Dataweave 2.0 Student Manual

August 31, 2020

Contents

Module 1

Fundamentals – Review++

WT 1-1 Import a basic Mule project into Anypoint Studio

Import the starter project

1. Start Anypoint Studio
2. Create a new workspace
3. Import the `apdw2-flights-starter.jar` project under the `studentFiles/mod01`

Create new project

4. Create a new project
   Creating a new project and copying only the files you minimally need for the class helps in containing the "noise" that is introduced with the starter project. Additionally, there is the extra benefit of not having to deal with students who are having compilation issues with the starter project.
   Furthermore, you become familiar with the starter project in case you would like to follow the manual that comes packaged with the class.
5. Create a new project and call it `dataweave`
6. From the `apdw2-flights-starter` copy the following files over to the new project:
   (a) `src/main/resources/airportInfoTiny.csv` to `src/main/resources`
   (b) `src/main/resources/examples/mockdata/deltaSoapResponsesToAllDestinations.xml` to `src/test/resources`
   (c) `src/test/resources/flight-example.json` to `src/test/resources`

WT 1-2 Fundamentals review++

In this WT the goal is to attempt (I am saying attempt because often enough we have participants who don't meet the prerequisites) to bring everyone at the same level by (1) reviewing fundamentals and (2) illustrating features of DW that we will be using throughout the class.

Create the flow, set the metadata

1. Rename the `dataweave.xml` to `mod1.xml`
2. Create a new flow named `mod1-review++`
   The reason for prefixing the flow name with the name of the flow is one of good practice. Such a convention will improve the readability of your flows by identifying the Mule Configuration file a flow is defined under by just looking at a Flow Reference's display name.
3. Drop a DW (aka `Transform Message`) processor to the process area of the flow
4. Define the payload input metadata to the `src/test/resources/flight-example.json`, set the name of the type to `flight_json`

5. Edit the sample data
6. Turn on the preview
7. Change the output to JSON

Construction

8. What are the semantics of `{}` in DW?
   (a) Object creation
9. What are the semantics of `[]` in DW?
   (a) Array creation

Fields

10. Three different ways of accessing the field `airline` out of the `payload`. What are they?
    (a) `payload.airline`
    (b) `payload["airline"]`
    (c) `payload[0]`

    Let me let you in a secret: Objects internally are represented as arrays—field access is a facade

11. Why DW stores objects as arrays?
    (a) Because DW is the only language I know of that allows the creation of objects with duplicate field names...

```
{
  a: 1,
  a: 2,
  a: 3
}
```

    ... and the only way I can access the second and third field is through an index access. But now we have more questions that need to be answered.

    (b) Why would a language allow for such a feature? That is duplicate fields within an object. Because of XML, how else you expect to be able to generate XML with tags that repeat:

```
%dw 2.0
output application/xml
---
"as": {
  a: 1,
  a: 2,
  a: 3
}
```

String concatenation

12. Two ways to concatenate strings
    (a) `"The flight is operated by " ++ payload.airline`
    (b) `"The flight is operated by ${payload.airline}"`
13. You have to be careful that the expression inside the `${}` returns a string, otherwise you will be getting type mismatch errors.

Conditional expressions

14. `if then else` conditional
    (a) `if (true) 1 else 0`
    (b) `if (false) 1 else 0`
15. Nullity conditional
    (a) `null default "Other value"`
    (b) `"The value" default "Other value"`
16. Conditional elements
    (a) Objects

```
{
  a: 1,
  (b: 2) if (true),
  (c: 3) if (false)
}
```

    (b) Arrays

```
[
  1,
  (2) if (true),
  (3) if (false)
]
```

Array access and Ranges

17. Array access
    (a) `[2,6,4,1,7][0]` evaluates to `2`
    (b) `[2,6,4,1,7][-1]` evaluates to `7`
18. Ranges
    (a) `0 to 5` evaluates to the `[0,1,2,3,4,5]` array
    (b) `5 to 0` evaluates to the `[5,4,3,2,1,0]` array
19. Ranges, Arrays, and Strings
    (a) `[2,6,4,1,7][1 to -2]` evaluates to the `[6,4,1,7]` sub-array
    (b) `[2,6,4,1,7][-1 to 0` reverses the array
    (c) `payload.airline[-3 to -1]` evaluates to the last characters in the string
    (d) `payload.airline[-1 to 0]` reverses the string

Common functions

20. `typeOf`
    This is a great function for debugging–again and it will help us identify the types of data we are working with. We will use it a few times to gain clarity when all else has failed.
    (a) `typeOf([])`
    (b) `typeOf()`
21. `sizeOf`
    (a) `sizeOf()`
    (b) `sizeOf(a: 1)`

(c) `sizeOf(0 to 100)`

(d) `sizeOf("ABC")`

22. `contains`

    (a) `[2,6,4] contains 2`

    (b) `"ABCD" contains "BC"`

23. `is`

    (a) ` is Object`

    (b) `[] is Array`

Expression chaining

24. Create an array of integers

    (a) Do you know what expression chaining is?

        i. `[2,5,3,7,8] map` $+1map$`-1`

    (b) We learned all about expression chains in elementary math!

        i. `1 + 2 - 3`

25. This is a good opportunity to briefly talk about the `map` function

    (a) Do you know what the `map` semantics are?

        i. `map` is a function

        ii. `map` is invoked using infix notation

        iii. `map` takes two arguments and evaluates to a value

            A. Left: an array

            B. Right: a λ (lambda) Expression (aka Anonymous Function). A λ function is a function that you define and apply in a specific context, very similar to an anonymous class (in OOP) that you define and instantiate once.

            C. Returns: another array whereby every element from the input array has been passed as an argument to the λ function.

Transform XML to JSON

26. Create a new flow and name it `mod1-xml2json`

27. Set the input payload metadata to `src/test/resources/deltaSoapResponsesToAllDestinations.xml`, name the new type `flights_xml`

28. Edit the sample data

29. Turn on the preview

30. change the output to `JSON`

31. Replace `{}` to `payload`

32. Explore the structure in the Preview and focus on the objects created with `return` fields repeating

33. Is this a valid JSON data structure?
    According to the JSON specification this is a valid JSON. But it is not appropriate.
    Transform the XML into a JSON collection containing the objects found under `return` tags

34. Use the `..*` selector to perform a recursive search and find fields named `return`

    ```
    payload..*return
    ```

35. Go to the first element in the sample data under the `return` tag

36. Add another `return` tag with a simple value

```
<return>
  <airlineName>Delta</airlineName>
  <code>A1B2C3</code>
  <departureDate>2018/03/20</departureDate>
  <destination>SFO</destination>
  <emptySeats>40</emptySeats>
  <origin>MUA</origin>
  <planeType>Boing 737</planeType>
  <price>400.0</price>
  <return>10</return>
</return>
```

37. Illustrate that `..*` performs a breadth-first search and the output contains an extra result all the way to the bottom.

38. Use the `.*` selector to perform a search at the right level–no longer do we receive the next `return` result.

```
payload.findflightResponse.*return
```

39. Restore your sample data by removing the additional nested `<return>10</return>` XML tag.

40. Ensure you make use of the namespace from the input data. Ignoring namespaces is not advised unless you are certain the data will always look the same, you will never have another `findFlightResponse` tag with a different meaning

```
ns ns2 http://soap.training.mulesoft.com/
---
payload.ns2#findFlightResponse.*return
```

41. Copy all the data from the preview

42. Create a new file under `src/test/resources` and call it `flights.json`

Transform JSON to XML

43. Create a new flow and name it `mod1-json2xml`

44. Drop a DW to the process area

45. Set the input payload metadata to `src/test/resources/flights.json`

46. Edit the sample data

47. Turn on the preview

48. Change the output to XML

49. Replace `{}` to `payload`

50. The error error says `Cannot coarse an array ... to a String`

    (a) The problem lies with XML not having any knowledge of arrays but just repeating elements to indicate sequences. No other format that I know of has such semantics, other formats have knowledge and serialization of the array type.

    (b) We need to proceed by eliminating the arrays

51. Create an appropriate XML for just two elements of the inputs

```
%dw 2.0
output application/xml
---
flights: {
  flight: payload[0],
  flight: payload[1]
}
```

52. Set the output to `application/dw` and identify the internal data structure we must aim for when generating XML

53. We need to generate XML for all elements not just the first two, change the code so that we now iterate over the collection of data in the `payload`

```
%dw 2.0
output application/dw
---
flights: payload map {
  flight: $
}
```

54. Switch the output back to XML results in errors because we are still having an array in our data structure

55. Change the output yet again to `application/dw`

56. Eliminate the array by enclosing the `map` in {()}

```
%dw 2.0
output application/dw
---
flights: {(payload map {
  flight: $
})}
```

The semantics of () are the usual precedence operators, however ***the semantics of parenthesis change when they appear on their own within {} enclosing (i) objects or (ii) arrays of objects*** *to the following:* ***Break every single object into pairs of keys and values****. The outer {} are there to construct a new object from all the pairs of keys and values. Hence why we end up with single object containing all the keys and their associated values for each object in the collection.*

57. *So far we solved this transformation by following a top-to-bottom solution. You can also solve this transformation by following a bottom-up approach.*
Change the expression back to just `payload` and eliminate the array first!

```
%dw 2.0
output application/dw
---
flights: {(payload)}
```

58. Organize the records around their own tag before we destroy the array and collapse the first level of containing objects.

```
%dw 2.0
output application/dw
---
flights: {(payload map flight: $)}
```

Note that objects with a single field can have the {} omitted

59. Finally, change the output back to XML

Module 2

Use Case: XML flights to JSON

As a developer I want to receive XML flight records and generate an enhanced JSON data structure with the corresponding flights. The JSON data structure is enhanced as follows:

- It contains the corresponding appropriate JSON structure contain the flights from the XML inputs
- It is enhanced with total seating capacity per flight
- It is enhanced with prices adjusted across a number of currencies
- It is enhanced such that important KPIs are contained at the same level as the flights

WT 2-1 Organize and Reuse DataWeave code

Create a new flow

1. Create a new Mule Configuration file and name it `mod2`, it will contain the solutions to all WTs from module 2.
2. Create a new flow named `mod2-functions`
3. Define the payload input metadata to the flights_xml
4. Edit the sample data
5. Turn on the preview
6. Change the output to `application/dw`
7. Change the body of the expression to `payload..*return`

Create a variable

8. Create a variable visible throughout the DW expression

```
var theTotalSeats = 400
```

9. Add the `totalSeats` field to the existing list of objects, do it for a single object then do it for all objects in the collection

```
%dw 2.0
output application/json
var theTotalSeats = 400
---
payload..*return[0] ++ {
  totalSeats: theTotalSeats
}
```

10. Do it now for all elements

```
%dw 2.0
output application/json
var theTotalSeats = 400
---
payload..*return map ($ ++ {
  totalSeats: theTotalSeats
})
```

++ we have already seen when concatenating strings we see it operating with objects as well because it is overloaded, more on overloading soon.

11. There is another way to add a field(s) to an existing object

```
%dw 2.0
output application/json
var theTotalSeats = 400
---
payload..*return map {
  ($)
  totalSeats: theTotalSeats
}
```

We have already seen $\{()\}$ when eliminating arrays, here these $()$ are applied to single objects with the same effect; i.e. destroy the object and retrieve the basic building blocks of the object, that is the keys and the associated values. These basic building blocks are then introduced in the new object created by the outermost object.
Pick the method you prefer to concatenate objects, I prefer the latter which is the one I shall be using for the duration of this class.

Fix the Boing typo

12. Add another field containing the `planeType` value

```
%dw 2.0
output application/dw
var theTotalSeats=400
---
payload..*return map {
  ($),
  planeType: $.planeType,
  totalSeats: theTotalSeats
}
```

A quick inspection of the output identifies two `planeType` fields containing the same value. This is a problem because we are interested in a single field containing the corrected spelling for Boeing. We should remove one of them and removing the first one is the right choice because this first one is the one we have no control over. The second `planeType` is the one we have full control because we are adding it!

13. Remove the first `planeType` field

```
%dw 2.0
output application/dw
var theTotalSeats=400
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType,
  totalSeats: theTotalSeats
}
```

This unary – operator is overloaded to also apply to objects and remove fields. You can keep applying it successively if you want to remove multiple fields–e.g. `$ - "planeType" - "price"`

14. Fix the `Boing` typo

```
%dw 2.0
output application/dw
var theTotalSeats=400
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: theTotalSeats
}
```

It is worth noting that `replace` and `with` are two separate functions where the result of `replace` is the first argument to `with`. It makes the code "talk to you in English"! This is the beauty of infix function invocation.

Calculate the total seats as a function of the `planeType` using `fun`

15. Create and apply a function and start unit-testing it

```
%dw 2.0
output application/dw
var theTotalSeats = 400
fun getTotalSeats(pt) = pt
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeats($.planeType)
}
```

`pt` is a user defined arbitrary name, denoting the sole input parameter
By unit testing we refer to the method by which small chunks of our functionality is tested before we put them all together. We are not referring to automated/regression testing.

16. Create the condition that identifies 737s over the other types of planes

```
%dw 2.0
output application/dw
var theTotalSeats = 400
fun getTotalSeats(pt) = pt contains "737"
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeats($.planeType)
}
```

17. Enclose the condition in an `if` expression

```
fun getTotalSeats(pt) = if (
  pt contains "737"
) 150 else 300
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeats($.planeType)
}
```

18. Change the function expression to allow for the 727 and 707 to be set to 150 seats

```
fun getTotalSeats(pt) = if (
  pt contains "737" or
  pt contains "707" or
```

```
  pt contains "727"
) 150 else 300
```

19. Fix the error `Cannot coerce String (737) to Boolean`

```
fun getTotalSeats(pt) = if (
  (pt contains "737") or
  (pt contains "707") or
  (pt contains "727")
) 150 else 300
```

Parenthesizing to enforce precedence is required in this context because `or` has higher precedence vs `contains` A chunk of the issues you will have when you start writing DW expressions on your own will stem from precedence rules.

20. Discuss issues with the `getTotalSeats` functions

    (a) We execute this function once per record

    (b) We are searching strings

    (c) We do this string search three times

    (d) The function is not that efficient, we could do better

Please do not think for a moment that in modern computing string searches as slow, they are fast and could be optimized in a number of ways. Nonetheless, this discussion has merit in the presence of large to very large data sets where the function is called once per record; i.e. every little bit helps!

Calculate efficiently the total seats as a function of `planeType` using a lambda expression

21. Create another function named `getTotalSeatsL`

```
var getTotalSeatsL = (pt) -> pt
```

L stands for Lambda, we store an anonymous function to a variable; i.e. we provide this anonymous function with a name. Additionally, the body of this function evaluates into the argument we passed–this encourages unit testing.

22. Apply the function to in the expression and get results

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType)
}
```

Applying the function as soon as possible and getting results as we further develop our function will only facilitate unit testing and code correctness.

23. Get the last three characters of the plane type

```
var getTotalSeatsL = (pt) -> pt[-3 to -1] as Number
```

We can now use the number to compare instead of doing a string search which will speed up the execution of our code.

24. Introduce closures (i.e. localized declarations) using `do {}`

```
var getTotalSeatsL = pt -> do {
  var pn = pt[-3 to -1] as Number
  ---
  pn
}
```

A closure is a construct that allows for the declaration of variables, functions, etc with a localized scope. The `---` serve the same purpose like the `---` we see in other DW expressions, they are section separators used to separate the declarations and the expression.

25. Add the conditional to the function

```
var getTotalSeatsL = pt -> do {
  var pn = pt[-3 to -1] as Number
  ---
  if (pn == 737 or pn == 707 or pn ==727) 150 else 300
}
```

You can use either one of these two functions to calculate the total seats; however, if you would like to use features such as function overloading you MUST stick with the fun.

Adjust price for currency

26. Create an object that contains currency exchange rates

```
var xes = {
  USD: 1.0,
  EUR: 0.9,
  GBP: 0.8,
  CAD: 1.3,
  AUD: 1.5,
  MXN: 25,
  INR: 72
}
```

We hard-code these currencies because we are within the confines of training. We can easily fetch these currencies dynamically from any data source and generate the map.

27. Create a function to calculate the price for a currency

```
var adjustFor = (p,c) -> p * xes[c]
```

28. Apply the function in prefix syntax

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price,"EUR")
}
```

29. Apply the function in infix syntax

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price,"EUR"),
  priceGBP: $.price adjustFor "GBP"
}
```

Functions with exactly two arguments get this infix application support! In fact, infix function application is encouraged because (1) it is natural in its application, (2) no need to use excessive parenthesis, and (3) allows for a more natural application of expression chains.

WT 2-2 Reuse DataWeave transformations

Store DW code in a file

1. Switch to the XML view of your file
2. Navigate under the mod2–functions flow and illustrate how the code is inline
3. Switch back to the graphical view (aka Message Flow)

4. Go to the properties of the DW processor under the mod2-functions
5. Click the Edit current target button (pencil icon)
6. Click the radio button File and type dw/transforms/mod2/functions in the text field to the right
7. Click OK *From the point of view of the DW properties UI nothing has changed. Nonetheless, with this action we have stored the DW code inside a new file under src/main/resources/dw/transforms/mod2 named functions.dwl*

Reuse the DW code from the file

8. Create a new flow named mod2-reuse
9. Drop a DW to the process area of the flow
10. Switch to the XML view
11. Locate the DW you just created
12. Remove the CDATA tag

```
<![CDATA[%dw 2.0 output application/java --- {}]]>
```

13. Introduce the / closing to the opening `<ee:set-payload />` tag, this should automatically remove the explicit closing tag
14. Add the attribute `resource` to the `<ee:set-payload />` tag

```
<ee:set-payload resource="dw/transforms/mod2/functions.dwl" />
```

This is the only way you could reuse the full transformation, i.e. by modifying the XML. Had you gone inside the UI and attempt to reuse the file, you would be overwriting it! That pencil button is a "one way trip", only there to store the file not reference it.

15. Switch back to the graphical view
16. Open the properties of the the DW processor under the `mod2-reuse`
17. Turn on the Preview *There is an issue that indicates that there is no metadata identifying what the payload is. You can fix this issue by just setting the metadata again. This issue is displayed because of DataSense. This issue is only visible when in Studio, if we start the server and deploy our app, DataSense is never in play.*
18. Set the input payload metadata to `flights_xml`
19. Turn on the Preview and validate you see the result

WT 2-3 Create and use DataWeave modules

Create a DW module

1. Create a new folder(s) under `src/main/resources`
2. In the text field type `dw/modules`
3. Create a new file under `dw.modules` and name it `Currency.dwl`
4. Type on line 1

```
%dw 2.0
```

DW modules can only contain declarations. Declarations such as variables, function, types, etc.

5. Navigate back to the DW processor under `mod2-reuse`
6. Copy the `xes` variable and the `adjustFor` function
7. Paste to `Currency.dwl` under line 1 and save

Use the module

8. Go back to the DW processor under `mod2-reuse`
9. Use the module by fully qualifying the function to adjust the price for the CAD currency

```
payload..* return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD"
}
```

10. Import the new module below the output directive

```
import dw::modules::Currency
```

11. Use the module again this time by taking advantage of the `import` to adjust the price for AUD

```
payload..* return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD"
}
```

12. Import the module again and provide an an alias to the module

```
import dw::modules::Currency as Curr
```

13. Use the module through the `Curr` alias next

```
payload..* return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD",
  priceMXN: $.price Curr::adjustFor "MXN"
}
```

14. Import all declarations to the current namespace

```
import * from dw::modules::Currency
```

15. Use directly the `adjustFor` function

```
payload..* return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD",
  priceMXN: $.price Curr::adjustFor "MXN",
  priceINR: $.price adjustFor "INR"
}
```

There is also an inline version of the `adjustFor` function which takes precedence. As such we are not using the function provided by the module. We can very easily verify by changing the body of the inline function, just change the body to 1000.

16. Modify the last `import` to selectively import declarations and provide them with aliases

```
import adjustFor as adj4 from dw::modules::Currency
```

17. Use the `adj4` alias

```
payload..* return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD",
  priceMXN: $.price Curr::adjustFor "MXN",
  priceINR: $.price adj4 "INR"
}
```

WT 2-4 Optional: Calculate KPIs using the Arrays Module

Inspect the Arrays module in the Documentation

1. Navigate to the Arrays module page in the documentation
2. Visit the `countBy` function documentation and understand the signature
3. Visit the `every` function documentation and understand the signature
4. Visit the `some` function documentation and understand the signature
5. Visit the `sumBy` function documentation and understand the signature

Add the KPI section

6. Add an object and set the record of flights to the object's `data` field

```
{
  data: payload..* return map {
    ($ - "planeType"),
    planeType: $.planeType replace /Boing/ with "Boeing",
    totalSeats: getPlaneTypeL($.planeType),
    priceEUR: adjustFor($.price ,"EUR"),
    priceGBP: $.price adjustFor "GBP",
    priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
    priceAUD: $.price Currency::adjustFor "AUD",
    priceMXN: $.price Curr::adjustFor "MXN",
    priceINR: $.price adj4 "INR"
  }

}
```

7. Add an `kpi` field above the `data` field and assign an empty object to it

```
{
  kpi: {

  },
  data: payload..* return map {
```

```
    ($ - "planeType"),
    planeType: $.planeType replace /Boing/ with "Boeing",
    totalSeats: getPlaneTypeL($.planeType),
    priceEUR: adjustFor($.price ,"EUR"),
    priceGBP: $.price adjustFor "GBP",
    priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
    priceAUD: $.price Currency::adjustFor "AUD",
    priceMXN: $.price Curr::adjustFor "MXN",
    priceINR: $.price adj4 "INR"
  }
}
```

We will use this new field to store the calculated KPIs in subsequent steps of this WT.

Assert all flights are Delta operated

8. Import the `Arrays` module

```
import every ,some ,countBy ,sumBy from dw::core::Arrays
```

9. Use the `every` function to assert whether all flights are Delta operated

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta"
},
```

You may want to refactor your code so that you execute `payload..*return` once. You already have seen how to do it, consider it a bonus exercise.

Assert the existence of full flights

10. Use the `some` to assert whether there are any full flights

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats == 0)
},
```

The result is `false`, i.e. there are no full flights. However, the second to last flight is a full flight because the `emptySeats` tag is set to 0. The problem lies with the comparison–we are comparing a string with a number. You can either use a type cast to `$.emptySeats as Number == 0` or the equality operator `~=` that autocasts its arguments.

11. Use the operator `~=` to fix the `Boolean` condition

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats ~= 0)
},
```

Calculate the number of full flights

12. Calculate the number of full flights

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats ~= 0),
  fullFlightsCount: payload..*return countBy ($.emptySeats ~= 0)
},
```

Sum the total number of empty seats across all flights

13. Sum up all the `emptySeats` across all flights

```
kpi: {
  allDeltaFlights: payload..* return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..* return some ($.emptySeats ~= 0),
  fullFlightsCount: payload..* return countBy ($.emptySeats ~= 0),
  totalEmptySeats: payload..* return sumBy $.emptySeats
},
```

Module 3

Defensive programming

WT 3-1 Function Overloading

Create a new flow

1. Create a new Mule Configuration file and name it `mod3`
2. Create a new flow named `mod3-data-matcher`
3. Drop a DW to the process area of the flow
4. Turn on the preview
5. Switch to the Source Only view
6. Change the output to `application/dw`

Nullity and function signatures

7. Explore nullity

```
%dw 2.0
output application/dw
---
typeOf(null)
```

By looking at the preview we can quickly realize that DW's typing system separates nullity from all other types. The benefit of such separation is that you can catch a good chunk of nullity errors at compile time.

8. Replace the expression with just `sizeOf`
9. Examine the signature in the preview

```
(arg0:Array | Object | Binary | String) -> ???
```

`sizeOf` expects a single argument. This argument can be one of listed types separated by a pipe symbol (|). This | indicates that this function has been overloaded.
Only through `application/dw` can we see the signature of functions.

10. Replace the expression with just `map`
11. Examine the signature in the preview

```
(arg0:Array | Null, arg1:Function | Function) -> ???
```

`map` is also an overloaded function, but this time it is a function that takes two arguments. If we were to reconstruct these overloaded functions defining `map` then we can tell (1) that there are two overloaded functions because we have a single pipe separating the types of the arguments and (2) the arguments of these overloaded functions will look as follows by using the position of the corresponding arguments:

- ***map**(**Array**,Function) – i.e. `Array` will match with the first `Function`*
- ***map**(**Null**, Function) – i.e. `Null` will match with the second `Function`*

Overload a function

12. Create a function that accepts the `Null` type

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
---
dataMatcher(null)
```

The : separates the name of the argument with the type of the argument. The same type of syntax can be used to specify types when needed.
You have created a function that takes only null values.

13. Pass an empty array instead

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
---
dataMatcher([])
```

You are now getting errors because our function is not defined to accept arrays.

14. Overload the function to also accept arrays

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
fun dataMatcher(a: Array) = "Array found"
---
dataMatcher([])
```

*Only a function declared through **fun** can be overloaded–in other words you cannot overload λ-expressions*

15. Display the signature of the `dataMatcher` function

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
fun dataMatcher(a: Array) = "Array found"
---
dataMatcher
```

16. Examine the signature

```
(arg0:Null | Array) -> ???
```

The signature clearly show the types the function accepts.

17. Overload the function for **Object**, **Number**, and **String**

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
fun dataMatcher(a: Array) = "Array found"
fun dataMatcher(o: Object) = "Object found"
fun dataMatcher(n: Number) = "Number found"
fun dataMatcher(s: String) = "String found"
---
dataMatcher
```

18. Examine the signature one last time

```
(arg0:Null | Array | Object | Number | String) -> ???
```

WT 3-2 Pattern Matching

Matching literals

1. Stay under the `mod3-data-matcher` flow
2. Delete the current expression
3. Use the `match` operator to match `Null` values

```
null match {
  case null -> "Null found"
}
```

This is the `match` operator, not to be confused with the `match` function.

4. Match the number 10

```
10 match {
  case null -> "Null found"
  case 10 -> "Ten"
}
```

Cases are evaluated in order of appearance, ensure more general cases are below the more specific ones.

5. Match the string "ABC"

```
"ABC" match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
}
```

6. Match the empty array

```
[] match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
}
```

7. Match the empty object

```
{} match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
  case {} -> "Empty object found"
}
```

8. Add elements to the object

```
{a: 1} match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
  case {} -> "Empty object found"
}
```

We are getting errors because there is no case matching this expression.
There is no way to match inner data of objects and arrays directly in a case–i.e. adding inner data at the case level will only result in syntax errors.

9. Add the default case for when there is no case matching your data

```
{a: 1} match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
  case {} -> "Empty object found"
  else -> {message: "No match", data: $}
}
```

In the body of the `else` case we are back-referencing our data with a single $. We can back-reference our data in all other cases in the same way. Further along we shall see how to define our own placeholder to back-reference our data so that we don't depend on a single $

Matching types

Lets use pattern matching to replicate function overloading from before.

10. Remove or comment-out the expression
11. Match the `Null` type

```
null match {
  case is Null -> "Null found"
  else -> $
}
```

*The `is` operator is used in the first case to match all values of type **Null**.*
We pro-actively set the default case to just evaluate back to the expression we are matching.

12. Match **Number**, **String**, **Object**, and **Array** types

```
[] match {
  case is Null -> {message: "Null found", data: $}
  case is Number -> {message: "Number found", data: $}
  case is String -> {message: "String found", data: $}
  case is Object -> {message: "Object found", data: $}
  case is Array -> {message: "Array found", data: $}
  else -> {message: "No match", data: $}
}
```

We are back-referencing the values we are matching using the $ for all our cases.

Matching by any condition

13. Replace the $ with a placeholder of our own choosing

```
[1] match {
  case n if (n is Null) -> {message: "Null found", data: n}
  case n if (n is Number) -> {message: "Number found", data: n}
  case s if (s is String) -> {message: "String found", data: s}
  case o if (o is Object) -> {message: "Object found", data: o}
  case a if (a is Array) -> {message: "Array found", data: a}
  else other -> {message: "No match", data: other}
}
```

These new placeholders are arbitrary, they can be anything of your choosing, from single letter to words–very similar to declaring variables.
Using this new syntax to define our placeholders and conditionals opens an unlimited set of Boolean expressions that could be defined on a per case basis.

14. Use pattern matching to distinguish the empty array, the array that has less than or equal to 100 elements, and the array that has larger than 100 elements

```
(0 to 200) match {
  case [] -> "Empty Array"
  case a if (a is Array and sizeOf(a) <= 100) -> "Non-empty array with less or equal to
  else a -> "Non-empty array with larger than 100 elements"
}
```

For this use case we combine all of the above:

- *Literal pattern matching for the empty array*
- *Conditional pattern matching on the type and size*
- *The default case for all other arrays*

WT 3-3 Error Handling

Create a new flow

1. Create a new Mule Configuration file and name it `mod3`
2. Create a new flow named `mod3-errors`
3. Drop a DW to the process area of the flow
4. Turn on the preview
5. Switch to the Source Only view
6. Change the output to `application/dw`

The error 10 / 0

7. Introduce an error

```
%dw 2.0
output application/dw
---
10 / 0
```

When the evaluation of DW expressions result in errors the whole expression errors out–there are no partial results
Imagine scenarios where we are working with sequences or records where some of these records our malformed, DW expressions will not provide you with partial results, instead the full expression will error out. Error handling will allow for such partial results.

The `dw::Runtime::try` function

8. Navigate to the Dataweave documentation
9. Examine the signature of the dw::Runtime:: try function found in the Dataweave Reference section
The `dw::Runtime::try` function takes a single argument. The argument is a λ-expression that takes no arguments itself.
The body of this λ-expression is the expression that we want to evaluate for errors.
This λ-expression is also referenced as a delegate function because through it we are delegating the execution of an expression to `dw::Runtime::try` function. The latter will determine whether our expression has errors or not.
The return value is a `TryResult<T>` structure. Thankfully, we have documentation that describes this structure.
10. Test the `10 / 0` expression

```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 0)
```

11. Examine the result

```
{
  success: false,
  error: {
    kind: "DivisionByZeroException",
    message: "Division by zero",
    location: "\n4| dw::Runtime::try(() -> 10 / 0)\n                                ^^^^^^",
    stack: [
      "main (anonymous:4:24)"
    ]
  }
}
```

We no longer have errors preventing us from getting results in the preview. Instead we are looking at a `TryResult` structure where the `success` field is set to `false` indicating that our expression has errors. Finally the error's description can be seen in the `error` field.

12. Test the `10 / 2` expression

```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 2)
```

13. Examine the result

```
{
  success: true,
  result: 5.0
}
```

Similarly we have `success` set to `true` but instead of `error` we have the `result` field set to the value our expression evaluates to.

The `guard` function

14. Chain the `match` operator

```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 0) match {
  case tr if (tr.success) -> tr.result
  else tr -> tr.error.message
}
```

When `success` is set to `true` just return the `result` otherwise arbitrarily just return the `error.message`

15. Extrapolate to a function that guards against errors

```
var guard = (fn) -> dw::Runtime::try( fn ) match {
  case tr if (tr.success) -> tr.result
  else tr -> tr.error.message
}
```

There is no need for a `guard` function. There are additional error handling functions that we shall see further along the material that provide for complete and flexible error handling. Nonetheless, going through the refactoring of our code into a function allows us to take a peek on functional programming.

16. Test the `guard` function with a correct expression

```
guard( () -> 10 / 2 )
```

17. Test the `guard` function with an erroneous expression

```
guard( () -> 10 / 0 )
```

WT 3-4 Optional: Partial Results

Let's see how we can get partial results now that we know how to capture errors. How many times you had your DW code fail because one or very few records were malformed? We will replicate such a scenario by creating an array containing the string representation of dates and then casting them into a date type. One of our dates will be malformed

List of String Dates

1. Stay in the `mod3-errors` flow
2. Create an array of string dates and assigned it in a variable

```
var dates = [
  "2020/01/01",
  "20100101",
  "2000/01/01"
]
```

It is the second date that is malformed and it is the cause of failure of our transformation.

3. Remove or comment out the current expression
4. Iterate over dates and cast into a `Date` type

```
dates map (
  $ as Date {format: "yyyy/MM/dd"}
)
```

There is an error thrown telling us that the second-string date cannot be casted.

Using `guard` and obtaining partial results

5. Delegate the execution of the type-casting expression to the `guard` function

```
dates map (
  guard( () -> $ as Date {format: "yyyy/MM/dd"} )
)
```

6. Finally `filter` for `Date` types and get your partial results while ignoring the rest

```
dates map (
  guard( () -> $ as Date {format: "yyyy/MM/dd"} )
)
filter ($ is Date)
```

7. We can expand on this solution such that we also obtain the errors

```
do {
  var parsedDates = dates map (
  guard( () -> $ as Date {format: "yyyy/MM/dd"} )
  )
  ---
  {
    success: parsedDates filter ($ is Date),
    failure: parsedDates filter  not ($ is Date)
  }
}
```

Note to instructors: This is a good exercise to have students work on their own for "bonus points", after all we have already covered all topics they need to know to get it done.

Module 4

Flights and Airports

In this module we work on the main use-case of the class. This use-case entails combining flights with airports and tranforming the data such that are communicated with external systems. The tasks we shall perform are as follows:

- Dynamically rename fields
- Combine data from two sources
- Explore in more detail functional programming
- Optimize your code in the absense of profiling tools
- Reorder objects to meet legacy system criteria
- Traverse and transform any type of structured data

In fact, the remainder of the class makes use of this use-case. We shall still apply unit-testing to solve smaller issues before we integrate the unit-tested expressions in our main transformation.

WT 4-1 Change field names

Create a new flow

1. Create a new Mule Configuration file and name it `mod4`
2. Create a new flow named `mod4-flights-airports`
3. Drop a DW to the process area of the flow
4. Turn on the preview
5. Switch to the Source Only view
6. Change the output to `application/dw`

Create the map

7. ...

The `mapObject` function

8. ...

Change the field names

9. ...

WT 4-2 Combine flights and airports

Explore the CSV file

1. ...

Parse the CSV airports

2. ...

Inject the airport to each flight

3. ...

Functions as values

4. ...

Curried functions

5. ...

A more efficient transformation

6. ...

Clean up the data

7. ...

WT 4-3 Reording fields

Create a new flow

1. ...

The `pluck` function

2. ...

The `pluck` function

3. ...

The `reduce` function

4. ...

Re-ordering fields

5. ...

Integrate field re-ordering to our main use-case

6. ...

Module 5

Recursion