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.
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 a best-practice one. 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`) 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 façade

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.
        i. 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 missmatch errors.

Conditional expressions

14. `if then else` conditional
    (a) `if (true) 1 else 0`
    (b) `if (false) 1 else 0`
15. Nullity conditional
    (a) lstinlinenull default "Other value"
    (b) lstinline"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` $+1$*map*`-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 argments 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

Variables, Functions, Modules

WT 2-1 Organize DataWeave code with variables and functions

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.

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

12. 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 {
  ($),
  totalSeats: getTotalSeats($.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.

13. 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 {
  ($),
  totalSeats: getTotalSeats($.planeType)
}
```

14. Enclose the confition in an `if` expression

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

15. 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
```

16. 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
```

Parenthesization 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.*

17. 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 λ expression

18. 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. Additionaly, the body of this function evaluates into the argument we passed–this encourages unit testing.

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

```
payload..*return map {
  ($),
  totalSeats: getTotalSeatsL($.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.

20. 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.

21. 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.*

22. 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`*. You can*

Adjust price for currency

23.

WT 2-2 Reuse DataWeave transformations

WT 2-3 Create and use DataWeave modules

Module 3

Defensive programming

Module 4

Operating on Arrays and Objects

Module 5

The Arrays and Objects Modules

Module 6

Flights and Airports

Module 7

Recursion