%dw 2.0

%dw 2.0

output application/xml

---

trains:{

"ticket": "wardha",

"price": 100

}

=

%dw 2.0

output application/xml

---

trains:{

"ticket": "wardha",

"price": 100

}

%dw 2.0

output json

---

typeOf("Hello")

="String"

%dw 2.0

output json

---

typeOf(1)

="Number"

output json

---

typeOf({})

="Object"

DataWeave allows repeated keys on Objects as well. This may seem

odd, but the support for this makes sense when we consider XML:

DataWeave [selectors](https://docs.mulesoft.com/mule-runtime/latest/dataweave-selectors) allow navigating any combination of Objects and Arrays to get to the data you need. Throughout this tutorial we will review some of the most important ones:

* Single-value selector : .
* Index selector: [n]
* Range selector: [n to m]
* Multi-value selector: .\*
* Descendants selector: ..

Single Value Selector

The single-value selector (.) allows you to lookup Object values by their key. Here’s an example:

#### Input:

{

"name": "Ana",

"age": 29,

"dynamicKey": "age"

}

#### DW Script:

%dw 2.0

output json

---

{

fixed: payload.age,

dynamic: payload[payload.dynamicKey]

}

#### Output:

{

"fixed": 29,

"dynamic": 29

}

Input:

{

"name": "Ana",

"age": 29

}

DW Script:

%dw 2.0

output json

---

payload.age

Output:

29

%dw 2.0

output json

---

payload[1]

#### Output:

"qa"

# Index Selector

Notice that by using 1 as the index, the script returned the second item in the Array. This is because Arrays in DataWeave are zero-indexed; the item in the first position of the Array has an index of 0, the second has an index of 1, and so on.

Just like Objects can be nested, so can Arrays, and you can retrieve nested Array items in the same way you do with the single-value selector, by stringing together index selectors.

There’s another important feature that should be noted. If you use positive numbers for the index, DataWeave will start selecting from the beginning of the Array, but if you use a negative number for the index, DataWeave will start selecting from the end of the Array. Since 0 is already reserved as the first element in the Array, and there is no such thing as -0, DataWeave starts indexing the last item of the Array from -1.

Input:

["prod", ["qa", "dev"] , ["to", 2, 3]]

%dw 2.0

output application/json

---

payload[-1][-2] = 2

payload[1][1] = dev

payload[2][0] = to

## Exercise

Given the following input, get to "qa".

### Input:

["prod", ["qa", "dev"]]

Try to get "qa" but using only negative indexes.

%dw 2.0

output json

---

payload[-1][0]

=qa

# Range Selector

If you need multiple sequential values from an Array, DataWeave allows you to select a range of values with the range selector ([n to m]). Instead of returning a single value like the index selector does, it will return an Array of values:

#### Input:

["prod", "qa", "dev"]

#### DW Script:

%dw 2.0

output json

---

payload[0 to 1]

#### Output:

["prod", "qa"]

Multi Value Selector

The multi-value selector (.\*) returns an Array containing any value that matches the key. The values returned are dependent on the key that’s passed in. The multi-value selector works on both Arrays and Objects, but in different ways.

Let’s check out Object first:

Input:

{

"name": "Emilia",

"name": "Isobel",

"name": "Euphemia",

"name": "Rose",

"surname": "Clarke"

}

DW Script:

%dw 2.0

output json

%dw 2.0

output json

---

payload.movies.\*title

=[

"The Terminator",

"Titanic",

"Avatar"

]

Or(simple)

payload.movies.title

=

"The Terminator"

---

payload.\*name

Output:

["Emilia", "Isobel", "Euphemia", "Rose"]

#### Input:

<movies>

<title>The Terminator</title>

<title>Titanic</title>

<title>Avatar</title>

<director>James Cameron</director>

</movies>



Descendants Selector

The descendants selector (..) is the perfect tool to use when you need the values for a certain key no matter where they appear in a piece of data. Here’s an example:

Input:

{

"echo": {"value": "Hello there!"},

"sequence": [

{

"echo": "Getting details...",

"try": {

"curl": "somelocation.com",

"echo": "Success!"

}

},

{

"grep": "Success"

}

]

}

DW Script:

%dw 2.0

output json

---

payload..echo

Output:

[{ "value": "Hello there!"}, "Getting details...", "Success!"]

#### Input:

{

"id": 1,

"id": 11,

"secondLevel": {

"id": 2,

"id": 22,

"thirdLevel": {

"id": 3,

"id": 33

}

}

}

{

"descendant": [

1,

2,

3

],

"multivalue": [

1,

11

],

"multivalueSecondLevel": [

2,

22

],

"multivalueThirdLevel": [

3,

33

],

"allTheIds": [

1,

11,

2,

22,

3,

33

]

}

%dw 2.0

output json

---

{

"descendant": payload..id,

"multivalue": payload.\*id,

"multivalueSecondLevel": payload.secondLevel.\*id,

"multivalueThirdLevel": payload.secondLevel.thirdLevel.\*id,

"allTheIds" : payload..\*id

}

[

[

"Max the Mule",

"Max’s house"

],

"1234 Mule Street",

false

]

=> [

%dw 2.0

output json

var data = {

"name": "Max the Mule",

"locations": [

{

"name": "Max’s house",

"address": "1234 Mule Street",

"state": "CA"

}

]

}

---

[

data..\*name,

data.locations[0][1],

data.locations.state == 'fa'

]

If Else

If/else expressions allow you to make decisions using logical operators and branch as a result.

if (<criteria\_expression>) <return\_if\_true> else <return\_if\_false>

#### Input:

{

"price": 150.00

}

%dw 2.0

output json

var action = if (payload.price > 140) "sell" else "hold"

---

{

price : payload.price,

action : action

}

{

"price": 150.00,

"action": "sell"

}

Input:

{

"price": 120.00

}

DW Script:

%dw 2.0

output json

var action = if (payload.price < 100) "buy" else "hold"

---

{

price : payload.price,

action : action

}

Output:

{

"price": 120.00,

"action": "hold"

}

Or

Named Functions

We create functions in the declarations section of the script using the fun keyword. This associates a set of functionality with a name. Here’s the basic syntax:

DW Script:

%dw 2.0

output json

fun add(n, m) = n + m

---

add(1,2)

= 3

fun <function\_name>(<arg1>, <arg2>, …, <argN>) = <body>

It’s usually good form to put the body on a new line and indent.

You can call functions with the following syntax:

<function\_name>(<arg1>, <arg2>, …, <argN>) =>

%dw 2.0

output json

---

isOdd(payload.value mod 2)

=true

%dw 2.0

output application/json

fun new(p,q) =

p / q

---

new(108,12)

=9

It is often useful to create a *scope* for functions, where we can declare variables and even more functions. Scopes are created using the do statement and work by making everything defined on its header available for use on its body but not beyond that limit.

In the example below, the diff function uses a scope to define two variables available only to the function itself:

DW Script:

%dw 2.0

output json

fun diff(n) = do {

var start = n[0]

var end = n[-1]

---

end - start

}

---

diff([1990, 1995, 2002, 2008, 2021])

Output:

31

Lambdas

DataWeave provides multiple ways to create functions. Just like we have named functions, we have functions without names, called lambdas. A lambda is a value in DataWeave, just like a String, an Object, and a Boolean. The syntax for a lambda is like so:

(<arg1>, <arg2>, …, <argN>) -> body

Recall that in order to call functions you need to use the following syntax:

<function\_name>(<arg1>, <arg2>, …, <argN>)

But lambdas don’t have names, that’s the whole point! In order to force a lambda to execute, we surround it in parentheses and append () to the end:

%dw 2.0

output json

var add = (n, m) -> n + m

---

add(2, 3)

Output

5

But that’s not very useful either, we already have a nicer syntax for the same thing with the fun keyword.

%dw 2.0

output json

---

(() -> 2 + 3)()

=5

Functions as Values

The usefulness of lambdas becomes apparent when we combine two ideas:

* Lambdas are values just like Strings, Objects, and Booleans
* Values can be passed to functions as arguments, as well as returned from functions.

In other words, lambdas become useful when you want to pass functions as arguments to other functions, or return a function from a function. A function that does this is referred to as a higher-order function (HOF).

# Filter:-

## filter(Array<T>, (item: T, index: Number) -> Boolean): Array<T>

Iterates over an array and applies an expression that returns matching values.

[9,2,3,4,5] filter (value, index) -> (value > 2)

**Output :-** [9,3,4,5]

The filter function takes two arguments, an Array and a Lambda. In situations like these, it’s important to specify what the lambda should do as well. In the case of filter, the lambda should take in two arguments: an item in the Array, and the index of that particular item.

%dw 2.0

output json

---

filter(payload, (n, idx) -> (n > 3))

= [4,66,776,10,5 ]

Infix Notation

So far, we’ve been calling filter using prefix notation. With prefix notation you put the function name before the arguments. If a function takes two arguments, like filter does, DataWeave allows you to call it with infix notation. Infix notation has the following syntax:

<arg1> <function\_name> <arg2>

This syntax is preferred for nearly every function that takes a lambda as its second argument

%dw 2.0

output json

var numbers = (1 to 5)

---

numbers

filter ((n, idx) -> (n mod 2) == 1)

filter ((n, idx) -> (n > 3))

Output:

[5]

%dw 2.0

output json

---

payload filter((n, idx) -> (n mod 5) == 0)

=[5,30,45,1040,105]

$, $$, $$$ Syntax

For functions that DataWeave provides, you can represent the first, second, and third arguments of the passed lambda as $, $$, and $$$, respectively. When you do this, you do not need to specify the arguments of the lambda when you pass it to the function.

%dw 2.0

output json

var numbers = (1 to 5)

---

numbers filter (($ mod 2) == 1)

The dollar-sign syntax gives us all the same functionality as when we reference something by its name. This means we can chain selectors and indexes right off of the dollar-sign in order to query data:

#### Input:

[

{

"name": "Jonas",

"family": "Kahnwald"

},

{

"name": "Mikke",

"family": "Nielsen"

},

{

"name": "Claudia",

"family": "Tiedemann"

},

{

"name": "Martha",

"family": "Nielsen"

},

{

"name": "Charlotte",

"family": "Doppler" }]

%dw 2.0

output json

---

payload filter ((n, index) -> n.name startsWith 'C') OR

payload filter ($.name startsWith 'C')

Output: [{

"name": "Claudia",

"family": "Tiedemann" },

{

"name": "Charlotte",

"family": "Doppler"

}]

The filter Function (and Function Type Signatures)

When dealing with functions, it’s important to know what kind of data is valid input, and what to expect as output. For example, we know the valid input to filter is an Array and a lambda, and it returns an Array. However, this isn’t quite descriptive enough because there is another function to be accounted for, the lambda. The lambda takes in two arguments, a single item of type Any, and a index of type Number. It returns a Boolean. We can use a syntax that is very close to DataWeave to define this:

filter(Array<Any>, ((Any, Number) -> Boolean)) : Array<Any>

The map Function

This function satisfies a very common use case in integration development: transforming every item in an Array to something else. Just like filter, map takes two parameters, an Array and a lambda, however, the lambda is structured differently than the one in filter. Here's the type definition for map:

map(Array<T>, ((T, Number) -> R)): Array<R>

There are two type variables in this definition, T, and R. T represents the type of items that the *input* Array contains. R represents the type of items that *output* Array contains. Since map’s job is to transform every item in an Array, it makes sense that the type of items in the input Array and type of items in the output Array are different. Knowing this, the lambda definition makes sense:

((T, Number) -> R)

The lambda’s job is to take in each item of type T from the input Array, as well as the index of that item, and return a new item that will be used in the output Array. Enough with the definitions, let’s check out a simple example that adds 1 to every value in an input array:

Input

[1,2,3,4,5]

%dw 2.0

output json

---

payload map (n, idx) -> {

value: n + 1,

index: idx

}

Input

[1,2,3,4,5]

DW Script

%dw 2.0

output json

---

payload map (n, idx) -> n + 1

Output

[2,3,4,5,6]

The reduce Function

The reduce function is about as close as we get to a general-purpose looping tool in DataWeave. It can be used to transform an Array to any other type. It can be used to perform the task of map, filter, distinctBy, groupBy, and other functions that take in Arrays. Here’s its function signature:

reduce(Array<T>, ((T, R) -> R)): R

The lambda can return anything at all, an Array, Object, String, etc. Whatever the type that is returned from the lambda is the same type that is returned from reduce. A common example for reduce is to show how you can sum all of the numbers in an Array. It is not a very useful example as the sum function does just that but it gives a small glimpse of what reduce

Use reduce to transform the input array into the object below. Keep in mind that the concatenation function ++ seen before for Strings also works between Objects as seen in the starting example, where you can also see how dynamic keys are defined in an Object.

Input

[

{"id": "1", "name": "dev"},

{"id": "2", "name": "test"},

{"id": "3", "name": "uat"},

{"id": "4", "name": "prod"}

]

%dw 2.0

output json

---

payload reduce ((item, accumulator= {}) -> accumulator ++ { (item.name):item.id } )

output:{

"dev": "1",

"test": "2",

"uat": "3",

"prod": "4"

}=> reduce() operate on arrays, but the concatenate operation ´++´ can operate on arrays, strings, objects, and several types of date combinations. Because I set the accumulator to an object and the second parameter of concatenation is another object, it does an object concatenation

#### Input

[1,2,3]

DW Script

%dw 2.0

output json

---

payload reduce (n, total) -> total + n

Output

6

#### Input

[1,2,3]

DW Script

%dw 2.0

output json

---

payload reduce (n, total = 100) -> total + n

Output

106