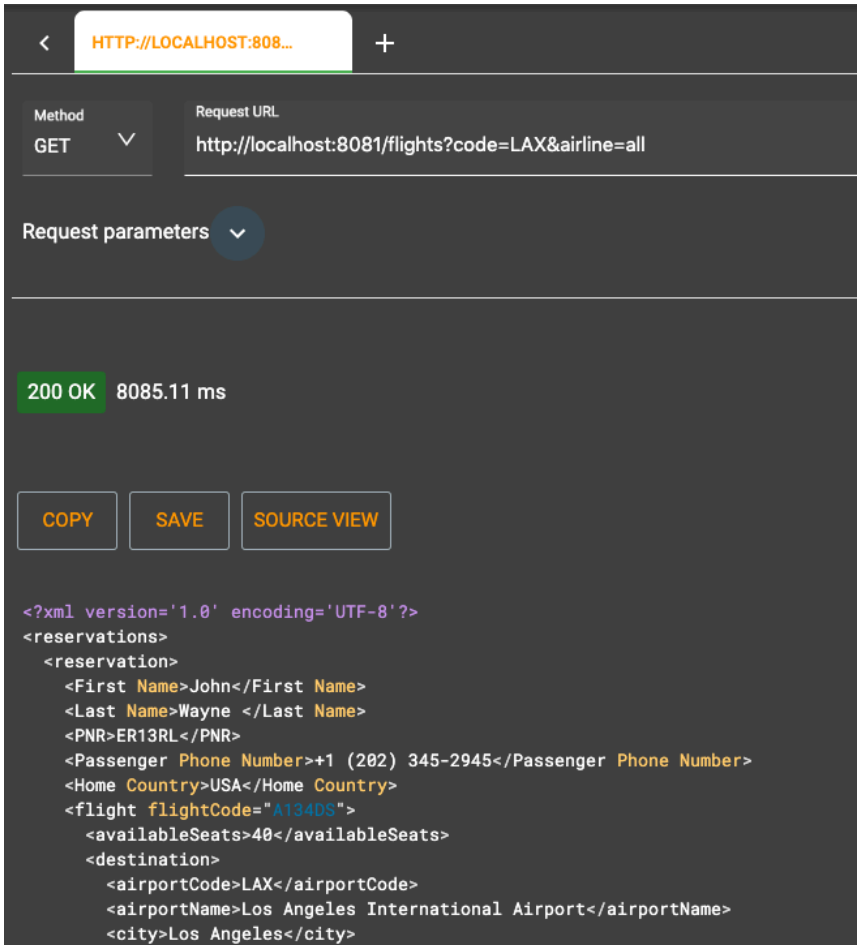
48. Copy the preview pane result and paste it into a new file named reservations.xml in src/main/resources.

Integrate the enrichPassengerRecords flow into the Mule application use case

49. Drag out a Flow Reference component and drop it to the end of the joinAirportDetailsToFlights flow.
50. From the Flow name drop-down list select the enrichPassengerRecords flow.



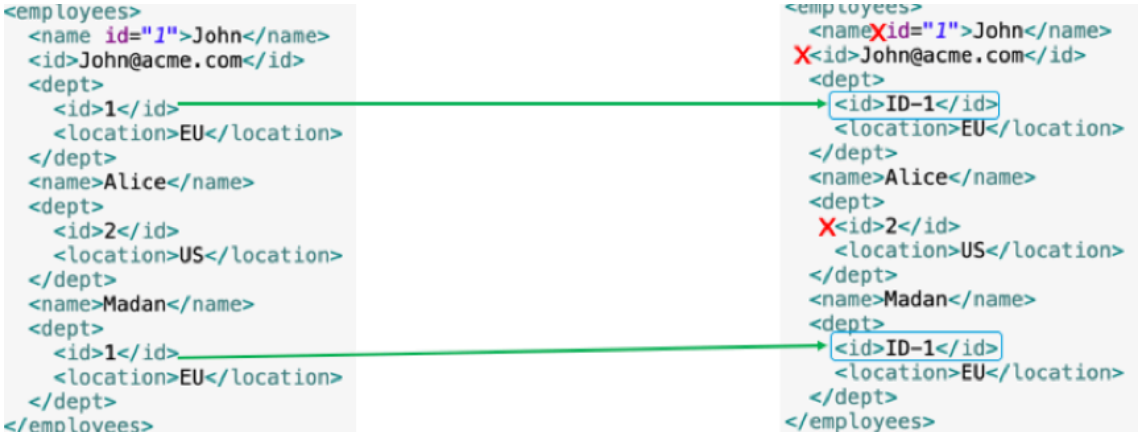
51. Debug the Mule application.
52. Submit a request to http://localhost:8081/flights?code=LAX&airline=all using a REST client such as cURL or the Advanced Rest Client, and verify the requests are accepted and processed, and the result returned is the reservations XML payload.



Walkthrough 6-4: Update and mask complex data structures

In this walkthrough, you use the update function to modify specific branches of a complex data structure, and you use the mask function to hide values for one or more keys in a complex nested data structure. You will:

- Conditionally modify selective branches of a complex data structure by using the update operator.
- Mask values for every occurrence of a certain key or key attribute of a complex data structure by using the mask function.



Use the update operator to format the longitude and latitude values

1. Return to the enrichPassengerRecords flow's Transform Message component.
2. Move the current body expression to a variable named reservationsXml.

```
var reservationsXml =
ComplexData::buildXmlObject(
       combineJoinResult(joinedPassengerAndFlights),
       "reservation"
)
---
reservationsXml
```

3. Open the documentation for the update operator; it's located under DataWeave Reference > DataWeave Operators.
4. Change the output directive to application/dw; this will help you build the update expressions with fewer errors in the Preview pane.
5. In the body expression pass reservationsXml to the update operator.

6. Use the type-ahead feature to help you to add a case statement to select each destination's longitude key and format the key's value to three decimal places; in the Preview pane you should see the longitude values rounded to three decimal places, while the latitude values are not rounded up.

```
reservationsXml update {
 case long at .reservations.*reservation.*flight.destination.longitude ->
 long as Number as String {format: "###.000"}
}
```

7. Add another case statement to select each destination's latitude key and format the key's value to three decimal places; in the Preview pane you should see the latitude values rounded to three decimal places.

```
reservationsXml update {
    case long at .reservations.*reservation.*flight.destination.longitude ->
        long as Number as String {format: "###.000"}
    case lat at .reservations.*reservation.*flight.*destination.latitude ->
        lat as Number as String {format: "###.000"}
}
```

Explore how multiple case statements can interfere with each other

8. Add a case statement first that simply copies the entire destination branch unchanged; you should see the latitude and longitude are no longer formatted to three decimal places.

```
reservationsXml update {
     case dest at .reservations.*reservation.*flight.*destination -> dest
     case long at .reservations.*reservation.*flight.destination.longitude ->
         long as Number as String {format: "###.000"}
     case lat at .reservations.*reservation.*flight.*destination.latitude ->
         lat as Number as String {format: "###.000"}
}
```

```
"ICAO": "KLAX",
"longitude": "33.94250107",
"latitude": "-118.4079971",
"altitude": "125",
```

9. Move the first case statement to the bottom of the update operator; the latitude and longitude should still not be formatted to three decimal places.

   Note: One update operator cannot update the same path multiple times, and the outermost branch wins. Then two case statements have selectors that match the same branch or branches, only the first case is evaluated.

Use multiple update operators to progressively transform various parts of a complex data structure

10. Move the destination case to a second update operator, and modify the expression to remove the "ICAO" key/value; in the Preview pane you should see the ICAO key/value removed.

```
reservationsXml update {
    case long at .reservations.*reservation.*flight.destination.longitude ->
       long as Number as String {format: "###.000"}
    case lat at .reservations.*reservation.*flight.*destination.latitude ->
        lat as Number as String {format: "###.000"}
  //case dest at .reservations.*reservation.*flight.*destination -> dest
}
update {
    case dest at .reservations.*reservation.*flight.*destination -> dest - "ICAO"
}
```

Conditionally update only American Airlines flight records

11. In the last update operator, add a case with a capitalization-insensitive conditional expression to test each flight object to see if the airlineName value is "american"; this should not change the current result in the Preview pane, but also not introduce any errors.

```
update {
    case dest at .reservations.*reservation.*flight.*destination -> dest - "ICAO"
    case flight at .reservations.*reservation.*flight
        if( lower(flight.airlineName) ~= "american" ) -> flight
}
```

12. Change the new value expression call a new function named updateAmerican with the case's flight variable.

```
update {
    case dest at .reservations.*reservation.*flight.*destination -> dest - "ICAO"
    case flight at .reservations.*reservation.*flight
        if( lower(flight.airlineName) ~= "american" ) -> updateAmerican(flight)
}
```

13. Implement the updateAmerican function to iterate over the key/values of the flight object; in the Preview pane you should not see any new errors introduced.

```
fun updateAmerican( flight ) =
flight mapObject (v,k) ->
(k) : v
```

14. In the Preview pane, scroll down and notice the American flights have the ICAO key/value put back, but the other airlines still have it removed, while all the airlines still show the longitude and latitude formatted to three decimal places; this is because of conflicting cases in the same update operator, while the latitude and longitude were updated by a previous update operator.

Note: You'll resolve the ICAO issue later in the walkthrough.

15. Use a match operator to match on the "airlineName" and "flightCode" key names; in the Preview pane you should not see any new errors introduced.

```
fun updateAmerican( flight ) =
flight mapObject (v,k) ->
    k match {
        case "airlineName" -> (k): v
        case "flightCode" -> (k): v
        else -> (k) : v
}
```

16. Change the new value for the "airlineName" case to capitalize the current value and append the string " Airlines".

```
fun updateAmerican( flight ) =
flight mapObject (v,k) ->
    k match {
        case "airlineName" -> (k): capitalize(v) ++ " Airlines"
        case "flightCode" -> (k): v
        else -> (k) : v
}
```

17. Change the new value for the "flightCode" case to uppercase the current value.

```
fun updateAmerican( flight ) =
flight mapObject (v,k) ->
    k match {
        case  "airlineName" -> (k): capitalize(v) ++ " Airlines"
        case "flightCode" -> (k): upper(v)
        else -> (k) : v
}
```

18. Add a case to remove the ICAO key/value again when this conflicting case of the update operator is executed.

```
fun updateAmerican( flight ) =
flight mapObject (v,k) ->
     k match {
        case  "airlineName" -> (k): capitalize(v) ++ " Airlines"
        case  "flightCode" -> (k): upper(v)
        case "ICAO" -> {}
        else -> (k) : v
}
```

Format every key at a certain level of a complex data structure

19. Add another update operator to call a new function named formatKeys to format the outer level passenger information keys of each reservation.

```
update {
    case dest at .reservations.*reservation.*flight.*destination -> dest - "ICAO"
    case flight at .reservations.*reservation.*flight
       if(lower(flight.airlineName) ~= "american") -> updateAmerican(flight)
}
update {
     case res at .reservations.*reservation -> formatKeys( res )
}
```

Note: You use another update operator so as not to conflict with previous update cases.

20. Add an import directive to import the entire dw::core::Strings module.

```
import * from dw::core::Strings
```

21. Implement the formatKeys function to map over the outer level keys of each reservation object and camelize each key; you'll need to combine Strings module functions to do this.

```
fun formatKeys( anObject ) =
anObject mapObject (v,k,i) ->
(camelize(underscore(k))): v
```

22. In the Preview pane, verify the outer level keys are formatted in camel-case, and the other updates also still apply, but the flight key's attributes have been removed.

```
{
  reservations: {
    reservation: {
      firstName: "John",
      lastName: "Wayne ",
      pnr: "ER13RL",
      passengerPhoneNumber: "+1 (202) 345-2945",
      homeCountry: "USA",
      flight: {
        price: "750",
        flightCode: "A134DS",
        availableSeats: "50",
```

23. Use the attributes selector k.@ in the mapObject operator to put back any attributes of the currently mapped key, and also handle the case where a key does not have any attributes.

```
fun formatKeys( anObject ) =
anObject mapObject (v,k,i) ->
    ( camelize(underscore(k)) )
    @( ( k.@ ) )
    : v
```

Note: You must evaluate the Object of key attributes k.@ using evaluation parentheses (k.@) to pass the key/value pairs of each attribute of the current key to the attribute delimiter @( ).

24. In the Preview pane verify the flight key's attributes have been put back.

```
{
  reservations: {
    reservation: {
      firstName: "John",
      lastName: "Wayne ",
      pnr: "ER13RL",
      passengerPhoneNumber: "+1 (202) 345-2945",
      homeCountry: "USA",
      flight @(code: "A134DS"): {
        price: "750",
```

Mask every passengerPhoneNumber values in the modified data structure at every level of the hierarchical complex data structure

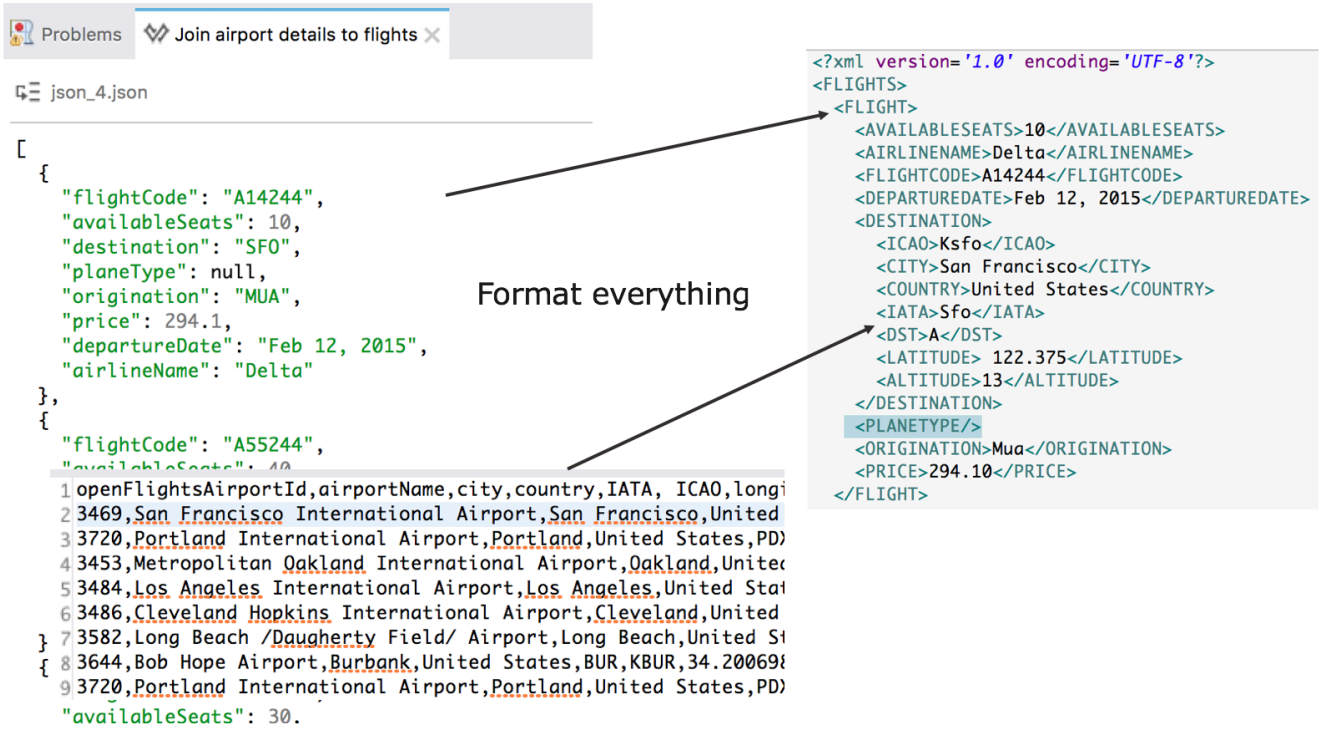
25. Pass the current body expression to the mask function to replace every passengerPhoneNumber value with the masking value "+** (***) ***-****"; you should see every phone number replaced with the mask string at any level of the complex data structure.

```
update {
        case res at .reservations.*reservation -> formatKeys( res )
}
dw::util::Values::mask "passengerPhoneNumber" with "+** (***) ***-****"
```

Walkthrough 6-5: Format and transform complex nested schema by using a recursive function

In the walkthrough, you use a match statement to write a recursive function that can traverse complex schemas and apply different transformation logic to each child node's value, based on criteria such as the data type or other conditional expressions. You will:

- Write a recursive DataWeave function.
- Recursively call a function on elements of a child array by using a map operator inside a match statement.
- Recursively call a function on each element of a child object by using a mapObject operator inside a match statement.
- Apply transformation expressions to every descendent of a complex schema by using a recursive function.



Write a function that matches the type of an input argument

1. Return to the dw-transforms tab.
2. In the joinAirportDetailsToFlights flow, select the first Transform Message component.
3. Change the output directive to application/xml.

4. Make sure the current body expression is

```
ComplexData::buildXmlObject(
    flightsJoinedWithAirportInfoV2( payload ), "flight"
)
```

5. In the Preview pane, verify valid XML appears.
6. Move the body expression to a variable named joinedFlightsXml.

```
var joinedFlightsXml =
ComplexData::buildXmlObject(
    flightsJoinedWithAirportInfoV2( payload ), "flight"
)
```

Create a function to format all the keys in a nested data structure

7. Declare a function named formatDataStructure with an argument named anyInput.

   Note: You can also define all these functions directly in the Transform Message component, but then they are not as reusable.

8. Add a match statement and a default expression to just pass through input unchanged.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
      else -> anyInput
 }
```

9. Change the current body expression to pass joinedFlightsXml to the formatDataStructure function.

```
formatDataStructure( joinedFlightsXml )
```

10. Look in the Preview pane; you should see there are no errors, but there is also no change to the result.

11. Add case conditions to the match statement for Array, Object, String, and Number types.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Array -> anyInput
       case is Object -> anyInput
       case is String -> anyInput
       case is Number -> anyInput
       else -> anyInput
 }
```

12. Look in the Preview pane; you should see there are no errors, but there is also no change to the result.

For array type inputs, recursively call formatDataStructure on each elements the input array

13. Change the Array case's new value expression to use a map operator to call formatDataStructure on each element of the array.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Array -> anyInput map formatDataStructure($)
       case is Object -> anyInput
       case is String -> anyInput
       case is Number -> anyInput
       else -> anyInput
 }
```

For object type inputs, format each key and recursively call formatDataStructure on each child value of the input object

14. Change the Object case's new value expression to use a mapObject operator to convert each key to uppercase.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Array -> anyInput map formatDataStructure($)
       case is Object -> anyInput mapObject (value, key ) ->
             { ( upper(key) ) : value }
       case is String -> anyInput
       case is Number -> anyInput
       else -> anyInput
 }
```

15. Look in the Preview pane; you should see only the outer level key is transformed to uppercase.

16. Call this formatDataStructure function on each value inside the mapObject operator.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Array -> anyInput map formatDataStructure($)
       case is Object -> anyInput mapObject (value, key ) ->
             { (upper(key) ) : formatDataStructure(value) }
       case is String -> anyInput
       case is Number -> anyInput
       else -> anyInput
 }
```

17. Look in the Preview pane; you should see every key is in uppercase, and the simple literal values print out the type, not the values.

```
{
  "FLIGHTS": {
    "FLIGHT": {
      "AVAILABLESEATS": 50,
      "AIRLINENAME": "Delta",
      "FLIGHTCODE": "A134DS",
      "DEPARTUREDATE": "Apr 11, 2018",
      "DESTINATION": {
        "AIRPORTCODE": "LAX",
        "AIRPORTNAME": "Los Angeles Inte
```

Note: formatDataStructure now descends to each child object, until formatDataStructure is called on the simple literal values.

Modify the formatKey function so that it also recursively modifies values of key/value pairs

18. Make a copy of the formatDataStructure function and rename the copied function to formatDataStructureV1.
19. In the formatDataStructureV1 expression, replace the two recursive calls to formatDataStructure with formatDataStructureV1.

```
fun formatDataStructureV1( anyInput ) =
 anyInput match {
       case is Array -> anyInput map formatDataStructureV1($)
       case is Object -> anyInput mapObject (value, key ) ->
             { (upper(key) ) : formatDataStructureV1(value) }
       case is String -> anyInput
       case is Number -> anyInput
       else -> anyInput
 }
```

20. Return to the formatDataStructure function and change the String case's expression to convert each value to lowercase.

```
case is String -> lower( anyInput )
```

21. Modify the else expression to make the default behavior to pass through the anInput argument unchanged.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Array -> anyInput map formatDataStructure($)
       case is Object -> anyInput mapObject (value, key ) ->
              { (upper(key) ) : formatDataStructure(value) }
       case is String -> lower( anyInput )
       case is Number -> anyInput
       else -> anyInput
 }
```

22. Look in the Preview pane; you should see every string type value is lowercase.

```
"FLIGHTCODE": "a134ds",
"DEPARTUREDATE": "apr 11, 2018",
"DESTINATION": {
  "AIRPORTCODE": "lax",
  "AIRPORTNAME": "los angeles interr
```

23. Change the Number case's expression to apply formatting to numbers to print out + for positive numbers.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Array -> anyInput map formatDataStructure($)
       case is Object -> anyInput mapObject (value, key ) ->
              { (upper(key) ) : formatDataStructure(value) }
       case is String -> lower( anyInput )
       case is Number -> anyInput as String {format: "+#,##0.00;-#"}
       else -> anyInput
 }
```

24. Look in the Preview pane; you should see the AVAILABLESEATS value is now formatted to two decimal places, with a + in front of positive numbers.

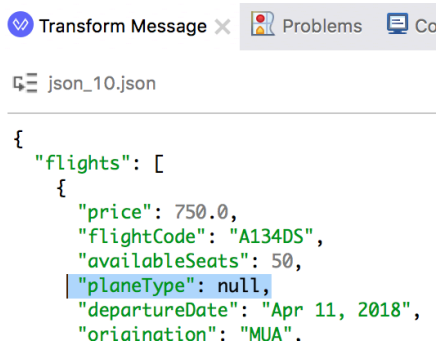
```
{
  "FLIGHTS": {
    "FLIGHT": {
      "AVAILABLESEATS": "+50.00",
      "AIRLINENAME": "delta",
      "FLIGHTCODE": "a134ds",
```

25. Change the output directive to application/xml; the Preview pane should display valid XML without any errors.

```
<?xml version='1.0' encoding='UTF-8'?>
<FLIGHTS>
  <FLIGHT>
    <AVAILABLESEATS>+50.00</AVAILABLESEATS>
    <AIRLINENAME>delta</AIRLINENAME>
    <FLIGHTCODE>a134ds</FLIGHTCODE>
    <DEPARTUREDATE>apr 11, 2018</DEPARTUREDATE>
    <DESTINATION>
      <AIRPORTCODE>lax</AIRPORTCODE>
      <AIRPORTNAME>los angeles international airport</AIRPORTNAME>
      <CITY>los angeles</CITY>
      <COUNTRY>united states</COUNTRY>
```

Add matching logic to convert null values to empty strings

26. In the Input pane, edit the sample input payload to change the planeType values of the first flight object to null.



27. Look in the Preview pane; you should see the planeType null value as a single self-closed XML tag.

```
      <COUNTRY>united states</COUNTRY>
      <TIMEZONE>-8</TIMEZONE>
      <TIMEZONE>america/los_angeles</TIMEZONE>
    </DESTINATION>
    <PLANETYPE/>
    <ORIGINATION>mua</ORIGINATION>
    <PRICE>+750.00</PRICE>
  </FLIGHT>
  <FLIGHT>
    <AVAILABLESEATS>+18.00</AVAILABLESEATS>
```

28. Add a case to match the Null type and then convert the input to an empty string.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Null -> ""
       case is Array -> anyInput map formatDataStructure($)
       . . .
```

Trim and format every key in the nested XML data structure

29. Change the formatDataStructure function to camelize every key, instead of converting every key to uppercase.

```
fun formatDataStructure( anyInput ) =
 anyInput match {
       case is Null -> ""
       case is Array -> anyInput map formatDataStructure($)
       case is Object -> anyInput mapObject (value, key ) ->
           { ( formatDataStructure(key) ) : formatDataStructure(value) }
       case aKey is Key -> dw::core::Strings::camelize(trim(aKey))
       case is String -> lower( anyInput )
       case is Number -> anyInput as String {format: "+#,##0.00;-#"}
       else -> anyInput
 }
```

30. Look in the Preview pane; you should see the new key formatting is applied to every nested key.

```
<?xml version='1.0' encoding='UTF-8'?>
<flights>
  <flight>
    <price>+750.00</price>
    <flightCode>a134ds</flightCode>
    <availableSeats>+50.00</availableSeats>
    <planeType/>
    <departureDate>apr 11, 2018</departureDate>
    <origination>mua</origination>
    <airlineName>delta</airlineName>
    <destination>
      <airportCode>lax</airportCode>
      <airportName>los angeles international airport</airportName>
      <city>los angeles</city>
      <country>united states</country>
```

Note: If you add the index 5 to the joinedAirportDetails function implementation, you should see the " ICAO" key name is trimmed and dasherized as "icao".

Test your solution

31. Run or debug the Mule application.
32. In a web client, submit GET requests to various airlines and destination codes; each request should return an XML response with formatted keys and values.

    http://localhost:8081?code=SFO&airline=delta

33. Stop the Mule application.

Note: If you get errors, a copy of the formatDataStruct solution is in your snippets file, or you can load up the solution project into your Anypoint Studio workspace.

Module 7: Reducing Data from Arrays

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

- Reduce and accumulate array elements to other output types using the reduce operator.
- Calculate key performance indicators from input collections by using the reduce operator.

Walkthrough 7-1: Accumulate transformations of array elements by using a reduce operator

In this walkthrough, you use a reduce operator to accumulate the results of a transformation expression applied iteratively to each element of an array. You also set various initial values for the reduce operator's accumulator to output to various data types such as a string, number, object, or another array. You will:

- Iterate over array elements and accumulate results by using the reduce operator.
- Set an initial accumulator value in a reduce operator to change the output type.
- Reduce an array of objects by using the reduce operator.

Create a new flow to write tests of the reduce function

1. Return to the dw-tests tab.
2. From the Mule Palette, drag out a new Transform Message component and drop it in the canvas to create a new flow.
3. Name the flow testReduce.
4. Select the Transform Message component and change the output directive to application/json.
5. In the header, define a new variable named array1 equal to the array [2,3,4,5].

```
var array1 = 2 to 5
```

Reduce a simple array of numbers

6. In the body expression, write an expression to reduce array1 over each iterated array element.

```
{
   reduceElement: array1 reduce (element, acc) -> element
}
```

7. Look in the Preview pane; you should see the result is 5, the last element of array1.
8. Add a key/value to use the reduce operation to use the default parameter for the current iterated array element.

```
{
   reduceElement: array1 reduce (element, acc) -> element,
   reduceElement2: array1 reduce $
}
```

9. Look in the Preview pane; you should see the result is still 5, the last array element.
10. Add a test to reduce over the accumulator, not the value.

```
reduceAccumulator: array1 reduce $$
```

11. Look in the Preview pane; you should see the result is 2, the first array element of the input array1.

Note: The reduce operation expression is a constant value, the initial acc value.

Write the last test using named parameters

12. Add a key/value to reduce over the accumulator, not the value.

```
reduceAccumulator2: array1 reduce (element, acc) -> acc
```

13. Look in the Preview pane; you should see the result is 2, the initial array element.

Note: The reduce operation expression acc is never modified by any of the element iterations, so the reduce operation ends set to the initial acc value, which by default is set to array1[0].

Change the initial accumulator value to 1 instead of 2

14. Copy the last key/value to a new key/value and set the accumulator to 1.

```
initAccumulator: array1 reduce (element, acc=1) -> acc
```

15. Look in the Preview pane; you should see the result is now 1, the initial value of the accumulator.

Note: The reduce operation expression set to acc, which is never changed between iterations, so the reduce operation ends set to the initial acc value, which was set to 1.

Sum up the elements of array1

16. Add a key/value to add each element of array1 to the accumulator.

```
sumNumbersFrom1: array1 reduce (element, acc=1) -> element + acc
```

17. Look in the Preview pane; you should see the answer is 15, the sum of the integers 1 to 5.
18. Copy and paste the last key/value and rename the new key sumNumbers, and change the acc parameter to initialize with its default value.

```
sumNumbers: array1 reduce (element, acc) -> element + acc
```

19. Look in the Preview pane; you should see the result is 14, the sum of the integers 2 to 5.

```
%dw 2.0
output application/json

var array1 = 2 to 5
---
{
    reduceElement: array1 reduce (element, acc) -> element,
    reduceElement2: array1 reduce $,
    reduceAccumulator: array1 reduce $$,
    reduceAccumulator2: array1 reduce (element, acc) -> acc,
    initAccumulator: array1 reduce (element, acc=1) -> acc,
    sumNumbersFrom1: array1 reduce (element, acc=1) -> element + acc,
    sumNumbersFrom1: array1 reduce (element, acc) -> element + acc
}
```

```
{
  "reduceElement": 5,
  "reduceElement2": 5,
  "reduceAccumulator": 2,
  "reduceAccumulator2": 2,
  "initAccumulator": 1,
  "sumNumbersFrom1": 15,
  "sumNumbersFrom1": 14
}
```

20. Add a key/value to reverse the order, so the next element is added after the acc value.

```
sumNumbersReverse: array1 reduce (element, acc) -> acc + element
```

21. Look in the Preview pane; you should see the result is 14.

    Note: This accumulates the result in the reverse order of the previous expression, but 2 + 3 + 4 + 5 is the same as 5 + 4 + 3 + 2. The results might not be equal if each element is a non-numeric type like a string, array or key/value.

22. Add a key/value to compare this result with the dw::core::Arrays::sumBy function.

```
sumNumbers2: array1 dw::core::Arrays::sumBy $
```

23. Look in the Preview pane; you should see the result is 14, the same as the previous sumNumbersReverse expression.

Concatenate the elements of array1

24. Add a key/value to concatenate each element of array1 to the accumulator.

```
concatStrings: array1 reduce (element, acc) -> acc ++ element
```

25. Look in the Preview pane; you should see the answer is the string "2345".
26. Change the expression to insert a comma (,) between each integer.

```
concatStrings2: array1 reduce (element, acc) -> acc ++ "," ++ element
```

27. Look in the Preview pane; you should see the answer is the string "2,3,4,5".

28. Add a key/value to concatenate the accumulator to the next element.

```
concatStringsReverse: array1 reduce (element, acc) -> element ++ "," ++ acc
```

29. Look in the Preview pane; you should see the answer is the string "5,4,3,2".
30. Add a key/value to concatenate the accumulator to the next element, and set an initial accumulator value of 1.

```
concatStringsReverseFrom1:
array1 reduce (element, acc=1) ->
   element ++ "," ++ acc
```

31. Look in the Preview pane; you should see the answer is the string "5,4,3,2,1".
32. Add a key/value to concatenate the accumulator to the next element, and set an initial accumulator value of 1.

```
concatStringsFrom1: array1 reduce (element, acc=1) -> acc ++ "," ++ element
```

33. Look in the Preview pane; you should see the answer is the string "1,2,3,4,5".

Concatenate the elements of an array of strings

34. In the header, declare a new variable named stringArray containing a few string elements.

```
var stringArray = ["one", "two", "three", "go!"]
```

35. Add a test to concatenate together these array elements forward, from left to right.

```
concateStrings: stringArray reduce (element, acc) -> acc ++ "," ++ element
```

36. Look in the Preview pane; you should see the result is the string "one,two,three,go!".
37. Add a test to concatenate together these array elements backwards, from right to left.

```
concatStringsReverse:
stringArray reduce (element, acc) ->
   element ++ "," ++ acc
```

38. Look in the Preview pane; you should see the result is the string "go!,three,two,one".

Count the number of strings in stringArray with at most 3 letters

39. Add a test to test if the size of the current element is at most 3, and set an initial accumulator to accurately count each time this test is true.

```
countSmallStrings: stringArray reduce (element, acc=0) ->
(if(sizeOf(element) < 4) 1 else 0) + acc
```

40. Look in the Preview pane; you should see the result of the reduce operation is 3.

41. Write another test that does not use the reduce operator to perform the same conditional count of the array elements.

```
countSmallStrings2: stringArray dw::core::Arrays::countBy(sizeOf($) < 4)
```

42. In the Preveiw pane, verify the result is also 3.

Use the reduce operator to flatten an array of arrays

43. In the header, add a variable named nestedArrays containing a few child arrays.

```
var nestedArray = [ [1], [2,3], ["hello"] ]
```

44. In the body expression, add a test to extract and concatenate together the elements of each child array.

```
flattenChildArrays: nestedArray reduce (element, acc=[]) -> acc ++ element
```

45. Look in the Preview pane; you should see the array is flattened.

```
"flattenChildArrays": [
    1,
    2,
    3,
    "hello"
  ]
```

Flatten an array of objects into a single object

46. In the header, define a variable that is an array of objects.

```
var objArray = [
   {one: 1},
   {two: 2, three: 3}
]
```

47. Add a test to concatenate each key/value of objArray into one object, in the same order as objArray.

```
flattenForwards: objArray reduce (element, acc={})  -> acc ++ element
```

48. Look in the Preview pane; you should see the result is a single object with key/values in the same order they occur in objArray.

```
flattenForwards: {
   "one": 1,
   "two": 2,
   "three": 3
}
```

49. Add a test with an equivalent expression that does not use the reduce operator.

```
flattenForwards2: {( objArray )}
```

50. Look in the Preview pane; you should see flattenForwards and flattenForwards2 produce the same result.
51. Add a test to concatenate each object of objArray into one object, in the reverse order as objArray.

```
flattenBackwards: objArray reduce (element, acc={})  -> element ++ acc
```

52. Look in the Preview pane; you should see the result is a single object with key values read backwards from objArray.

```
flattenBackwards: {
   "two": 2,
   "three": 3,
   "one": 1
}
```

53. Add a test with an equivalent expression that does not use the reduce operator.

```
flattenBackwards2: {( objArray[-1 to 0] )}
```

Walkthrough 7-2: Calculate key performance indicators (KPIs) by using reduce operators

In this walkthrough, you extend the skills covered in the last walkthrough by using more complex initial accumulator values in reduce operators to summarize statistics over an input collection. This produces data that can be used to track key performance indicators (KPIs). You will:

- Use the accumulator in a reduce operator to collect key performance data from an input collection.
- Append accumulated KPI data to the original input collection.

Initialize a reduce operator's accumulator to track some key performance indicators from an input collection

1. In Anypoint Studio, select the dw-transforms tab.
2. In the joinAirportDetailsToFlights flow, select the Transform Message component.
3. Change the output directive to application/json.
4. Define a new function named deltaFlightsStats that accepts an input argument named flightsInput.

   ```
   fun deltaFlightKpis(flightsInput)
   ```

5. Implement the function to just return the input argument flightsInput.

   ```
   fun deltaFlightKpis(flightsInput) = flightsInput
   ```

6. Comment out the current body expression and change it to call deltaFlightKpis with input flightsJoinedWithAirportInfoV2(payload).

   ```
   ---
   deltaFlightKpis( flightsJoinedWithAirportInfoV2(payload) )
   ```

7. Look in the Preview pane; you should see an array of flights joined with airport details.
8. Modify the deltaFlightsKpis function to call the reduce operator with an accumulator set to an initial object to accumulate some KPIs for the number of flight elements and the sum of all the prices, and for now just return the initial accumulator.

```
fun deltaFlightKpis(flightsInput : Array) =
flightsInput reduce (
       ( flight,
         acc={
              delta: {count: 0, price: 0}
         }
       ) ->
       acc
)
```

9. Look in the Preview pane; you should see the initial accumulator object.

```
{
  "delta": {
    "count": 0,
    "price": 0
  }
}
```

Accumulate KPIs for delta flights

10. Add a match statement based on the current flight object's airlineName, and return the current accumulator value.

```
fun deltaFlightKpis(flightsInput : Array) =
flightsInput reduce (
       ( flight,
         acc={
              delta: {count: 0, price: 0}
         }
       ) ->
       flight.airlineName match {
              else -> acc
       }
)
```

11. Look in the Preview pane; you should see the same initial accumulator object.

12. Add a case to the match statement to test if the current flight object's airlineName matches "Delta", ignoring case, and if there is a match, return a string indicating at least one flight object contains "Delta".

```
flight.airlineName match {
  case delta if(lower(delta) contains "del") -> "at least one Delta flight"
  else -> acc
}
```

13. Look in the Preview pane; you should see the string "at least one Delta flight".
14. Modify the delta case expression to add 1 to the accumulator's count, and add the current flight price to the accumulator's price.

```
fun deltaFlightKpis(flightsInput : Array) =
flightsInput reduce (
  ( flight,
    acc = { delta: {count: 0, price: 0} }
  ) ->
  flight.airlineName match {
    case delta if(lower(delta) contains "del") ->
      delta: {
        price: acc.delta.price + flight.price,
        count: acc."delta".count + 1
      }
    else -> acc
  }
)
```

15. Look in the Preview pane; you should see the delta key with the total number of delta flights and the sum of the flight prices.

```
{
  "delta": {
    "price": 1445.99,
    "count": 3
  }
}
```

Calculate statistics from the reduce operator's KPIs

16. Refactor the current body expression into a do statement using a local variable named deltaStats.

```
do {
  var deltaStats = deltaFlightKpis(flightsJoinedWithAirportInfoV2(payload) )
  ---
  deltaStats
}
```

17. In the do statement's header, after the deltaStats variable, add a variable named aveDeltaPrice to calculate the average delta flight price from the deltaStats variable, and change the do statement's body expression to aveDeltaPrice.

```
do {
  var deltaStats = deltaFlightKpis(flightsJoinedWithAirportInfoV2(payload) )
  var aveDeltaPrice = (
    (deltaStats..price[0] as Number) / (deltaStats..count[0] as Number ) )
  ---
  aveDeltaPrice
}
```

18. Look in the Preview pane; you should see the average Delta flight price:

```
481.99666666666666666666666666667
```

19. Properly format aveDeltaPrice to two decimal places.

```
do {
  var deltaStats = deltaFlightKpis(flightsJoinedWithAirportInfoV2(payload) )
  var aveDeltaPrice = (
    (deltaStats..price[0] as Number) / (deltaStats..count[0] as Number ) )
    as String {format: "#,###.00"} as Number
  ---
  aveDeltaPrice
}
```

20. Look in the Preview pane; you should see the average price is displayed as a Number properly formatted to two decimal places.