**http://www.a2cart.com/mule-esb-interview-questions/**

**Choice Router :**

Do not use choice routers to conditionally set variables, instead opt for a set-variable component and conditionally set the value using DataWeave.

We can use like this

**<set-variable value='#[if(condition) "1" else "2"]' doc:name="x" variableName="x"/>**

OR MEL or dataweave not Choice

Choice router isn’t meant to be used in every situation you would use an if-then-else statement. Instead, the choice router is meant to be used when you need to use conditional logic that will dictate how your message will be routed through your program.

For conditional throwing exception use find suitable validator components

**Frequently used Connectors in Projects:**

Custom file filter for File connector

[Apache Commons IO](http://commons.apache.org/proper/commons-io/javadocs/api-2.5/index.html) ships with Mule runtime and provides various file filters that can easily be used in Mule application.

File inbound endpoint element only allows configuring filter that implements java.io.FileFilter or java.io.FilenameFilter.

Write your own java customer filter using extending these classes and declare as beans and refer it from connector as reference.

**<file:inbound-endpoint path="input" moveToDirectory="output" responseTimeout="10000" doc:name="File">**

**<filter ref="fileFilterWrapper"></filter>**

**</file:inbound-endpoint>**

<https://blogs.mulesoft.com/dev/connectivity-dev/using-advanced-file-filters-in-file-inbound-endpoint/>

<https://blogs.perficient.com/2017/02/28/enabling-https-for-mule-application/>

By default, http.port = 8081, https.port = 8082, Cloudhub forwards port 80 to http.port (8081), 443 to port https.port (8082) to your worker VM which runs your application.

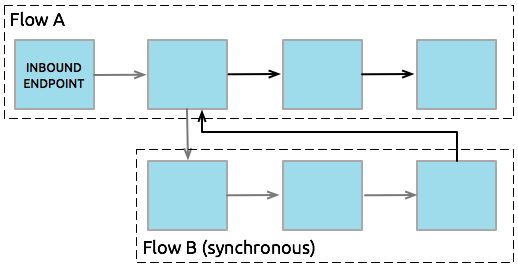
**Types of Flows**

When its execution is triggered by another flow in an application, a flow exists as one of three types:

|  |  |  |
| --- | --- | --- |
| **1** | **Subflow** | A subflow processes messages **synchronously** (relative to the flow that triggered its execution) and always inherits both the processing strategy and exception strategy employed by the triggering flow. While a subflow is running, processing on the triggering flow pauses, then resumes only after the subflow completes its processing and hands the message back to the triggering flow. |
| **2** | **Synchronous Flow** | A synchronous flow, like a subflow, processes messages **synchronously** (relative to the flow that triggered its execution). While a synchronous flow is running, processing on the triggering flow pauses, then resumes only after the synchronous flow completes its processing and hands the message back to the triggering flow. However, unlike a subflow, this type of flow *does not* inherit processing or exception strategies from the triggering flow.  This type of flow processes messages along a single thread, which is ideally suited to transactional processing. |
| **3** | **Asynchronous Flow** | An asynchronous flow simultaneously and **asynchronously** processes messages in parallel to the flow that triggered its execution. When a flow passes a message to an asynchronous flow, thus triggering its execution, it simultaneously passes a copy of the message to the next message processor in its own flow. Thus, the two flows – triggering and triggered – execute simultaneously and independently, each finishing on its own. This type of flow *does not* inherit processing or exception strategies from the triggering flow.  This type of flow processes messages along multiple threads. |

### About Synchronous Message Processing

When a flow triggers a synchronous flow or subflow, it passes programmatic control to the triggered flow and suspends its own message processing activity. For example, when the synchronous Flow B completes its sequence of message processing events, it passes programmatic control back to Flow A. The message that exits Flow B replaces the message in Flow A (see image below).



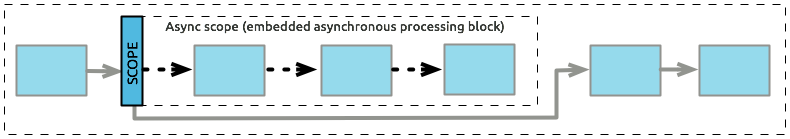
Since the Flow A and Flow B hand off programmatic control to each other and, by implication, all processing occurs on the same thread, each event in the message processing sequence can be tracked. This setup is is ideal for ensuring **transactional processing**.

| **Type of Flow** | **Component** | **Execution Relative to Triggering Flow** | **Exception and Processing Strategies** |
| --- | --- | --- | --- |
| Subflow | Flow Reference | synchronous | inherited |
| Synchronous Flow | Flow Reference | synchronous | not inherited |
| Asynchronous Flow | Flow Reference wrapped within an [Async Scope](https://docs.mulesoft.com/mule-runtime/3.8/async-scope-reference) | asynchronous | not inherited |

# Async Scope Reference

An **async scope** is a branch processing block that executes simultaneously with the parent message flow. This type of processing block can prove useful for executing time-consuming operations (such as printing a file or connecting to a mail server) — as long as those operations do not require sending a response back to the initiating flow. In other words, the main flow can continue execution while it initiates and processes the asynchronous scope; it does not have to pause until the last message processor embedded in the asynchronous flow has completed its task.

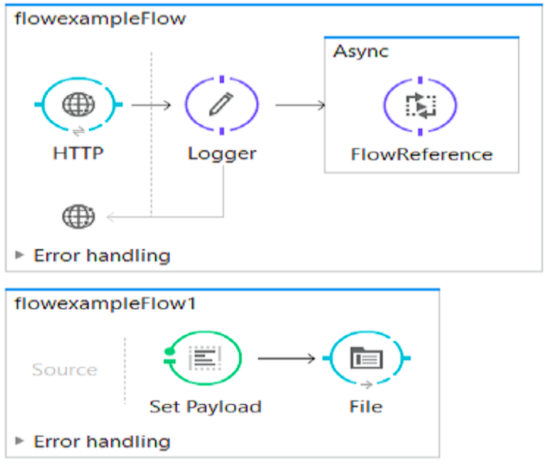
To facilitate this simultaneous branch processing, the async scope sends one copy of the message it has received to the first embedded message processor in its own processing block; at the same time it sends another copy of the message to the next message processor in the main flow (see below).



Since they operate on a copy of the message on a different thread, async scopes cannot, by definition, support request-response exchange patterns. Instead, they must implement one of several supported one-way processing strategies, as detailed in the configuration section, below.

If no processing strategy is configured for the async scope, Mule applies a queued-asynchronous processing strategy.

For asynchronous flows, the flow reference must be wrapped with async scope in the triggering flow.



## Async Scopes versus Asychronous Flows

An async scope is similar to an [**asynchronous flow**](https://docs.mulesoft.com/mule-runtime/3.8/flows-and-subflows) in that:

* It processes the message asynchronously with the main flow, so the message is simultaneously processed in the async scope without pausing the processing in the main flow thread
* Does not pass data from the scope back into the main flow thread
* It can have its own processing strategy

However, unlike an asynchronous flow, an async scope:

* Exists in-line with the main flow thread
* Is not called by a flow reference component
* Is not re-usable
* Cannot have its own exception handling strategy – it inherits this from the flow in which it resides

## Async Scopes versus Subflows

An async scope is similar to a [**subflow**](https://docs.mulesoft.com/mule-runtime/3.8/flows-and-subflows) in that it inherits the exception strategy of the main flow.

However, unlike a subflow, an async scope:

* Processes messages asynchronously
* Does not pass data back to the main flow
* Exists in-line with the main flow thread
* Is not called by a flow reference component
* Is not re-usable

\*Even though the Async scope receives a copy of the Mule message, the payload is not copied. The same payload objects are referenced by both Mule messages: One that continues down the original flow, and the one processed by the Async scope.

In other words, if the payload of your message is a mutable object (for example a bean with different fields in it) and a message processor in your async scope changes the value of one of the fields, the message processors outside of the Async scope see the changed values.

## Replacing versus Modifying Object References

If you replace, that is, change the reference completely inside the async scope, then both the payload and the flow variable in the original thread continue to have their original values.

If you modify, that is, make a change in the object referenced, but leave the same reference, the payload is modified for the original thread, but is preserved for the flow variable

**<flow name="replace">**

**<http:inbound-endpoint address="http://localhost:9000/replacepayload" exchange-pattern="request-response" />**

**<set-payload value="original payload" />**

**<set-variable value="original flowvar" variableName="testflowvar"/>**

**<logger level="WARN" message="original payload: #[payload]" />**

**<logger level="WARN" message="original flowvar: #[flowVars['testflowvar']]" />**

**<async>**

**<set-payload value="new payload" />**

**<set-variable value="new flowvar" variableName="testflowvar"/>**

**<logger level="WARN" message="Payload in async: #[payload]" />**

**<logger level="WARN" message="Flowvar in async: #[flowVars['testflowvar']]" />**

**</async>**

**<scripting:component>**

**<scripting:script engine="groovy">**

**<scripting:text>**

**Thread.sleep(3000)**

**return payload**

**</scripting:text>**

**</scripting:script>**

**</scripting:component>**

**<logger level="WARN" message="Payload after async: #[payload]" />**

**<logger level="WARN" message="Flowvar after async: #[flowVars['testflowvar']]" />**

**</flow>**

**<flow name="modify">**

**<http:inbound-endpoint address="http://localhost:9000/modifypayload" exchange-pattern="request-response" />**

**<set-payload value="#[['key':'originalvalue']]" />**

**<set-variable value="#[['key':'originalvalue']]" variableName="testflowvar"/>**

**<logger level="WARN" message="original payload: #[payload]" />**

**<logger level="WARN" message="original flowvar: #[flowVars['testflowvar']]" />**

**<async>**

**<set-payload value="#[payload.key = 'new payload'; return payload]" />**

**<set-variable value="#[['key':'new value']]" variableName="testflowvar"/>**

**<logger level="WARN" message="Payload in async: #[payload]" />**

**<logger level="WARN" message="Flowvar in async: #[flowVars['testflowvar']]" />**

**</async>**

**<scripting:component>**

**<scripting:script engine="groovy">**

**<scripting:text>**

**Thread.sleep(3000)**

**return payload**

**</scripting:text>**

**</scripting:script>**

**</scripting:component>**

**<logger level="WARN" message="Payload after async: #[payload]" />**

**<logger level="WARN" message="Flowvar after async: #[flowVars['testflowvar']]" />**

**</flow>**

### Configuring a Processing Strategy

Configuring a processing strategy is optional. Unless you explicitly define a different one, Mule applies the queued-asynchronous processing strategy to the scope. You can configure the **Processing Strategy** of the async scope to one of the following available processing strategies.

| **Strategy** | **Description** |
| --- | --- |
| Asynchronous Processing Strategy | Same as queued-asynchronous processing strategy (which is what Mule applies if no other processing strategy is configured) except that it doesn’t use a queue. Use this only if for some reason you do not want your processing to be distributed across nodes. |
| Custom Processing Strategy | A user-written processor strategy. |
| Queued-Asynchronous Processing Strategy | Uses a queue to decouple the flow’s receiver from the rest of the steps in the flow. It works the same way in a scope as in a flow. Mule applies this strategy unless another is specified. Select this if you want to fine-tune this processing strategy by:   * Changing the number of threads available to the flow. * Limiting the number of messages that can be queued. * Specifying a queue store to persist data. |
| Queued Thread Per Processor Processing Strategy | Not applicable to most use cases. Writes messages to a queue, then every processor in the scope runs sequentially in a different thread. |
| Thread Per Processor Processor Strategy | Not applicable to most use cases. Every processor in the scope runs sequentially in a different thread. |

## DataWeave Body

Regardless of the input and output types, the data model for the output is always described in standard DataWeave code, and it is this model that the input is transformed into.

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

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

When you write code in the DataWeave body, you define an expression that generates one of the data types listed above, even a literal string Hello world is a valid DataWeave body.

You just need to remember that each expression produces an Object, an Array, or a Variable.

*To visualize the canonical DataWeave model of your data to get a better visual reference, set the output type of your transform to application/dw. Your transform then outputs your data as a DataWeave expression, which resembles a JSON object.*

### Simple Values

Simple values can be of the following types:

* [**String**](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#string) : Double quoted ("Hello") or Single quoted ('Hello')
* [**Boolean**](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#boolean) : Literals true or false
* [**Number**](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#number) : Decimal and Integer values are supported (ex: 2.0)
* [**Date**](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#dates) : IS0-8601 enclosed by "|" (ex:|2003-10-01T23:57:59Z|)
* [**Regex**](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#regular-expressions) : Regex expression enclosed by "/" (ex:/(\d+)-(\d+)/)

### **Arrays**

Arrays are represented as a sequence of value expressions.

Input

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

Transform

%dw 1.0

%output application/json

---

[ "My", "three", "words" ]

### **Objects**

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

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**.

Constants : You can declare in header and can use in body

### **Scoped Variables:** To declare a variable in the DataWeave body, the following syntax is supported: **using (<variable-name> = <expression>)** and it must be written before defining the contents of the literal that it exists in. To reference an already initialized variable, you can just call it by the name you defined for it as with any other variable, or you can also write it in the form **$<variable-name>**.

Transform

%dw 1.0

%output application/xml

%var pi=3.14 //constants

---

entry: using (firstName = "Bear", lastName = "Grylls") {

person: using (user = firstName, gender = "male") {

name: user,

gender: gender

},

sn: lastName, **(1)** The reference **lastName** is valid because it is within scope

gen: gender **(2)** The reference **gender** is invalid because gender was declared in the **person** object, and this reference exists outside the scope of that object.

}

**Expressions**

DataWeave allows you to put logic in your script using expression values. All expressions in DataWeave return a value, these can be categorized into:

Operators

Selectors

Flow Control Expressions

#### When Otherwise

The keyword **when** conditionally evaluates a part of your DataWeave code, depending on if an expression evaluates to true or to false. You can make a single line conditional, or enclose a whole section in curly brackets. In case the **when** expression evaluates to **false**, its corresponding part of the code is ignored, and the code that corresponds to the **otherwise** expression is executed.

Transform

%dw 1.0

%output application/json

---

{

currency: "USD"

} when payload.country == "USA"

otherwise

{

currency: "EUR"

}

You can also chain several **otherwise** expressions together, like in the example below:

Transform

%dw 1.0

%output application/json

---

{

currency: "USD"

} when payload.country =="USA"

otherwise

{

currency: "GBP"

} when payload.country =="UK"

otherwise

{

currency: "EUR"

}

|  |
| --- |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see what expressions are compiled before or after this one. |

#### Unless Otherwise

The keyword **unless** conditionally evaluates a part of your DataWeave code, depending on if an expression evaluates to true or to false. You can make a single line conditional, or enclose a whole section in curly brackets. In case the **unless** expression evaluates to **true**, its corresponding part of the code is ignored, and the code that corresponds to the **otherwise** expression is executed.

Transform

%dw 1.0

%output application/json

---

{

currency: "EUR"

} unless payload.country == "USA"

otherwise

{

currency: "USD"

}

|  |
| --- |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see what expressions are compiled before or after this one. |

#### Default

Assigns a default value in case no value is found in the input field.

Transform

%dw 1.0

%output application/json

---

{

currency: payload.currency default "USD"

}

|  |
| --- |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see what expressions are compiled before or after this one. |

#### Pattern matching

Pattern matching executes on the first pattern that matches the specified expression. DataWeave supports four different types of patterns:

* literal
* type/traits
* regex
* expression

Each pattern type can be either named or unnamed. The example below is not actual DataWeave code, but rather a model for how matching works, you can see more concrete examples on each of the sections that follow:

value match {

(<name>:)?<pattern> -> <when matched>,

(<name>:)?<pattern> -> <when matched>,

default -> <when none of them matched>

}

|  |  |
| --- | --- |
| For simpler use cases where all you need is a boolean result based on if a value matches or not, see the [Matches Operator](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#matches). | |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see what expressions are compiled before and after this one. |

##### Literal Pattern

Matches when the evaluated value equals a simple literal value.

Transform

%dw 1.0

%output application/json

---

a: payload.string match {

"Emiliano" -> true,

"Mariano" -> false

},

b: payload.string match {

str: "Emiliano" -> { "matches": true, value: str },

str: "Mariano" -> { "matches": false, value: str }

}

### Input: JSON

Input

{

"string": "Emiliano"

}

### Output: JSON

Output

{

"a": true,

"b": {

"matches": true,

"value": "Emiliano"

}

}

In this example, the first field simply matches the value in 'payload.string' and returns a boolean, the second field performs the same match, but returns an object that contains both a boolean and a reference to the validated value.

##### Expression Pattern

Matches when running a certain expression over the evaluated value returns true.

Transform

%dw 1.0

%output application/json

---

{

a: payload.string match {

str when str == "Mariano" -> str ++ " de Achaval",

str when str == "Emiliano" -> str ++ " Lesende"

},

b: payload.number match {

n when n < 3 -> "lower",

n when n > 3 -> "higher"

}

}

### Input: JSON

Input

{

"string": "Emiliano",

"number": 3.14

}

### Output: JSON

Output

{

"a": "Emiliano Lesende",

"b": "higher"

}

In this example, the first field matches the value of 'payload.string' against two alternatives and conditionally appends a different string to it; the second field evaluates if the value in 'payload.number' is larger or smaller than 3 and returns "lower" or "higher" accordingly.

##### Match Type

Matches when the evaluated value is of the specified type

Transform

%dw 1.0

%output application/json

---

{

a: payload.a match {

:object -> "OBJECT",

:string -> "STRING",

:number -> "NUMBER",

:boolean -> "BOOLEAN",

:array -> "ARRAY",

:null -> "NULL"

},

b: payload.b match {

y is :object -> { type: "OBJECT", y: y },

y is :string -> { type: "STRING", y: y },

y is :number -> { type: "NUMBER", y: y },

y is :boolean -> { type: "BOOLEAN", y: y },

y is :array -> { type: "ARRAY", y: y },

y is :null -> { type: "NULL", y: y }

}

}

### Input: JSON

Input

{

"a": "Emiliano",

"b": 3.14

}

### Output: JSON

Output

{

"a": "STRING",

"b": {

"type": "NUMBER",

"y": 3.14

}

}

In this example, the first field evaluates the type of 'payload.a' and returns a different string with the type name depending on what type it matches with (this could be easier done through the [Type Of operator](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#type-of)), the second field returns an object with the same type name as a string and a reference to the evaluated value.

##### Match Regex

Matches when the evaluated value fits a given regular expression

Transform

%dw 1.0

%output application/json

---

{

a: payload.phones map ($ match {

/\+(\d+)\s\((\d+)\)\s(\d+\-\d+)/ -> { country: $[0], area: $[1], number: $[2] },

/\((\d+)\)\s(\d+\-\d+)/ -> { area: $[1], number: $[2] }

}),

b: payload.phones map ($ match {

phone: /\+(\d+)\s\((\d+)\)\s(\d+\-\d+)/ -> { country: phone[0], area: phone[1], number: phone[2] },

phone: /\((\d+)\)\s(\d+\-\d+)/ -> { area: phone[1], number: phone[2] }

})

}

### Input: JSON

Input

{

"phones": [

"+1 (415) 229-2009",

"(647) 456-7008"

]

}

### Output: JSON

Output

{

"a": [

{

"country": "+1 (415) 229-2009",

"area": "1",

"number": "415"

},

{

"area": "647",

"number": "456-7008"

}

],

"b": [

{

"country": "+1 (415) 229-2009",

"area": "1",

"number": "415"

},

{

"area": "647",

"number": "456-7008"

}

]

}

In this example, the payload includes two elements in an array, and in both cases the [Map operator](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map) to cycle through the array. It then evaluates each element against a regular expression and outputs a different object depending on what kind of match is found.

## System Values

DataWeave provides a set of values that are automatically assigned by the system.

### Now

Returns the present moment in [(:datetime)](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#dates) type.

Transform

%dw 1.0

%output application/json

---

{

a: now,

b: now.day,

c: now.minutes

}

Output

{

"a": "2015-12-04T18:15:04.091Z",

"b": 4,

"c": 15

}

|  |
| --- |
| See [DataWeave Selectors](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors) for a list of possible selectors to use here. |

### Random

Returns a random number of type [(:number)](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#number) between 0 and 1

Transform

%dw 1.0

%output application/json

---

{

price: random \* 1000

}

## Calling External Flows

From a DataWeave transform, you can trigger the calling of a different flow in your Mule application, and whatever the flow returns is what the expression returns.

You can do this through the following expression:

lookup(“flowName”,$)

Which takes two parameters:

* The name of the flow that must be called
* The payload to send to this flow, as a map

Transform

%dw 1.0

%output application/json

---

{

a: lookup("mySecondFlow",{b:"Hello"})

}

Mule Flow

<flow name="mySecondFlow">

<set-payload doc:name="Set Payload" value="#[payload.b + ' world!' ]"/>

</flow>

Output

{

"a": "Hello world!"

}

Please note that only the payload returned by the invoked flow will be assigned (i.e. all other message’s properties such as flowVars and sessionVars will not be overridden when using the lookup function).

|  |
| --- |
| The lookup function does not support calling [subflows](https://docs.mulesoft.com/mule-runtime/3.8/flows-and-subflows#types-of-flows). |

## Calling Global MEL Functions from DataWeave Code

If you define a global [Mule Expression Language](https://docs.mulesoft.com/mule-runtime/3.8/mule-expression-language-mel) (MEL) function in your Mule project, you can then invoke it anywhere in your DataWeave code, without need for any special syntax.

To create one such global function, you must edit your Mule project’s XML file and enclose any functions that you wish to define in the following set of tags, which must be placed in the global elements section, before any of the flows are defined.

<configuration doc:name="Configuration">

<expression-language>

<global-functions>

</global-functions>

</expression-language>

</configuration>

In this space you can use any MEL expression to define custom functions, for example:

<configuration doc:name="Configuration">

<expression-language>

<global-functions>

def newUser() {

return ["name" : "mariano"]

}

def upperName(user) {

return user.name.toUpperCase()

}

</global-functions>

</expression-language>

</configuration>

With that in place, in the DataWeave code of your Transform Message element you can just refer to these functions. Note that the inputs and outputs of these functions can even be objects and arrays.

%dw 1.0

%output application/json

---

{

"foo" : newUser(),

"bar": upperName(newUser())

}

Even with these external functions in place, you should be able to preview the output of this transform, updated in real time as you edit it.

## Read

(content :string, mimeType :string,readerOptions :object) ⇒ :any

The read function returns the result of parsing the content parameter with the specified mimeType reader.

The first argument points the content that must be read, the second is the format in which to write it. A third optional argument lists reader configuration properties.

Transform

%dw 1.0

%output application/xml

---

output: read(payload.root.xmlblock, "application/xml").foo

### Input: XML

Input

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

<root>

<xmlblock><![CDATA[<foo>bar</foo>]]></xmlblock>

</root>

### Output: XML

Output

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

<output>bar</output>

In the example above, what was in the CDATA element isn’t parsed by the DataWeave reader by default, that’s why the **read** operator must be used to interpret it.

## Write

(value :any, mimeType :string,writerOptions :object) ⇒ :string

The write function returns a string with the serialized representation of the value in the specified mimeType.

The first argument points to the element that must be written, the second is the format in which to write it. A third optional argument lists writer configuration properties. See [Output Directive](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#output-directive) and its sub-sections for a full list of available configuration options for each different format.

Transform

%dw 1.0

%output application/xml

---

{

output: write(payload, "application/csv", {"separator" : "|"})

}

### Input: JSON

Input

[{"Name": "Mr White",

"Email": "white@mulesoft.com",

"Id": "1234",

"Title": "Chief Java Prophet"},

{"Name": "Mr Orange",

"Email": "orange@mulesoft.com",

"Id": "4567",

"Title": "Integration Ninja"}]

### Output: XML

Output

<?xml version='1.0' encoding='US-ASCII'?>

<output>Name|Email|Id|Title

Mr White|white@mulesoft.com|1234|Chief Java Prophet

Mr Orange|orange@mulesoft.com|4567|Integration Ninja

</output>

## Log

(prefix :string,value :any)

Returns the specified value and also logs the value in the DataWeave representation with the specified prefix.

Transform

%dw 1.0

%output application/json

---

{

result: log("Logging the array",[1,2,3,4])

}

Output

{

"result": [1,2,3,4]

}

Output to Logger

Logging the array [1,2,3,4]

Note that besides producing the expected output, it also logs it.

## Precedence Table

This table lists the order in which different DataWeave expressions are compiled. The result of compiling something at one level may be used as an input for expressions in higher levels, but not vice-versa. Expressions are ordered in the table from first compiled to last.

| **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/3.8/dataweave-operators#AND) | Conditional And | 7 |
| [OR](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#OR) | Conditional OR | 8 |
| [Default](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#default), [[Pattern Matching]](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#Pattern Matching), [Matches](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#matches), [Map](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map), [Map Object](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map-object), [Group By](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#group-by), [Filter](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#filter) | Default Value / Pattern Matching / Binary Operators | 9 |
| [When Otherwise](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#when-otherwise), [Unless Otherwise](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#unless-otherwise) | Conditional Expressions | 10 |

## Closer Look at an Example Transformation

Input

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

<note>

<to>Tove</to>

<from>Jani</from>

<heading>Reminder</heading>

<body>Don't forget me this weekend!</body>

</note>

Transform

%dw 1.0

%output application/json

%var date='01-MAR-2015'

---

{

letter : payload,

sent : date

}

Output as JSON

{

"letter": { **(1)**

"note": { **(2)**

"to": "Tove",

"from": "Jani",

"heading": "Reminder", **(3)**

"body": "Don't forget me this weekend!"

}

},

"sent": "01-MAR-2015" **(4)**

|  |
| --- |
| }  1 The "payload" input is parsed into an Object.  2 As previously stated, Objects are sequences of key:value pairs. Note how each element name from the XML input is parsed into a key followed by a colon : and then the value.  3 The value may be a Simple Value, as is the case of the heading field, or an object, as is the case in note #2.  4 This value arises from a variable 'date', which is defined in a directive in the DataWeave header.  Whenever you make a transformation from JSON to XML, make sure that the resulting output is valid as an XML file. Specifically, make sure that there’s a single parent tag, JSON supports having multiple elements at the highest level while XML doesn’t. Likewise, whenever you transform from XML to JSON, make sure the resulting output is valid as a JSON file. Specifically, make sure that there are no repeated keys inside the same parent. XML supports having this but JSON doesn’t. |

# Selectors

The complex structure of Objects and Arrays can be navigated using Selector Expressions. Each selector expression returns either an object, an array, or a simple type. A selector always operates within a given context, which can be a reference to a variable, an object literal, an array literal, or the invocation of a function. As DataWeave processes a selector, a new context is set for further selectors, so you can navigate through the complex structures of arrays and objects by using chains of selectors, who’s depth is limited only by the depth of the current context.

There are 5 types of selector expression:

* **Single Value selector** .<key-name>
* **Multi Value selector** .\*<key-name>
* **Descendants Selector** ..<key-name>
* **Indexed Selector** [<index>]
* **Attribute Selector** .@<attribute-name>

Applying the **Single level Explicit Selector**, the **Descendants Selector**, or the **Indexed Selector** returns the value of the key:value pair that matches the expression.

**Note**: Each of these selector expressions supports having a '?' appended at the end of the chain. This changes the expression into a query that checks upon the existence of the key. The return type in this case is a boolean true or false.

This document also details how you can [access different elements](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors#reference-elements-from-an-incoming-mule-message) of the incoming Mule Message and environment:

* [The Payload of a Mule Message](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors#the-payload-of-a-mule-message)
* [Inbound Properties from a Mule Message](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors#inbound-properties-from-a-mule-message)
* [Outbound Properties from a Mule Message](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors#outbound-properties-from-a-mule-message)
* [Flow Variables from a Mule Message](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors#flow-variables-from-a-mule-message)
* [Accessing System and Spring Properties](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-selectors#accessing-system-and-spring-properties)

## Single Value selector

Value selectors may be applied over an :object or an :array.

### Over :object

This selector returns the first value whose key matches the expression, that is, **payload.name**, which returns the value whose key matches **name**.

Transform

%dw 1.0

%output application/xml

---

{

address: payload.people.person.address

}

### Input: JSON

Input

{

"people": {

"size" : 1,

"person": {

"name": "Nial",

"address": {

"street": {

"name": "Italia",

"number": 2164

},

"area": {

"zone": "San Isidro",

"name": "Martinez"

}

}

}

}

}

### Output: XML

Output

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

<address>

<street>

<name>Italia</name>

<number>2164</number>

</street>

<area>

<zone>San Isidro</zone>

<name>Martinez</name>

</area>

</address>

### Over :array

This selector is applied on each of the elements of the array that are of type :object and returns an array with all the selected values

Transform

%dw 1.0

%output application/json

---

payload.people.person.address.street

### Input: JSON

Input

{

"people": [ **(1)**

{

"person": {

"name": "Nial",

"address": {

"street": {

"name": "Italia",

"number": 2164

},

"area": {

"zone": "San Isidro",

"name": "Martinez"

}

}

}

},

{

"person": {

"name": "Coty",

"address": {

"street": {

"name": "Monroe",

"number": 323

},

"area": {

"zone": "BA",

"name": "Belgrano"

}

}

}

}

]

}

|  |  |
| --- | --- |
| **1** | As 'people' is an array, this sets the context for searching across both 'person' instances. The result from this is always an array. |

### Output: JSON

Output

[ **(1)**

{

"name": "Italia",

"number": 2164

},

{

"name": "Monroe",

"number": 323

}

]

|  |  |
| --- | --- |
| **1** | As the context is an array, the output is always an array. An array is returned even if there’s a single matching value. |

### Alternative syntax

You can also select a single value through the alternate syntax ["<key-name>"]. For example **payload["price"]** returns the value whose key matches **price**. This is valid for both arrays and objects.

Transform

%dw 1.0

%output application/json

---

payload["people"]

The main advantage of this syntax is that you can easily replace the name of a key with a variable to create a more dynamic expression.

Transform

%dw 1.0

%output application/json

---

payload.items[flowVars.item]

The above example receives a payload that contains a list of items, and a flow variable that indicates which of these items to take from the payload.

## Multi Value selector

Multi value selector can either be applied over an :object or an :array.

### Over :object

This selector returns an array with all the values whose key matches the expression.

Transform

%dw 1.0

%output application/json

---

{

users: payload.users.\*user

}

### Input: XML

Input

<users>

<user>Mariano</user>

<user>Martin</user>

<user>Leandro</user>

</users>

### Output: JSON

Output

{

"users": [

"Mariano",

"Martin",

"Leandro"

]

}

### Over :array

The selector is applied on each of the elements of the array that are of type :object and returns an array with all the selected values.

### Alternative syntax

You can also select multiple values through the alternate syntax [**"<key-name>"]. For example \*payload[**"price"]\* returns all the values whose key matches **price**. This is valid for both arrays and objects.

Transform

%dw 1.0

%output application/json

---

payload.items[\*item]

## Indexed Selector

The index selector returns the element at the specified position, it can be applied over an :array, an :object or a :string

### Over :array

This selector can be applied to String literals, Arrays and Objects. In the case of Objects, the value of the key:value pair found at the index is returned. The index is zero based.

1. If the index is bigger or equal to 0, it starts counting from the beginning.
2. If the index is negative, it starts counting from the end where -1 is the last element.

Transform

%dw 1.0

%output application/json

---

payload.people[1]

### Input: JSON

Input

{

"people": [

{

"name": "Nial",

"address": "Martinez"

},

{

"name": "Coty",

"address": "Belgrano"

}

]

}

### Output: JSON

Output

{

"name": "Coty",

"address": "Belgrano"

}

When using the Index Selector with a String, the string is broken down into an array, where each character is an index.

Transform

%output application/json

---

{

name: "MuleSoft"[0]

}

Output

{

"name": "M"

}

### Over :string

The selector picks the character at a given position, treating the string as an array of characters.

1. If the index is bigger or equal to 0, it starts counting from the beginning.
2. If the index is negative, it starts counting from the end.

Transform

%dw 1.0

%output application/json

---

{

name: "Emiliano"[0]

}

Output

{

"name": "E"

}

## Over :object

The selector returns the value of the key : value pair at the specified position.

## Range selector

### Over :array

Range selectors limit the output to only the elements specified by the range on that specific order. This selector allows you to slice an array or even invert it.

Transform

%dw 1.0

%output application/json

---

{

slice: [0,1,2][0..1],

last: [0,1,2][-1..0]

}

Output

{

"slice": [

0,

1

],

"last": [

2,

1,

0

]

}

### Over :string

The Range selector limits the output to only the elements specified by the range on that specific order, treating the string as an array of characters. This selector allows you to slice a string or even invert it.

Transform

%dw 1.0

%output application/json

---

{

slice: "DataWeave"[0..1],

last: "DataWeave"[-1..0]

}

Output

{

"slice": "Da",

"last": "evaeWataD"

}

## Attribute Selector Expressions

In order to query for the attributes on an Object, the syntax **.@<key-name>** is used. If you just use **.@** (without <key-name>) it returns an object containing each key:value pair in it.

Transform

%dw 1.0

%output application/json

---

{

item: {

type : payload.product.@type,

name : payload.product.brand,

attributes: payload.product.@

}

}

### Input: XML

Input

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

<brand>Samsung</brand>

</product>

### Output: JSON

Output

{

"item:" {

"type": "tv",

"name": "Samsung",

"attributes": { **(1)**

"id": 1,

"type": tv

}

}

}

|  |  |
| --- | --- |
| **1** | The third element in this output retrieves an object with all of the attributes in it, in this case both the id and the type. |

Transform

%dw 1.0

%output application/json

---

{

item: {

attributes : payload.product.@,

name : payload.product.brand

}

}

Output

{

"item": {

"attributes": {

"id": 1,

"type": "tv"

},

"name": "Samsung"

}

}

### Selecting the key value pair

As selectors only return the value of a key:value pair, in order to get both the key and value, you can use a type conversion to object.

Transform

%dw 1.0

%output application/xml

---

user: payload.name as :object **(1)**

|  |  |
| --- | --- |
| **1** | Using the **as :object** transforms the value into an object that contains the key as well as the value. Without this conversion to object, the returned XML body would simply be <user>Mariano</user>. |

### Input: JSON

Input

{

"name": "Mariano",

"lastName" : "Doe"

}

### Output: XML

Output

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

<user>

<name>Mariano</name>

</user>

## Descendants Selector

This selector is applied to the context using the form **..<field-name>** and retrieves the values of all matching key:value pairs in the sub-tree under the current context. Regardless of the hierarchical structure these fields are organized in, they are all placed at the same level in the output.

Transform

%dw 1.0

%output application/json

---

{

names: payload.people..name **(1)**

}

### Input: JSON

Input

{

"people": {

"person": {

"name": "Nial",

"address": {

"street": {

"name": "Italia",

"number": 2164

},

"area": {

"zone": "San Isidro",

"name": "Martinez"

}

}

}

}

}

### Output: JSON

Output

{

"names": [

"Nial",

"Italia",

"Martinez"

]

}

|  |  |
| --- | --- |
| **1** | In this example, all of the fields that match the key "name" are placed in a list called "names" regardless of their cardinality in the tree of the input data. |

### Selecting all the Descendant Key Value Pairs

Transform

%dw 1.0

%output application/xml

---

{

names: payload.people..name as :object**(1)**

}

|  |  |
| --- | --- |
| **1** | The **as: object** makes the expression return an object rather than an array (which would be returned by default). This implies that each value has a key. Without this conversion, in XML the returned array would be a single long string of characters comprised of all three names merged into one. |

### Input: JSON

Input

{

"people": {

"person": {

"name": "Nial",

"address": {

"street": {

"name": "Italia",

"number": 2164

},

"area": {

"zone": "San Isidro",

"name": "Martinez"

}

}

}

}

}

### Output: XML

Output

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

<names>

<name>Nial</name>

<name>Italia</name>

<name>Martinez</name>

</names>

## Selectors modifiers

There are two selectors modifiers: ? and !. The question mark returns true or false whether the keys are present on the structures. The exclamation mark evaluates the selection and fails if any key is not present.

### Key Present

Returns true if the specified key is present in the object.

Transform

%dw 1.0

%output application/xml

---

present: payload.name?

### Input: JSON

Input

{

"name": "Annie"

}

### Output: XML

Output:

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

<present>true</present>

In the example above, if a 'name' key does exist in the input, it returns **true**.

This operation also works with attributes:

Transform

%dw 1.0

%output application/json

---

{

item: {

typePresent : payload.product.@type?

}

}

### Input: XML

Input

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

<brand>Samsung</brand>

</product>

### Output: JSON

Output

{

"item": {

"typePresent": true

}

}

You can also use this validation operation as part of a filter:

Transform

%dw 1.0

%output application/xml

---

users: payload.users.\*name[?($ == "Mariano")]

### Input: XML

Input

<users>

<name>Mariano</name>

<name>Luis</name>

<name>Mariano</name>

</users>

### Output: XML

Output

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

<users>

<name>Mariano</name>

<name>Mariano</name>

</users>

The example above selects key:value pairs with value "Mariano" ⇒ {name: Mariano, name: Mariano}

### Assert Present

Returns an exception if any of the specified keys are not found.

Transform

%dw 1.0

%output application/xml

---

present: payload.lastName!**(1)**

|  |  |
| --- | --- |
| **1** | Throws the exception "There is no key named 'lastName'". |

### Input: JSON

Input

{

"name": "Annie"

}

### Output: XML

Exception: "There is no key named 'lastName'"

## Reference Elements From an Incoming Mule Message

Often, you want to use the different elements from the Mule Message that arrive at the DataWeave Transformer in your transform. The following sections show you how to reference each of these.

### The Payload of a Mule Message

You can take the **Payload** of the Mule message that reaches the DataWeave transformer and use it in your transform body.

%dw 1.0

%output application/xml

---

{

a: payload

}

You can also refer to sub elements of the payload through the dot syntax payload.user.

|  |
| --- |
| If the metadata for the payload’s inner contents are known to Studio, an autocomplete function helps you out. |

You can optionally also define the payload as an input directive in the header, although this isn’t required.

%dw 1.0

%input payload application/xml

%output application/xml

---

{

a: payload

}

### Inbound Properties from a Mule Message

You can take **Inbound Properties** from the Mule message that arrives to the DataWeave transformer and use them in your transform body. To refer to one of these, simply call it through the matching [Mule Expression Language (MEL)](https://docs.mulesoft.com/mule-runtime/3.8/mule-expression-language-mel) expression.

In MEL, there are two supported syntaxes to call for an inbound property:

* inboundProperties.name
* inboundProperties['name']

|  |
| --- |
| The first method only works if the variable name doesn’t include any periods or spaces. |

%dw 1.0

%output application/xml

---

{

a: inboundProperties.userName

}

|  |
| --- |
| If the metadata about these inbound properties is known to Studio, an autocomplete function helps you out. |

You can optionally also define the inbound property as a constant directive in the header, although this isn’t required as you can directly call the property as shown above.

%var inUname = inboundProperties['userName']

### Outbound Properties from a Mule Message

You can take any **Outbound Properties** in the Mule message that arrives to the DataWeave transformer and use it in your transform body. To refer to it, simply call it through the matching [Mule Expression Language (MEL)](https://docs.mulesoft.com/mule-runtime/3.8/mule-expression-language-mel) expression.

In MEL, there are two supported syntaxes to call an outbound property:

* outboundProperties.name
* outboundProperties['name']

|  |
| --- |
| The first method only works if the variable name doesn’t include any periods or spaces. |

%dw 1.0

%output application/xml

---

{

a: outboundProperties.userName

}

|  |
| --- |
| If the metadata about these outbound properties is known to Studio, an autocomplete function helps you out. |

You can optionally also define the outbound property as a constant directive in the header, although this isn’t required as you can directly call the property as shown above.

%var outUname = outboundProperties['userName']

### Flow Variables from a Mule Message

You can take any **Flow Variable** in the Mule message that arrives at the DataWeave transformer and use it in your transform body. To refer to it, simply call it through the matching [Mule Expression Language (MEL)](https://docs.mulesoft.com/mule-runtime/3.8/mule-expression-language-mel) expression.

In MEL, there are two supported syntaxes to call a flow variable:

* flowVars.name
* flowVars['name']

|  |
| --- |
| The first method only works if the variable name doesn’t include any periods or spaces. |

%dw 1.0

%output application/xml

---

{

a: flowVars.userName

}

|  |
| --- |
| If the metadata about these flow variables is known to Studio, an autocomplete function helps you out. |

You can optionally also define the flow variable as a constant directive in the header, although this isn’t required as you can directly call the variable as shown above.

%var uname = flowVars['userName']

### Accessing System and Spring Properties

You can reference any **Property** (System or Spring) that exists in the server while DataWeave is processing your transformation, to do so use the **p('prop\_name')** function or **'${prop\_name}'**.

%dw 1.0

%output application/xml

---

{

a: p('userName'),

b: '${userName}'

}

@@

# DataWeave Operators

In [DataWeave](https://docs.mulesoft.com/mule-runtime/3.8/dataweave) you can carry out many different operations on the elements of a DataWeave transform. This document serves as a reference for all of the available operators in the DataWeave language. See [all operators sorted by type](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#dataweave-operators-sorted-by-type)

* For an introduction to the essentials of the language, see [DataWeave Language Intro](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction)
* For information on DataWeave’s accepted types, see [DataWeave Types](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types)
* For an index of all available operators, categorized by the types you need to supply as parameters, see [DataWeave Operators Sorted by Type](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#operators-sorted-by-type).
* See [Functions and Lambdas](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#functions-and-lambdas) to learn how to create your own DataWeave functions.

|  |
| --- |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see the order in which DataWeave expressions are compiled. |

## Map

### Using Map to Return an Array

(':array', ':function') ⇒ :array

Returns an array that is the result of applying a transformation function (lambda) to each of the elements. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

---

users: ["john", "peter", "matt"] map upper $

Output

{

"users": [

"JOHN",

"PETER",

"MATT"

]

}

In the following example, custom names are defined for the index and value parameters of the map operation, and then both are used to construct the returned value. In this case, value is defined as **firstName** and its index in the array is defined as **position**.

Transform

%dw 1.0

%output application/json

---

users: ["john", "peter", "matt"] map ((firstName, position) -> position ++ ":" ++ upper firstName)

Output

{

"users": [

"0:JOHN",

"1:PETER",

"2:MATT"

]

}

### Using Map on an Object

(':object', ':function') ⇒ ':array'

Returns an array with the values that result out of applying a transformation function (lambda) to each of the values in the object. The keys of the original object are all ignored by this operation and the object is treated as an array. To have access to the keys, you can use the operation **mapObject** instead. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**. The index refers to the position of a key:value pair when the object is treated as an array.

|  |
| --- |
| See [Map Object](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map-object) if what you want is to process both keys and values instead of just values. |

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices map (

'$$':{

dollars: $,

localCurrency: $ \* conversionRate

}

)

Input

<prices>

<basic>9.99</basic>

<premium>53</premium>

<vip>398.99</vip>

</prices>

Output

{

"priceList": [

{

"0": {

"dollars": "9.99",

"localCurrency": 134.3655

}

},

{

"1": {

"dollars": "53",

"localCurrency": 712.85

}

},

{

"2": {

"dollars": "398.99",

"localCurrency": 5366.4155

}

}

]

}

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of in this example, you must either enclose it in quote marks or brackets. '' or ($$) are both equally valid. |

In the example above, as key and value are not defined, they’re identified by the placeholders **$$** and **$**. For each key:value pair in the input, an object is created and placed in an array of objects. Each of these objects contains two properties: one of these directly uses the value, the other multiplies this value by a constant that is defined as a directive in the header.

The mapping below performs exactly the same transform, but it defines custom names for the properties of the operation, instead of using $ and $$. Here, position is defined as referring to the array index, and money to the value in that index. The reference to the array index (named position, in this case) is optional.

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices map ((money, position) ->

'$position':{

dollars: money,

localCurrency: money \* conversionRate

}

)

The reference to the array index is optional. This is also a valid example:

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices map ((money) ->

{

dollars: money,

localCurrency: money \* conversionRate

}

)

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of position in this example, you must either enclose it in brackets or enclose it in quote marks adding a $ to it, otherwise the name of the property is taken as a literal string. '$position' or (position) are both equally valid. |

## Map Object

(':object', ':function') ⇒ ':object'

Similar to Map, but instead of processing only the values of an object, it processes both keys and values as a tuple. Also instead of returning an array with the results of processing these values through the lambda, it returns an object, which consists of a list of the key:value pairs that result from processing both key and value of the object through the lambda.

The lambda is invoked with two parameters: **key** and the **value**. If these parameters are not named, the key is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices mapObject (

'$$':{

dollars: $,

localCurrency: $ \* conversionRate

}

)

Input

<prices>

<basic>9.99</basic>

<premium>53</premium>

<vip>398.99</vip>

</prices>

Output

{

"priceList": {

"basic": {

"dollars": "9.99",

"localCurrency": 134.3655

},

"premium": {

"dollars": "53",

"localCurrency": 712.85

},

"vip": {

"dollars": "398.99",

"localCurrency": 5366.4155

}

}

}

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of in this example, you must either enclose it in quote marks or brackets. '' or ($$) are both equally valid. |

In the example above, as key and value are not defined, they’re identified by the placeholders **$$** and **$**. For each key:value pair in the input, the key is preserved and the value becomes an object with two properties: one of these is the original value, the other is the result of multiplying this value by a constant that is defined as a directive in the header.

The mapping below performs exactly the same transform, but it defines custom names for the properties of the operation, instead of using $ and $$. Here, 'category' is defined as referring to the original key in the object, and 'money' to the value in that key.

Transform

%dw 1.0

%output application/json

%var conversionRate=13.45

---

priceList: payload.prices mapObject ((money, category) ->

'$category':{

dollars: money,

localCurrency: money \* conversionRate

}

)

|  |
| --- |
| Note that when you use a parameter to populate one of the keys of your output, as with the case of **category** in this example, you must either enclose it in brackets or enclose it in quote marks adding a $ to it, otherwise the name of the property is taken as a literal string. '$category' or (category) are both equally valid. |

## Pluck

(':object', ':function') ⇒ ':array'

Pluck is useful for mapping an object into an array. Pluck is an alternate mapping mechanism to mapObject. Like mapObject, pluck executes a lambda over every key:value pair in its processed object as a tuple, but instead of returning an object, it returns an array, which may be built from either the values or the keys in the object.

The lambda is invoked with two parameters: **key** and the **value**. If these parameters are not named, the key is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

---

result: {

keys: payload.prices pluck $$,

values: payload.prices pluck $

}

Input

<prices>

<basic>9.99</basic>

<premium>53</premium>

<vip>398.99</vip>

</prices>

Output

{

"result": {

"keys": [

"basic",

"premium",

"vip"

],

"values": [

"9.99",

"53",

"398.99"

]

}

}

## Filter

### Using Filter on an Array

(':array', ':function') ⇒ ':array'

Returns an array that only contains those that pass the criteria specified in the lambda. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

---

{

biggerThanTwo: [0, 1, 2, 3, 4, 5] filter $ > 2

}

Output

{

"biggerThanTwo": [3,4,5]

}

### Using Filter on an Object

(':object', ':function') ⇒ ':object'

Returns an object with the key:value pairs that pass the acceptance criteria defined in the lambda. If these parameters are not named, the index is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/xml

---

filtered: {

aa: "a", bb: "b", cc: "c", dd: "d"

} filter $ == "d" **(1)**

|  |  |
| --- | --- |
| **1** | Filters the all key:value pairs with value "d" ⇒ {dd:d} |

Output

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

<filtered>

<dd>d</dd>

</filtered>

The usage above is performing the filter operation after the transformation block has occurred, if this is not a requirement then filter can be done prior to the transformation.

Transform

%dw 1.0

%output application/json

---

filtered: (payload filter ($.\*data contains "b") map {

data: $.data

}) **(2)**

|  |  |
| --- | --- |
| **1** | Filters the all key:value pairs with "data" value "b" ⇒ {data:b} before the transformation block has occurred. For large, complex transformations this is more efficient than filtering after the transformation block has occurred. |

Input

[

{"data": "a"},

{"data": "b"},

{"data": "c"},

{"data": "d"}

]

Output

{

"filtered": [

{

"data": "b"

}

]

}

|  |
| --- |
| If you require to filter by key, you need to use [mapObject](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map-object) and **when**. For example, to filter the last example by key:  %dw 1.0  %output application/xml  ---  filtered: {  aa: "a", bb: "b", cc: "c", dd: "d"  } mapObject ({ ($$): $ } when $$ as :string == "dd" otherwise {}) |

## Remove

### Using Remove on an Array

(':array', ':name') ⇒ ':array'

When running it on an array, it returns another array where the specified value is removed.

Transform

%dw 1.0

%output application/json

---

{

aa: ["a", "b", "c"] - "b"

}

Output

{

"aa": ["a", "c"]

}

### Using Remove on an Object

(':object', ':name') ⇒ ':object'

When running it on an object, it returns another object where the specified keys are removed.

Transform

%dw 1.0

%output application/json

---

myObject: {aa: "a", bb: "b"} - "aa"

Output

{

"myObject": {

"bb": "b"

}

}

The above example removes the key value pair that contains the key 'aa' from {aa: "a", bb: "b"} ⇒ {bb: "b"}

## Remove by Matching Key and Value

(':object', ':object') ⇒ ':object'

Works just like [remove](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#using remove on an object) on objects, but only removes an element when there is a match of not just the key but of the key + value pair . It returns another object where the specified keys are removed.

Transform

%dw 1.0

%output application/json

---

myObject: {aa: "a", aa:"c", bb: "b"} -- { aa:"a"}

Output

{

"myObject": {

"aa": "c",

"bb": "b"

}

}

The above example removes the key value pair that contains both the key 'aa' and value "a", but not the one that contains only a matching key but not value.

## AND

The expression **and** (in lower case) can be used to link multiple conditions, its use means that all of the linked conditions must evaluate to true for the expression as a whole to evaluate to true.

Transform

%dw 1.0

%output application/json

---

{

currency: "USD"

} when payload.country == "USA" and payload.currency == "local"

otherwise

{

currency: "EUR"

}

In the example above, currency is "EUR", unless the payload has BOTH conditions met.

|  |
| --- |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see what expressions are compiled before or after this one. |

## OR

The expression **or** (in lowercase) can be used to link multiple conditions. Its use means that either one or all of the linked conditions must evaluate to true for the expression as a whole to evaluate to true. This example combines the usage of **OR** with the [when and otherwise](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#when-otherwise) expressions.

Transform

%dw 1.0

%output application/json

---

{

currency: "EUR"

} when payload.country == "Italy" or payload.country == "Germany" or payload.country == "Spain" or payload.country == "Portugal" or payload.country == "France" or payload.country == "Greece"

otherwise

{

currency: "USD"

}

In the example above, currency is "EUR", only when one of the conditions evaluates to true.

|  |
| --- |
| Check the [Precedence Table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#precedence-table) to see what expressions are compiled before or after this one. |

## IS

(':any', condition) ⇒ ':boolean'

Evaluates if a condition validates to true and returns a boolean value. Conditions may include and and or operators.

Transform

%dw 1.0

%output application/xml

---

ROOT: payload.root.\*order mapObject (

ORDER:{

itemsCollectionPresent: $ is :object and $.items?

}

)

Input

<root>

<order>

<items> 155 </items>

</order>

<order>

<items> 30 </items>

</order>

<order>

null

</order>

</root>

Output

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

<ROOT>

<ORDER>

<itemsCollectionPresent>true</itemsCollectionPresent>

</ORDER>

<ORDER>

<itemsCollectionPresent>true</itemsCollectionPresent>

</ORDER>

<ORDER>

<itemsCollectionPresent>false</itemsCollectionPresent>

</ORDER>

</ROOT>

## Concat

The concat operator is defined using double plus signs. You must have spaces on both sides of them.

### Using Concat on an Array

(':array', ':array') ⇒ ':array'

When using arrays, it returns the resulting array of concatenating two existing arrays.

Transform

%dw 1.0

%output application/json

---

{

a: [0, 1, 2] ++ [3, 4, 5]

}

Output

{

"a": [0, 1, 2, 3, 4, 5]

}

### Using Concat on a String

(':string', ':string') ⇒ ':string'

Strings are treated as arrays of characters, so the operation works just the same with strings.

Transform

%dw 1.0

%output application/json

---

{

name: "Mule" ++ "Soft"

}

Output

{

"name": MuleSoft

}

### Using Concat on an Object

(':object', ':object') ⇒ ':object'

Returns the resulting object of concatenating two existing objects.

Transform

%dw 1.0

%output application/xml

---

concat: {aa: "a"} ++ {cc: "c"}

Output

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

<concat>

<aa>a</aa>

<cc>c</cc>

</concat>

The example above concatenates object {aa: a} and {cc: c} in a single one ⇒ {aa: a , cc: c}

## Contains

Evaluates if an array or list contains in at least one of its indexes a value that validates to true and returns a boolean value. You can search for a literal value, or match a regex too.

### Using Contains on an Array

(':array', ':any') ⇒ ':boolean'

You can evaluate if any value in an array matches a given condition:

Transform

%dw 1.0

%output application/json

---

ContainsRequestedItem: payload.root.\*order.\*items contains "3"

Input

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

<root>

<order>

<items>155</items>

</order>

<order>

<items>30</items>

</order>

<order>

<items>15</items>

</order>

<order>

<items>5</items>

</order>

<order>

<items>4</items>

<items>7</items>

</order>

<order>

<items>1</items>

<items>3</items>

</order>

<order>

null

</order>

</root>

Output

{

"ContainsRequestedItem": true

}

### Using Contains on a String

(':string', ':regex') ⇒ ':boolean'

You can also use contains to evaluate a substring from a larger string:

Transform

%dw 1.0

%output application/json

---

ContainsString: payload.root.mystring contains "me"

Input

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

<root>

<mystring>some string</mystring>

</root>

Output

{

"ContainsString": true

}

Instead of searching for a literal substring, you can also match it against a regular expression:

Transform

%dw 1.0

%output application/json

---

ContainsString: payload.root.mystring contains /s[t|p]ring/

Input

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

<root>

<mystring>A very long string</mystring>

</root>

Output

{

"ContainsString": true

}

## Type Coercion using as

Coerce the given value to the specified type.

|  |
| --- |
| DataWeave by default attempts to convert the type of a value before failing, so using this operator to convert is sometimes not required but still recommended. |
| Check the [type coercion table](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#type-coercion-table) to see what conversions between what types are allowed in DataWeave. |

### Coerce to string

(':any', ':type') ⇒ ':string'

Any simple types can be coerced to string. If formatting is required (such as for a number or date) the format schema property can be used.

Date and number format schemas are based on Java [DateTimeFormatter](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) and [DecimalFormat](https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html).

Transform

%dw 1.0

%output application/json

---

{

a: 1 as :string {format: "##,#"},

b: now as :string {format: "yyyy-MM-dd"},

c: true as :string

}

Output

{

"a": "1",

"b": "2015-07-07",

"c": "true"

}

### Coerce to number

(':string', ':type') ⇒ ':number'

A string can be coerced to number. If the given number has a specific format the schema property can be used.

Any format pattern accepted by [DecimalFormat](https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html) is allowed.

Transform

%dw 1.0

%output application/json

---

{

a: "1" as :number

}

Output

%dw 1.0

%output application/json

---

{

"a": 1

}

#### Coerce a date to number

(':time', ':type') ⇒ ':number'

When coercing a date to a number, there is an extra parameter you can add – 'unit' – to specify what unit of time to use,

Transform

%dw 1.0

%output application/json

---

{

mydate1: |2005-06-02T15:10:16Z| as :number {unit: "seconds"},

mydate2: |2005-06-02T15:10:16Z| as :number {unit: "milliseconds"}

}

Output

{

"mydate1": 1117725016,

"mydate2": 1117725016000

}

|  |
| --- |
| Only the values 'seconds' and 'milliseconds' are valid for using in the 'unit' parameter. |

### Coerce to date

(':string', ':type')/(':number', ':type') ⇒ ':date'

Date types can be coerced from string or number.

Any format pattern accepted by [DateTimeFormatter](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) is allowed.

Transform

%dw 1.0

%output application/json

---

{

a: 1436287232 as :datetime,

b: "2015-10-07 16:40:32.000" as :localdatetime {format: "yyyy-MM-dd HH:mm:ss.SSS"}

}

Output

{

"a": "2015-07-07T16:40:32Z",

"b": "2015-10-07 16:40:32.000"

}

Through this operator you can also take a value that’s already structured as a date, and transform it into a differently formatted date, for example:

Transform

%output application/json

---

{

myDate: ((payload as :string) as :date {format: "yyyyMMdd"}) as :string {format: "MM-dd-yyyy"}

}

### Coerce to Object

(':any', ':type') ⇒ ':object'

You can coerce your input into a custom object type of whatever class you want.

Transform

%dw 1.0

%output application/json

---

{

payload as :object {class : "soa.sfabs.SOAResponseInfoType\$ServiceInfo"}

}

|  |
| --- |
| Keep in mind that if the class name contains any '$' characters, they must be escaped with a backslash (\). |

## Type Of

(':any') ⇒ ':type'

Returns the type of a provided element (eg: '":string"' , '":number"' )

Transform

%dw 1.0

%output application/json

---

isString: typeOf payload.mystring

Input

{

"mystring":"a string"

}

Output

{

"isString": ":string"

}

## Flatten

(':array') ⇒ ':array'

If you have an array of arrays, this function can flatten it into a single simple array.

Transform

%dw 1.0

%output application/json

---

flatten payload

Input

[

[3,5],

[9,5],

[154,0.3]

]

Output

[

3,

5,

9,

5,

154,

0.3

]

## Size Of

(':array')/(':string')/(':object') ⇒ ':number'

Returns the number of elements in an array (or anything that can be converted to an array such as a string).

Transform

%dw 1.0

%output application/json

---

{

arraySize: sizeOf [1,2,3],

textSize: sizeOf "MuleSoft",

objectSize: sizeOf {a:1,b:2}

}

Output

{

"arraySize": 3,

"textSize": 8,

"objectSize": 2

}

## Array Push

(:array', ':any') ⇒ ':array'

Pushes a new element to the end of an array.

Transform

%dw 1.0

%output application/json

---

aa: [0, 1, 2] + 5

Output

{

"aa": [0, 1, 2, 5]

}

## Remove from Array

(':array', ':any') ⇒ ':array'

Removes an element from an array when it matches the specified value. If multiple elements in the array match the value, they will all be removed.

Transform

%dw 1.0

%output application/json

---

{

a: [0, 1, 1, 2] - 1,

b: [{a: "a"}] - {a: "a"}

}

Output

{

"a": [0,2],

"b": []

}

## Remove Matching from Array

(':array', ':array') ⇒ ':array'

Removes a set of elements from an array when an element in the base array matches one of the values in the substracted array. If multiple elements in the array match a value, they will all be removed.

Transform

%dw 1.0

%output application/json

---

a: [0, 1, 1, 2] -- [1,2]

Output

{

"a": [0],

}

## Average of Array

(':array') ⇒ ':number'

Creates an average of all the values in an array and outputs a single number. The array must of course contain only numerical value in it.

Transform

%dw 1.0

%output application/json

---

{

a: avg [1..1000],

b: avg [1, 2, 3]

}

Output

{

"a": 500.5,

"b": 2.0

}

## Reduce

(':array', ':function') ⇒ ':any'

Apply a reduction to the array using just two parameters: the accumulator (**$$**), and the value (**$**). By default, the accumulator starts at the first value of the array.

Transform

%dw 1.0

%output application/json

---

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

Output

{

"sum": 15

}

Transform

%dw 1.0

%output application/json

---

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

Output

{

"concat": "abcd"

}

In some cases, you may not want to use the first element of the array as an accumulator. To set the accumulator to something else, you must define this in a lambda.

Transform

%dw 1.0

%output application/json

---

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

Output

{

"concat": "zabcd"

}

## Join By

(':array', ':string') ⇒ ':string'

Merges an array into a single string value, using the provided string as a separator between elements.

Transform

%dw 1.0

%output application/json

---

aa: ["a","b","c"] joinBy "-"

Output

{

"aa": "a-b-c"

}

## Split By

(':string', ':string')/(':string', ':regex') ⇒ ':array'

Performs the opposite operation as Join By. It splits a string into an array of separate elements, looking for instances of the provided string and using it as a separator.

Transform

%dw 1.0

%output application/json

---

split: "a-b-c" splitBy "-"

Output

{

"split": ["a","b","c"]

}

## Order By

(':array', ':function')/(':object', ':function') ⇒ ':array'/':object'

Returns the provided array (or object) ordered according to the value returned by the lambda. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

---

orderByLetter: [{ letter: "d" }, { letter: "e" }, { letter: "c" }, { letter: "a" }, { letter: "b" }] orderBy $.letter

Output

{

"orderByLetter": [

{

"letter": "a"

},

{

"letter": "b"

},

{

"letter": "c"

},

{

"letter": "d"

},

{

"letter": "e"

}

]

}

|  |
| --- |
| The **orderBy** function doesn’t have an option to order in descending order instead of ascending. What you can do in these cases is simply invert the order of the resulting array.  Transform  %dw 1.0  %output application/json  ---  orderDescending: ([3,8,1] orderBy $)[-1..0]  Output  { "orderDescending": [8,3,1] } |

## Group By

(':array', ':function') ⇒ ':object'

Partitions an array into a Object that contains Arrays, according to the discriminator lambda you define. The lambda is invoked with two parameters: **index** and the **value**. If these parameters are not named, the index is defined by default as **$$** and the value as **$**.

Transform

%dw 1.0

%output application/json

---

"language": payload.langs groupBy $.language

Input

{

"langs": [

{

"name": "Foo",

"language": "Java"

},

{

"name": "Bar",

"language": "Scala"

},

{

"name": "FooBar",

"language": "Java"

}

]

}

Output

{

"language": {

"Scala": [

{"name":"Bar", "language":"Scala"}

],

"Java": [

{"name":"Foo", "language":"Java"},

{"name":"FooBar", "language":"Java"}

]

}

}

## Distinct By

(':array', ':function') ⇒ ':array'

Returns only unique values from an array that may have duplicates. The lambda is invoked with two parameters: **index** and **value**. If these parameters are not defined, the index is defined by default as $$ and the value as $.

Transform

%dw 1.0

%output application/json

---

{

book : {

title : payload.title,

year: payload.year,

authors: payload.author distinctBy $

}

}

Input

{

"title": "XQuery Kick Start",

"author": [

"James McGovern",

"Per Bothner",

"Kurt Cagle",

"James Linn",

"Kurt Cagle",

"Kurt Cagle",

"Kurt Cagle",

"Vaidyanathan Nagarajan"

],

"year":"2000"

}

Output

{

"book": {

"title": "XQuery Kick Start",

"year": "2000",

"authors": [

"James McGovern",

"Per Bothner",

"Kurt Cagle",

"James Linn",

"Vaidyanathan Nagarajan"

]

}

}

## Zip Arrays

(':array', ':array') ⇒ ':array'

Given two or more separate lists, the zip function can be used to merge them together into a single list of consecutive n-tuples. Imagine two input lists each being one side of a zipper: similar to the interlocking teeth of a zipper, the zip function interdigitates each element from each input list, one element at a time.

Transform

%dw 1.0

%output application/json

---

{

a: [0, 1, 2, 3] zip ["a", "b", "c", "d"],

b: [0, 1, 2, 3] zip "a",

c: [0, 1, 2, 3] zip ["a", "b"]

}

Output

{

"a": [

[0,"a"],

[1,"b"],

[2,"c"],

[3,"d"]

],

"b": [

[0,"a"],

[1,"a"],

[2,"a"],

[3,"a"]

],

"c": [

[0,"a"],

[1,"b"]

]

}

Note that in example b, since only one element was provided in the second array, it was matched with every element of the first array. Also note that in example c, since the second array was shorter than the first, the output was only as long as the shortest of the two.

Here is another example of the zip function with more than two input lists.

Transform

%dw 1.0

%output application/json

---

payload.list1 zip payload.list2 zip payload.list3

Input

{

"list1": ["a", "b", "c", "d"],

"list2": [1, 2, 3],

"list3": ["aa", "bb", "cc", "dd"],

"list4": [["a", "b", "c"], [1, 2, 3, 4], ["aa", "bb", "cc", "dd"]]

}

Output

[

[

"a",

1,

"aa"

],

[

"b",

2,

"bb"

],

[

"c",

3,

"cc"

]

]

## Unzip Array

(':array') ⇒ ':array'

Performs the opposite function of [[zip arrays]](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#zip arrays), that is: given a single array where each index contains an array with two elements, it outputs two separate arrays, each with one of the elements of the pair. This can also be scaled up, if the indexes in the provided array contain arrays with more than two elements, the output will contain as many arrays as there are elements for each index.

Transform

%dw 1.0

%output application/json

---

{

a: unzip [[0,"a"],[1,"b"],[2,"c"],[3,"d"]],

b: unzip [ [0,"a"], [1,"a"], [2,"a"], [3,"a"]],

c: unzip [ [0,"a"], [1,"a","foo"], [2], [3,"a"]]

}

Output

{

"a":[

[0, 1, 2, 3],

["a", "b", "c", "d"]

],

"b": [

[0,1,2,3],

["a","a","a","a"]

],

"c": [

[0,1,2,3]

]

}

Note even though example b can be considered the inverse function to the example b in [[zip array]](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#zip array), the result is not analogous, since it returns an array of repeated elements instead of a single element. Also note that in example c, since the number of elements in each component of the original array is not consistent, the output only creates as many full arrays as it can, in this case just one.

## Replace

(':string', ':regex', ':function') ⇒ ':string'

Replaces a section of a string for another, in accordance to a regular expression, and returns a modified string.

Transform

%dw 1.0

%output application/json

---

b: "admin123" replace /(\d+)/ with "ID"

Output

{

"b": "adminID"

}

## Matches

(':string', ':regex') ⇒ ':boolean'

Matches a string against a regular expression, and returns **true** or **false**.

Transform

%dw 1.0

%output application/json

---

b: "admin123" matches /(\d+)/

Output

{

"b": false

}

|  |
| --- |
| For more advanced use cases where you need to output or conditionally process the matched value, see [Pattern Matching](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#pattern-matching). |

## Starts With

(':string', ':string') ⇒ ':boolean'

Returns true or false depending on if a string starts with a provided substring.

Transform

%dw 1.0

%output application/json

---

{

a: "Mariano" startsWith "Mar",

b: "Mariano" startsWith "Em"

}

Output

{

"a": true,

"b": false

}

## Ends With

(':string', ':string') ⇒ ':boolean'

Returns true or false depending on if a string ends with a provided substring.

Transform

%dw 1.0

%output application/json

---

{

a: "Mariano" endsWith "no",

b: "Mariano" endsWith "to"

}

Output

{

"a": true,

"b": false

}

## Find

(':string', ':string')/.(':string', ':regex') ⇒ ':array'

Given a string, it returns the index position within the string at which a match was matched. If found in multiple parts of the string, it returns an array with the various idex positions at which it was found. You can either look for a simple string or a regular expression.

Transform

%dw 1.0

%output application/json

---

{

a: "aabccde" find /(a).(b)(c.)d/,

b: "aabccdbce" find "a",

c: "aabccdbce" find "bc"

}

Output

{

"a": [[0,0,2,3]],

"b": [0,1],

"c": [2,6]

}

## Match

(':string', ':regex') ⇒ ':string'

Match a string against a regular expression. Match returns an array that contains the entire matching expression, followed by all of the capture groups that match the provided regex.

It can be applied to the result of any evaluated expression, and can return any evaluated expression. See the Match operator in [the DataWeave Language Introduction](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction).

Transform

%dw 1.0

%output application/json

---

hello: "anniepoint@mulesoft.com" match /([a-z]\*)@([a-z]\*).com/

Output

{

"hello": [

"anniepoint@mulesoft.com",

"anniepoint",

"mulesoft"

]

}

In the example above, we see that the search regular expression describes an email address. It contains two capture groups, what’s before and what’s after the @. The result is an array of three elements: the first is the whole email address, the second matches one of the capture groups, the third matches the other one.

## Scan

(':string', ':regex') ⇒ ':array'

Returns an array with all of the matches in the given string. Each match is returned as an array that contains the complete match, as well as any capture groups there may be in your regular expression.

Transform

%dw 1.0

%output application/json

---

hello: "anniepoint@mulesoft.com,max@mulesoft.com" scan /([a-z]\*)@([a-z]\*).com/

Output

{

"hello": [

[

"anniepoint@mulesoft.com",

"anniepoint",

"mulesoft"

],

[

"max@mulesoft.com",

"max",

"mulesoft"

]

]

}

In the example above, we see that the search regular expression describes an email address. It contains two capture groups, what’s before and what’s after the @. The result is an array with two matches, as there are two email addresses in the input string. Each of these matches is an array of three elements, the first is the whole email address, the second matches one of the capture groups, the third matches the other one.

## Similar

(':any', ':any') ⇒ ':boolean'

Evaluates if two values are similar, regardless of their type. For example, the string "1234" and the number 1234 aren’t equal, but they are recognized as similar.

Transform

%dw 1.0

%output application/json

---

{

a: "1234" == 1234,

b: "1234" ~= 1234,

c: "true" == true,

d: "true" ~= true

}

Output

{

"a": false,

"b": true,

"c": false,

"d": true

}

## Upper

(':string') ⇒ ':string'

Returns the provided string in uppercase characters.

Transform

%dw 1.0

%output application/json

---

{

name: upper "mulesoft"

}

Output

{

"name": MULESOFT

}

## Lower

(':string') ⇒ ':string'

Returns the provided string in lowercase characters.

Transform

%dw 1.0

%output application/json

---

{

name: lower "MULESOFT"

}

Output

{

"name": mulesoft

}

## Camelize

(':string') ⇒ ':string'

Returns the provided string in camel case.

Transform

%dw 1.0

%output application/json

---

{

a: camelize "customer",

b: camelize "customer\_first\_name",

c: camelize "customer name"

}

Output

{

"a": "customer",

"b": "customerFirstName",

"c": "customer name"

}

## Capitalize

(':string') ⇒ ':string'

Returns the provided string with every word starting with a capital letter and no underscores. It also replaces underscores with spaces and puts a space before each capitalized word.

Transform

%dw 1.0

%output application/json

---

{

a: capitalize "customer",

b: capitalize "customer\_first\_name",

c: capitalize "customer NAME",

d: capitalize "customerName",

}

Output

{

"a": "Customer",

"b": "Customer First Name",

"c": "Customer Name",

"d": "Customer Name"

}

## Dasherize

(':string') ⇒ ':string'

Returns the provided string with every word separated by a dash.

Transform

%dw 1.0

%output application/json

---

{

a: dasherize "customer",

b: dasherize "customer\_first\_name",

c: dasherize "customer NAME"

}

Output

{

"a": "customer",

"b": "customer-first-name",

"c": "customer-name"

}

## Underscore

(':string') ⇒ ':string'

Returns the provided string with every word separated by an underscore.

Transform

%dw 1.0

%output application/json

---

{

a: underscore "customer",

b: underscore "customer-first-name",

c: underscore "customer NAME"

}

Output

{

"a": "customer",

"b": "customer\_first\_name",

"c": "customer\_NAME"

}

## Pluralize

(':string') ⇒ ':string'

Returns the provided string transformed into its plural form.

Transform

%dw 1.0

%output application/json

---

{

a: pluralize "box",

b: pluralize "wife",

c: pluralize "foot"

}

Output

{

"a": "boxes",

"b": "wives",

"c": "feet"

}

## Singularize

(':string') ⇒ ':string'

Returns the provided string transformed into its singular form.

Transform

%dw 1.0

%output application/json

---

{

a: singularize "boxes",

b: singularize "wives",

c: singularize "feet"

}

Output

{

"a": "box",

"b": "wife",

"c": "foot"

}

## Trim

(':string') ⇒ ':string'

Removes any excess spaces at the start and end of a string.

Transform

%dw 1.0

%output application/json

---

{

"a": trim " my long text "

}

Output

{

"a": "my long text"

}

## Substring using ****..**** or ****to****

(':string') ⇒ ':string'

Extracts a set of characters out of a string, based on the position that the first and last character of the desired substring occupy in the character array. If you use negative numbers, you can also inverse the order in which characters are set.

Transform

%dw 1.0

%output application/json

---

{

"a": "abcdefg"[0..4],

"b": "abcdefg"[-1..-4],

"d": "abcdefg"[0 to 4],

"e": "abcdefg"[-1 to -4]

}

Output

{

"a": "abcde",

"b": "gfed",

"d": "abcde",

"e": "gfed"

}

## Ordinalize

(':number') ⇒ ':string'

Returns the provided numbers set as ordinals.

Transform

%dw 1.0

%output application/json

---

{

a: ordinalize 1,

b: ordinalize 8,

c: ordinalize 103

}

Output

{

"a": "1st",

"b": "8th",

"c": "103rd"

}

## Basic Math Operations

### Sum

Transform

%dw 1.0

%output application/xml

---

plus : 2 + 2.5

### Minus

Transform

%dw 1.0

%output application/xml

---

minus : 2.5 - 2

### Multiply

Transform

%dw 1.0

%output application/xml

---

multiply : 2.5 \* 2

### Division

Transform

%dw 1.0

%output application/xml

---

division : 10 / 2

## Max

(':array')/(':object') ⇒ ':number'

Returns the highest number in an array or object.

Transform

%dw 1.0

%output application/json

---

{

a: max [1..1000],

b: max [1, 2, 3],

d: max [1.5, 2.5, 3.5]

}

Output

{

"a": 1000,

"b": 3,

"d": 3.5

}

## Min

(':array')/(':object') ⇒ ':number'

Returns the lowest number in an array or object.

Transform

%dw 1.0

%output application/json

---

{

a: min [1..1000],

b: min [1, 2, 3],

d: min [1.5, 2.5, 3.5]

}

Output

{

"a": 1,

"b": 1,

"d": 1.5

}

## Round

(':number') ⇒ ':number'

Rounds the value of a number to the nearest integer

Transform

%dw 1.0

%output application/json

---

{

a: round 1.2,

b: round 4.6,

c: round 3.5

}

Output

{

"a": 1,

"b": 5,

"c": 4

}

## Sqrt

(':number') ⇒ ':number'

Returns the square root of the provided number

Transform

%dw 1.0

%output application/json

---

{

a: sqrt 4,

b: sqrt 25,

c: sqrt 100

}

Output

{

"a": 2.0,

"b": 5.0,

"c": 10.0

}

## Pow

(':number', ':number') ⇒ ':number'

Returns the result of the first number a to the power of the number following the pow operator.

Transform

%dw 1.0

%output application/json

---

{

a: 2 pow 3,

b: 3 pow 2,

c: 7 pow 3

}

Output

{

"a": 8,

"b": 9,

"c": 343

}

## Ceil

(':number') ⇒ ':number'

Rounds a number upwards, returning the first full number above than the one provided.

Transform

%dw 1.0

%output application/json

---

{

a: ceil 1.5,

b: ceil 2.2,

c: ceil 3

}

Output

{

"a": 2,

"b": 3,

"c": 3

}

## Floor

(':number') ⇒ ':number'

Rounds a number downwards, returning the first full number below than the one provided.

Transform

%dw 1.0

%output application/json

---

{

a: floor 1.5,

b: floor 2.2,

c: floor 3

}

Output

{

"a": 1,

"b": 2,

"c": 3

}

## Abs

(':number') ⇒ ':number'

Returns the absolute value of a number,

Transform

%dw 1.0

%output application/json

---

{

a: abs -2,

b: abs 2.5,

c: abs -3.4,

d: abs 3

}

Output

{

"a": 2,

"b": 2.5,

"c": 3.4,

"d": 3

}

## Mod

(':number', ':number') ⇒ ':number'

Returns the remainder after division of the first number by the second one

Transform

%dw 1.0

%output application/json

---

{

a: 3 mod 2,

b: 4 mod 2,

c: 2.2 mod 2

}

Output

{

"a": 1,

"b": 0,

"c": 0.2

}

## Now

Returns a datetime object with the current date and time.

Transform

%dw 1.0

%output application/json

---

{

current\_time: now

}

Output

{

"current\_time": "2016-10-20T17:15:06.196Z"

}

## Date Time Operations

There are several operators that deal with [date related types](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#dates), which include date, time, localtime, datetime, localdatetime, period, timezone.

### Get Time Unit

(':date')/(':time')/(':localtime')/(':datetime')/(':localdatetime')/(':period') ⇒ (':date')/(':time')/(':localtime')/(':period')

You can extract a particular time unit from any date related type as shown below:

Transform

%dw 1.0

%output application/json

---

{

a: |2003-10-01|.day,

b: |2003-10-01|.month,

c: |2003-10-01|.year,

d: |2003-10-01T23:57:59Z|.hour,

e: |2003-10-01T23:57:59Z|.minutes,

f: |2003-10-01T23:57:59Z|.seconds,

g: |2003-10-01T23:57:59-03:00|.offsetSeconds,

h: |23:57:59Z|.hour,

i: |23:57:59.700|.nanoseconds,

j: |23:57:59.700|.milliseconds,

k: |2003-10-01T23:57:59Z|.dayOfWeek,

l: |2003-10-01T23:57:59Z|.dayOfYear,

m: |P3Y2M10D|.years

}

Output

{

"a": 1,

"b": 10,

"c": 2003,

"d": 23,

"e": 57,

"f": 59,

"g": -10800,

"h": 23,

"i": 700000000,

"j": 700,

"k": 3,

"l": 274,

"m": 3

}

### Shift Time Zone

(':datetime', ':timezone') ⇒ ':datetime'

Shift a date time to the specified timezone.

Transform

%dw 1.0

%output application/json

---

a: |2014-01-01T14:00-03:00| >> |-08:00|

Output

{

"a": "2014-01-01T09:00-08:00"

}

### Append Time

(':date', ':time')/(':date', ':localtime')/(':time', ':date')/(':localtime', ':date') ⇒ (':localtime')/(':datetime')/(':localdatetime')

You can append a date to a time (or localtime) object so as to provide a more precise value.

Transform

%dw 1.0

%output application/json

---

{

a: |2003-10-01| ++ |23:57:59|,

b: |2003-10-01| ++ |23:57:59Z|

}

Output

{

"a": "2003-10-01T23:57:59",

"b": "2003-10-01T23:57:59Z"

}

Note that the order in which the two objects are appended is irrelevant, so logically a ':date' + ':time' will result in the same as a '#:time' + ':date'.

### Append Time Zone

(':datetime', ':timezone')/(':time', ':timezone')/(':localtime', ':timezone')/(':localdatetime', ':timezone') ⇒ (':localtime')/(':localdatetime')

Appends a time zone to a date type value.

Transform

%dw 1.0

%output application/json

---

a: |2003-10-01T23:57:59| ++ |-03:00|

Output

{

"a": "2003-10-01T23:57:59-03:00"

}

### Adding a Period of Time

(':time', ':period')/(':datetime', ':period')/(':localtime', ':period')/(':localdatetime', ':period') ⇒ (':date')/(':time')/(':localtime')/(':datetime')/(':localdatetime')

Add or subtract a period of time from a given date or time type object.

Transform

%dw 1.0

%output application/json

---

a: |2003-10-01T23:57:59Z| + |P1Y|

Output

{

"a": "2004-10-01T23:57:59Z"

}

### Subtracting a Period of Time

(':time', ':period')/(':datetime', ':period')/(':localtime', ':period')/(':localdatetime', ':period') ⇒ (':date')/(':time')/(':localtime')/(':datetime')/(':localdatetime')

The same logically applies to subtracting time periods from a date or time type object.

Transform

%dw 1.0

%output application/json

---

{

a: |2003-10-01| - |P1Y|,

b: |2003-10-01T23:57:59Z| - |P1Y|

}

Output

{

"a": "2002-10-01",

"b": "2002-10-01T23:57:59Z"

}

Note that when a subtraction operation includes a time object and a period, the order in which both elements are placed is indiferent since it would be impossible to subtract a date from a period, so |2003-10-01| - |P1Y| returns the same as |P1Y| - |2003-10-01|.

### Subtracting two Dates

(':date', ':date')/(':datetime', ':datetime')/('#:time', ':time')/(':localtime', ':localtime')/(':localdatetime', ':localdatetime') ⇒ ':period'

When subtracting one date or time type object from another, what we logically get is the difference between these times expressed as a time period.

Transform

%dw 1.0

%output application/json

---

{

a: |23:59:56-03:00| - |22:59:56-00:00|,

b: |2003-10-01| - |2002-09-23|

}

Output

{

"a": "PT-4H",

"b": "P-1Y-8D"

}

@@

# DataWeave Value Types

The [DataWeave](https://docs.mulesoft.com/mule-runtime/3.8/dataweave) language functions essentially as a template engine that describes an output structure through the use of elements that may be of various different types.

This document covers the different types of objects you can use to construct your output.

## Array

**Type** ⇒ ':array'

Arrays are represented as a sequences of **value** expressions.

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

**Array Literal**

%dw 1.0

%output application/json

---

[ "My", "three", "words" ]

### Conditional Elements

Arrays can define conditional **values** based on a condition. Wrap the value expression between parentheses and use the **when** keyword with the condition.

Transform

%dw 1.0

%output application/json

---

[(1) when true, (2) when false]

Output

[1]

## Object

**Type** ⇒ ':object'

Objects are represented as a collection of **key**: **value** pairs.

1. Object: { 'Key' : Value }
2. Key : 'Qualified Name' 'Attributes'
3. Qualified Name: 'namespace prefix#name' where the 'namespace prefix#' part is optional
4. Name: String that represents the name.
5. Attributes: @('Qualified Name'= **value**,…​)

|  |
| --- |
| Strings must be double quoted to be recognized as strings. |

### Single Value Objects

If an Object has only one key:value pair, the enclosing curly brackets { } are not required:

Example

%dw 1.0

%output application/xml

---

name: "Annie"

### Conditional Elements

Objects can define conditional **key**: **value** pairs based on a conditional expression. Wrap the key:value expression between parentheses and use the **when** keyword with the condition.

%dw 1.0

%output application/xml

---

file: {

name: "transform",

(extension: "zip") when payload.fileSystem?

}

This example outputs an additional field called "extension" only when the fileSystem property is present in payload (this field may contain any value, not just "true").

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

<file>

<name>transform</name>

<extension>zip</extension>

</file>

If absent:

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

<file>

<name>transform</name>

</file>

### Dynamic Elements

Dynamic elements allow you to add the result of an expression as key:value pairs of an object.

Transform

%dw 1.0

%output application/json

---

{

a: "a",

(["b","c","d"] map {'$': $})

}

Output

{

"a": "a",

"b": "b",

"c": "c",

"d": "d"

}

### Dynamic keys

In order to specify a key via an expression, the expression should be wrapped in parentheses.

Transform

%dw 1.0

%output application/json

---

{name: 'Data Weave'} mapObject {(upper $$ as :string) : $}

Output

{

"NAME": "Data Weave"

}

### Conditional Attributes

Attributes can be conditional based on a given condition. Wrap the key:value expression in parentheses and use the **when** keyword with the condition.

Transform

%dw 1.0

%output application/xml

---

name @((company: "Acme") when false, (transform: "Anything") when true): "DataWeave"

Output

<?xml version='1.0' encoding='US-ASCII'?>

<name transform="Anything">DataWeave</name>

### Dynamic Attributes

Dynamic attributes allow you to add the result of an expression as key:value pairs of the attributes set.

Input

{

"company": "Mule",

"product": "DataWeave"

}

Transform

%dw 1.0

%output application/xml

---

transformation @((payload)): "Transform from anything to anything"

Output

<?xml version='1.0' encoding='US-ASCII'?>

<transformation company="Mule" product="DataWeave">Transform from anything to anything</transformation>

## String

**Type** ⇒ ':string'

A string can be defined by the use of double quotes or single quotes.

{

doubleQuoted: "Hello",

singleQuoted: 'Hello',

}

### String interpolation

String interpolation allows you to embed variables or expressions directly in a string.

Transform

%dw 1.0

%output application/json

%var name = "Shoki"

---

{

Greeting: "Hi, my name is $name",

Sum: "1 + 1 = $(1 + 1)"

}

Output

{

"Greeting": "Hi, my name is Shoki",

"Sum": "1 + 1 = 2"

}

## Number

**Type** ⇒ ':number'

There is only one number type that supports both floating point and integer numbers. There is no loss of precision in any operation, the engine always stores the data in the most performant way that doesn’t compromise precision.

## Boolean

**Type** ⇒ ':boolean'

A boolean is defined by the keywords 'true' and 'false'.

## Dates

Dates in DataWeave follow the [ISO-8601 standard](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) and are defined between '|' characters.

The date system supports:

* DateTime
* Local DateTime
* Time
* Local Time
* Period
* TimeZone (ISO-8601 timezone standard only, not the region or daylight timezone)
* Date

### Date

**Type** ⇒ ':date'

Represented as 'Year'-'Month'-'Date'

The type **Date** has no time component at all (not even midnight).

Transform

%dw 1.0

%output application/json

---

c: |2003-10-01|

Output

{

"c": "2003-10-01"

}

### Time

**Type** ⇒ ':time'

Represented as 'Hour':'Minutes':'Seconds'.'Milliseconds'

Transform

%dw 1.0

%output application/json

---

c: |23:59:56|

Output

{

"c": "23:59:56"

}

### TimeZone

**Type** ⇒ ':timeZone'

Timezones must include a + or a - to be defined as such. |03:00| is a time, |+03:00| is a timezone.

Transform

%dw 1.0

%output application/json

---

c: |-08:00|

Output

{

"c": "-08:00"

}

### DateTime

**Type** ⇒ ':datetime'

Date time is the conjunction of 'Date' + 'Time' + 'TimeZone'.

Transform

%dw 1.0

%output application/json

---

a: |2003-10-01T23:57:59-03:00|

Output

{

"a": "2003-10-01T23:57:59-03:00"

}

### Local Date Time

**Type** ⇒ ':localdatetime'

Local date time is the conjunction of 'Date' + 'Time'.

Transform

%dw 1.0

%output application/json

---

a: |2003-10-01T23:57:59|

Output

{

"a": "2003-10-01T23:57:59"

}

### Period

**Type** ⇒ ':period'

Specifies a period of time. Examples |PT9M| ⇒ 9 minutes , |P1Y| ⇒ 1 Year

Transform

%dw 1.0

%output application/json

---

a: |23:59:56| + |PT9M|

Output

{

"a": "00:08:56"

}

### Date decomposition

In order to access the different parts of the date, special selectors must be used.

Transform

%dw 1.0

%output application/json

---

{

day: |2003-10-01T23:57:59Z|.day,

month: |2003-10-01T23:57:59Z|.month,

year: |2003-10-01T23:57:59Z|.year,

hour: |2003-10-01T23:57:59Z|.hour,

minutes: |2003-10-01T23:57:59Z|.minutes,

seconds: |2003-10-01T23:57:59Z|.seconds,

offsetSeconds: |2003-10-01T23:57:59-03:00|.offsetSeconds,

nanoseconds: |23:57:59.700|.nanoseconds,

milliseconds: |23:57:59.700|.milliseconds,

dayOfWeek: |2003-10-01T23:57:59Z|.dayOfWeek,

dayOfYear: |2003-10-01T23:57:59Z|.dayOfYear

}

Output

{

"day": 1,

"month": 10,

"year": 2003,

"hour": 23,

"minutes": 57,

"seconds": 59,

"offsetSeconds": -10800,

"nanoseconds": 700000000,

"milliseconds": 700,

"dayOfWeek": 3,

"dayOfYear": 274

}

### Changing the Format of a Date

You can specify a date to be in any format you prefer through using **as** in the following way:

Transform

%dw 1.0

%output application/json

---

formatedDate: |2003-10-01T23:57:59| as :string {format: "YYYY-MM-dd"}

Output

{

"formatedDate": "2003-10-01"

}

If you are doing multiple similar conversions in your transform, you might want to define a custom type as a directive in the header and set each date as being of that type.

Transform

%dw 1.0

%output application/json

%type mydate = :string { format: "YYYY/MM/dd" }

---

{

formatedDate1: |2003-10-01T23:57:59| as :mydate,

formatedDate2: |2015-07-06T08:53:15| as :mydate

}

Output

{

"formatedDate1": "2003/10/01",

"formatedDate2": "2015/07/06"

}

## Regular Expression

**Type** ⇒ ':regex'

Regular Expressions are defined between /. For example /(\d+)/ for represents multiple numerical digits from 0-9. These may be used as arguments in certain operations that act upon strings, like Matches or Replace, or on operations that act upon objects and arrays, such as filters.

## Iterators

**Type** ⇒ ':iterator'

This type is based in the [iterator Java class](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html). The iterator contains a collection, and includes methods to iterate through and filter it.

|  |
| --- |
| Just like the Java class, the iterator is designed to be consumed only once. For example, if you then pass this value to a [logger](https://docs.mulesoft.com/mule-runtime/3.8/logger-component-reference) would result in consuming it and it would no longer be readable to further elements in the flow. |

## Custom Types

You can define your own custom types in the header of your transform, then in the body you can define an element as being of that type.

To do so, the directive must be structured as following: %type name = java definition

For example:

%dw 1.0

%type currency = :number { format: "##"}

%type user = :object { class: “my.company.User”}

|  |
| --- |
| Usually it’s a good idea to extend an existing type rather than creating one from scratch.  For example, above :string defines currency as extending the string type. |

To then assign an element as being of the custom type you defined, use the operation as :type after defining a field:

%dw 1.0

%type currency = :number { format: "##"}

%type user = :object { class: “my.company.User”}

---

customer:payload.user as :user

### Defining Types For Type Coercion

#### Format

The metadata 'format' key is used for formatting numbers and dates.

Input

<items>

<item>

<price>22.30</price>

</item>

<item>

<price>20.31</price>

</item>

</items>

Transform

%dw 1.0

%output application/json

%type currency = :number { format: "##"}

---

books: payload.items.\*item map

book:

price: $.price as :currency

Output

{

"books": [

{

"book": {

"price": 22.30

}

},

{

"book": {

"price": 20.31

}

}

]

}

In Anypoint Studio, you can define several more values, like separators, quote characters and escape characters. See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task).

## Functions and Lambdas

**type** → :function

In DataWeave, function and lambdas (anonymous functions) are first-class citizen and they can be used inside operators such as a map, mapObject, etc, and can even be assigned to a variable. When using lambdas within the body of a DataWeave file in conjunction with an operator such as [map operator](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map), its attributes can either be explicitly named or left anonymous, in which case they can be referenced as $, $$, etc.

### Assign to a var

You can define a function as a variable with a [constant directive](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-language-introduction#define-constant-directive) through '%var'

Transport

%dw 1.0

%output application/json

%var toUser = (user) -> {firstName: user.givenName, lastName: user.sn}

---

{

"user" : toUser({ givenName : "Annie", sn : "Point" })

}

Output

{

"user": {

"firstName": "Annie",

"lastName": "Point"

}

}

### Named attributes with an Operator

This example uses a lambda with an attribute that’s explicitly named as 'name'.

Input

%dw 1.0

%output application/json

---

users: ["john", "peter", "matt"] map ((name) -> upper name)

Transform

{

"users": ["JOHN","PETER","MATT"]

}

### Anonymous attributes with an Operator

This example uses a lambda with an attribute that’s not explicitly named, and so is referred to by default as '$'.

Transform

%dw 1.0

%output application/json

---

users: ["john", "peter", "matt"] map upper $

Output

{

"users": ["JOHN","PETER","MATT"]

}

### Declare using function directive

You can declare functions in the Header and these can be invoked at any point in the Body, you can also declare functions anywhere in the body. You refer to them using the form **function-name()** passing an expression in between the parentheses for each necessary argument. Each expression between the parentheses is evaluated and the result is passed as an argument used in the execution of the function body.

Transform

%dw 1.0

%output application/json

%function toUser(user){firstName: user.givenName, lastName: user.sn}

---

{

"user" : toUser({ givenName : "Annie", sn : "Point" })

}

Output

{

"user": {

"firstName": "Annie",

"lastName": "Point"

}

}

## Operators Sorted by Type

Below is an index that includes all of the different operators in DataWeave, sorted by the types of the parameters it accepts. Each operator displays what type is accepted on each of its arguments, not all arguments are required.

|  |
| --- |
| When you provide an operator with properties that don’t match the expected types, DataWeave automatically [attempts to coerce](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#type-coercion-table) the provided property to the required type. |

### Operations Performed on any Type

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [typeOf](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#typ-Of) | (':any') |
| [as](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#as-(type-coercion)) | (':any', ':type') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#push) | (':any', ':array') |

### Operations Performed on ':number'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#sum) | (':number', ':number') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#minus) | (':number', ':number') |
| [\*](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#multiply) | (':number', ':number') |
| [/](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#division) | (':number', ':number') |
| [round](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#round) | (':number') |
| [sqrt](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#sqrt) | (':number') |
| [pow](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#pow) | (':number', ':number') |
| [ceil](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#ceil) | (':number') |
| [floor](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#floor) | (':number') |
| [abs](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#abs) | (':number') |
| [mod](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#mod) | (':number', ':number') |
| [ordinalize](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#ordinalize) | (':number') |

### Operations Performed on ':array'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [min](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#min) | (':array') |
| [max](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#max) | (':array') |
| [sizeOf](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#size-of) | (':array') |
| [sum](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#sum) | (':array') |
| [flatten](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#flatten) | (':array') |
| [orderBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#order-by) | (':array', ':function') |
| [reduce](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#reduce) | (':array', ':function') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#array-push) | (':array', ':any') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#array-remove) | (':array', ':any') |
| [--](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#array-remove-all) | (':array', ':array') |
| [map](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map) | (':array', ':function') |
| [avg](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#array-average) | (':array') |
| [filter](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#filter) | (':array', ':function') |
| [contains](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#contains) | (':array', ':any') |
| [distinctBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#distinct-by) | (':array', ':function') |
| [joinBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#join-by) | (':array', ':string') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#using-oncat-on-an-array) | (':array', ':array') |
| [groupBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#group-by) | (':array', ':function') |
| [zip](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#zip-arrays) | (':array', ':array') |
| [unzip](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#unzip-array) | (':array') |

### Operations Performed on ':string'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [trim](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#trim) | (':string') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#using-concat-on-a-string) | (':string', ':string') |
| [sizeOf](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#size-of) | (':string') |
| [capitalize](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#capitalize) | (':string') |
| [lower](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#lower) | (':string') |
| [upper](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#upper) | (':string') |
| [camelize](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#camelize) | (':string') |
| [dasherize](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#dasherize) | (':string') |
| [underscore](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#underscore) | (':string') |
| [singularize](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#singularize) | (':string') |
| [pluralize](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#pluralize) | (':string') |
| [splitBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#split-by) | (':string', ':string') |
| [splitBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#split-by) | (':string', ':regex') |
| [find](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#find) | (':string', ':regex') |
| [find](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#find) | (':string', ':string') |
| [replace](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#replace) | (':string', ':regex', ':function') |
| [startsWith](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#starts-with) | (':string', ':string') |
| [endsWith](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#ends-with) | (':string', ':string') |
| [match](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#match) | (':string', ':regex') |
| [matches](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#matches) | (':string', ':regex') |
| [scan](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#scan) | (':string', ':regex') |
| [contains](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#contains) | (':string', ':string') |
| [contains](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#contains) | (':string', ':regex') |

### Operations Performed on ':object'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [sizeOf](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#size-of) | (':object') |
| [orderBy](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#order-by) | (':object', ':function') |
| [map](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#using-map-on-an-object) | (':object', ':function') |
| [mapObject](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#map-object) | (':object', ':function') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#using-oncat-on-an-object) | (':object', ':object') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#using-remove-on-an-object) | (':object', ':name') |
| [--](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#remove-by-matching-key-and-value) | (':object', ':object') |
| [pluck](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#pluck) | (':object', ':function') |

### Operations Performed on ':datetime'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [>>](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#shift-time-zone) | (':datetime', ':timezone') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':datetime', ':period') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#subtracting-a-period-of-time) | (':datetime', ':period') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#subtracting-two-dates) | (':datetime', ':datetime') |

### Operations Performed on ':date'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#subtracting-two-dates) | (':date', ':date') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time) | (':date', ':localtime') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time) | (':date', ':time') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time-zone) | (':date', ':timezone') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':date', ':period') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':date', ':period') |

### Operations Performed on ':time'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#subtracting-two-dates) | ('#:time', ':time') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time) | ('#:time', ':date') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':time', ':period') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':time', ':period') |

### Operations Performed on ':localtime'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#subtracting-two-dates) | (':localtime', ':localtime') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time) | (':localtime', ':date') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time-zone) | (':localtime', ':timezone') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':localtime', ':period') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':localtime', ':period') |

### Operations Performed on ':localdatetime'

| **Operator** | **Accepted types for each argument** |
| --- | --- |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#subtracting-two-dates) | (':localdatetime', ':localdatetime') |
| [++](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#append-time-zone) | (':localdatetime', ':timezone') |
| [+](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':localdatetime', ':period') |
| [-](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#adding-a-period-of-time) | (':localdatetime', ':period') |

## Type Coercion Table

In DataWeave, types can be coerced from one type to other using the [AS Operator](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-operators#as). This table shows the possible combinations and the properties from the schema that are used in the transformation.

|  |  |  |  |
| --- | --- | --- | --- |
| When you [provide an operator](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#DataWeave Operators Sorted by Type) with properties that don’t match the expected types, DataWeave automatically attempts to coerce the provided property to the required type. | | | |
| Source | Target | Property |
| :object | :array | (1) |
| :range | :array |  |
| :number | :binary |  |
| :string | :binary |  |
| :string | :boolean |  |
| :number | :datetime | unit |
| :localdatetime | :datetime |  |
| :string | :datetime | format / locale (3) |
| :datetime | :localdate |  |
| :localdatetime | :localdate |  |
| :string | :localdate | format / locale (3) |
| :datetime | :localdatetime |  |
| :string | :localdatetime | format / locale (3) |
| :datetime | :localtime |  |
| :localdatetime | :localtime |  |
| :time | :localtime |  |
| :string | :localtime | format / locale (3) |
| :datetime | :number | unit |
| :string | :number | format / locale (3) |
| :array | :object |  |
| :string | :period |  |
| :string | :regex |  |
| :datetime | :string | format / locale (3) |
| :localdatetime | :string | format / locale (3) |
| :localtime | :string | format / locale (3) |
| :localdate | :string | format / locale (3) |
| :timetype | :string | format / locale (3) |
| :period | :string |  |
| :timezone | :string |  |
| :number | :string | format / locale (3) |
| :boolean | :string |  |
| :range | :string | (2) |
| :type | :string |  |
| :trait | :string |  |
| :datetime | :time |  |
| :localdatetime | :time |  |
| :localtime | :time |  |
| :string | :time | format |
| :datetime | :timezone |  |
| :time | :timezone |  |
| :string | :timezone |  |

(1) Returns and array with all the values of the object. (2) Returns a string with all the values of the range using "," as the separator (3) To make the locale value explicit, you can use language, country or both, as detailed in [Locale.forLanguageTag(String)](https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html#forLanguageTag-java.lang.String-).

@@@

# Formats

DataWeave supports different types of data formats. Each format has an associated reader and a writer, and in some cases custom types are allowed. Each reader and writer defines configuration properties for customization.

## Java

### Canonical Model

The mapping between Java objects to DataWeave types is quite simple:

| **Java Type** | **DataWeave Type** |
| --- | --- |
| Collections/Array/Iterator/Iterable | [:array](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#array) |
| String/CharSequence/Char/Enum/Class | [:string](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#string) |
| int/Short/Long/BigInteger/Flat/Double/BigDecimal | [:number](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#number) |
| Calendar/XmlGregorainCalendar | [:datetime](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#datetime) |
| TimeZone | [:timezone](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#timezone) |
| sql.Date/util.Date | [:date](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#date) |
| Bean/Map | [:object](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#object) |
| InputStream/Array[Byte] | [:binary](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#binary) |
| java.lang.Boolean | [:boolean](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-types#boolean) |

### Custom Types

### Metadata property **class**

Java developers use the 'class' metadata key as hint for what class needs to be created and sent as an input. If this is not explicitly defined, DataWeave tries to infer from the context or it assigns it the default values:

* java.util.HashMap for **objects**
* java.util.ArrayList for **lists**

Transform

%dw 1.0

%type user = :object { class: "com.anypoint.df.pojo.User"}

%output application/xml

---

{

name : "Mariano",

age : 31

} as :user

The above code defines the type of the required input as an instance of 'com.anypoint.df.pojo.User'.

### :enum

In order to put an enum value in a java.util.Map, the DataWeave java module defines a custom type called :enum. It allows you to specify that a given string should be handled as the name of a specified enum type. It should always be used with the class property with the java class name of the enum.

### Defining a Metadata Type

In the Transform Message component, you can define a Java type through the following methods:

* By Providing a sample object

|  |
| --- |
| See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task). |

## CSV

### Canonical Model

CSV content is modeled in DataWeave as a list of objects, where every record is an object and every field in it is a property. For example:

Input

Name,Last Name

Mariano, De achaval

Leandro, Shokida

DataWeave representation

[

{

Name: "Mariano",

"Last Name": " De achaval"

},

{

Name: "Leandro",

"Last Name": " Shokida"

}

]

### Reader Properties

In CSV you can assign any special character as the indicator for separating fields, toggling quotes, or escaping quotes. Make sure you know what special characters are being used in your input, so that DataWeave can interpret it correctly.

When defining an input of type CSV, there are a few optional parameters you can add in the XML definition of your Mule project to customize how the data is parsed.

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| separator | char | , | Character that separates one field from another |
| quote | char | " | Character that delimits the field values |
| escape | char | \ | Character used to escape occurrences of the separator or quote character within field values |
| bodyStartLineNumber | number | 0 | The line number where the body starts. |
| ignoreEmptyLine | bool | true | defines if empty lines are ignored |
| header | bool | true | Indicates if the first line of the output shall contain field names |
| headerLineNumber | number | 0 | the line number where the header is located |

|  |
| --- |
| When header=true you can then access the fields within the input anywhere by name. Ex: payload.userName.  When header=false you must access the fields by index, referencing first the entry and then the field, Ex: payload[107][2] |

These properties can be either set via the XML of your Mule project:

<dw:transform-message metadata:id="33a08359-5085-47d3-aa5f-c7dd98bb9c61"

doc:name="Transform Message">

<dw:input-payload

<!-- Boolean that defines if the first line in the data contains headers -->

<dw:reader-property name="header" value="false" />

<!-- Character that separates fields, `','` by default -->

<dw:reader-property name="separator" value="," />

<!-- Character that defines quoted text, `" "` by default -->

<dw:reader-property name="quote" value="&quot;" />

<!-- Character that escapes quotes, `\` by default -->

<dw:reader-property name="escape" value="\" />

</dw:input-payload>

<dw:set-payload>

<![CDATA[

%dw 1.0

%output application/java

---

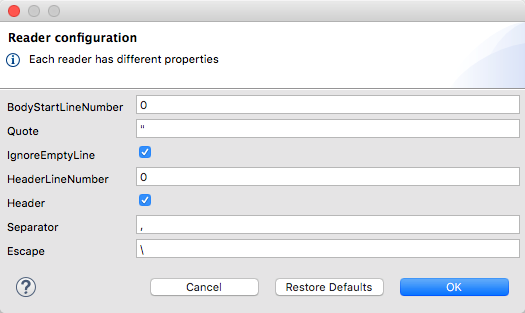
// Your transformation script goes here

]]>

</dw:set-payload>

</dw:transform-message>

Or via the UI of the Transform Message component:



See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task).

### Writer Properties

When defining an output of type CSV, there are a few optional parameters you can add to the output directive to customize how the data is parsed:

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| separator | char | , | Character that separates one field from another |
| encoding | string |  | The character set to be used for the output |
| quote | char | " | Character that delimits the field values |
| escape | char | \ | Character used to escape occurrences of the separator or quote character within field values |
| lineSeparator | string | system line ending default | line separator to be used. Example: "\r\n" |
| header | bool | true | Indicates if the first line of the output shall contain field names |
| quoteHeader | bool | false | Indicates header values should be quoted |
| quoteValues | bool | false | Indicates if every value should be quoted whether or not it contains special characters within |

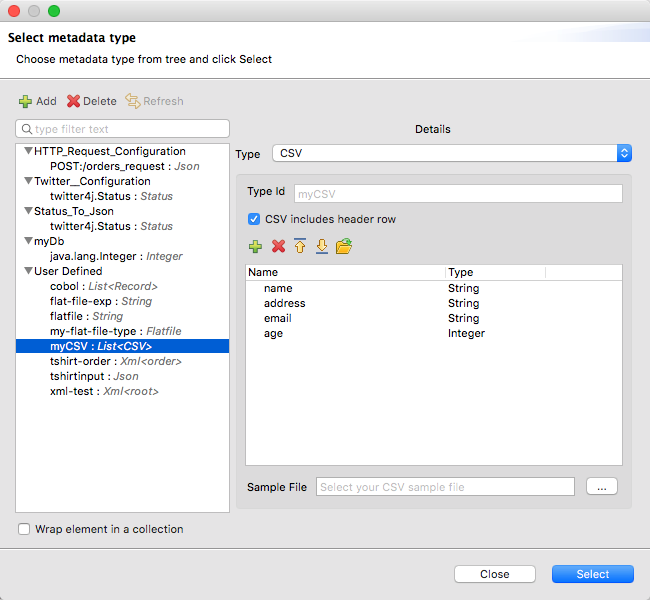
All of these parameters are optional. A CSV output directive might for example look like this:

%output text/csv separator=";", header=false, quoteValues=true

### Defining a Metadata Type

In the Transform Message component, you can define a CSV type through the following methods:

* By Providing a sample file
* Via a graphical editor that allows you to set up each field manually



|  |
| --- |
| See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task). |

## Excel

### Canonical Model

An excel workbook is a sequence of sheets, in DataWeave this is mapped to an object where each sheet is a key. Only one table is allowed per excel sheet. A table is expressed as an array of rows. A row is an object where its keys are the columns and the values the cell content.

For example:

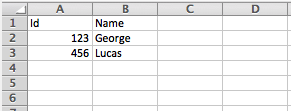


Figure 1. Input

DataWeave representation

%output application/xlsx header=true

---

{

Sheet1: [

{

Id: 123,

Name: George

},

{

Id: 456,

Name: Lucas

}

]

}

### Reader Properties

When defining an input of type excel, there are a few optional parameters you can add in the XML definition of your Mule project to customize how the data is parsed.

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| header | bool | true | defines if the excel tables contain headers. When set to false, column names are used. (A, B, C, …​) |
| ignoreEmptyLine | bool | true | defines if empty lines are ignored |
| tableOffset | string | A1 | The position of the first cell of the tables |

These properties can be either set via the XML of your Mule project:

<dw:transform-message metadata:id="33a08359-5085-47d3-aa5f-c7dd98bb9c61"

doc:name="Transform Message">

<dw:input-payload

<!-- Boolean that defines if the first line in the data contains headers -->

<dw:reader-property name="header" value="true" />

<!-- Boolean that defines if empty lines are ignored -->

<dw:reader-property name="ignoreEmptyLine" value="false" />

<!-- Defines that defines what cell to start reading from. In this case Column A is ignored, and all rows above 9 -->

<dw:reader-property name="tableOffset" value="B9" />

</dw:input-payload>

<dw:set-payload>

<![CDATA[

%dw 1.0

%output application/java

---

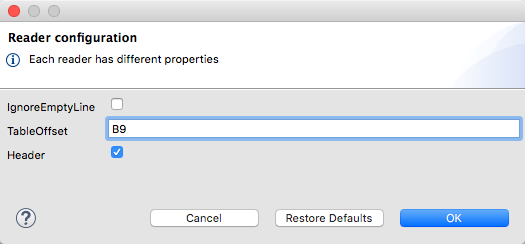
// Your transformation script goes here

]]>

</dw:set-payload>

</dw:transform-message>

Or via the UI of the Transform Message component:



See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task).

### Writer Properties

When defining an output of type excel, there are a few optional parameters you can add to the output directive to customize how the data is parsed:

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| header | bool | true | defines if the excel tables contain headers. When there are no headers, column names are used. (A, B, C, …​) |
| ignoreEmptyLine | bool | true | defines if empty lines are ignored |
| tableOffset | string | A1 | The position of the first cell of the tables |

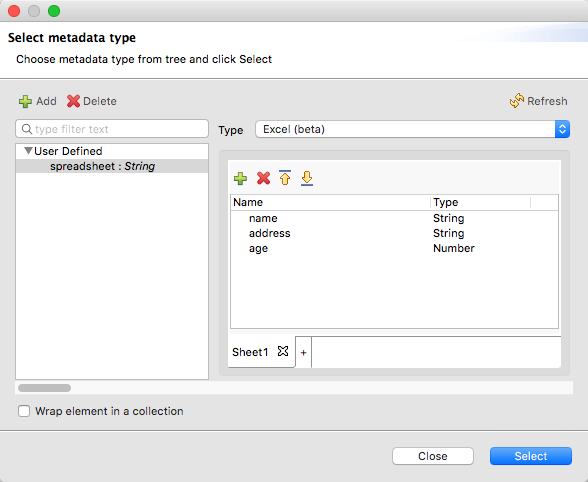
All of these parameters are optional. An excel output directive might for example look like this:

%output application/xlsx header=true

### Defining a Metadata Type

In the Transform Message component, you can define a excel type through the following methods:

* Via a graphical editor that allows you to set up each field manually



|  |
| --- |
| See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task). |

## XML

### Canonical Model

The XML data-structure is mapped to DataWeave objects that may contain other objects as values to their keys. Repeated keys are supported. For example:

Input

<users>

<company>MuleSoft</company>

<user name="Leandro" lastName="Shokida"/>

<user name="Mariano" lastName="Achaval"/>

</users>

DataWeave representation

{

users: {

company: "MuleSoft",

user @(name: "Leandro",lastName: "Shokida"): "",

user @(name: "Mariano",lastName: "Achaval"): ""

}

}

### Reader Properties

When defining an input of type XML, there are a few optional parameters you can add in the XML definition of your Mule project to customize how the data is parsed.

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| optimizeFor | string | speed | specifies the strategy to be used by the reader. Posible values = memory/speed |
| nullValueOn | string | 'empty' | If a tag with empty or blank text should be read as null. |
| indexedReader | boolean | true | Picks which reader modality to use. The indexed reader is faster but uses up a greater amount of memory, whilst the unindexed reader is slower but uses up less memory |
| maxEntityCount | integer | 1 | Limits the number of times that an entity can be referenced within the XML code. This is included to guard against [denial of service attacks](https://en.wikipedia.org/wiki/Billion_laughs). |
| externalEntities | boolean | false | Defines if references to entities that are defined in a file outside the XML are accepted as valid. It’s recommended to avoid these for [security reasons](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing) as well. |

These properties can be either set via the XML of your Mule project:

<dw:transform-message metadata:id="33a08359-5085-47d3-aa5f-c7dd98bb9c61"

doc:name="Transform Message">

<dw:input-payload>

<!-- specifies the strategy to be used by the reader -->

<dw:reader-property name="optimizeFor" value="speed" />

<!-- If a tag with empty or blank text should be read as null. -->

<dw:reader-property name="nullValueOn" value="empty" />

</dw:input-payload>

<dw:set-payload>

<![CDATA[

%dw 1.0

output application/xml

---

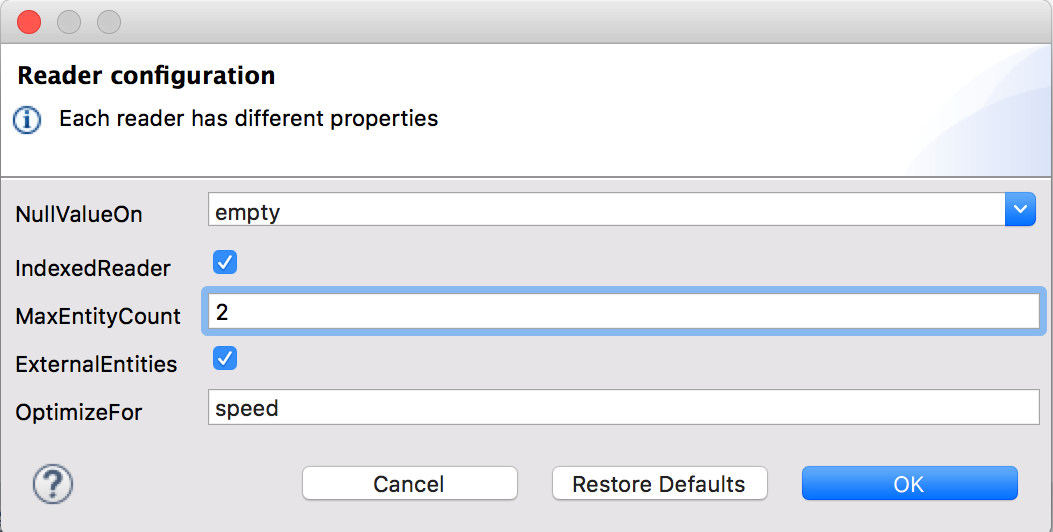
// Your transformation script goes here

]]>

</dw:set-payload>

</dw:transform-message>

Or via the UI of the Transform Message component:



See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task).

### Writer Properties

When defining an output of type XML, there are a few optional parameters you can add to the output directive to customize how the data is parsed:

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| indent | boolean | true | Defines if the XML code will be indented for better readability, or if it will be compressed into a single line |
| inlineCloseOn | string | never | Defines whether an empty XML child element appears as single self-closing tag or with an opening and closing tag. The value empty sets it to output self-closing tags. |
| encoding | string | UTF-8 | The character set to be used for the output |
| bufferSize | number | 153600 | The size of the buffer writer |
| inlineCloseOn | string |  | When the writer should use inline close tag. Possible values = empty/none |
| skipNullOn | string |  | Possible values = elements/attributes/everywhere. See [Skip Null On](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-formats#skip-null-on) |
| writeDeclaration | boolean | true | Defines if the XML declaration will be included in the first line (available since Mule 3.8.4) |

%output application/xml indent=false, skipNullOn="attributes"

The inlineCloseOn parameter defines if the output is structured like this (by default):

<someXml>

<parentElement>

<emptyElement1></emptyElement1>

<emptyElement2></emptyElement2>

<emptyElement3></emptyElement3>

</parentElement>

</someXml>

or like this (set with a value of "empty"):

<payload>

<someXml>

<parentElement>

<emptyElement1/>

<emptyElement2/>

<emptyElement3/>

</parentElement>

</someXml>

</payload>

#### Skip Null On

You can specify whether your transform generates an outbound message that contains fields with "null" values, or if these fields are ignored entirely. This can be set through an attribute in the output directive named **skipNullOn**, which can be set to three different values: **elements**, **attributes**, or **everywhere**.

When set to: \* **elements**: A key:value pair with a null value is ignored. \* **attributes**: An XML attribute with a null value is skipped. \* **everywhere**: Apply this rule to both elements and attributes.

### Defining a Metadata Type

In the Transform Message component, you can define a XML type through the following methods:

* By Providing a sample file
* By pointing to a schema file

|  |
| --- |
| See [To Define Input and Output Structure of a Transformation](https://docs.mulesoft.com/studio/6/input-output-structure-transformation-studio-task). |

### Custom Types

#### :cdata

XML defines a custom type named :cdata, it extends from string and is used to identify a CDATA XML block. It can be used to tell the writer to wrap the content inside CDATA or to check if the input string arrives inside a CDATA block. :cdata inherits from the type :string.

Transform

%dw 1.0

%output application/xml

---

{

users:

{

user : "Mariano" as :cdata,

age : 31 as :cdata

}

}

Output

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

<users>

<user><![CDATA[Mariano]]></user>

<age><![CDATA[31]]></age>

</users>

## JSON

### Canonical Model

JSON data-structures are mapped to DataWeave data-structures in a straight forward way as they share a lot of similarities.

### Writer Properties

When defining an output of type JSON, there are a few optional parameters you can add to the output directive to customize how the data is parsed:

| **Parameter** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| indent | boolean | true | Defines if the JSON code will be indented for better readability, or if it will be compressed into a single line |
| encoding | string | UTF-8 | The character set to be used for the output |
| bufferSize | number | 153600 | The size of the buffer writer |
| inlineCloseOn | string |  | When the writer should use inline close tag. Possible values = empty/none |
| skipNullOn | string |  | Possible values = elements/attributes/everywhere. See [Skip Null On](https://docs.mulesoft.com/mule-runtime/3.8/dataweave-formats#skip-null-on) |
| duplicateKeyAsArray | boolean | false | JSON language doesn’t allow duplicate keys with one same parent, this usually raises an exception. If set to true, the output contains a single key that points to an array containing all the values assigned to it. |

%output application/json indent=false, skipNullOn="arrays"

#### Skip Null On

You can specify whether this generates an outbound message that contains fields with "null" values, or if these fields are ignored entirely. This can be set through an attribute in the output directive named **skipNullOn**, which can be set to three different values: **elements**, **attributes**, or **everywhere**.

When set to: \* **elements**: A key:value pair with a null value is ignored. \* **attributes**: An XML attribute with a null value is skipped. \* **everywhere**: Apply this rule to both elements and attributes.

# DataWeave Memory Management

## Memory vs Disk Usage

DataWeave uses the system’s memory as a buffer while processing a transformation unless a certain threshold is exceeded, in that case it resorts to using the system’s hard disk as a buffer. By default, this threshold is set at **1572864 bytes**, but this value may be changed. The value refers to memory usage of each individual Transform Message component, not to an aggregate of all the ones in the project.

To change the threshold value at which memory is no longer used as a buffer, you must add a system property com.mulesoft.dw.buffersize and assign it the number (in bytes) of your new threshold. System properties may be defined in several ways, for example by editing the mule-app.properties file located in your project’s src/main/app folder, see [system properties](https://docs.mulesoft.com/mule-runtime/3.8/configuring-properties#system-properties) for more details and more ways you can set these.

The value you assign to this property affects your entire Mule application, affecting each instance of the Transform Message component individually.

## Immediate vs Deferred Execution

By default, DataWeave processes the transformation of a message as soon as the component is called out in the flow, you can change this behavior so that the DataWeave transformation returns a WeaveOutputHandler which is only processed when read by another component. This handler is capable of deferring writing the Mule Message’s payload until there is a stream available to write it to. This allows for the DataWeave output to remain outside of the heap as processing continues on other components in the flow.

To set this up, in the XML of your <dw:transform-message> component, add a mode attribute. This attribute accepts the values **immediate** or **deferred**.

* With **immediate** the output is an **inputStream** (default mode)
* With **deferred** the output is a WeaveOutputHandler

Below is an example that sets this attribute to "deferred":

<dw:transform-message doc:name="Transform Message" mode="deferred">

<dw:set-payload>

<![CDATA[

%dw 1.0

%output application/xml

---

payload

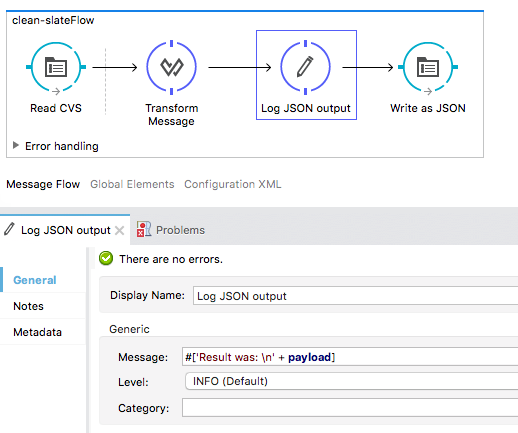
]]>

</dw:set-payload>

</dw:transform-message>

### Using The WeaveOutputHandler

Keep in mind that when using deferred execution, the returned message will contain a WeaveOutputHandler object rather than a String representation. For example, consider a logger you wish to log your payload with:



The output of this logger will appear as the following:

org.mule.transport.file.FileMessageReceiver: Lock obtained on file: /Users/mulesoft/inputCSV.csv

org.mule.api.processor.LoggerMessageProcessor: Result was:

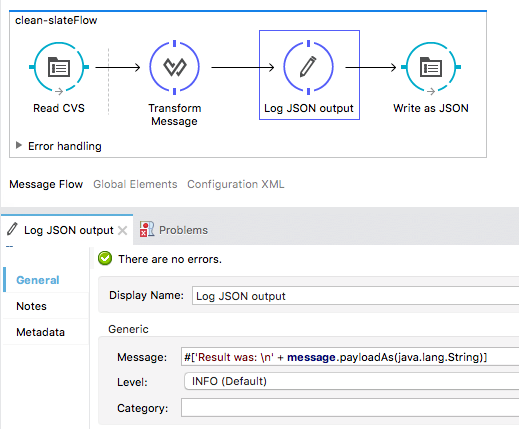
com.mulesoft.weave.mule.WeaveMessageProcessor$WeaveOutputHandler@3528770d

Initialising: 'File.dispatcher.657964187'. Object is: FileMessageDispatcher

org.mule.lifecycle.AbstractLifecycleManager: Starting: 'File.dispatcher.657964187'. Object is: FileMessageDispatcher

org.mule.transport.file.FileConnector: Writing file to: /Users/mulesoft/output/inputCSV.csv

If you wish to log the payload as a String representation, you’ll need to request the payload in a String representation. This can be achieved by using the expression #[message.payloadAs(java.lang.String)].



|  |
| --- |
| This expression is the equivalent of consuming the DataWeave output and transforming it into a String. Even when this expression is used in the context of a logger. The payload will reach the next processor as a String. It’s also important to note that once consumed as such, the entire payload will exist in memory. |

The output of this logger will appear as the following.

org.mule.transport.file.FileMessageReceiver: Lock obtained on file: /Users/josh/inputCSV.csv

org.mule.api.processor.LoggerMessageProcessor: Result was:

{

"people": [

{

"id": 1,

"firstName": "Max",

"lastName": "Mule"

},

{

"id": 2,

"firstName": "Sally",

"lastName": "Mule"

}

]

}

org.mule.lifecycle.AbstractLifecycleManager: Initialising: 'File.dispatcher.2036619369'. Object is: FileMessageDispatcher

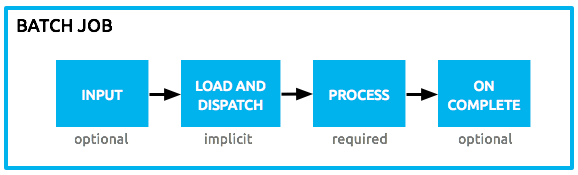
org.mule.lifecycle.AbstractLifecycleManager: Starting: 'File.dispatcher.2036619369'. Object is: FileMessageDispatcher

org.mule.transport.file.FileConnector: Writing file to: /Users/josh/output/inputCSV.csv

**Batch processing :**

Batch processing in Mule takes place within four phases (see table below). Within Studio’s visual editor, batch jobs manifest as flow-like objects that are visually divided according to the phases of batch processing.

| **Phase** | | **Configuration** |
| --- | --- | --- |
| 1 | Input | optional |
| 2 | Load and Dispatch | implicit, not exposed in a Mule application |
| 3 | Process | required |
| 4 | On Complete | optional |



**Process :**

