Mule 3

Dataweave 1.0

Reference :

<https://docs.mulesoft.com/mule-runtime/3.9/dataweave>

<https://docs.mulesoft.com/mule-runtime/3.9/dataweave-language-introduction#precedence-table>

DataWeave File Structure -.dwl

1. Header –defines directives
2. Body –describes the output structure

%dw 1.0

%output application/xml

---

{

user: {

name: payload.user\_name,

lastName: payload.user\_lastName

}

}

Dataweave Header

1. Version - %dw 1.0
2. Output type - %output application/json
3. Input type- %input payload application/xml
4. Namespace - %namespace ns0 <http://abc.com/service>
5. Constants - %var conversionRate=13.15
6. Functions - %var toUser= (user) ->{ fistName: user.name, lastName:user.lastName”}

**Define Constant Directive**

You can define a constant in the header, you can then reference it (or its child elements, if any exist) in the DataWeave body.

%dw 1.0

%var conversionRate=13.15

%output application/json

---

{

price\_dollars: payload.price,

price\_localCurrency: payload.price \* conversionRate

}

#### Define Function Directive

You can define a function in the header, you can then call it in any part of the DataWeave body, including arguments.

%dw 1.0

%var toUser = (user) -> {name: user.name, lastName: user.lastName}

%output application/json

---

{

user: toUser(payload)

}

### Body

The body contains the **expression** that generates the output structure. Regardless of the types of the input and output, the data model for the output is always described in the standard DataWeave language, and this model that the transform executes.

The data model of the produced output can consist of three different types of data:

1. Objects: Represented as collection of key value pairs
2. Arrays: Represented as a sequence of comma separated values
3. Simple literals

When you write your DataWeave file, you define an expression that generates one of the data types listed above.

### Simple Literal Types

Literals can be of the following types:

1. String : Double quoted ("Hello") or Single quoted ('Hello')
2. Boolean : Literals true or false
3. Number : Decimal and Integer values are supported (ex: 2.0)
4. Dates : IS0-8601 enclosed by "|" (ex:|2003-10-01T23:57:59Z|)
5. Regex : Regex expression enclosed by "/" (ex:/(\d+)-(\d+)/)

This is a String literal expression

### Arrays

Arrays are sequences of **expressions**.

[ 1, 2 + 2, 3 \* 3, $x ]

### Objects

These are represented as a comma separated sequence of key:value pairs surrounded by curly braces { }.

Transform

%dw 1.0

%output application/xml

---

myoutput:{

name : "Jill",

payload : payload.id + 3

}

Output

<?xml version="1.0" encoding="UTF-8"?>

<myoutput>

<name>Jill</name>

<payload>5</payload>

</myoutput>

Note that both the keys and the values may be **expressions**.

## DataWeave Canonical Model

As covered above, DataWeave uses three basic data types: Objects, Arrays, and Simple Types, the execution of a DataWeave transformation always produces one of these three types of data.

The body of every DataWeave transformation is a single expression that defines the structure and contents of one such element (which can be an Object, Array, or Simple Literal).

This expression can be built using any of the following elements:

* Objects
* Arrays
* Simple literals
* Variable and Constant references

Scoped Variable:

Variables in header – Global scope

Eg: %dw 1.0

%output **application/json**

**%var fi=1.3**

In Body –limited scope

Declare variables – using(<variable-name>=<Expression>)

Eg:

%dw 1.0

%output application/json

---

using (x=2) 3+x

using (y=2)[1,y,5]

**Scoped to object literal**

%dw 1.0

%output application/json

---

{

person: using (user=”Greg”, gender=”male” ){

name: user,

gender: gender

}

}

%dw 1.0

%output application/json

---

Entry: using (fname=”Ranasinghe”,lname=”Kulatunge”){

student: using (grade=”5d” , class=”lynx”){

grd: grade,

class: class

}

firstName: fname,

lastName: lname

}

**Attribute Selectors:**

<product id="1" type="tv">

<brand>Samsung</brand>

</product>

%dw 1.0

%output application/json

---

item: {

type: [payload.product.@type](mailto:payload.product.@type),

name: payload.product.brand,

attribs: payload.product.@

}

o/p

{

item: {

type: tv,

name: Samsung,

attributes: { **(1)**

id: 1,

type: tv

}

}

}

**Pluck Operator: (Use to map Object -> Array) returns Array**

payload.prices **pluck** $$ - retrieve all the keys

payload.prices **pluck** $ -retrieve all the values

%dw 1.0

%output **application/json**

%var conversionRate=3.156

---

priceList : payload.prices **pluck** (value,key)-> {

(key) : {

price: value

}

//(key) :value \* conversionRate as :number

//"sss":payload.prices.basic mapObject $

}

OR

priceList : payload.prices **pluck** {

($$) : {

price: $

}

//(key) :value \* conversionRate as :number

//"sss":payload.prices.basic mapObject $

}

i/p

====

<?xml version=*'1.0'* encoding=*'UTF-8'*?>

<prices>

<basic>9.99</basic>

<premium>53</premium>

<vip>398.99</vip>

</prices>

o/p

====

{

"priceList": [

{

"basic": {

"price": "9.99"

}

},

{

"premium": {

"price": "53"

}

},

{

"vip": {

"price": "398.99"

}

}

]

}

**Map function:**

Method-1

%dw 1.0

%output **application/json**

---

shirts: payload **map** {

size: **upper** $.size,

description:$.description,

count:$.count

}

$ selector - item

$$ selector- index

Method -2

%dw 1.0

%output **application/json**

---

shirts: payload **map** (item,index)->{

size: **upper** item.size,

description:item.description,

count:index

}

**Filter operator**

===========

%dw 1.0

%output **application/json**

---

shirts: payload **map** {

size: **upper** $.size,

description:$.description,

count:$.count

} **filter** $.count > 0 **and** $.count<100 **and** $.size =="M"

Filter 0< S.count<100

**groupBy operator:**

%dw 1.0

%output **application/json**

---

shirts: payload **map** {

size: **upper** $.size,

description:$.description,

count:$.count

} **filter** $.count > 0 **and** $.count<3000 **groupBy** $.size

Output:

{

"shirts": {

"M": [

{

"size": "M",

"description": "Connect 2016",

"count": 151

},

{

"size": "M",

"description": "RAML",

"count": 23

}

],

"XXL": [

{

"size": "XXL",

"description": "Connect 2016",

"count": 329

}

]

}

}

M2:

%dw 1.0

%output **application/json**

---

patents:payload.items **groupBy** $.author **pluck** {

owner: $$,

categories:$.type

}

i/p:

{

"items": [

{

"type" : "book",

"label" : "Java Curiosities",

"author" : "Luis Miguel",

"image": "http://www.gravatar.com/avatar/a133c7d8d9245d063b136732272ea77e",

"url": "http://javacuriosities.blogspot.com.ar/"

},

{

"type" : "CD",

"label" : "No culpes a la noche",

"author" : "Luis Miguel",

"image": "http://www.gravatar.com/avatar/a133c7d8d9245d063b136732272ea77e"

},

{

"type" : "book",

"label" : "Mule in Action",

"author" : "Victor Romero"

},

{

"type" : "CD",

"label" : "Yesterday",

"author" : "The Beatles"

}

]

}

o/p:

{

"patents": [

{

"owner": "Victor Romero",

"categories": [

"book"

]

},

{

"owner": "The Beatles",

"categories": [

"CD"

]

},

{

"owner": "Luis Miguel",

"categories": [

"book",

"CD"

]

}

]

}

| **Operator** | **Description** | **Level** |
| --- | --- | --- |
| using, all unary operators | All [unary operators](https://en.wikipedia.org/wiki/Unary_operation) | 1 |
| As | Type Coercion expression | 2 |
| \* / | Multiplicative | 3 |
| + - >> | Additive | 4 |
| >= ⇐ < > is | Relational / Type Comparison | 5 |
| != ~= == | Equality evaluators | 6 |
| [AND](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#AND) | Conditional And | 7 |
| [OR](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#OR) | Conditional OR | 8 |
| [Default](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#default), [[Pattern Matching]](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-language-introduction#Pattern%20Matching), [Matches](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#matches), [Map](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#map), [Map Object](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#map-object), [Group By](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#group-by), [Filter](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-operators#filter) | Default Value / Pattern Matching / Binary Operators | 9 |
| [When Otherwise](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-language-introduction#when-otherwise), [Unless Otherwise](https://docs.mulesoft.com/mule-runtime/3.9/dataweave-language-introduction#unless-otherwise) | Conditional Expressions | 10 |

# Reduce(:Array to :object)

## reduce(Array<T>, (item: T, accumulator: T) -> T): T | Null

### Parameters

| **Name** | **Description** |
| --- | --- |
| item | Item in the given list. It provides the value to reduce. Can also be referenced as $. |
| acc | An accumulator (also referenced as $$) to apply to items in the input array. It starts with the first value of the array by default. |

%dw 2.0

output application/json

---

{

"sum" : [0, 1, 2, 3, 4, 5] reduce ($$ + $),

"sum" : [0, 1, 2, 3, 4, 5] reduce ((val, acc) -> acc + val)

}

o/p

{

"sum": 15,

"sum": 15

}

i/p

%dw 2.0

output application/json

---

{

"concat" : ["a", "b", "c", "d"] reduce ($$ ++ $),

"concat" : ["a", "b", "c", "d"] reduce ((val, acc) -> acc ++ val)

}

o/p

{

"concat": "abcd",

"concat": "abcd"

}

Eg:

%dw 1.0

%output **application/json**

%var arr= [{number : 10, times : 8},{number : 2, times : 5},{number : 7, times : 7},{number : 6, times : 3},{number : 4, times : 7}]

---

//[10,40,60] reduce $$+$

//sum: {

// number: arr reduce ((val,acc=0) -> acc + val.number),

// times: arr reduce ((val,acc=0) -> acc + val.times),

// total : arr reduce ((val,acc=0)-> acc + val.times)

//

// }

payload **reduce** ((val,acc={}) -> acc ++ {(val.ClubID):val.price})

i/p

===

[

{

"ClubID": "A1234",

"Name": "Barcelona FC",

"Country": "Spain",

"price": "30.00"

}, {

"ClubID": "B1234",

"Name": "Chelsea FC",

"Country": "Spain",

"price": "56.76"

}, {

"ClubID": "C1234",

"Name": "Arsenal FC",

"Country": "Englanda",

"price" : "100.67"

}]

o/p

===

{

"A1234": "30.00",

"B1234": "56.76",

"C1234": "100.67"

}

Eq:

payload reduce ((val,acc={})-> acc ++ {(val.ClubID): val ++ "ww":"5"})

OR

payload **map** {

($.ClubID):$ **mapObject** { ($$):$ }

o/p

[

{

"A1234": {

"ClubID": "A1234",

"Name": "Barcelona FC",

"Country": "Spain",

"price": "30.00"

}

},

{

"B1234": {

"ClubID": "B1234",

"Name": "Chelsea FC",

"Country": "Spain",

"price": "56.76"

}

},

{

"C1234": {

"ClubID": "C1234",

"Name": "Arsenal FC",

"Country": "Englanda",

"price": "100.67"

}

}

]

**Functions:**

%function add(a ,b ) a+b

%function calculate(a,b) a-b **when** a>b **otherwise** b-a

---

{

sum:add(21,2),

cal:calculate(7,20)

}

**String Manipulation:**

M1:

contacts:payload.users.\*user **map** {

firstName: ($.name **splitBy** /\s+/)[0],

lastName: ($.name **splitBy** /\s+/)[1],

email: **lower** $.name **replace** /\s+/ **with** "." ++ "@acme.com.ar",

address: $.street

}

M2:

%dw 1.0

%output **application/json**

%function splitBySpace(name) name **splitBy** /\s+/

---

contacts: payload.users.\*user **map** **using**(names= splitBySpace($.name)){

firstName:names[0],

lastName:names[1],

email:**lower** $.name **replace** /\s+/ **with** "." ++ "@gmail.com",

address:$.name

}

**Input:**

<users>

<user>

<name>Mariano Achaval</name>

<phone>152235465654</phone>

<street>Laprida 924</street>

</user>

<user>

<name>Martin Alejandro Cousido</name>

<phone>15332255555</phone>

<street>Acassuso 2280</street>

</user>

</users>

**Output:**

{

"contacts": [

{

"firstName": "Mariano",

"lastName": "Achaval",

"email": "mariano.achaval@acme.com.ar",

"address": "Laprida 924"

},

{

"firstName": "Martin",

"secondName": "Alejandro",

"lastName": "Cousido",

"email": "martin.alejandro.cousido@acme.com.ar",

"address": "Acassuso 2280"

}

]

}