Dataweave

# Introduction

DataWeave is the MuleSoft expression language for accessing and transforming data that travels through a Mule app. Mule runtime is responsible for running the script and expressions in our Mule application. DataWeave is strongly integrated with Mule runtime.

DataWeave files are categorized into two main sections:

Header -> Which defined directives

Body ----> Which describes output structure

These two are delimited by a separator “---“.

The header contains directives, where we can define the following:

1. DataWeave version
2. Input types & sources
3. Output type
4. Namespaces to import
5. Constants
6. Functions etc.

In body section, we can write the required transformation logic that defines output structure.

**Example Dataweave code:**

%dw 2.0

output application/json

var user = {"firstName": "Murali", "lastName" : "TMR"}

var a = 2

fun FName(data) = (data.firstName)

var Fullname = (data) -> (data.firstName ++ ' ' ++ data.lastName)

---

{

"A-Value" : a,

"FirstName" : FName(user),

"Fullname" : Fullname(user)

}

# DataWeave Playground

Url: <http://34.205.75.56:8081/>

# Dataweave Operators

DataWeave 2.0 supports several operators, including mathematical operators, equality operators, and operators such as prepend and append.

## Mathematical Operators

DataWeave 2.0 supports the most common mathematical operators:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | For addition. |
| - | For subtraction. |
| \* | For multiplication. |
| / | For division. |

### Example

%dw 2.0

output application/json

---

{ "mathOperators" : [

{ "2 + 2" : (2 + 2) },

{ "2 - 2" : (2 - 2) },

{ "2 \* 2" : (2 \* 2) },

{ "2 / 2" : (2 / 2) }

]

}

## Equality and Relational Operators

DataWeave 2.0 supports the following equality and relational operators:

| **Operator** | **Description** |
| --- | --- |
| < | For less than. |
| > | For greater than. |
| <= | For less than or equal to. |
| >= | For greater than or equal to. |
| == | For equal to. |
| ~= | Equality operator that tries to coerce one value to the type of the other when the types are different. |

### Example

%dw 2.0

output application/json

---

{ "relational" : [

{ "1 < 1" : (1 < 1) },

{ "1 > 2" : (1 > 2) },

{ "1 <= 1" : (1 <= 1) },

{ "1 >= 1" : (1 >= 1) }

],

"equality" :

[

(1 == 1),

(1 == 2),

("true" == true),

("true" ~= true),

(['true', false] ~= [true, 'false']),

('1' ~= 1)

]

}

## Logical Operators

DataWeave 2.0 supports the following logical operators:

| **Operator** | **Description** |
| --- | --- |
| not | Negates the result of the input. See also, !. |
| ! | Negates the result of the input. See also, not. |
| and | Returns true if the result of all inputs is true, false if not. |
| or | Returns true if the result of any input is true, false if not. |

Note: Though the semantics of not and ! are the same, their precedence differs. not true or true is executed as not (true or true), so it returns false, whereas !true or true returns true because the ! only applies to the first true. !(true or true) returns false.

### Example

%dw 2.0

output application/json

var myArray = [1,2,3,4,5]

var myMap = myArray map not (($ mod 2) == 0)

---

{

"not" : [

"notTrue" : not true,

"notFalse" : not false,

"myMapWithNot" : myMap

],

"and" : [

"andTrueFalse" : true and false,

"andIsTrue" : (1 + 1 == 2) and (2 + 2 == 4),

"andIsFalse" : (1 + 1 == 2) and (2 + 2 == 2)

],

"or" : [

"orTrueFalse" : true or false,

"orIsTrue" : (1 + 1 == 2) or (2 + 2 == 2),

"orIsFalse" : (1 + 1 == 1) or (2 + 2 == 2)

],

"!-vs-not" : [

     "example-!" : (! true or true),

     "example-not" : (not true or true)

]}

## Prepend, Append, and Remove Operators for Arrays

DataWeave 2.0 supports operators for appending and prepending items within an array:

| **Operator** | **Description** |
| --- | --- |
| >> | Prepends data on the left-hand side of the operator to items in the array on the right-hand side. For example, 1 >> [2] results in [ 1, 2 ], prepending 1 to 2 in the array. |
| << | Appends data on the right-hand side of the operator to items in the array on the left-hand side. For example, [1] << 2 results in [ 1, 2 ], appending 2 to 1 in the array. |
| + | Appends data on the right-hand side of the operator to items in the array on the left-hand side. For example, [1] + 2 results in [ 1, 2 ], appending 2 to 1 in the array. The array is always on the left-hand side of the operator. |
| - | Removes a specified element of any supported type from an array. |

### Example

%dw 2.0

output application/json

---

{

"prepend-append" : [

   // Array on right side when prepending.

{ "prepend" : 1 >> [2] },

   // Array is on left side when appending.

{ "append-number" : [1] << 2 },

// + always appends within the array

{ "append-with-+" : [1] + 2 },

{ "append-with-+" : [2] + 1 },

{ "removeNumberFromArray" : ( [1,2,3] - 2 ) },

{ "removeObjectFromArray" : ( [ {a : "b"}, {c : "d"} , { e : "f"} ] - { c : "d"} ) }

]

}

## Flow Control Operators

| **Operator** | **Description** |
| --- | --- |
| if else | An if operator evaluates a conditional expression and returns the value under the if only if the conditional expression is true. Otherwise, it returns the expression under else. Every if expression must have a matching else expression. See [if else](https://docs.mulesoft.com/mule-runtime/4.3/dataweave-flow-control#control_flow_if_else) for an example. |
| else if | An else operator chains expressions together within an if-else construct by incorporating else if. See [else if](https://docs.mulesoft.com/mule-runtime/4.3/dataweave-flow-control#control_flow_else_if) for an example. |

### Examples

#### Example 1: if-else

%dw 2.0

var myVar = { country : "FRANCE" }

output application/json

---

if (myVar.country == "USA")

{ currency: "USD" }

else { currency: "EUR" }

#### Example 2: else-if

%dw 2.0

var myVar = { country : "UK" }

output application/json

---

if (myVar.country =="USA")

    { currency: "USD" }

else if (myVar.country =="UK")

    { currency: "GBP" }

else { currency: "EUR" }

# Core

This module contains core DataWeave functions for data transformations. It is automatically imported into any DataWeave script.

| **Function** | **Description** |
| --- | --- |
| ++ | Concatenates two values. |
| -- | Removes specified values from an input value. |
| contains | Returns true if an input contains a given value, false if not. |
| distinctBy | Iterates over an array and returns the unique elements in it. |
| endsWith | Returns true if a string ends with a provided substring, false if not. |
| startsWith | Returns true or false depending on whether the input string starts with a matching prefix. |
| filter | Iterates over an array and applies an expression that returns matching values. |
| flatten | Turns a set of subarrays (such as [ [1,2,3], [4,5,[6]], [], [null] ]) into a single, flattened array (such as [ 1, 2, 3, 4, 5, [6], null ]). |
| map | Iterates over items in an array and outputs the results into a new array. |
| mapObject | Iterates over an object using a mapper that acts on keys, values, or indices of that object. |
| matches | Checks if an expression matches the entire input string. |
| pluck | Useful for mapping an object into an array, pluck iterates over an object and returns an array of keys, values, or indices from the object. |
| reduce | Applies a reduction expression to the elements in an array. |
| replace | Performs string replacement |
| sizeOf | Returns the number of elements in an array. It returns 0 if the array is empty. |
| avg | Returns the average of numbers listed in an array. |
| sum | Returns the sum of numeric values in an array. |
| splitBy | Splits a string into a string array based on a value that matches part of that string. It filters out the matching part from the returned array. |
| to | Returns a range with the specified boundaries. |
| trim | Removes any blank spaces from the beginning and end of a string. |
| typeOf | Returns the type of a value. |

More functions: <https://docs.mulesoft.com/mule-runtime/4.3/dw-core>

# String

Functions in the Core (dw::Core) module are imported automatically into your DataWeave scripts. To use other modules, you need to import the module or functions you want to use by adding the import directive to the head of your DataWeave script, for example:

import dw::core::Strings

import camelize, capitalize from dw::core::Strings

import \* from dw::core::Strings

String module or its functions can be imported in ways as shown above.

| **Function** | **Description** |
| --- | --- |
| dasherize | Replaces spaces, underscores, and camel-casing in a string with dashes (hyphens). |
| camelize | Returns a string in camel case based on underscores in the string. |
| capitalize | Capitalizes the first letter of each word in a string. |
| isAlpha | Checks if the text contains only Unicode digits. A decimal point is not a Unicode digit and returns false. |
| isAlphaNumeric | Checks if the text contains only Unicode letters or digits. |
| isNumeric | Checks if the text contains only Unicode digits. |

More functions: <https://docs.mulesoft.com/mule-runtime/4.3/dw-strings>

## Example:

%dw 2.0

import \* from dw::core::Strings

output application/json

---

{

sizeof: sizeOf("Hello World!"),

trim: trim(" hello world "),

upper : upper("hello"),

lower : lower("HELLO"),

splitBy: "Hello World" splitBy(" "),

concat: "Hello" ++ "World",

contains: "Hello" contains "k",

equals: "Hello" == "Hello",

charAtIndex: "Hello"[3],

subString: "Hello" [0 to 3],

startsWith: "Hello" startsWith "hell",

endsWith: "Hello" endsWith "lo",

replace: "hello" replace "ll" with "LL",

matches: "hello" matches /[\w]+/,

//functions from Strings

dasherize: dasherize("hello world"),

camelize: camelize("hello\_world"),

capitalize: capitalize("hello world"),

isAlpha: isAlpha("hello"),

isAlphaNumeric: isAlphanumeric("hello1345world"),

isNumeric: isNumeric("123")

}

# Arrays

Basic operation on arrays;

## Example:

%dw 2.0

var myArr1 = [1,2,3,4,5]

var myArr2 = ["A", "B", 1, 2, 3, 1]

output application/json

---

{

sizeOf: sizeOf(myArr1),

slicing: myArr1[2 to -1],

reverse: myArr1[-1 to 0],

sum: sum(myArr1),

avg: avg(myArr1),

contains: myArr1 contains 2,

filter: myArr1 filter ($ > 2) ,

"prepend": 15 >> myArr1,

"append": myArr1 << 20,

"myarr1 + myarr2": myArr1 + myArr2,

flatten: flatten(myArr1 + myArr2),

distinctBy: myArr2 distinctBy ($),

"myarr1 ++ myarr2": myArr1 ++ myArr2,

"myarr1 -- myarr2": myArr1 -- myArr2,

"myarr2 -- myarr1": myArr2 -- myArr1,

}

# Miscellaneous

## Example

%dw 2.0

var myArr1 = [1,2,3,4,5]

var myArr2 = ["A", "B", 1, 2, 3, 1]

fun toUpper(aString) = upper(aString)

fun multiplyBy3(num) = num \* 3

output application/json

---

{

function: toUpper("hello"),

groupBy: myArr1 groupBy ( if ($ < 3) "smaller" else "larger"),

groupBy2: myArr2 groupBy ( if ($ matches /[A-Z]/) "alphabet" else "numeric"),

reduceAdd: myArr1 reduce ((item, accumulator = 0) -> (accumulator + item)),

reduceConcat: myArr2 reduce ((item, accumulator = 0) -> (accumulator ++ item)),

reduceObject: myArr1 reduce ((item, accumulator = {}) -> accumulator ++ {'key$item': item}),

map: myArr1 map ($ \* 3 ),

mapWithFunction: myArr1 map ( multiplyBy3($) ),

map2: myArr1 map ((item, index) -> {

'index-$index': item,

index: index,

item: item

}),

mapAndFilter: myArr1 filter ( $ < 3) map ($ \* 3 )

}