Module 4: Use Case—Flights and Airports

Review: Use Case – Flights and Airports

As a developer I want to combine two sets of data, (1) A list of JSON flights and (2) a list of CSV airports, into a JSON data structure so that I communicate them with a third party legacy system.

The JSON data structure contains:

- dynamically renaming fields based upon a provided map
- destination airport details injected per flight
- reordering the fields of each object to meet the legacy system requirements.
- optimizing the performance of our algorithm
- fixes to bad data

Goal

MuleSoft

Inputs

```
[
  {
    "airlineName": "Delta",
    "code": "A1B2C3",
    "departureDate": "2018/03/20",
    "destination": "SFO",
    "emptySeats": "40",
    "origin": "MUA",
    "planeType": "Boing 737",
    "price": "400.0"
  },
  …
]
```

```
openFlightsAirportId,airportName,city,country,IATA
492,London Luton Airport,London,UK,LTN
499,Jersey Airport,Jersey,Jersey,JER502,London
Gatwick Airport,London,UK,LGW
…
```

Expression

```
var fs2rn = {
        airlineName: "carrier",
        departureDate: "date",
        emptySeats: "seats",
        planeType: "plane"
}
var airports = readUrl(
        "classpath://airportInfoTiny.csv",
        "application/csv",
        {
        header: true,
        bodyStartLineNumber: 0,
        separator: ","
        }
        )
distinctBy $.IATA
groupBy $.IATA
---
payload map (
        $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
        ($),
        airport: airports[$.destination]
}
map (
        $ reorder (8 to 0)
)
```

Output

```
[
 {
  AIRPORT: [
   {
    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"
   }
  ],
  PRICE: 400.0,
  PLANE: "boing 737",
  ORIGIN: "mua",
  SEATS: 40,
  DESTINATION: "sfo",
  DATE: |2018-03-20| as Date {format: "yyyy/MM/dd"},
  CODE: "a1b2c3",
  CARRIER: "delta"
 },
 …
]
```

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

- Dynamically rename fields
- Read and parse CSV files
- Iterate, search, and combine data
- Discuss and practice the functional programming paradigm
- Identify and correct slow portions of the DataWeave expression
- Reorder objects to satisfy legacy system prerequisites

Iterating Objects

The `mapObject` Function

- `mapObject(Object,(v:Any,k:Key,i:Number) -> Object):Object`
  - Iterates over individual key, value pairs of the object in order
  - Expects two arguments
    - An object to iterate over
    - A lambda-expression to be applied against every single key, value pair
      - This lambda-expression returns an object
  - Returns an object
    - This objects is the concatenation of all objects returned by the lambda-expression

Walkthrough 4-1: Change field names

- Use the `mapObject` function
- Understand how to evaluate fields through DW expressions
- Dynamically change fields based upon a provided map

Reading and Parsing files

The `readUrl` Function

- `readUrl(String | Binary, String, Object)`
  - Takes three arguments
    - Either a url-type of a classpath indicating the location of the file or binary data
      - This is the only mandatory argument
    - A string containing a MIME type indicating the type of data the file contains
      - The default value is application/dw
    - An object containing reader properties that determine how the file will be parsed
      - E.g. {header: true,bodyStartLineNumber: 0,separator: ","}
      - Every format has its own set of reader and writer properties
      - All the supported formats along with the reader and writer properties can be found here.

Functional Programming

Functional programming paradigm

- Functions are at the core of functional programming
  - Composing and applying functions is what a functional program looks like
- Functions are first-class citizens
  - They are values
    - They can be assigned to variables
      - E.g. `var id = (e) -> e`
    - They can be passed as arguments
      - E.g. `[3,1,2] map (e,i) -> e+i`
    - They can returned as values from other functions
      - E.g.

```
var add = (n1) -> (n2) -> n1 + n2
var add10 = add(10)
```

Curried functions

- Defined by Haskell Brooks Curry
- The concept of curring
  - is the technique of converting a function that takes multiple arguments into a sequence of functions that each take a single argument

```
var add = (n1,n2) -> n1 + n2
var addC = (n1) -> (n2) -> n1 + n2
var add10 = addC(10)
var twenty = acc10(10)
Var thirty = addC(10)(20)
```

  - is related to partial application of functions
    - Real-world applications
      - Callback functions that their inputs are not available at the same time, think UI
      - Generate new functions out of a single definition, i.e. function factories
        - This could apply to DW.
    - Curried functions is a subset of functions supporting partial application

```
var addThree = (n1,n2) -> (n3) -> n1 + n2 + n3
```

Performance Tuning

Gauging performance

- Logger processor
- `log()` DataWeave function
- No profilers
- No debuggers
- Logs provide with discrete executions of preset data
  - There is nothing to speak of when it comes to general executions
  - Accepting all kind of different inputs
- What is needed is an abstract way of thinking, calculating performance, and identifying bottlenecks

Complexity of Algorithms

- Here’s comes old-school Big-O notation to the rescue
- Big-O is concerned with abstract values
  - Constants are irrelevant
- Big-O is a polynomial
  - E.g $3*N + N*M + M^2$
  - Drawing an X and Y graph where Y is the data set, and X is the time it takes for your algorithm to complete is telling
  - Do you see linear or exponential growth
- In DW you must identify the iteration and the data iterating over
  - Abstract the data with variables and start building the polynomial
  - Identify parts of the polynomial that grow faster than others
  - These parts are your bottlenecks and should be optimized

Walkthrough 4-2: Combine flights and airports

- Read and parse data from a CSV file
- Combine each flight with the corresponding airport
- Discuss and experiment with functional programming
- Calculate the Big-O for your algorithm
- Identify and fix bottlenecks

Additional functions to iterate data

The `pluck` function

- `pluck(Object,(v:Any,k:Key,i:Number) -> T) : Array<T>`
  - Iterates over individual key, value pairs of the object, in order
  - Expects two arguments
    - An object to iterate over
    - A lambda-expression to be applied against every single key, value pair
      - This lambda-expression returns a value
  - Returns an array
    - The array contains result of applying the lambda-expression to each key, value pair

The `reduce` function

- `reduce(Array<T>, (e: T, acc: R) -> R) : R`
  - Iterates of the array
  - Expects two arguments
    - The array to iterate over
    - A lambda-expression that is applied for each one of the elements in the array. This lambda-expression expects two arguments
      - The current element from the input array to be processed
      - The accumulator (acc for short) that determines how the data are accumulated
  - Returns the value of the lambda-expression of the last iteration
- Lets trace `[3,1,2] reduce (e, acc=0) -> acc+e`
  - 1st iteration: `(3,0) -> 0+3`
  - 2nd iteration: `(1,3) -> 3+1`
  - 3rd iteration: `(2,4) -> 4+2`

Walkthrough 4-3: Reordering fields

- Use the `pluck` function
- Use the `reduce` function
- Build a reorder an object function using `map`
- Build a more efficient version of reorder using `reduce`
- Apply the reorder function to the flights and airports use-case

Summary

Summary

- `mapObject` will allow for iterating and manipulating objects resulting into new objects
- `readUrl` can read and parse files containing structured data
- Functional programming is the act of composing and applying functions where functions are first-class citizens
- `pluck` iterates over objects but generates arrays as output
- Applying Big-O notation is currently the best way to evaluate the performance of your algorithms and to identify bottlenecks
- `reduce` is a general use function that accepts arrays and is able to generate any types of data