

# DataWeave

Version 2.5, 2024-01-26

# Table of Contents

Overview .....	2
Release Notes .....	3
DataWeave Quickstart .....	5
Language Guide .....	23
Scripts .....	30
Selectors .....	41
Supported Data Formats .....	60
Streaming in DataWeave .....	175
Indexed Readers in DataWeave .....	185
Flatfile Schemas .....	186
Type System .....	201
Value Constructs for Types .....	228
Type Coercion with DataWeave .....	245
Variables .....	250
Predefined Variables .....	259
Flow Control in DataWeave .....	264
Pattern Matching in DataWeave .....	269
External Functions Available to DataWeave .....	275
Troubleshooting .....	279
Define Functions .....	287
Create Custom Modules and Mappings .....	294
Working with Functions and Lambdas in DataWeave .....	305
Memory Management .....	308
Versioning Behavior in DataWeave .....	310
DataWeave Examples .....	311
Extract Data .....	315
Select XML Elements .....	342
Set Default Values .....	345
Set Reader and Writer Configuration Properties .....	348
Perform a Basic Transformation .....	350
Map Data .....	352
Map and Flatten an Array .....	359
Map an Object .....	360
Map the Objects within an Array .....	366
Map Based On an External Definition .....	369
Rename Keys .....	371
Output a Field When Present .....	374
Change Format According to Type .....	376

Regroup Fields .....	378
Zip Arrays Together .....	381
Pick Top Elements .....	386
Change the Value of a Field .....	388
Exclude Fields from the Output .....	392
Use Constant Directives .....	394
Define a Custom Addition Function .....	399
Define a Function that Flattens Data in a List .....	402
Flatten Elements of Arrays .....	404
Use Regular Expressions .....	410
Output self-closing XML tags .....	413
Insert an Attribute into an XML Tag .....	414
Remove Certain XML Attributes .....	416
Pass XML Attributes .....	418
Include XML Namespaces .....	420
Remove Objects Containing Specified Key-Value Pairs .....	423
Reference Multiple Inputs .....	425
Merge Fields from Separate Objects .....	429
Parse Dates .....	433
Add and Subtracting Dates .....	435
Change a Time Zone .....	437
Format Dates and Times .....	440
Work with Multipart Data .....	444
Conditionally Reduce a List Via a Function .....	446
Pass Functions as Arguments .....	448
Change a Script's MIME Type Output (Mule) .....	450
Look Up Data in an Excel (XLSX) File (Mule) .....	453
Look Up Data in CSV File (Mule) .....	455
Decode and Encode Base64 (Mule) .....	457
Call Java Methods (Mule) .....	461
Read and Write a Flat File (Mule) .....	467
Use a Reader Property through a Connector (Mule) .....	474
Use Dynamic Writer Properties (Mule) .....	476
Extract Key/Value Pairs with Pluck Function (Mule) .....	477
DataWeave Reference .....	483
DataWeave Operators .....	485
System Properties .....	502
Precedence Rules .....	507
dw::Core .....	512
dw::core::Arrays .....	638
dw::core::Binaries .....	662

dw::core::Dates .....	668
dw::core::Numbers .....	687
dw::core::Objects .....	694
dw::core::Periods .....	704
dw::core::Strings .....	713
dw::core::Types .....	760
dw::core::URL .....	796
dw::Crypto .....	803
dw::extension::DataFormat .....	808
dw::module::Multipart .....	810
dw::Mule .....	825
dw::Runtime .....	829
dw::System .....	860
dw::util::Coercions .....	862
dw::util::Diff .....	884
dw::util::Math .....	890
dw::util::Timer .....	899
dw::util::Tree .....	903
dw::util::Values .....	918
dw::xml::Dtd .....	930

DataWeave is the programming language designed by MuleSoft for data transformation. It is also the expression language Mule runtime engine uses to configure components and connectors.

DataWeave enables you to build a simple solution for a common use case for integration developers: read and parse data from one format, transform the data, and write it out as a different format. For example, a DataWeave script can receive a CSV file as input and transform it into an array of complex JSON objects, or receive an XML input and write the data out to a flat file format. DataWeave enables developers to focus on the transformation logic instead of thinking about the specifics of reading, parsing, and writing specific data formats in a performant way.

- To learn more about DataWeave, visit [the language guide](#).
- To look up reference information about DataWeave 2.4.0 operators and functions, visit [the reference guide](#).
- To see sample scripts to solve common transformation scenarios, visit [DataWeave examples](#).
- To try DataWeave scripts interactively, visit [DataWeave Playground](#).
- To get started with DataWeave 2.4.0 for Mule runtime engine (Mule) version 4.4 and later, visit [the quickstart](#).

## Compatibility

The following table specifies which version of DataWeave is bundled with each Mule runtime engine release:

Mule Version	DataWeave Version
4.5	2.5
4.4	2.4
4.3	2.3
4.2	2.2
4.1	2.1
3.9	1.2
3.8	1.1
3.7	1.0

# Overview

DataWeave is the programming language designed by MuleSoft for data transformation. It is also the expression language Mule runtime engine uses to configure components and connectors.

DataWeave enables you to build a simple solution for a common use case for integration developers: read and parse data from one format, transform the data, and write it out as a different format. For example, a DataWeave script can receive a CSV file as input and transform it into an array of complex JSON objects, or receive an XML input and write the data out to a flat file format. DataWeave enables developers to focus on the transformation logic instead of thinking about the specifics of reading, parsing, and writing specific data formats in a performant way.

- To learn more about DataWeave, visit [the language guide](#).
- To look up reference information about DataWeave 2.4.0 operators and functions, visit [the reference guide](#).
- To see sample scripts to solve common transformation scenarios, visit [DataWeave examples](#).
- To try DataWeave scripts interactively, visit [DataWeave Playground](#).
- To get started with DataWeave 2.4.0 for Mule runtime engine (Mule) version 4.4 and later, visit [the quickstart](#).

## Compatibility

The following table specifies which version of DataWeave is bundled with each Mule runtime engine release:

Mule Version	DataWeave Version
4.5	2.5
4.4	2.4
4.3	2.3
4.2	2.2
4.1	2.1
3.9	1.2
3.8	1.1
3.7	1.0

# Release Notes

DataWeave is the programming language designed by MuleSoft for data transformation. It is also the expression language Mule runtime engine uses to configure components and connectors.

Release notes for each DataWeave version include bug fixes, compatibility information, and patch update release details, in addition to new features and other information about DataWeave releases:

- [DataWeave 2.5.0](#)
- [DataWeave 2.4.0](#)

## New Features in This Release

The 2.5.0 version of DataWeave introduces the following new features and enhancements:

Support for backward compatibility with previous 2.x versions of DataWeave:

- Compatibility flags retain previous DataWeave behavior at the Mule application level. For details, see <https://docs.mulesoft.com/dataweave/latest/dataweave-system-properties.html>.
- Syntax of an earlier version of DataWeave is supported through the `%dw` directive (such as `%dw 2.4`).

Setting the directive to an earlier version of DataWeave avoids any syntax-breaking changes when the DataWeave runtime engine runs the script. This script-level setting enables you to retain earlier behavior in some scripts while using the latest behavior in others. DataWeave syntax did not change between DataWeave versions 2.1 through 2.4. See discussion of `%dw` in [DataWeave Header](#).

Extended format support:

- ProtoBuf format support is available. See <https://docs.mulesoft.com/dataweave/latest/dataweave-formats-protobuf.html>.
- cXML support in the XML format is available for reading and creating doctype directives (DTDs). See <https://docs.mulesoft.com/dataweave/latest/dataweave-formats-xml.html>.

Extended type support:

- Loading type definitions from Java classes, JSON schemas, and XML schemas, and using the definitions in DataWeave scripts is supported. See <https://docs.mulesoft.com/dataweave/latest/dataweave-selecting-types.html>.
- Creating new DataWeave types from existing types is supported. See [Reuse Types](#).
- Specifying type parameters (similar to generics in other programming languages) of a function at the call site is supported. For details and examples, see [Type Parameters](#) and <https://docs.mulesoft.com/dataweave/latest/dataweave-functions-lambdas.html>.
- Introducing a Metadata Assignment Operator (`<~`) which enables you to set the metadata of any

value without using the `as` operator. See [Metadata Assignment Operator](#).

Memory management:

- DataWeave now uses a centralized memory service provided by the Mule runtime when executing in that context.

DataWeave module features:

- The `Dtd` module (`dw::xml::Dtd`) is new.
- The `toString` function adds the `locale` parameter.
- The `Core annotation @UntrustedCode()` changes to `@UntrustedCode(privileges: Array<String>)`.
- The `concatWith` function is new.
- The `version` function is new.

## DataWeave and Mule

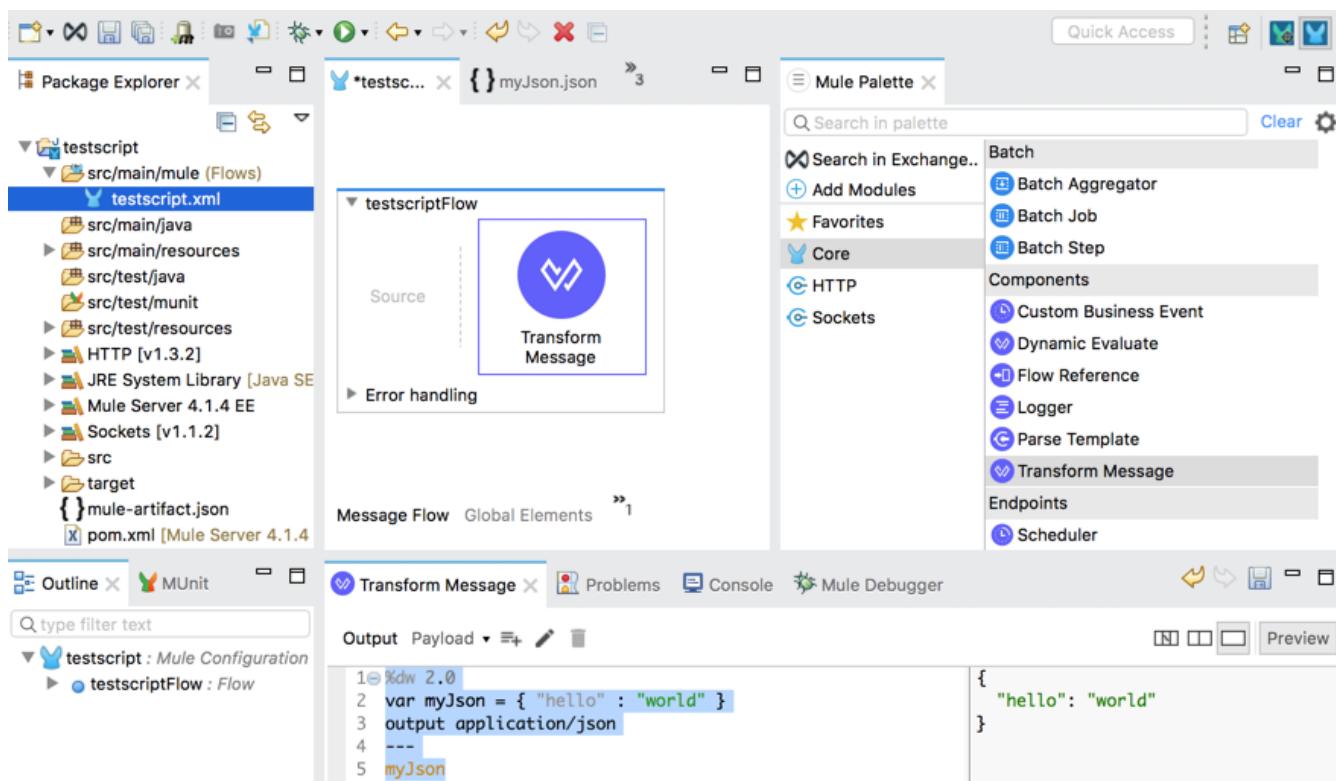
DataWeave 2.5.0 is bundled with the Mule 4.5.0 release. For details about the Mule release, see [Mule Runtime Engine 4.5.0 Release Notes](#).

# DataWeave Quickstart

In this guide, you run DataWeave 2.0 scripts on sample data, without relying on external data sources. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

The examples introduce some key DataWeave concepts that you can explore further whenever you are ready, and they show how to turn the Transform Message component into a DataWeave playground.

To run the scripts, you copy them into the source code area of a Transform Message component in [Studio](#), then view the results in the component's **Preview** pane. You also load content from files so that your DataWeave script can act on it. Once you feel comfortable with DataWeave examples here and elsewhere in the docs, you can use the DataWeave playground to practice writing your own DataWeave scripts on sample data.



The figure shows a Transform Message component in the center canvas, within **testingscriptFlow**. Below the Studio canvas, the **Transform Message** tab includes a DataWeave script in the source code area with output in the **Preview** pane.

## Prerequisites

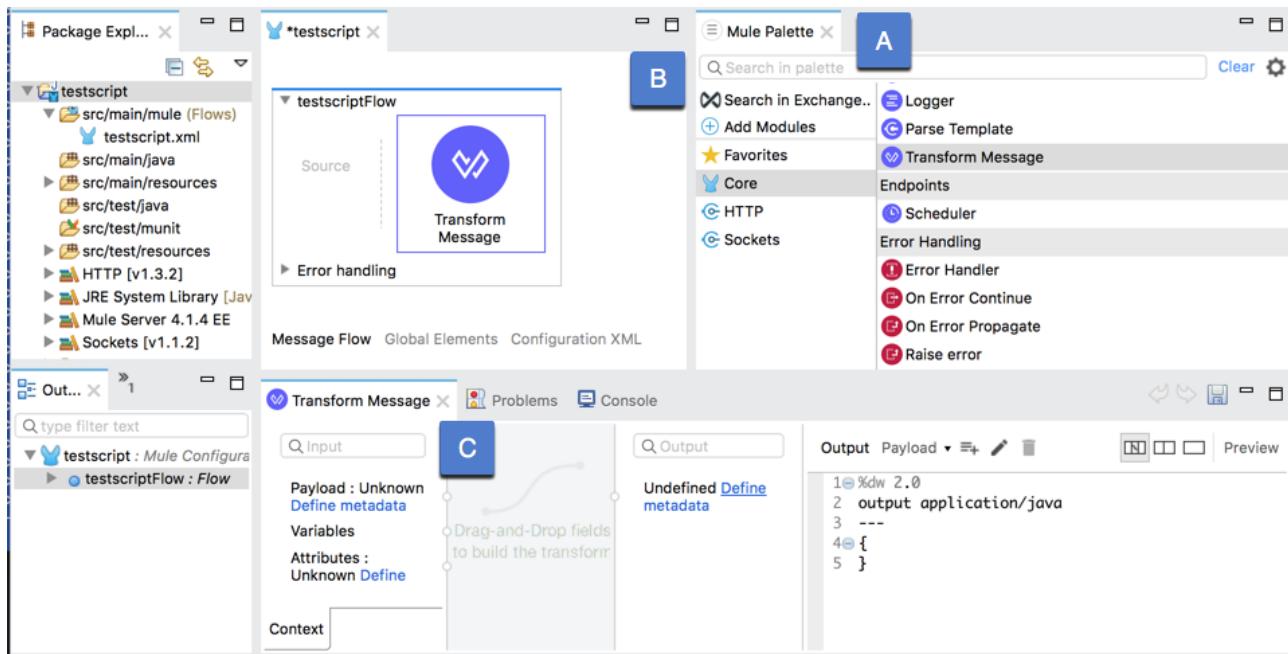
[Studio 7](#) is required. Versions 7.3 or 7.2.3 are recommended. Other Studio 7 versions are untested with this guide.

Once Studio is installed, you need to set up a project with a Transform Message component. See [Set Up a Project in Studio](#).

# Set Up a Project in Studio

Set up a Mule project that serves as a DataWeave playground:

1. In Studio, click **File** → **New** → **Mule Project** to create a Mule project.
2. Provide the name **testscript** for the project, and click **Finish**.
3. From the **Mule Palette** tab of your new project, click **Core**, and then drag the **Transform Message** component into the Studio canvas.



(A) Mule Palette tab

(B) Studio canvas with Transform Message component

(C) Transform Message tab

4. In the **Transform Message** tab, click **Preview** (on the far right) to open the Preview pane, and click the *empty* rectangle next to **Preview** to expand the source code area.



The source code area on the left is the place where you are going to add DataWeave scripts, and the Preview pane on the right is where you view the output of the scripts.

5. Proceed to **Start Scripting**.

# Start Scripting

1. [Concatenate Two Strings into a Single String](#)
2. [Transform JSON Input to XML Output](#)
3. [Learn About Supported Data Types](#)
4. [Define and Use a DataWeave Variable as Input](#)
5. [Use a DataWeave Function in a DataWeave Variable](#)
6. [Read, Transform, and Select Content from an Input](#)
7. [Read File Contents with a DataWeave Function](#)
8. [Map Elements from an Array into an Object](#)
9. [Pluck Values from an Object into an Array](#)
10. [Map and Merge Fields](#)

## Concatenate Two Strings into a Single String

Begin with a simple DataWeave script that concatenates two strings ("hello" and "World") together into a single string ("helloWorld").



The screenshot shows the Mule Studio interface with the 'Transform Message' tab selected. The source code area contains the following DataWeave script:

```
1 %dw 2.0
2 output application/json
3 ---
4 { myString: ("hello" ++ "World") }
```

The preview area shows the resulting JSON output:

```
{ "myString": "helloWorld" }
```

1. Replace the current script in the source code area of the **Transform Message** tab with this one:

```
%dw 2.0
output application/json
---
{ myString: ("hello" ++ "World") }
```

- The body of the DataWeave script contains a key-value pair (`{ myString: ("hello" ++ "World") }`). The value of this input object is a DataWeave expression (`("hello" ++ "World")`) for concatenating the strings "hello" and "World" into a single string, "helloWorld".

When you are ready, you can learn about the DataWeave `++` function used to concatenate the strings.

- The header of the script is all the content above the three dashes, `---`. It includes important directives, including one for specifying the output format `application/json`. You can learn

more about [Scripts](#) when you are ready.

2. See the JSON output in the **Preview** pane:

```
{ "myString": "helloWorld" }
```

3. Proceed to [Transform JSON Input to XML Output](#).

## Transform JSON Input to XML Output

Many integrations require transformations from one format to another. This procedure uses the **output** directive to produce XML output from JSON input.



The screenshot shows the 'Transform Message' interface. The top bar includes tabs for 'Transform Message' and 'Problems', and icons for back, forward, save, and close. Below the tabs is a toolbar with buttons for 'Output', 'Payload', and preview options. The main area is divided into two sections: 'Source' and 'Preview'. The 'Source' section contains the following DWScript code:

```
1 %dw 2.0
2   output application/xml
3   ---
4   { "myString" : ("helloWorld") }
```

The 'Preview' section shows the resulting XML output:

```
<?xml version='1.0' encoding='UTF-8'?>
<myString>helloWorld</myString>
```

1. Replace the body of the current script in the source code area with JSON output from [Concatenate Two Strings into a Single String](#), and change the **output application/json** directive to **output application/xml**:

```
%dw 2.0
output application/xml
---
{ "myString" : ("helloWorld") }
```

2. See the XML output in the **Preview** pane:

```
<?xml version='1.0' encoding='UTF-8'?>
<myString>helloWorld</myString>
```

Notice that the **"myString"** key of the input JSON object `{ "myString" : ("helloWorld") }` is converted to the root element of the XML output and that the concatenated string becomes the value of that XML object. So the XML output is `<myString>helloWorld</myString>`. That output is preceded by a standard XML declaration that specifies the XML version and encoding.



Without a *single* key to serve as a root node for XML output (for example, if the input is simply `("helloWorld")`), the transformation to XML will fail with a yellow warning (!) in the source code area. The warning message is **Trying to output non-whitespace characters outside main element tree**. If you like, you can try to produce this warning on your own.

DataWeave supports many output and input formats. You can learn more about [Supported Data Formats](#) when you are ready.

3. Proceed to [Learn About Supported Data Types](#)

## Learn About Supported Data Types

Now provide a DataWeave script that simply introduces you to a variety of supported data types and shows how to add comments to a script.

1. Replace the current script in the source code area of the **Transform Message** tab with this one:

```
%dw 2.0
output application/json
---
{
    /*
     * A multi-line
     * comment here.
    */
    myString: "hello world",
    myNumber: 123,
    myFloatingPointNumber: 123.456,
    myVeryBigNumber: 12341234134123412341234123,
    myDate: |2018-12-07|,
    myTime: |11:55:56|,
    myDateTime: |2018-10-01T23:57:59-03:00|,
    myBoolean: true,
    myArray: [ 1, 2, 3, 5, 8 ],
    myMixedArray: [ 1, 2, "blah", { hello: "there" } ],
    myObjectKeyValuePair: { innerKey: "innerValue" },
    myObjectWithConditionalField: { a : { b : 1, ( c : 2 ) if true, (d : 4) if false
} },
    myNull: null,
    myBinary: "abcd1234123" as Binary
    //A one-line comment here.
}
```

DataWeave supports multi-line comments within `/* */` markup and single-line comments after forward slashes (`//`). It also supports many data types, shown after the colon (`:`) in key-value pairs, such as `myString: "hello world"` and `myNumber: 123`. These types include strings (surrounded by quotation marks, `"`), numbers, date and time structures (within pipes, `||`), Booleans (`true` and `false`), arrays (within square brackets, `[]`), JSON-like objects (key-value structures within curly braces, `{}`), `null`, and binaries. When you are ready for more on this topic, you can review [DataWeave types](#).

2. See the JSON output in the **Preview** pane:

```
{
  "myString": "hello world",
  "myNumber": 123,
  "myFloatingPointNumber": 123.456,
  "myVeryBigNumber": 12341234134123412341234123,
  "myDate": "2018-12-07",
  "myTime": "11:55:56",
  "myDateTime": "2018-10-01T23:57:59-03:00",
  "myBoolean": true,
  "myArray": [ 1, 2, 3, 5, 8 ],
  "myMixedArray": [ 1, 2, "blah", { "hello": "there" } ],
  "myObjectKeyValuePair": { "innerKey": "innerValue" },
  "myObjectWithConditionalField": { "a": { "b": 1, "c": 2 } },
  "myNull": null,
  "myBinary": "abcd1234123"
}
```

### 3. Proceed to [Define and Use a DataWeave Variable as Input](#)

## Define and Use a DataWeave Variable as Input

Now try a simple DataWeave script that outputs the value of the DataWeave variable `myJson`. You set the variable using the `var` directive in the script's header.

### 1. Replace the current script in the source code area of the **Transform Message** tab with this one:

```
%dw 2.0
var myJson = { "hello" : "world" }
output application/json
---
myJson
```

A JSON object (`{ "hello" : "world" }`) is defined as the `myJson` variable in the script's header. You can learn more about [Variables](#) when you are ready.

### 2. See the output in the **Preview** pane:

```
{
  "hello": "world"
}
```

### 3. Proceed to [Use a DataWeave Function in a DataWeave Variable](#)

## Use a DataWeave Function in a DataWeave Variable

Now try a script that uses the DataWeave `avg` function in a DataWeave variable (`myJson`) to get averages of two sets of numbers.

```
1 %dw 2.0
2 var myJson = {
3     a: avg([1, 1000]),
4     b: avg([1, 2, 3])
5 }
6 output application/json
7 ---
8 myJson
```

{  
    "**a**": 500.5,  
    "**b**": 2.0  
}

1. Replace the current script in the source code area with this one:

```
%dw 2.0
var myJson = {
    a: avg([1, 1000]),
    b: avg([1, 2, 3])
}
output application/json
---
myJson
```

The `avg` functions here get invoked when you add `myJson` to the body of the script, producing the calculated averages within the JSON object you can see in the **Preview** pane. The structures `[1, 1000]` and `[1, 2, 3]` are **arrays**. You can learn more about `avg` when you are ready.

2. Preview the output:

```
{  
    "a": 500.5,  
    "b": 2.0  
}
```

3. Proceed to [Read, Transform, and Select Content from an Input](#)

## Read, Transform, and Select Content from an Input

Now try a more complicated script that reads XML input, transforms it to JSON, and only selects the contents of the `car` element.

1. Replace the current script in the source code area with this one:

```
%dw 2.0
var myRead = read("<car><color>red</color></car>",
                  "application/xml")
output application/json
---
{
  mySelection : myRead.car
}
```

If you encounter an issue previewing this example, try changing `myRead.car` to `myRead."car"`. Learn more about the `read` function, [Supported Data Formats](#), and [Selectors](#) when you are ready.

## 2. Preview the output:

```
{
  "mySelection": {
    "color": "red"
  }
}
```

## 3. Proceed to [Read File Contents with a DataWeave Function](#)

### Read File Contents with a DataWeave Function

Now use `readUrl` to read the contents of a file in the Studio `src/main/resources` folder so you can use that content as sample data for a DataWeave script.

1. Add a file by right-clicking the `src/main/resources` folder in the **Package Explorer** tab, then navigating to **New** → **File**, providing the file name `myJson.json` for that file, and clicking **Finish**.
2. From `src/main/resources`, provide and save the following sample content in the `myJson.json` tab (or within the **Source** sub-tab of the `myJson.json` tab, if it is present):

```
{
  "hello": "world"
}
```

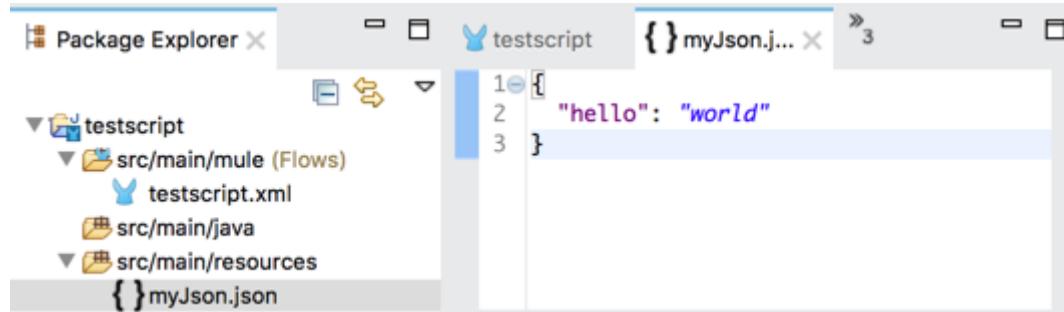


Figure 1. `myJson.json` in Studio

3. Returning to the Transform Message component within the **testscript** tab, replace the current script with one that uses `readUrl` to read the JSON contents from your file:

```
%dw 2.0
output application/json
---
readUrl("classpath://myJson.json", "application/json")
```

Learn more about the [readUrl](#) function when you are ready.

4. View the matching output in the **Preview** pane.

```
{
  "hello": "world"
}
```

Note that you can also load the contents of a file through a metadata type in the Transform Message component. That procedure is covered later, in [Run Examples with Longer Payloads](#). It uses the `myJson.json` file you just created.

5. Proceed to [Map Elements from an Array into an Object](#)

## Map Elements from an Array into an Object

Almost all integrations require data mappings. Here, you map elements within an array to keys and values of a JSON object:

1. Replace the current script in the source code area with this one:

```
%dw 2.0
output application/json
---
{
  (
    ["a", "b", "c"] map ((value, index) -> {
      (index): value
    })
  )
}
```

The `map` function iterates over the array on the left to apply the lambda (anonymous function) on the right (`((value, index) -> { (index): value })`) to elements in that array. The lambda uses [named parameters](#) (`value` and `index`) to select the values and indices from the array and populate a JSON object with key-value pairs. Learn about [map](#), and when you are ready, compare [map](#) with [mapObject](#), which takes an object as input.

2. Preview the output:

```
{  
  "0": "a",  
  "1": "b",  
  "2": "c"  
}
```

### 3. Proceed to [Pluck Values from an Object into an Array](#)

## Pluck Values from an Object into an Array

Now use the DataWeave `pluck` function to iterate over values in a JSON object and output those values into an array.

### 1. Replace the contents of the source code area with a script that uses `pluck` on a JSON object:

```
%dw 2.0  
output application/json  
---  
{  
  "0": "a",  
  "1": "b",  
  "2": "c"  
} pluck ((value) -> value)
```

Notice that the input object matches the `output` of the `map` example and that the output array matches the `input` from the `map` example. Learn more about `pluck` when you are ready.

### 2. Preview the output:

```
[  
  "a",  
  "b",  
  "c"  
]
```

### 3. Proceed to [Map and Merge Fields](#).

## Map and Merge Fields

Now try a more complex example that maps and merges fields from items in separate arrays. The point here is simply to provide a taste of DataWeave's ability to handle more complex mappings and transformations needed for some integrations.

### 1. Replace the current script in the source code area with this one:

```
%dw 2.0
var myVar = [
    { bookId: 101,
        title: "world history",
        price: "19.99"
    },
    {
        bookId: 202,
        title: 'the great outdoors',
        price: "15.99"
    }
]
var myVar2 = [
    {
        bookId: 101,
        author: "john doe"
    },
    {
        bookId: 202,
        author: "jane doe"
    }
]
output application/json
---
myVar map (item, index) -> using (id = item.bookId) {
    "id" : id,
    "topic" : item.title,
    "cost" : item.price as Number,
    (myVar2 filter ($.bookId contains id) map (item) -> {
        author : item.author
    })
}
```

When you are ready to explore the language further, you can learn how the `filter` function used near the end returns `author` values from the array in the `myVar2` variable. You can read about [type coercion with DataWeave](#) to see how `as` works with the data type in the line `"cost" : item.price as Number`, to coerce input strings like `"19.99"` into numbers like `19.99`. You can see how `using` (shown in `using (id = item.bookId)`) enables you to create [local DataWeave variables](#) in a script.

## 2. Preview the output:

```
[
  {
    "id": "101",
    "topic": "world history",
    "cost": 19.99,
    "author": "john doe"
  },
  {
    "id": "202",
    "topic": "the great outdoors",
    "cost": 15.99,
    "author": "jane doe"
  }
]
```

3. Proceed to [Try Out More DataWeave Examples](#).

## Try Out More DataWeave Examples

Now you are ready to run DataWeave examples on your own. In your DataWeave playground, you can run examples from the docs whenever you want to discover more about the DataWeave language. You can play with the examples and use them to start your own scripts.



For examples that use `payload` to reference input data (such as the `contains<T>(@StreamCapable items: Array<T>, element: Any): Boolean` or `mapObject` function examples), you can avoid payload-related issues by using techniques described in [Run Examples that Act on a Payload](#).

1. Find many more examples to try out in Studio here:
  - [DataWeave Operators](#)
  - [DataWeave Reference](#): Docs on the DataWeave function modules provide many examples that use DataWeave functions.
  - [Flow Control in DataWeave](#)
  - [Pattern Matching in DataWeave](#)
  - [DataWeave Cookbook](#)
  - [Define Functions](#)
2. Proceed to [Next Steps](#).

## Run Examples that Act on a Payload

In DataWeave, `payload` is a built-in Mule Runtime variable that holds the contents of a [Mule message](#). It enables you to retrieve the body of a message simply by typing `payload` into the DataWeave script. The docs often refer to this content as "the payload."

To try out DataWeave examples that use `payload` to get the contents of a Mule message, you have

some options:

- [Run Examples with Short Payloads](#): You can use a DataWeave variable.
- [Run Examples with Longer Payloads](#): Add the payload content through a file. This technique also works for short payloads.

Alternatives that require a running Mule app (such as using [Set Payload](#)) are not covered here but are introduced in [Next Steps](#).

## Run Examples with Short Payloads

For short examples with a few lines of sample data, you can convert the input payload to a variable. You simply copy the payload's content into a DataWeave variable (`var`) in the header of a DataWeave script. Then, in the body of the script, you replace `payload` with the name of the new variable to reference that variable.

1. Start with this script in the source code area:

```
%dw 2.0
output application/json
---
ContainsRequestedItem: payload.root.*order.*items contains "3"
```

Notice that you cannot preview it yet:



2. Now modify this example so that it uses some sample input instead of attempting to use `payload` to input content of a Mule message that does not exist:
  - a. Add a `myInput` DataWeave variable supplying some input data the script can read, and change the Mule `payload` variable in the body of the script to the DataWeave `myInput` variable.

```
%dw 2.0
var myInput = read("<root>
<order>
  <items>1</items>
  <items>3</items>
</order>
<order>
  <items>2</items>
</order>
</root>",
"application/xml")
output application/json
---
ContainsRequestedItem: myInput.root.*order.*items contains "3"
```

- Preview the results:

```
{
  "ContainsRequestedItem": true
}
```

When you are ready, learn more about the [contains](#) function and about the built-in [payload](#) variable, in [Predefined Variables](#).

## Run Examples with Longer Payloads

For DataWeave examples with many lines of sample data, consider creating a metadata type through Transform Message. Metadata types can accept a local file that contains sample data.

- Use the [readUrl](#) example to create [src/main/resources/myJson.json](#) in Studio.

You can skip this step if you still have the file in Studio. The next steps show how to use the contents of this file as your payload.

- Now replace the current script from the source code area of the Transform Message with one that selects the payload of the input:

```
%dw 2.0
output application/json
---
payload
```

- Notice that you cannot preview a payload now because it does not exist yet.

```

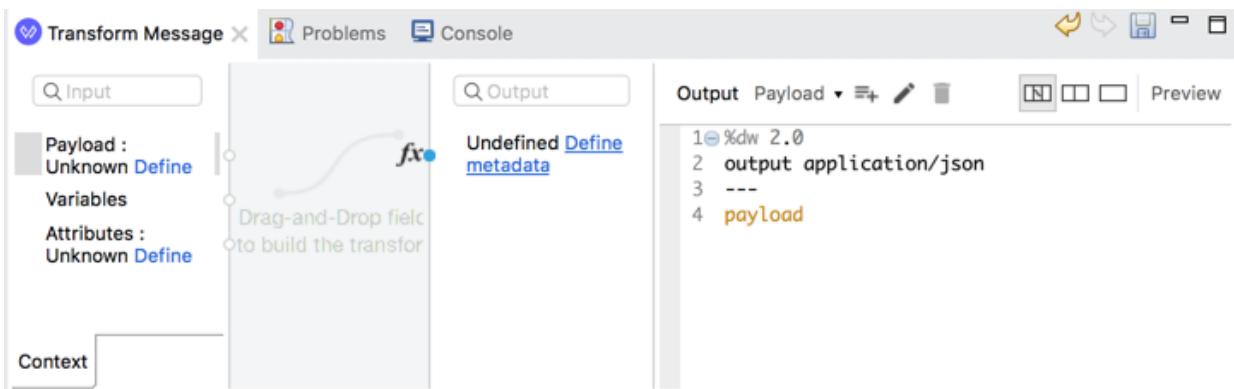
1 %dw 2.0
2 output application/json
3 ---
4 payload

```

Create required sample data to execute preview

4. Now provide a simple JSON object as the payload of the Transform Message:

- In the **Transform Message** tab, click the *left-most* rectangle from the **Preview** button to open the columned, graphical view next to the source code area:



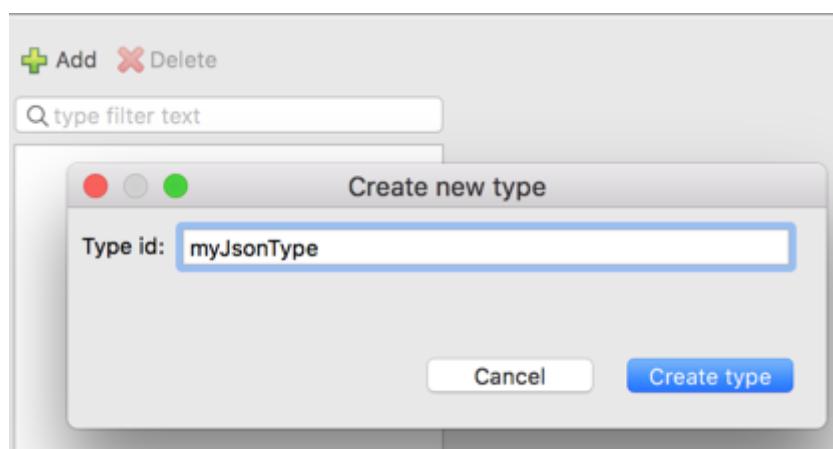
If you mouse over that rectangle, you can see the label **Show Graphic**.

- In the left-most column of the **Transform Message** tab, find **Payload: Unknown**, and click **Define metadata**.

If you do not see the **Define metadata** link, simply right-click the **Payload:** entry in the left-most column, and then click **Set Metadata** to open the **Select metadata type** dialog.

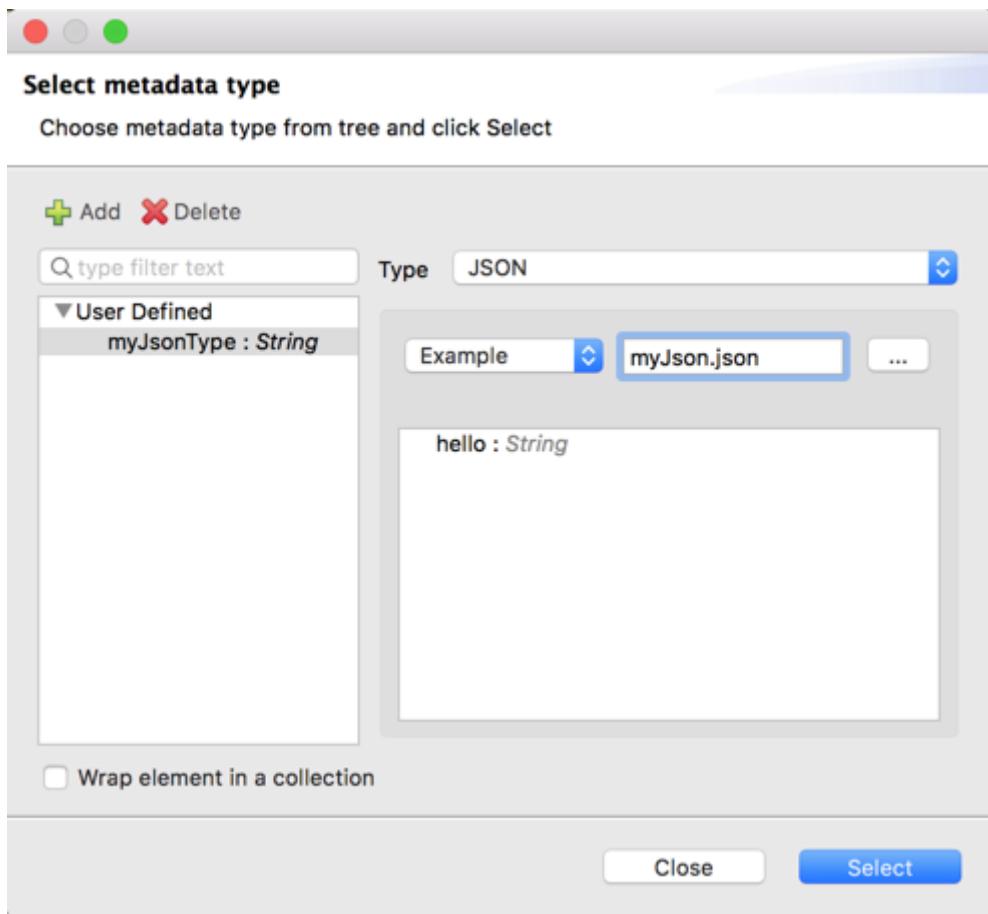
5. Now load the contents of `myJson.json` (created in the `readUrl` example) into the Transform Message component:

- Click **+Add** to open the **Create new type** dialog.
- In the dialog, provide the **Type id** `myJsonType`, and click **Create type**.



- Back in the **Select metadata type** dialog that opens, select **JSON** from the **Type** drop-down

menu.



- d. Below the new type, change **Schema** to **Example** (as shown above).
- e. Use the navigation button with the ellipsis (...) to find `src/main/resources/myJson.json`, and click **Open**, which displays the structure of the file contents (`hello: String`) in the **Select metadata type** window.
- f. Now click **Select** to load the contents of the file into the message payload.
- g. Notice that the JSON object from `myJson.json` is now in the Preview pane.

If necessary, you can click **Preview** to open the Preview pane.

- h. Now click the *empty* rectangle next to the **Preview** button to open the source code area, and change the body of the script to `payload.hello`, retaining the existing DataWeave header content.

Notice that the Preview pane now contains only the value of the payload: "`world`".

Here is what this example looks like in Studio:



```
1 %dw 2.0
2   output application/json
3   ---
4   payload.hello
```

"world"

Learn more about [Selectors](#) when you are ready.

## Next Steps

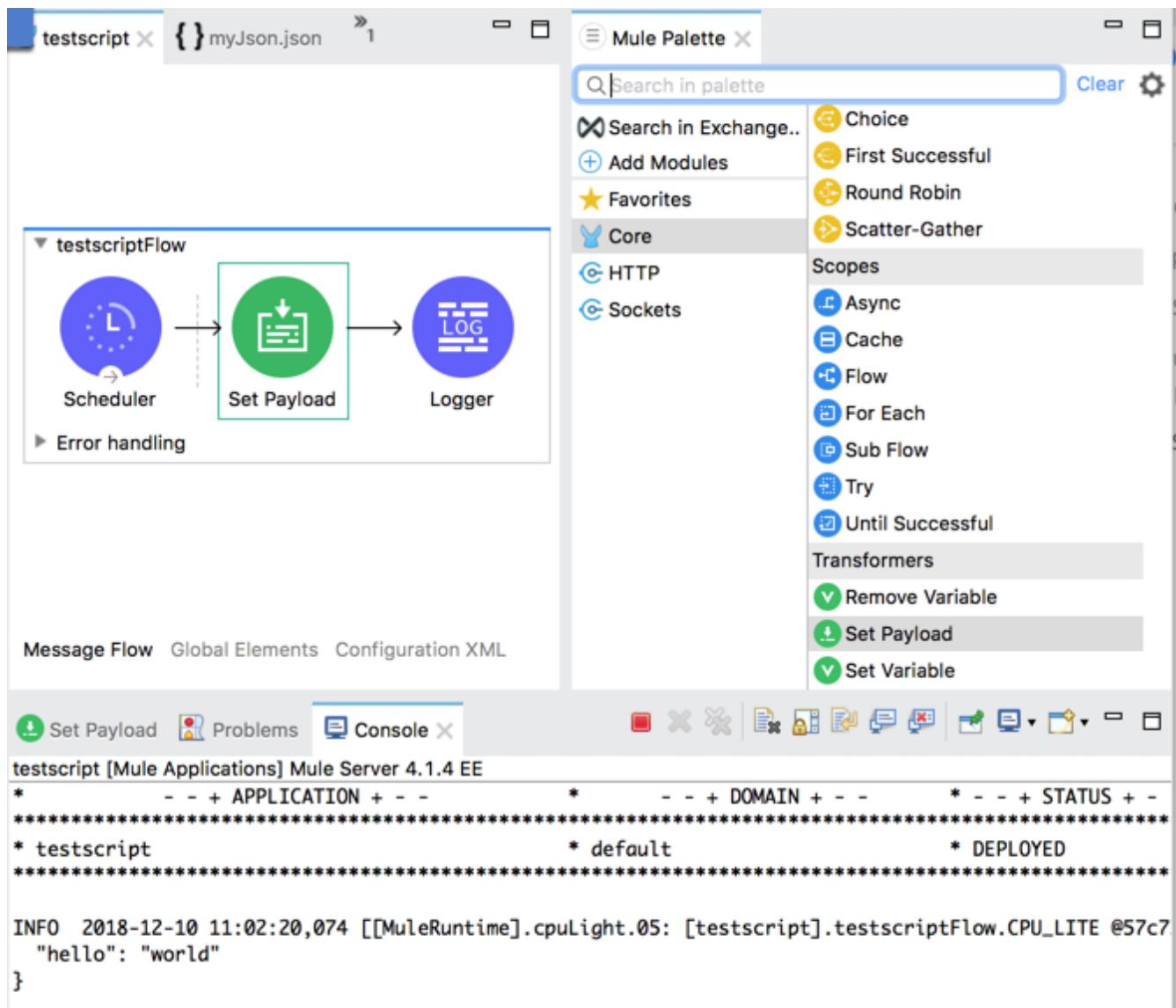
- To get started with Mule app development and data mapping through the Studio UI, see [Mule App Development Tutorial](#).
- Beyond the Transform Message component, many Mule connectors, modules, and Core components accept DataWeave selectors and short DataWeave expressions in their **fx** fields.

To learn about the components, you can start from [Mule Components](#).

To try out sample data in a running Mule app, without relying on external data sources, you can use these Core components with or without Transform Message:

- [Set Payload](#) to provide content for the payload of a Mule event.
- [Set Variable](#) to create content in a Mule event [variable](#).
- [Scheduler](#) to trigger the regular generation of Mule events.
- [Logger](#) to view output and issues logged in the Studio console.

Here is an example that uses some of these components:



- The **fx** value of Set Payload is set to `output application/json --- { hello : "world"}`.
- The **fx** value of the Logger is set to `payload`, which looks like `##[payload]` in the UI and the XML configuration file for the project.
- The **Console** tab of the *running* Mule app (`testscript`) displays the payload from Set Payload.

# Language Guide

DataWeave is a functional language used in Mule applications to perform data transformations. Before you begin to use DataWeave to code your own powerful and complex data transformations, you must understand the basic concepts of programming and the core features of functional programming languages.

DataWeave enables you to take advantage of the benefits of functional programming, including:

- Pure functions that always produce the same output given the same input, making them easier for you to debug.
- Immutable variables that do not change their value during the execution, adding stability to your code.
- Function signatures that provide additional transparency about what affects a function's execution, because pure functions depend on only their input parameters to produce a result.
- Functional languages produce code that's easier to maintain because the code is concise and clear.
- A call-by-need strategy (also known as *lazy evaluation*) that improves performance, because expressions are evaluated only when they are needed and their result is stored to avoid evaluating an expression multiple times if the input parameters are the same.

## DataWeave Language Basic Concepts

DataWeave incorporates concepts that are common to most programming languages. All DataWeave scripts consist of a *header* that contains important directives, including the output format of the transformation, and a *body* that contains the expression to generate a result.

DataWeave supports different *data structures*, including *simple*, *complex*, and *composite* types. In most of your DataWeave transformations, you use *data selectors*, *data operators*, and *functions* alongside *arrays* and *objects* to perform data mappings or transformations.

You can also import prebuilt DataWeave *modules* to extend the set of functions available to use in your scripts.

## Architecture of a DataWeave Script

A DataWeave script consists of a *header* and a *body*, separated by three dashes (---). The header contains language directives (`import`, for example), defines the output format of the transformation, and can also contain variable and function declarations. The body contains the expression that produces the resulting output, usually a data mapping or a transformation.

Here is an example of a basic DataWeave transformation:

```
%dw 2.0
output application/json
---
payload
```

## Data Types

DataWeave represents data using values, each of which has a data type associated with it. The type system enables you to apply constraints to variables and function parameters. The supported data types include simple types such as `String`, `Boolean`, `Number`, and `Date`, as well as composite and complex types such as `Array`, `Object`, and `Any`.

For a complete list of all supported data types, see [Type System](#).

## Data Selectors

Data selectors enable you to access fields within a data structure so that you can retrieve data from the payload or any other variable created during the DataWeave script execution. Different types of selectors provide you with a variety of options that you can use to extract data from structures.

For example, you can use the *single-value selector* `payload.userName` to retrieve the single value `userName`, where `userName` represents an actual key in the input message.

See [Selectors](#) for a complete list of data selectors and their usage.

## Functions

A function is a code block that contains a group of operations that are executed when the function is called in your application. Functions can accept different parameters as input that can then be used by the operations defined in the function. Additionally, every function returns a value after it finishes processing, and this value can then be used as input for a different function.

Each DataWeave script that you write in any component (like a Transform Message component or Set Payload component) works as an independent program, meaning that all function definitions and calls are unique to the script defined in the component.

See [Define Functions](#) for additional function definition details.

### Function Definition vs. Function Call

DataWeave enables you to [define your own functions](#) in the header of the script, creating a blueprint for the process that defines the operations that are executed upon the function call. You can also define them in [DataWeave mapping and module files](#).

When you *call* a function from a DataWeave expression, you are instructing DataWeave to run all the operations defined in that function at that exact point of the script's execution.

For example, `upper` is a built-in function defined in the String module. You can call this function to convert a text value to uppercase. Also, you can use a selector to retrieve a value from the payload,

and then execute the `upper()` function call with this value as the input parameter `upper(payload.someKey)`.

DataWeave supports both *prefix* notation (`function (param1, param2)`) and *infix* notation (`param1 function param2`) for function calls. Note that you can use infix notation only with functions that accept only two parameters.

See also, [Function Signatures](#).

## Data Operators

Data operators are symbols that enable you to perform operations on data to produce a result. DataWeave supports operators that are common in most programming languages like mathematical (+, -, /, \*), relational (<, >, <=, >=, ==, ~=), and logical (not, !, and, or) operators.

DataWeave also supports the following operators:

- Append (`<<`, `+`), Prepend (`>>`), and Remove (`-`), which enable you to manage data in an array.
- Scope and Flow control operators (`do`, `using`, `if`, `else`) that enable you to create a scope where new variables and functions can be declared, or to execute an expression based on a condition.
- Update (`update`), which enables you to update specific fields of a data structure.

See [DataWeave Operators](#) for a complete list of supported operators, their usage, and examples.

## Import Function Modules

By default, DataWeave scripts import the `dw::Core` function module. To use functions from different modules, import the modules to your script by adding the `import` directive.

See [DataWeave Reference](#) for instructions on using the `import` directive.

## Arrays

An array is a data structure that can contain any number of elements of the supported DataWeave types. To declare an array, use brackets ([ and ]) to enclose a comma-separated list of elements, for example: `[1,2,3]`.

Arrays also support the application of conditional logic to define conditional elements (`[(value1) if condition]`). For example, the expression `[(1) if true, (2) if false, (3) if payload != null]` defines the array `[1,3]` (assuming that `payload` is not `null`).

## Objects

An object is a collection of zero or more `Key` and `value` pairs:

- `Key`, which is formed by:
  - `Name`, composed of a `String` value as the local name and an optional `Namespace`.
  - `Attributes`, a collection of zero or more `Name:value` pairs, where `value` can be any possible value.

- **Value**, comprising any supported type, including another object, a function call, or an array.

To declare an object, use braces ({ and }) to enclose a comma-separated list of **Key:value** pairs:

```
{
  keyName @keyAttributeName :"keyAttributeValue": "value",
  User @(role:"admin"): upper("max the mule"),
  myArray: [1,2,3]
}
```

Objects also support the definition of conditional elements:

```
{
  (userName: name) if !isBlank(name),
  (userRole: role) if !isBlank(role)
}
```

## Functional Programming Principles

The functional programming paradigm enables developers to build applications by using pure function composition and immutable data. Because functional programming is declarative, functions in DataWeave are just expressions that return a value, instead of imperative statements that change program states.

DataWeave follows the core principles of functional programming, including pure functions, first-class functions, high-order functions, function composition, lambda expressions, and immutable data.

### Pure Functions

Pure functions in programming are similar to pure functions in mathematics in that, given a certain input, they always return the same output. Also, executing a function has no impact on other parts of a program, other than producing a result. All functions in DataWeave are pure; they are not affected by external values that could change the result of the execution.

### First-Class Functions

The pure functions in DataWeave are also “first-class citizens” in that they support operations that are available to any other data type, including being stored in variables, passed as parameters, and returned as results.

### High-Order Functions

High-order functions in programming also have the same concept as in mathematics. These functions take one or more functions as parameters and return a function as a result. The result of a high-order function can then be used as an input parameter in another function, enabling the concept of *function composition*.

## Function Composition

Function composition is the process by which functions are combined to build complex operations. In this case, the result of executing one function serves as an input parameter for the next function. In DataWeave, all data transformations are a chain of expressions, functions composed to obtain a final result that outputs the data as needed.

### Example of Function Composition

The following example uses the function `pluck` to map an object into an array. Then the resulting array serves as input for the function `filter`, which selects the elements in the array that have a `Role` equal to `ADMIN`. During the mapping process, the function `upper` transforms the `role` value:

```
%dw 2.0
output application/json
var collaborators =
{
    Max: {role: "admin", id: "01"},
    Einstein: {role: "dev", id: "02"},
    Astro: {role: "admin", id: "03"},
    Blaze: {role: "user", id: "04"}
}
---
collaborators pluck ((value,key,index)-> {
    "Name": key,
    "Role": upper(value.role),
    "ID": value.id
})
filter ((item, index) -> item.Role == "ADMIN")
```

The following example shows the same expression, but it uses prefix notation. This syntax helps visualize how `filter` takes as the first parameter the result of `pluck`.

```
%dw 2.0
output application/json
var collaborators =
{
    Max: {role: "admin", id: "01"},
    Einstein: {role: "dev", id: "02"},
    Astro: {role: "admin", id: "03"},
    Blaze: {role: "user", id: "04"}
}
---
filter(
    pluck(
        collaborators, (value,key,index)->
    {
        "Name": key,
        "Role": upper(value.role),
        "ID": value.id
    }),
    (item, index) -> item.Role == "ADMIN")
```

The output of both expressions is:

```
[
{
    "Name": "Max",
    "Role": "ADMIN",
    "ID": "01"
},
{
    "Name": "Astro",
    "Role": "ADMIN",
    "ID": "03"
}]
```

## Lambda Expressions

Lambda expressions, also known as *anonymous functions* or *function literals*, are function definitions that do not have an identifier name and can be passed as parameters to high-order functions. Lambda syntax is expressed in the form **(a,b) → c**, which can be read as “Given a certain input (**a** and **b**), execute the following (**c**).”

For example, the **map** function in DataWeave accepts a lambda expression as one of its parameters: **map(Array<T>, (item: T, index: Number) → R): Array<R>**. **map** iterates over the items in an array and executes the lambda expression that is passed as a parameter. This lambda expression consists of two parameters that represent the value (**item**) and index (**index**) of each array element during the execution, and a function definition to execute during the iteration **R**.

The following code example shows a `map` function that receives a lambda expression to concatenate the index and the value of each element in the array it iterates on, and then returns these values as a new array:

#### Source

```
%dw 2.0
output application/json
---
["Max", "the", "Mule"] map (item, index) -> index ++ " - " ++ item
```

#### Output

```
[ "0 - Max",
  "1 - the",
  "2 - Mule"
]
```

See [Working with Functions and Lambdas in DataWeave](#) for additional information about Lambda expressions.

## Immutable Data

In DataWeave, variables cannot change their assigned value after they are defined. This ensures that there are no unexpected results from executing functions. If you want to store a new value that is the result of an operation, you must declare a new data structure.

## Differences with Imperative Programming

If you have a development background in working with object-oriented programming languages, it is important to understand that functional programming separates data from functions and does not use loop control statements.

## Separation of Data and Functions

DataWeave does not use assignment statements, which ensures that data remains immutable. Because there are no getter and setter methods, nor classes containing data structures and behavior, each function works only with the data you provide as input. If you need to modify any existing data structure, you create a new copy and apply functions to modify its values.

## Control Flow Statements

There are no loop control statements like `for` or `while` in DataWeave. In DataWeave, instead of defining how to execute a procedure using iteration, you focus on your current requirements when you write the expressions.

For example, to make changes to a list, you use the `map` function to apply changes to the elements in the collection. Additionally, you can use the `filter` function to remove specific elements from a list, or the `reduce` function to collapse all values from an array into one single value.

However, you can use `if-else` statements to apply a condition when executing an expression:

```
%dw 2.0
var myVar = { country : "FRANCE" }
output application/json
---
if (myVar.country == "USA")
  { currency: "USD" }
else { currency: "EUR" }
```

## DataWeave Examples

For usage examples of `map`, `reduce`, and `filter`, see:

- [Map Data](#)
- [Conditionally Reduce a List Via a Function](#)
- [Merge Fields from Separate Objects](#)

## See Also

- [Mule Message Structure](#)
- [Mule Events](#)
- [Core Components](#)
- [Connectors and Modules](#)
- [Transform Message Component](#)

## Scripts

DataWeave is the primary data transformation language for use in Mule flows. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

You can write standalone DataWeave scripts in Transform Message components, or you can write inline DataWeave expressions to transform data *in-place* and dynamically set the value of various properties, such as configuration fields in an event processor or global configuration element. Inline DataWeave expressions are enclosed in `#[ ]` code blocks. For example, you can use a DataWeave expression to set conditions in a Choice router or to set the value of a Set Payload or Set Variable component.

The DataWeave code in this example sets a timestamp variable to the current time using the

DataWeave `now()` function:

*Example: Simple Inline DataWeave Script*

```
<set-variable value="#[now()]" variableName="timestamp" doc:name="Set timestamp" />
```

You can also store DataWeave code in [external files](#) and read them into other DataWeave scripts, or you can factor DataWeave code into modules (libraries) of reusable DataWeave functions that can be shared by all the components in a Mule app.

## The Structure of DataWeave Scripts

DataWeave scripts and files are divided into two main sections:

- The Header, which defines directives that apply to the body expression (optional).
- The Body, which contains the expression to generate the output structure.

When you include a header, the header appears above the body separated by a delimiter consisting of three dashes: `---`.

Here is an example of a DataWeave file with an output directive declared in the header, followed by a DataWeave expression to create a user object that contains two child key/value pairs:

*Example: Simple DataWeave Script*

```
%dw 2.0
output application/xml
---
{
    user: {
        firstName: payload.user_firstname,
        lastName: payload.user_lastname
    }
}
```

## DataWeave Header

This example shows keywords (such as `import` and `var`) you can use for header directives.

## DataWeave Script

```
%dw 2.0
import * from dw::core::Arrays
var myVar=13.15
fun toUser(obj) = {
    firstName: obj.field1,
    lastName: obj.field2
}
type Currency = String { format: "##" }
ns ns0 http://www.abc.com
output application/xml
---
/*
 * Body here.
 */

```

- **%dw**: To maintain backward compatibility with a previous version of DataWeave and avoid introducing syntax-breaking changes from the current version, the **%dw** directive supports the specification of a previous DataWeave version on a per script basis. Use of this directive is optional. The default version is 2.0 if the directive is not present. DataWeave versions from 2.0 to 2.4 are identical because no syntax changes occurred in those versions.

Example: **%dw 2.0**

- **output**: Commonly used directive that specifies the mime type that the script outputs.

Example: **output application/xml**

Valid values: [Supported Data Formats](#).

Default: If no output is specified, the default output is determined by an algorithm that examines the inputs (payload, variables, and so on) used in the script:

1. If there is no input, the default is **output application/java**.
2. If all inputs are the same mime type, the script outputs to the same mime type. For example, if all input is **application/json**, then it outputs **output application/json**.
3. If the mime types of the inputs differ, and no output is specified, the script throws an exception so that you know to specify an output mime type.

Note that only one output type can be specified.

- **import**: For importing a DataWeave function module. See [DataWeave Functions](#).
- **var**: Global variables for defining constants that you can reference throughout the body of the DataWeave script:

*Example*

```
%dw 2.0
var conversionRate=13.15
output application/json
---
{
    price_dollars: payload.price,
    price_localCurrency: payload.price * conversionRate
}
```

For details, see [Variables](#).

- **type**: For specifying a custom type that you can use in the expression.

For a more complete example, see [Type Coercion with DataWeave](#).

- **ns**: Namespaces, used to import a namespace.

*Example*

```
%dw 2.0
output application/xml

ns ns0 http://www.abc.com
ns ns1 http://www.123.com
---
{
    ns0#myroot: {
        ns1#secondroot: "hello world"
    }
}
```

- **fun**: For creating custom functions that can be called from within the body of the script.

*Example*

```
%dw 2.0
output application/json
fun toUser(user) = {firstName: user.name, lastName: user.lastName}
---
{
    user: toUser(payload)
}
```

## Including Headers in Inline DataWeave Scripts

You can include header directives when you write inline DataWeave scripts by flattening all the lines in the DataWeave script into a single line. For smaller DataWeave scripts, this allows you to

quickly apply header directives (without having to add a separate Transform Message component to set a variable), then substitute the variable in the next Event processor.

For example, here is the Mule configuration XML to create the same valid XML output as the previous Transform Message component:

*Example: Simple Inline DataWeave Script*

```
<set-payload value="#[output application/xml --- { myroot: payload } ]" doc:name="Set Payload" />
```

Note that the DataWeave documentation provides numerous [transformation examples](#).

## DataWeave Body

The DataWeave body contains an expression that generates the output structure. Note that MuleSoft provides a canonical way for you to work on data with the DataWeave model: a query, transform, build process.

Here is simple example that provides JSON input for a DataWeave script:

*Example: JSON Input*

```
{  
    "message": "Hello world!"  
}
```

This DataWeave script takes the entire payload of the JSON input above and transforms it to the [application/xml](#) format.

*Example: Script that Outputs application/xml*

```
%dw 2.0  
output application/xml  
---  
payload
```

The next example shows the XML output produced from the DataWeave script:

*Example: XML Output*

```
<?xml version='1.0' encoding='UTF-8'?>  
<message>Hello world!</message>
```

The script above successfully transforms the JSON input to XML output.

## Errors (Scripting versus Formatting Errors)

A DataWeave script can throw errors due to DataWeave coding errors and due to formatting errors.

So when transforming one data format to another, it is important to keep in mind the constraints of both the language and the formats. For example, XML requires a single root node. If you use the [DataWeave script above](#) in the attempt to transform this JSON input to XML, you will receive an error ([Unexpected internal error](#)) because the JSON input lacks a single root:

*Example: JSON Input*

```
{  
  "size" : 1,  
  "person": {  
    "name": "Yoda"  
  }  
}
```

A good approach to the creation of a script is to normalize the input to the JSON-like [application/dw](#) format. In fact, if you get an error, you can simply transform your input to [application/dw](#). If the transformation is successful, then the error is likely a formatting error. If it is unsuccessful, then the error is a coding error.

This example changes the output format to [application/dw](#):

*Example: DataWeave Script that Outputs application/dw*

```
%dw 2.0  
output application/dw  
---  
payload
```

You can see that the script successfully produces [application/dw](#) output from the [JSON input example](#) above:

*Example: application/dw Output*

```
{  
  size: 1,  
  person: {  
    name: "Yoda"  
  }  
}
```

So you know that the previous error ([Unexpected internal error](#)) is specific to the format, not the coding. You can see that the [application/dw](#) output above does not provide a single root element, as required by the XML format. So, to fix the script for *XML* output, you need to provide a single root element to your script, for example:

*Example: Script that Outputs application/xml*

```
%dw 2.0
output application/xml
---
{
    "myroot" : payload
}
```

Now the output meets the requirements of XML, so when you change the output directive back to **application/xml**, the result produces valid XML output.

*Example: XML Output Containing a Single XML Root*

```
<?xml version='1.0' encoding='UTF-8'?>
<myroot>
    <size>1</size>
    <person>
        <name>Yoda</name>
    </person>
</myroot>
```

## DataWeave Comments

Comments that use a Java-like syntax are also accepted by DataWeave.

```
%dw 2.0
output application/json
---
/* Multi-line
 * comment syntax */
payload
// Single-line comment syntax
```



To avoid possible errors in Anypoint Studio, use the multi-line comment syntax(**/\*comment here\*/**) if the first line in the body of your DataWeave script is a comment. Subsequent script comments can use any syntax (**//comment** or **/\*comment\*/**)

## Escape Special Characters

In DataWeave, you use the backslash (\) to escape special characters that you are using in an input string.

See [Escaping Special Characters](#) for more details.

The *input* strings in the DataWeave scripts escape special characters, while the *output* escapes in

accordance with the requirements of the output format, which can vary depending on whether it is [application/json](#), [application/xml](#), [application/csv](#), or some other format.

This example escapes the internal double quotation mark that is surrounded by double quotation marks.

#### *DataWeave Example*

```
%dw 2.0
output application/json
---
{
    "a": "something",
    "b": "dollar sign (\$)",
    "c": 'single quote (\')',
    "d": "double quote (\")",
    "e": `backtick (\`)"
}
```

Notice that the JSON output also escapes the double quotation marks to make the output valid JSON but does not escape the other characters:

#### *JSON Output*

```
{
    "a": "something",
    "b": "dollar sign ($)",
    "c": "single quote (')",
    "d": "double quote (\")",
    "e": "backtick (`)"}
```

The following example escapes the same characters but outputs to XML.

#### *DataWeave Example*

```
%dw 2.0
output application/xml
---
{
    xmlExample:
    {
        "a": "something",
        "b": "dollar sign (\$)",
        "c": 'single quote (\')',
        "d": "double quote (\")",
        "e": `backtick (\`)"}
```

The XML output (unlike JSON output) is valid without escaping the double quotation marks:

#### XML Output

```
<?xml version='1.0' encoding='UTF-8'?>
<xmLExample>
  <a>something</a>
  <b>dollar sign ($)</b>
  <c>single quote ('')</c>
  <d>double quote ("")</d>
  <e>backtick (`)</e>
</xmLExample>
```

## Rules for Declaring Valid Identifiers

To declare a valid identifier, its name must meet the following requirements:

- It must begin with a letter of the alphabet (a-z), either lowercase or uppercase.
- After the first letter, the name can contain any combination of letters, numbers, and underscores (\_).
- The name cannot match any DataWeave reserved keyword (see [Reserved Keywords](#) for a complete list).

Here are some examples of valid identifiers:

- `myType`
- `abc123`
- `a1_3BC_22`
- `Z___4`
- `F`

The following table provides examples of invalid identifiers and a description of what makes them invalid:

Identifier	Issue
<code>123456</code>	Cannot start with a number.
<code>value\$</code>	Cannot contain special characters.
<code>_number</code>	Cannot start with an underscore.
<code>type</code>	Cannot be a reserved keyword.

## Reserved Keywords

Valid identifiers cannot match any of the reserved DataWeave keywords:

- **and**  
Reserved as a [logical operator](#)
- **as**  
Reserved for [type coercion](#)
- **async**  
Reserved
- **case**  
Reserved for `case` statements, for example, to perform [pattern matching](#) or for use with the [update operator](#)
- **default**  
Reserved for assigning [default values](#) for parameters
- **do**  
Reserved for `do` statements
- **else**  
Reserved for use in if-else and else-if statements

[See Flow Control in DataWeave.](#)

- **enum**  
Reserved
- **false**  
Reserved as a Boolean value
- **for**  
Reserved
- **fun**  
Reserved for function definitions in [DataWeave Header](#)
- **if**  
Reserved for if-else and else-if statements

[See Flow Control in DataWeave.](#)

- **import**  
Reserved for the `import` directive in [DataWeave Header](#)
- **input**  
Reserved
- **is**  
Reserved
- **ns**  
Reserved for namespace definitions in [DataWeave Header](#)
- **null**  
Reserved as the value `null`

See [Null \(dw::Core Type\)](#).

- **or**  
Reserved as a [logical operator](#)
- **output**  
Reserved for the `output` directive in [DataWeave Header](#)
- **private**  
Reserved
- **throw**  
Reserved
- **true**  
Reserved as a Boolean value
- **type**  
Reserved for type definitions in [DataWeave Header](#)
- **unless**  
Reserved
- **using**  
Reserved for backward compatibility and replaced by `do`

See [Scope and Flow Control Operators](#).

- **var**  
Reserved for variable definitions in the [DataWeave Header](#)
- **yield**  
Reserved

## dwl File

In addition to specifying DataWeave scripts in the Transform and other components, you can also specify the scripts in a `.dwl` file. In Studio projects, your script files are stored in `src/main/resources`.

In the Mule app XML, you can use the  `${file::filename}` syntax to send a script in a `dwl` file through any XML tag that expects an expression. For example, see the `when expression=" ${file::someFile.dwl}"` in the Choice router here:

```

<http:listener doc:name="Listener" config-ref="HTTP_Listener_config" path="/test">
    <http:response>
        <http:body><![CDATA[#[$file::transform.dwl]]]></http:body>
    </http:response>
</http:listener>
<choice doc:name="Choice">
    <when expression="#{file::someFile.dwl}">
        <set-payload value="It's greater than 4!" doc:name="Set Payload" />
    </when>
    <otherwise>
        <set-payload value="It's less than 4!" doc:name="Set Payload" />
    </otherwise>
</choice>

```

## See Also

- [Selectors](#)
- [DataWeave Functions](#)
- [DataWeave Cookbook](#)
- [Supported Data Formats](#)
- [Functions and Lambdas](#)

## Selectors

DataWeave selectors traverse the structures of objects and arrays and return matching values. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

A selector always operates within a context, which can be a reference to a variable, an object literal, an array literal, or the invocation of a DataWeave function.

Selector Type	Syntax	Return Type
Single-value	<code>.keyName</code>	Any type of value that belongs to a matching key
Multi-value	<code>.*keyName</code>	Array of values of any matching keys
Descendants	<code>..keyName</code>	Array of values of any matching descendant keys
Dynamic	See <a href="#">Dynamic Selector</a> .	
Key-value pair	<code>.@keyName</code>	Object with the matching key

Selector Type	Syntax	Return Type
Index	[<index>]	Value of any type at selected array index. Use negative numbers to index from the end of an array, for example, -1 for the last element in the array. Use of numbers beyond the array size returns <code>null</code> .
Range	[<index> to <index>]	Array with values from selected indexes
XML attribute	@, .@keyName	String value of the selected attribute
Namespace	keyName.#	String value of the namespace for the selected key
Key present	keyName?, keyName.@type?	Boolean ( <code>true</code> if the selected key of an object or XML attribute is present, <code>false</code> if not)
Assert present	keyName!	String: Exception message if the key is not present
Filter	[?(boolean_expression)]	Array or object containing key-value pairs if the DataWeave expression returns <code>true</code> . Otherwise, returns the value <code>null</code> .
Metadata	.^someMetadata	Returns the value of specified metadata for a Mule payload, variable, or attribute. The selector can return the value of class (.^class), content length (.^contentLength), encoding (.^encoding), mime type (.^mimeType), media type (.^mediaType), raw (.^raw), and custom (.^myCustomMetadata) metadata. For details, see <a href="#">Metadata Selector .^someMetadata</a> .



The following examples use these selectors. For additional examples of selectors, see [Extract Data](#).

## Rules for Matching

In DataWeave, a name that matches the key of an object (such as the key "someName" in the JSON object { "someName" : "Data Weave" }) can select data from that object. For a simple example, see [Single-Value Selector](#).

## Single-Value Selector

This selector matches the value of the first key-value pair in which the key matches the given selector name. The selector can be applied to an Object or to an array. On an array, the selector applies to all Object values inside the array and ignores all values that are not Object values. If no key matches, the value `null` is returned.



For information about the dot selector `(.)` and how to use it on DataWeave types, refer to [Selecting Types](#).

## Example: Using a Single-Value Selector on an Object

This example shows how the selector works on a simple JSON object.

### Input Payload

The input payload is an object with the key `"name"` and the value `"DataWeave"`:

```
{ "name": "Data Weave" }
```

### DataWeave Source

The DataWeave script uses `payload.name` to select the value of the object that has the key `"name"`. Note that `payload.name` and `payload."name"` are both valid ways to perform the selection.

```
%dw 2.0
output application/json
---
payload.name
```

### Output

The script outputs the String value of the input object, "Data Weave":

```
"Data Weave"
```

## Example: Using a Single-Value Selector on an Array

On an array, the selector applies to every element.

### Input Payload

The input payload is an array that contains two objects with the same key, `"name"`:

```
[  
  {  
    "name": "Arg"  
  },  
  {  
    "name": "Japan"  
  }  
]
```

## DataWeave Source

The DataWeave script uses `payload.name` to select the value of any objects in input payload that have the key "name":

```
%dw 2.0  
output application/json  
---  
payload.name
```

## Output

The script outputs an array that contains the values of the input objects with the key "`"name": "Arg"`" and "`"Japan"`":

```
["Arg", "Japan"]
```

## Namespace Selector

The DataWeave namespace selector supports the use of a name that matches a namespace or a local part. To match a namespace, the name must match the namespace and the local part. If the name only matches the local part, the selector matches all namespaces with that local part, regardless of the namespace.

Note that XML namespaces and their prefixes are defined in the `xmlns` attribute of an XML element. For example, the element `<h:table xmlns:h="http://www.w3.org/TR/html4/">` defines a namespace that assigns the prefix `h` to the namespace `http://www.w3.org/TR/html4/`. All elements in that namespace contain the `h` prefix. For example, in the element `<h:table/>`, `h` is the prefix for the namespace, and `table` is the local part.

### Example: Using a Namespace Selector

This example selects values from XML elements using the namespace and local part.

#### Input Payload

The input payload contains XML elements, `h:table` and `f:table`, that have different namespaces but the same local names (`table`):

```

<root>

  <h:table xmlns:h="http://www.w3.org/TR/html4/">
    <h:tr>
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
  </h:table>

  <f:table xmlns:f="https://www.w3schools.com/furniture">
    <f:tr>
      <f:name>African Coffee Table</f:name>
      <f:width>80</f:width>
      <f:length>120</f:length>
    </f:tr>
  </f:table>

</root>

```

## DataWeave Source

The DataWeave script selects XML content from specified namespaces. The script's header creates namespace (`ns`) variables `html` and `furniture` to store namespaces from the input payload. To select the children of `h:table` into element `a` and the children of `f:table` elements into element `b`, the script uses the `html` and `furniture` namespace variables with a namespace selector (#) and local part (`table`):

```

%dw 2.0
output application/xml
ns html http://www.w3.org/TR/html4/
ns furniture https://www.w3schools.com/furniture
---
root: {
  a: payload.root.html#table,
  b: payload.root.furniture#table
}

```

## Output

The script outputs children of the `h:table` element under element `a` and the children of the `f:table` under element `b`:

```

<?xml version='1.0' encoding='UTF-8'?>
<root>
  <a>
    <h:tr xmlns:h="http://www.w3.org/TR/html4/">
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
  </a>
  <b>
    <f:tr xmlns:f="https://www.w3schools.com/furniture">
      <f:name>African Coffee Table</f:name>
      <f:width>80</f:width>
      <f:length>120</f:length>
    </f:tr>
  </b>
</root>

```

## Example: Using a Local Name to Select XML Values

This example selects XML values using a local name **table** from the **h:table** element.

### Input Payload

The input payload contains XML with a **table** element that contains the namespace **"http://www.w3.org/TR/html4/":**

```

<root>
  <h:table xmlns:h="http://www.w3.org/TR/html4/">
    <h:tr>
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
  </h:table>
</root>

```

### DataWeave Source

The DataWeave script selects values of children in the **table** element by using a local name of the XML element **table** instead of its namespace:

```

%dw 2.0
output application/xml
---
root: { a: payload.root.table }

```

### Output

The script outputs XML that contains the values of children in the `table` element:

```
<?xml version='1.0' encoding='UTF-8'?>
<root>
  <a>
    <h:tr xmlns:h="http://www.w3.org/TR/html4/">
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
  </a>
</root>
```

## Attribute Selector

The attribute selector returns the first attribute value that matches the selected name expression. If no key matches, the selection returns the value `null`.

The following example selects the value of an XML attribute.

### Input Payload

The input payload is an XML `user` element that contains the attribute `name="Weave"`:

```
<user name="Weave"/>
```

### DataWeave Source

The DataWeave script selects the value of the `name` attribute from the input payload (`<user name="Weave"/>`). Notice that the `@` indicates an attribute selection.

```
%dw 2.0
output application/json
---
payload.user.@name
```

### Output

The DataWeave script outputs the value of `name` attribute.

```
"Weave"
```

## Multi-Value Selector

Instead of returning the value of the first matching key in an array of objects, the multi-value selector (\*) returns an array containing values to the matching key (for example, `*user` where `user` is the key). The selector does not return the values of descendants, only those at the specified level.

If no key matches, the selection returns the value `null`.

## Input Payload

The input payload contains an XML array of `user` elements.

```
<users>
  <user>Weave</user>
  <user>BAT</user>
  <user>TF</user>
</users>
```

## DataWeave Source

The DataWeave script uses `*user` to select the values of all `user` elements from the input payload.

```
%dw 2.0
output application/json
---
payload.users.*user
```

## Output

The script outputs an array of `user` values.

```
[ "Weave", "BAT", "TF" ]
```

## Descendant Selector

The descendant selector returns a list of all children and their descendants. You can directly chain this selector to any other selector without using a single `..`. For example, `payload..` recursively returns an array of all the child values, the values of *their* children, and so on. You can also chain the selector to another element (for example, with `payload..user`) to select the values of each `user` key and its descendants, or you can use `payload..*name` to select the values of all `name` descendants.

### Example: Selecting Each Descendant

This example selects each descendant of the input payload.

## Input Payload

The input payload contains a set of elements that are nested at different levels.

```
<users>
  <user>
    <name>Weave</name>
    <user>
      <name>BAT</name>
      <user>
        <name>BDD</name>
      </user>
    </user>
  </user>
</users>
```

### DataWeave Source

The DataWeave script uses `..` to recursively select all elements in the input payload and return them in a JSON array.

```
%dw 2.0
output application/json
---
payload..
```

### Output

```
[  
 {  
   "user": {  
     "name": "Weave",  
     "user": {  
       "name": "BAT",  
       "user": {  
         "name": "BDD"  
       }  
     }  
   }  
 },  
 {  
   "name": "Weave",  
   "user": {  
     "name": "BAT",  
     "user": {  
       "name": "BDD"  
     }  
   }  
 },  
 "Weave",  
 {  
   "name": "BAT",  
   "user": {  
     "name": "BDD"  
   }  
 },  
 "BAT",  
 {  
   "name": "BDD"  
 },  
 "BDD"  
 ]
```

## Example: Selecting Descending Values

This example selects the descending `name` values.

### Input Payload

The input payload contains a set of `name` elements that are nested at different levels.

```
<users>
  <user>
    <name>Weave</name>
    <user>
      <name>BAT</name>
      <name>Munit</name>
      <user>
        <name>BDD</name>
      </user>
    </user>
  </user>
</users>
```

### DataWeave Source

The DataWeave script uses `..` to select the values of all first `name` elements from the input payload and output those values into a JSON array.

```
%dw 2.0
output application/json
---
payload..name
```

### Output

The script outputs a JSON array with all `name` values.

```
[ "Weave", "BAT", "BDD" ]
```

### Example

This example selects the descending `name` values.

### Input Payload

The input payload contains a set of `name` elements that are nested at different levels.

```
<users>
  <user>
    <name>Weave</name>
    <user>
      <name>BAT</name>
      <name>Munit</name>
      <user>
        <name>BDD</name>
      </user>
    </user>
  </user>
</users>
```

### DataWeave Source

The DataWeave script uses `..*` to select the values of all `name` elements from the input payload and output those values into a JSON array, including the `BAT` and `Munit` values of the repeated `name` keys that are at the same level in the XML hierarchy.

```
%dw 2.0
output application/json
---
payload..*name
```

### Output

The script outputs a JSON array with all `name` values.

```
[ "Weave", "BAT", "Munit", "BDD" ]
```

### Example: Selecting Descending Values

This example selects the descending `Name` values of the `Item` elements.

#### Input Payload

The input payload contains a set of `Name` elements that are nested at different levels.

```

<Example>
  <Brand>
    <Id>32345</Id>
    <logo>circle</logo>
    <Item>
      <Name>Perfume</Name>
      <Item>
        <Name>Bosque</Name>
      </Item>
    </Item>
    <Item>
      <Name>t-Shirt</Name>
    </Item>
  </Brand>
  <Brand>
    <Id>435678C</Id>
    <logo>circle</logo>
    <Item>
      <Name>t-Shirt2</Name>
      <Item>
        <Name>t-Shirt red</Name>
        <Item>
          <Name>t-Shirt red with logo</Name>
        </Item>
      </Item>
    </Item>
  </Brand>
</Example>

```

## DataWeave Source

The DataWeave script uses `..*` to select the values of all `Name` elements nested in the `Item` elements from the input payload and output those values into a XML array.

```

%dw 2.0
output application/xml
---
{
  Supermarket: {
    Item: {
      Value: payload..*Item.Name
    }
  }
}

```

## Output

The script outputs a XML array with all `name` values.

```

<?xml version='1.0' encoding='UTF-8'?>
<Supermarket>
  <Item>
    <Value>Perfume</Value>
    <Value>t-Shirt</Value>
    <Value>Bosque</Value>
    <Value>t-Shirt2</Value>
    <Value>t-Shirt red</Value>
    <Value>t-Shirt red with logo</Value>
  </Item>
</Supermarket>

```

## Dynamic Selector

The syntax for dynamic selection depends on what you are selecting:

- Single Value: `payload[(nameExpression)]`
- Multi Value: `payload[*nameExpression)]`
- Attribute: `payload[@(nameExpression)]`
- Key Value: `payload[&(nameExpression)]`
- Single Value with a Namespace: `payload.ns0#"$(nameExpression)"`

### Example: Dynamically Selecting a Single Value

This example shows how to dynamically select a single value.

#### Input Payload

The input payload is an array of objects. The first object has the key "`ref`" and the value "`name`". The second has the key "`name`" and the value "`Data Weave`":

```
{ "ref": "name", "name": "Data Weave" }
```

#### DataWeave Source

The DataWeave script dynamically selects the value of the "`name`" key:

```
%dw 2.0
output application/json
---
payload[(payload.ref)]
```

Notice that it passes `payload.ref` within the parentheses of dynamic selector `[()]`. The script works because the value of "`ref`" is "`name`", which matches the key "`name`".

## Output

The script outputs the value of the object that has the "name" key:

```
"Data Weave"
```

## Example: Dynamically Selecting a Single Value with a Namespace

This example shows how to dynamically select a single value that contains a namespace.

### Input Payload

The input payload contains a `<root>` element with two child elements, one of which has the namespace `http://www.w3.org/TR/html4/`. Both child elements (`<f:table>` and `<h:table>`) have the local name `table`:

```
<root ref="table">
  <f:table xmlns:f="https://www.w3schools.com/furniture">Manzana</f:table>
  <h:table xmlns:h="http://www.w3.org/TR/html4/">Banana</h:table>
</root>
```

### DataWeave Source

The DataWeave script dynamically selects the value of the element that has the namespace `http://www.w3.org/TR/html4/`:

```
%dw 2.0
output application/json
ns h http://www.w3.org/TR/html4/
---
payload.root.h#"$(payload.root.@ref)"
```

Notice that the expression `payload.root.@ref` uses the attribute selector (`@`) on the `ref` attribute of the `root` element to select the value `table`, which matches local name `table` in the element `<h:table xmlns:h="http://www.w3.org/TR/html4/">Banana</h:table>`.

## Output

The script outputs the value of the element that has the namespace `http://www.w3.org/TR/html4/`:

```
"Banana"
```

## Use of Selectors on Content Stored in Variables

All selectors work with the [predefined Mule Runtime variables](#) and with [DataWeave variables](#).

To select [Mule event](#) data, such as the payload, an attribute, or variable data within the Mule event,

use predefined Mule Runtime variables, such as `payload`, `attributes`, and `vars`.

Extracted values are handled as literal values of one of the supported DataWeave value types.

Data to extract	Syntax
<b>Payload</b>	<code>payload</code> , for example: <code>payload.name</code>  If the <code>payload</code> is <code>{"name" : "somebody"}</code> , <code>payload.name</code> returns <code>"somebody"</code> .  For more on the Mule payload, see <a href="#">Message Payload</a> .
<b>Attribute</b>	<code>attributes.&lt;myAttributeName&gt;</code>  For examples, see <a href="#">Attributes</a> .
<b>Variable</b>	<code>&lt;myVariableName&gt;</code>  To avoid name collisions, you must prepend <code>vars</code> to the variable name:  <code>vars.&lt;myVariableName&gt;</code>  For more on Mule variables, see <a href="#">Variables in Mule Apps</a> .
<b>Error object</b>	<code>error</code>  For information on errors in the flow, you can use <code>#[\$error.cause]</code> .
<b>Flow</b>	<code>flow</code>  For the flow name in the Logger: <code>#[\$flow.name]</code>  Note that <code>flow.name</code> does not work in some Core components, such as Set Payload and Transform Message.  For more on flows, see <a href="#">Flows and Subflows</a> .

## Selecting Types

Using the dot selector `(.)` over DataWeave `types` enables you to declare new types from existing ones. Combined with the `module loader`, DataWeave can also load and translate declarations from a custom module file into DataWeave type directives that can be accessed in the same way as types from any other DataWeave module.

### Declare a New Type from an Existing Type

The following example declares the `NameType` from the `User` type. Using the dot selector in `User.name` to declare the `name` variable as `NameType` enables the `typeOf()` function to return the primitive type, `"String"`. The DataWeave variable `name` is declared as a `NameType` value `"Seba"`:

*DataWeave script:*

```
%dw 2.0
type User = {name: String}
type NameType = User.name
var name: NameType = "Seba"
---
typeOf(name)
```

*Output:*

```
"String"
```

## Compose a Type from an Existing Type's Field

The following example uses the dot selector (.) to declare a new type **Address** from one key in the complex type **User**:

*DataWeave script:*

```
%dw 2.0
output application/json
type User = {
    address : {country: String, city: String, street: String, number: Number},
    userName : String
}
type Address = User.address

var userAddress: Address = {country: "Argentina", city: "Rosario", street: "Calle
Falsa", number: 123}
---
userAddress
```

*Output:*

```
{
  "country": "Argentina",
  "city": "Rosario",
  "street": "Calle Falsa",
  "number": 123
}
```

## Select Types from Union Types

The following example shows how you can use type selection with **Union** types that consist of **Object** types:

*DataWeave script:*

```
%dw 2.0
output application/json
type ID = {id: String, firstName: String, age: Number} | {id: Number, firstName: String, secondName: String}

var nameAge: ID = {firstName: "Seba", id: "123", age: 28}
var fullName: ID = {id: 38123, firstName: "Seba", secondName: "Elizalde"}

var nameString: ID.firstName = "Seba"
var idNumber: ID.id = 38123
---
[nameAge, fullName, nameString ++ " is of type " ++ typeOf(nameString), idNumber ++ " is of type " ++ typeOf(idNumber)]
```

*Output:*

```
[{"name": "Seba"}, {"id": 38123}, "Seba is of type String", "38123 is of type Number"]
```

The following script fails because the second type is not an **Object** type and there is no way to reference it with the selector:

*DataWeave script:*

```
%dw 2.0
output application/json
type UnionType = {name: String} | Number
type Fail = UnionType.name
---
{}
```

*Output:*

```
Cannot do selection on Type: Number
4| type Fail = UnionType.name
```

## Select Types from Namespaces

The following example shows type selection of a namespace:

*DataWeave script:*

```
%dw 2.0
output application/json
ns ns1 http://acme.com
type User = {
    ns1#name : String,
    name : Number,
}
var nameString: User.ns1#name = "Seba"
var nameNumber: User.name = 123
---
{
    "NameString" : typeOf(nameString),
    "NameNumber" : typeOf(nameNumber)
}
```

*Output:*

```
{
    "NameString": "String",
    "NameNumber": "Number"
}
```

## Select Types with Type Parameters

This example shows how type selection works with type parameters, which are similar to generics in other programming languages:

*DataWeave script:*

```
%dw 2.0
output application/json
type WithParameters<A, B> = {first: A, second: B, nestedObject: {message: A}}
---
{
    a: true is WithParameters<Boolean, Number>.first,
    b: 4592 is WithParameters<String, Number>.second,
    c: "sdf" is WithParameters<String, Number>.nestedObject.message,
}
```

*Output:*

```
{  
  "a": true,  
  "b": true,  
  "c": true  
}
```

## Preserve Metadata of a Type

This example shows that type selection preserves metadata associated with the type:

*DataWeave script:*

```
%dw 2.0  
output application/json  
type User = {  
  birthDate: Date {format: "dd-MMM-yy"},  
  userName : String {schema: "value"}  
}  
type FormattedDate = User.birthDate  
type UserName = User.userName  
var formattedDate: FormattedDate = "10-SEP-15" as Date {format: "dd-MMM-yy"}  
var otherFormatDate = "23-10-2022" as Date {format: "dd-MM-yyyy"}  
var(userName) = "Messi" as String {schema: "value"}  
var(otherUserName) = "Di María" as String {schema: "otherValue"}  
---  
{  
  formattedDate: formattedDate is FormattedDate,  
  userName: userName is UserName,  
  otherFormatDate: otherFormatDate is FormattedDate,  
  otherUserName: otherUserName is UserName  
}
```

*Output:*

```
{  
  "formattedDate": true,  
  "userName": true,  
  "otherFormatDate": false,  
  "otherUserName": false  
}
```

## Supported Data Formats

DataWeave can read and write many types of data formats, such as JSON, XML, and many others. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can

use the version selector in the DataWeave table of contents.

DataWeave supports the following formats as input and output:

MIME Type	ID	Supported Formats
application/avro	avro	Avro Format
application/csv	csv	CSV Format
application/dw	dw	DataWeave Format (dw) for testing a DataWeave expression
application/flatfile	flatfile	Flat File Format, COBOL Copybook Format, Fixed Width Format
application/java	java	Java Format, Enum Format for Java
application/json	json	JSON Format
application/octet-stream	binary	Binary Format
application/protobuf, application/x-protobuf	protobuf	Protobuf Format
application/xlsx	excel	Excel Format (XLSX)
application/xml	xml	XML Format, CData Custom Type
application/x-ndjson	ndjson	Newline Delimited JSON Format (ndjson)
application/x-www-form-urlencoded	urlencoded	URL Encoded Format
application/yaml	yaml	YAML Format
multipart/form-data	multipart	Multipart Format
text/plain	text	Text Plain Format
text/x-java-properties	properties	Text Java Properties

## DataWeave Readers

DataWeave can read input data as a whole by loading it into memory or by indexing it in local storage and, for some data formats, DataWeave can read data sequentially in parts by streaming the input. The indexed reading strategy implements a mechanism to avoid out-of-memory issues when processing large files. The streaming strategy can improve performance but restricts file access to sequential access (which disallows random access) because you can access only the part of the file that is being read and stored in memory.

Read Strategy	Description	Supported Formats
In-Memory	<p>This strategy parses the entire document and loads it into memory, enabling random access to data. When using this strategy, a DataWeave script can access any part of the resulting value at any time.</p>	<p>DataWeave can read all supported formats using this strategy.</p>
Indexed	<p>This strategy parses the entire document and uses disk space to avoid out-of-memory issues on large files, enabling random access to data. When using this strategy, a DataWeave script can access any part of the resulting value at any time.</p> <p>When processing a <b>String</b> with a size larger than 1.5 MB, DataWeave automatically splits the value in chunks to avoid out-of-memory issues. This feature works only with <b>JSON</b> and <b>XML</b> input data.</p> <p>For additional details, see <a href="#">Indexed Readers in DataWeave</a></p>	<ul style="list-style-type: none"> <li>• <b>CSV</b></li> <li>• <b>JSON</b></li> <li>• <b>XML</b></li> </ul>
Streaming	<p>This strategy partitions the input document into smaller items and accesses the data sequentially, storing the current item in memory. A DataWeave selector can access only the portion of the file that is getting read.</p> <p>For additional details, see <a href="#">Streaming in DataWeave</a></p>	<ul style="list-style-type: none"> <li>• <b>CSV</b></li> <li>• <b>JSON</b></li> <li>• <b>Excel</b> (XLSX)</li> <li>• <b>XML</b></li> </ul>

## Using Reader and Writer Properties

In some cases, it is necessary to modify or specify aspects of the format through format-specific properties. For example, you can specify CSV input and output properties, such as the **separator** (or delimiter) to use in the CSV file. For Cobol copybook, you need to specify the path to a schema file using the **schemaPath** property.

You can append reader properties to the MIME type (`outputMimeType`) attribute for certain components in your Mule app. Listeners and Read operations accept these settings. For example, this On New File listener example identifies the `,` separator for a CSV input file:

*Example: Properties for the CSV Reader*

```
<file:listener doc:name="On New File" config-ref="File_Config"
outputMimeType='application/csv; separator=",">
    <scheduling-strategy>
        <fixed-frequency frequency="45" timeUnit="SECONDS"/>
    </scheduling-strategy>
    <file:matcher filenamePattern="comma_separated.csv" />
</file:listener>
```

Note that the `outputMimeType` setting above helps the CSV *reader* interpret the format and delimiter of the input `comma_separated.csv` file, not the writer.

To specify the output format, you can provide the MIME type and any writer properties for the writer, such as the CSV or JSON writer used by a File Write operation. For example, you might need to write a pipe (`|`) delimiter in your CSV output payload, instead of some other delimiter used in the input. To do this, you append the property and its value to the `output` directive of a DataWeave expression. For example, this Write operation specifies the pipe as a `separator`:

*Example: output Directive for the CSV Writer*

```
<file:write doc:name="Write" config-ref="File_Config" path="my_transform">
    <file:content><![CDATA[#[output application/csv separator="|" ---
payload]]]></file:content>
</file:write>
```

The following example shows a DataWeave script in which the `output` directive specifies two writer properties, one for the CSV field separator, another to indicate that there is no header.

```
%dw 2.0
output application/csv separator=";", header=false
---
payload
```

Writer properties depend on the format. The following example uses the JSON writer property, `skipNullOn`:

```
%dw 2.0
output application/json skipNullOn="everywhere"
---
payload
```

The following example shows a DataWeave script that uses an invisible ASCII character as a

separator for the CSV output. The separator property value is `\u001E`. `\u` is the escape sequence representation and `001E` is the hexadecimal representation of the record separator (RS) ASCII character.

```
%dw 2.0
output application/csv separator = "\u001E"
---
[
  {
    "Name" : "Tom",
    "UID" : 2569,
    "ShipOrderID" : "ui-288188"
  },
  {
    "Name" : "Alan",
    "UID" : 7756,
    "ShipOrderID" : "xj-232142"
  },
  {
    "Name" : "Dan",
    "UID" : 7821,
    "ShipOrderID" : "uk-259459"
  }
]
```

The CSV example produces the following output:

```
Name|UID|ShipOrderID
Tom|2569|ui-288188
Alan|7756|xj-232142
Dan|7821|uk-259459
```

Scripts that use the `read` or `readUrl` methods can also specify reader properties through the `readerProperties` parameter. See examples in [read](#). However, note that use of reader properties is more common in message sources, such as the HTTP listener, as shown in the [CSV reader example](#).

It is also possible to use a Mule variable as a value of a configuration property. For more complete examples, see [Set Reader and Writer Configuration Properties](#).

## Setting MIME Types

You can specify the MIME type for the input and output data that flows through a Mule app.

For DataWeave transformations, you can specify the MIME type for the output data. For example, you might set the `output` header directive of an expression in the Transform Message component or a Write operation to `output application/json` or `output application/csv`.

This example sets the MIME type through a File Write operation to ensure that a format-specific

writer, the CSV writer, outputs the payload in CSV format:

*Example: MIME Type for the CSV Writer*

```
<file:write doc:name="Write" config-ref="File_Config" path="my_transform">
    <file:content><![CDATA[#[output application/csv --- payload]]]></file:content>
</file:write>
```

Starting in Mule 4.3.0, you can set the `output` directive using the format ID alone, instead of using the MIME type. For example, you might set the `output` header directive of an expression in the Transform Message component or a Write operation to `output json` or `output csv`. You can also use the format ID to differentiate the format of the output data from the MIME type in the output header, and you can use a custom MIME type. For example, you can write JSON data but customize the MIME type to `application/problem+json` by using the DataWeave directive `output application/problem+json with json`. See [Change a Script's MIME Type Output](#) for examples that customize the MIME type output of a script.

The `with` keyword separates the output MIME type from the DataWeave writer for the script's output. For example, you can specify 'output application/csv with binary' to output the `application/csv` MIME type when using the `binary` writer. This setting is necessary in cases shown in [Use Dynamic Writer Properties \(Mule\)](#) and [Use Reader and Writer Properties in DataWeave Functions](#). It is also possible to append writer properties, such as in the use case `output application/my-custom-type with json indent=false`.

For input data, format-specific readers for Mule sources (such as the On New File listener), Mule operations (such as Read and HTTP Request operations), and DataWeave expressions attempt to infer the MIME type from metadata that is associated with input payloads, attributes, and variables in the Mule event. When the MIME type cannot be inferred from the metadata (and when that metadata is not static), Mule sources and operations allow you to specify the MIME type for the reader. For example, you might set the MIME type for the On New File listener to `outputMimeType='application/csv'` for CSV file *input*. This setting provides information about the file format to the CSV reader.

*Example: MIME Type for the CSV Reader*

```
<file:listener doc:name="On New File"
    config-ref="File_Config"
    outputMimeType='application/csv'>
</file:listener>
```

Note that reader settings *are not* used to perform a transformation from one format to another. They simply help the reader interpret the format of the input.

You can also set special reader and writer properties for use by the format-specific reader or writer of a source, operation, or component. See [Using Reader and Writer Properties](#).

## See Also

[Transform Message Component](#)

## Avro Format

MIME type: `application/avro`

ID: `avro`

Avro is a binary data format that uses a schema to structure its data. DataWeave relies on the schema to parse the data. Avro data structures are mapped to DataWeave data structures.

### Java Value Mapping

The following table shows how Avro types map to DataWeave types.

Avro Type	DataWeave Type
<code>long</code>	<code>Number</code>
<code>int</code>	<code>Number</code>
<code>double</code>	<code>Number</code>
<code>boolean</code>	<code>Boolean</code>
<code>string</code>	<code>String</code>
<code>fixed</code>	<code>String</code>
<code>bytes</code>	<code>Binary</code>
<code>enum</code>	<code>String</code>
<code>map</code>	<code>Object</code>
<code>array</code>	<code>Array</code>
<code>null</code>	<code>Null</code>

### Example: Use an Avro Schema

The following example shows how to specify a schema that the writer uses to output an Avro data structure.

#### Input

An Avro schema looks something like this.

*schema.json:*

```
{  
  "type": "record",  
  "name": "userInfo",  
  "namespace": "my.example",  
  "fields": [  
    {  
      "name": "username",  
      "type": "string",  
      "default": "NONE"  
    },  
    {  
      "name": "age",  
      "type": "int",  
      "default": -1  
    },  
    {  
      "name": "phone",  
      "type": "string",  
      "default": "NONE"  
    },  
    {  
      "name": "housenum",  
      "type": "string",  
      "default": "NONE"  
    }  
  ]  
}
```

## Source

The `schemaUrl` property in the header of this DataWeave script passes a schema (`schema.json`) to the DataWeave writer. The writer uses the schema to structure content from the body of the script and output the results in Avro format.

```
%dw 2.0
output application/avro schemaUrl="classpath://schema.json"
---
[{
    username: "Mariano",
    age: 35,
    phone: "213",
    housenum: "123"
},
{
    username: "Leandro",
    age: 29,
    phone: "213",
    housenum: "123"
},
{
    username: "Christian",
    age: 25,
    phone: "213",
    housenum: "123"
}]
}
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
schemaUrl (Required)	String	''	The URL for the Avro schema. Valid URL schemes are <code>classpath://</code> , <code>file://</code> , or <code>http://</code> . For the reader, this property is optional but defaults to the schema embedded in the input Avro file. The reader requires an embedded schema. For the writer, DataWeave requires a schema value.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer. The value must be greater than 8.

Parameter	Type	Default	Description
<code>deferred</code>	<code>Boolean</code>	<code>false</code>	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
<code>schemaUrl</code> (Required)	<code>String</code>	<code>''</code>	The URL for the Avro schema. Valid URL schemes are <code>classpath://</code> , <code>file://</code> , or <code>http://</code> . For the reader, this property is optional but defaults to the schema embedded in the input Avro file. The reader requires an embedded schema. For the writer, DataWeave requires a schema value.

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>application/avro</code>

## Binary Format

MIME type: `application/octet-stream`

ID: `binary`

The Binary data format handles binary content, such as an image or PDF. Such content is represented as a `Binary` type.

Note that Binary content is stored into RAM memory.

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

There are no reader properties for this format.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to true, and defers the script's execution until the generated content is consumed.  Valid values are true or false.

## Supported MIME Types

This format supports the following MIME types.

MIME Type
application/octet-stream
application/x-binary

## COBOL Copybook Format

MIME Type: application/flatfile

ID: flatfile

A COBOL copybook is a type of flat file that describes the layout of records and fields in a COBOL data file.

The Transform Message component provides settings for handling the COBOL copybook format. For example, you can import a COBOL definition into the Transform Message component and use it for your Copybook transformations.



COBOL copybook in DataWeave supports files of up to 15 MB, and the memory requirement is roughly 40 to 1. For example, a 1-MB file requires up to 40 MB of memory to process, so it's important to consider this memory requirement in conjunction with your TPS needs for large copybook files. This is not an exact figure; the value might vary according to the complexity of the mapping instructions.

## Importing a Copybook Definition

When you import a Copybook definition, the Transform Message component converts the definition to a flat file schema that you can reference with `schemaPath` property.

To import a copybook definition:

1. Right-click the input payload in the Transform component in Studio, and select **Set Metadata** to open the Set Metadata Type dialog.

Note that you need to create a metadata type before you can import a copybook definition.

2. Provide a name for your copybook metadata, such as `copybook`.
3. Select the Copybook type from the **Type** drop-down menu.
4. Import your copybook definition file.
5. Click Select.

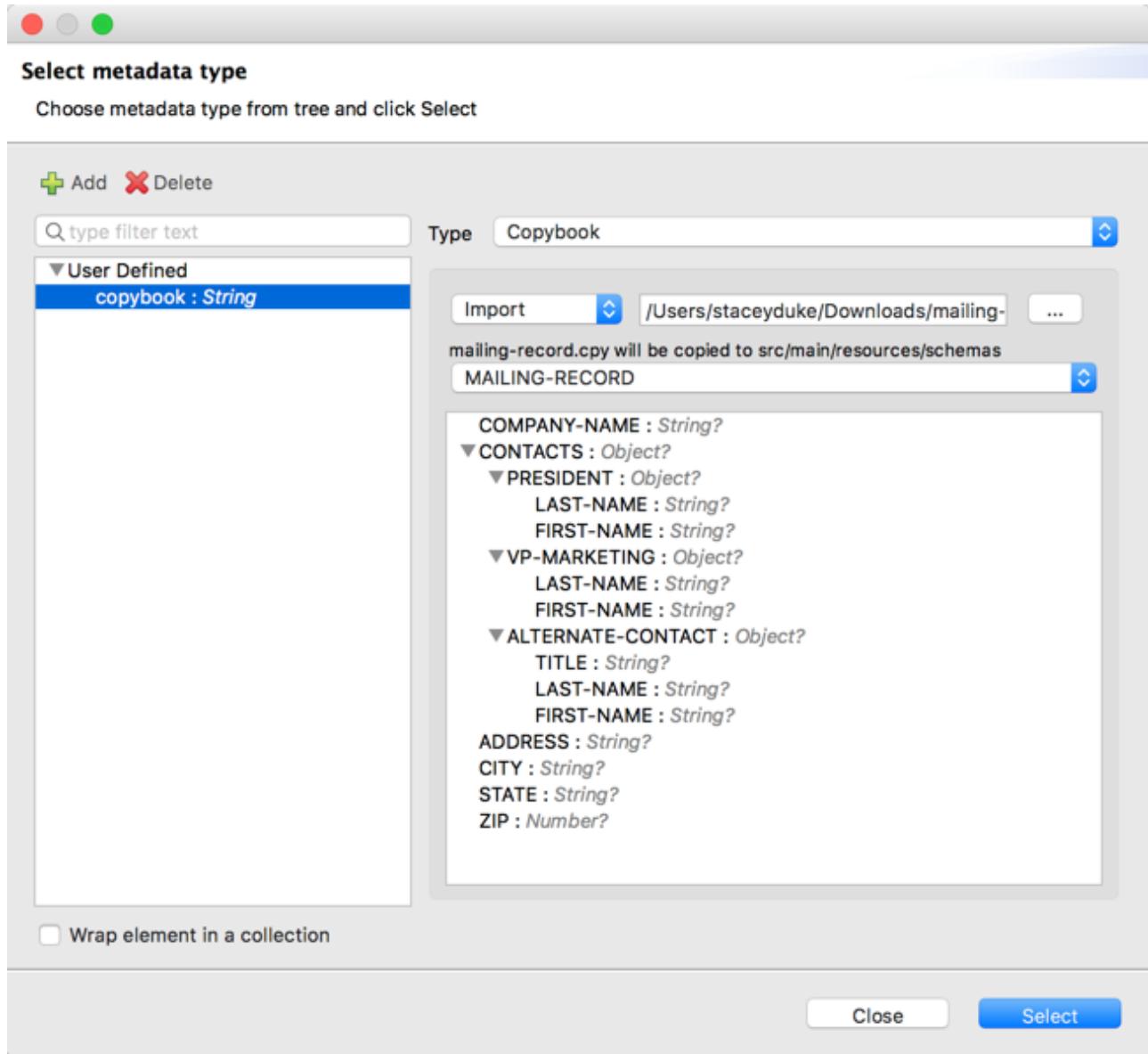


Figure 2. Importing a Copybook Definition File

For example, assume that you have a copybook definition file (`mailing-record.cpy`) that looks like this:

```

01 MAILING-RECORD.
  05 COMPANY-NAME          PIC X(30).
  05 CONTACTS.
    10 PRESIDENT.
      15 LAST-NAME        PIC X(15).
      15 FIRST-NAME       PIC X(8).
    10 VP-MARKETING.
      15 LAST-NAME        PIC X(15).
      15 FIRST-NAME       PIC X(8).
    10 ALTERNATE-CONTACT.
      15 TITLE             PIC X(10).
      15 LAST-NAME        PIC X(15).
      15 FIRST-NAME       PIC X(8).
  05 ADDRESS              PIC X(15).
  05 CITY                 PIC X(15).
  05 STATE                PIC XX.
  05 ZIP                  PIC 9(5).

```

- Copybook definitions must always begin with a **01** entry. A separate record type is generated for each **01** definition in your copybook (there must be at least one **01** definition for the copybook to be usable, so add one using an arbitrary name at the start of the copybook if none is present). If there are multiple **01** definitions in the copybook file, you can select which definition to use in the transform from the dropdown list.
- COBOL format requires definitions to only use columns 7-72 of each line. Data in columns 1-5 and past column 72 is ignored by the import process. Column 6 is a line continuation marker.

When you import the schema, the Transform component converts the copybook file to a flat file schema that it stores in the `src/main/resources/schema` folder of your Mule project. In flat file format, the copybook definition above looks like this:

```

form: COPYBOOK
id: 'MAILING-RECORD'
values:
- { name: 'COMPANY-NAME', type: String, length: 30 }
- name: 'CONTACTS'
  values:
    - name: 'PRESIDENT'
      values:
        - { name: 'LAST-NAME', type: String, length: 15 }
        - { name: 'FIRST-NAME', type: String, length: 8 }
    - name: 'VP-MARKETING'
      values:
        - { name: 'LAST-NAME', type: String, length: 15 }
        - { name: 'FIRST-NAME', type: String, length: 8 }
    - name: 'ALTERNATE-CONTACT'
      values:
        - { name: 'TITLE', type: String, length: 10 }
        - { name: 'LAST-NAME', type: String, length: 15 }
        - { name: 'FIRST-NAME', type: String, length: 8 }
- { name: 'ADDRESS', type: String, length: 15 }
- { name: 'CITY', type: String, length: 15 }
- { name: 'STATE', type: String, length: 2 }
- { name: 'ZIP', type: Integer, length: 5, format: { justify: ZEROS, sign: UNSIGNED } }
}

```

After importing the copybook, you can use the `schemaPath` property to reference the associated flat file through the `output` directive. For example: `output application/flatfile schemaPath="src/main/resources/schemas/mailling-record.ffd"`

## Supported Copybook Features

Not all copybook features are supported by the COBOL Copybook format in DataWeave. In general, the format supports most common usages and simple patterns, including:

- USAGE of DISPLAY, BINARY (COMP), COMP-5, and PACKED-DECIMAL (COMP-3). For character encoding restrictions, see [Character Encodings](#).
- PICTURE clauses for numeric values consisting only of:
  - '9' - One or more numeric character positions
  - 'S' - One optional sign character position, leading or trailing
  - 'V' - One optional decimal point
  - 'P' - One or more decimal scaling positions
- PICTURE clauses for alphanumeric values consisting only of 'X' character positions
- Repetition counts for '9', 'P', and 'X' characters in PICTURE clauses (as in `9(5)` for a 5-digit numeric value)
- OCCURS DEPENDING ON with `controlVal` property in schema. Note that if the control value is nested inside a containing structure, you need to manually modify the generated schema to

specify the full path for the value in the form "container.value".

- REDEFINES clause (used to provide different views of the same portion of record data - see details in section below)

*Unsupported* features include:

- Alphanumeric-edited PICTURE clauses
- Numeric-edited PICTURE clauses, including all forms of insertion, replacement, and zero suppression
- Special level-numbers:
  - Level 66 - Alternate name for field or group
  - Level 77 - Independent data item
  - Level 88 - Condition names (equivalent to an enumeration of values)
- SIGN clause at group level (only supported on elementary items with PICTURE clause)
- USAGE of COMP-1 or COMP-2 and of clause at group level (only supported on elementary items with PICTURE clause)
- VALUE clause (used to define a value of a data item or conditional name from a literal or another data item)
- SYNC clause (used to align values within a record)

## REDEFINES Support

REDEFINES facilitates dynamic interpretation of data in a record. When you import a copybook with REDEFINES present, the generated schema uses a special grouping with the name '\*' (or '\*1', '\*2', and so on, if multiple REDEFINES groupings are present at the same level) to combine all the different interpretations. You use this special grouping name in your DataWeave expressions just as you use any other grouping name.

Use of REDEFINES groupings has higher overhead than normal copybook groupings, so MuleSoft recommends that you remove REDEFINES from your copybooks where possible before you import them into Studio.

## Character Encodings

BINARY (COMP), COMP-5, or PACKED-DECIMAL (COMP-3) usages are only supported with single-byte character encodings, which use the entire range of 256 potential character codes. UTF-8 and other variable-length encodings are not supported for these usages (because they're not single-byte), and ASCII is also not supported (because it doesn't use the entire range). Supported character encodings include ISO-8859-1 (an extension of ASCII to full 8 bits) and other 8859 variations and EBCDIC (IBM037).

REDEFINES requires you to use a single-byte-per-character character encoding for the data, but any single-byte-per-character encoding can be used unless BINARY (COMP), COMP-5, or PACKED-DECIMAL (COMP-3) usages are included in the data.

## Common Copybook Import Issues

The most common issue with copybook imports is a failure to follow the COBOL standard for input line regions. The copybook import parsing ignores the contents of columns 1-6 of each line, and ignores all lines with an '\*' (asterisk) in column 7. It also ignores everything beyond column 72 in each line. This means that all your actual data definitions need to be within columns 8 through 72 of input lines.

Tabs in the input are not expanded because there is no defined standard for tab positions. Each tab character is treated as a single space character when counting copybook input columns.

Indentation is ignored when processing the copybook, with only level-numbers treated as significant. This is not normally a problem, but it means that copybooks might be accepted for import even though they are not accepted by COBOL compilers.

Both warnings and errors might be reported as a result of a copybook import. Warnings generally tell of unsupported or unrecognized features, which might or might not be significant. Errors are notifications of a problem that means the generated schema (if any) will not be a completely accurate representation of the copybook. You should review any warnings or errors reported and decide on the appropriate handling, which might be simply accepting the schema as generated, modifying the input copybook, or modifying the generated schema.

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>allowLenientWithBinaryNotEndElement</code>	Boolean	<code>false</code>	<p>When the schema contains elements of type Binary or Packed, the <code>lenient</code> option does not allow short records, regardless of the last element's format type. When you set this property to <code>true</code>, the validation applies only to cases in which the record ends with type Binary or Packed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>enforceRequires</code>	Boolean	<code>false</code>	<p>Produces an error when set to <code>true</code> if a required value is missing.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>missingValues</code>	<code>String</code>	<code>null</code>	<p>Fill character used to represent missing values. To activate a non-default setting, set the <code>useMissCharAsDefaultForFill</code> property to <code>true</code>, and use one of the following values to <code>missingValues</code>:</p> <ul style="list-style-type: none"> <li>• <code>none</code> (for the reader) or <code>NONE</code> (for the writer): Treats all data as values.</li> <li>• <code>spaces</code> (for the reader) or <code>SPACES</code> (for the writer): Interprets a field consisting of only spaces as a missing value. Default for flat file and fixed-width formats.</li> <li>• <code>zeroes</code> (for the reader) or <code>ZEROES</code> (for the writer): Interprets numeric fields consisting of only <code>0</code> characters and character fields consisting of only spaces as missing values.</li> <li>• <code>nulls</code> (for the reader) or <code>NULLS</code> (for the writer): Interprets a field consisting only of <code>0</code> bytes as a missing value. Default for COBOL copybook schema.</li> </ul>
<code>notTruncateDependingOnSubjectNotPresent</code>	<code>Boolean</code>	<code>false</code>	<p>Fills the entire group when the DEPENDING ON subject is not present.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>recordParsing</code>	<code>String</code>	<code>'strict'</code>	<p>Specifies the expected type of separation between lines or records:</p> <ul style="list-style-type: none"> <li>• <code>strict</code>: Line break is expected at exact end of each record. <code>strict</code> is the default.</li> <li>• <code>lenient</code>: Line break is used, but records can be shorter or longer than the schema specifies. Do not use <code>lenient</code> if your payload lacks line breaks. The other options to <code>recordParsing</code> support records that lack line breaks.</li> <li>• <code>noTerminator</code>: Records follow one another with no separation. This option is preferred for fixed-length records that lack a line break.</li> <li>• <code>singleRecord</code>: The entire input is a single record.</li> </ul> <p>Note that schemas with type <code>Binary</code> or <code>Packed</code> don't allow for line break detection, so setting <code>recordParsing</code> to <code>lenient</code> only allows long records to be handled, not short ones. These schemas also currently only work with certain single-byte character encodings (so not with UTF-8 or any multibyte format).</p>
<code>retainEmptyStringFieldsOnParsing</code>	<code>Boolean</code>	<code>false</code>	<p>Allow parsing behavior to keep missing string value fields with a default value in the output map</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>schemaPath</code> (Required)	<code>String</code>	<code>null</code>	Path to the schema definition. Specifies the location in your local disk of the schema file that parses your input.
<code>segmentIdent</code>	<code>String</code>	<code>null</code>	Segment identifier in the schema for fixed-width or COBOL copybook schemas. Required when parsing a single segment or record definition if the schema includes multiple segment definitions.

Parameter	Type	Default	Description
structureIdent	String	null	Structure identifier in the schema for flat file schemas. Required when parsing a structure definition if the schema includes multiple structure definitions.
truncateDependingOn	Boolean	false	For COBOL copybook, truncates DEPENDING ON values to the length used.  Valid values are <code>true</code> or <code>false</code> .
useMissCharAsDefaultForFill	Boolean	false	By default, the flat file reader and writer use spaces for missing characters and ignore the setting of the <code>missingValues</code> property. When you set this property to <code>true</code> , DataWeave honors the setting of the <code>missingValues</code> property. Introduced in DataWeave 2.3 (2.3.0-20210823) for the September 2021 release of Mule 4.3.0-20210823.  Valid values are <code>true</code> or <code>false</code> .
zonedDecimalStrict	Boolean	false	For COBOL copybook, uses the 'strict' ASCII form of sign encoding for zoned decimal values.  Valid values are <code>true</code> or <code>false</code> .

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
encoding	String	null	The encoding to use for the output, such as UTF-8.

Parameter	Type	Default	Description
<code>enforceRequires</code>	<code>Boolean</code>	<code>false</code>	<p>Produces an error when set to <code>true</code> if a required value is missing.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>missingValues</code>	<code>String</code>	<code>null</code>	<p>Fill character used to represent missing values. To activate a non-default setting, set the <code>useMissCharAsDefaultForFill</code> property to <code>true</code>, and use one of the following values to <code>missingValues</code>:</p> <ul style="list-style-type: none"> <li>• <code>none</code> (for the reader) or <code>NONE</code> (for the writer): Treats all data as values.</li> <li>• <code>spaces</code> (for the reader) or <code>SPACES</code> (for the writer): Interprets a field consisting of only spaces as a missing value. Default for flat file and fixed-width formats.</li> <li>• <code>zeroes</code> (for the reader) or <code>ZEROES</code> (for the writer): Interprets numeric fields consisting of only <code>0</code> characters and character fields consisting of only spaces as missing values.</li> <li>• <code>nulls</code> (for the reader) or <code>NULLS</code> (for the writer): Interprets a field consisting only of <code>0</code> bytes as a missing value. Default for COBOL copybook schema.</li> </ul>
<code>notTruncateDependingOnSubjectNotPresent</code>	<code>Boolean</code>	<code>false</code>	<p>Fills the entire group when the DEPENDING ON subject is not present.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>recordTerminator</code>	<code>String</code>	<code>null</code>	<p>Line break for a record separator. Valid values: <code>lf</code>, <code>cr</code>, <code>crlf</code>, <code>none</code>. Note that in Mule versions 4.0.4 and later, this is only used as a separator when there are multiple records. Values translate directly to character codes (<code>none</code> leaves no termination on each record).</p>
<code>schemaPath</code> (Required)	<code>String</code>	<code>null</code>	<p>Path to the schema definition. Specifies the location in your local disk of the schema file that parses your input.</p>

Parameter	Type	Default	Description
segmentIdent	String	null	Segment identifier in the schema for fixed-width or COBOL copybook schemas. Required when parsing a single segment or record definition if the schema includes multiple segment definitions.
structureIdent	String	null	Structure identifier in the schema for flat file schemas. Required when parsing a structure definition if the schema includes multiple structure definitions.
trimValues	Boolean	false	Trim values that are longer than the width of a field.  Valid values are <code>true</code> or <code>false</code> .
truncateDependingOn	Boolean	false	For COBOL copybook, truncates DEPENDING ON values to the length used.  Valid values are <code>true</code> or <code>false</code> .
useMissCharAsDefaultForFill	Boolean	false	By default, the flat file reader and writer use spaces for missing characters and ignore the setting of the <code>missingValues</code> property. When you set this property to <code>true</code> , DataWeave honors the setting of the <code>missingValues</code> property. Introduced in DataWeave 2.3 (2.3.0-20210823) for the September 2021 release of Mule 4.3.0-20210823.  Valid values are <code>true</code> or <code>false</code> .
zonedDecimalStrict	Boolean	false	For COBOL copybook, uses the 'strict' ASCII form of sign encoding for zoned decimal values.  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

The COBOL Copybook format supports the following MIME types.

MIME Type
<code>*/flatfile</code>

## CSV Format

MIME type: [application/csv](#)

ID: [csv](#)

The CSV data format is represented as a DataWeave array of objects in which each object represents a row. All simple values are represented as strings.

The DataWeave reader for CSV input supports the following parsing strategies:

- Indexed
- In-Memory
- Streaming

By default, the CSV reader stores input data from an entire file in-memory if the file is 1.5MB or less. If the file is larger than 1.5 MB, the process writes the data to disk. For very large files, you can improve the performance of the reader by setting a streaming property to true.

For additional details, see [DataWeave Readers](#).

## Examples

The following examples show uses of the CSV format.

- [Example: Represent CSV Data](#)
- [Example: Stream CSV Data](#)

### Example: Represent CSV Data

The following example shows how DataWeave represents CSV data.

#### Input

The following sample data serves as input for the DataWeave source.

```
name,lastname,age,gender
Mariano,de Achaval,37,male
Paula,de Estrada,37,female
```

#### Source

The DataWeave script transforms the CSV input payload to the DataWeave (dw) format and MIME type.

```
%dw 2.0
output application/dw
---
payload
```

## Output

The DataWeave script produces the following output.

```
[{
  {
    name: "Mariano",
    lastname: "de Achaval",
    age: "37",
    gender: "male"
  },
  {
    name: "Paula",
    lastname: "de Estrada",
    age: "37",
    gender: "female"
  }
]
```

## Example: Stream CSV Data

By default, the CSV reader stores input data from an entire file in-memory if the file is 1.5MB or less. If the file is larger than 1.5 MB, the process writes the data to disk. For very large files, you can improve the performance of the reader by setting a `streaming` property to `true`. To demonstrate the use of this property, the next example streams a CSV file and transforms it to JSON.

## Input

The structure of the CSV input looks something like the following. Note that a streamed file is typically much longer.

*CSV File Input for Streaming Example (truncated):*

```
street,city,zip,state,beds,baths,sale_date
3526 HIGH ST,SACRAMENTO,95838,CA,2,1,Wed May 21 00:00:00 EDT 2018
51 OMAHA CT,SACRAMENTO,95823,CA,3,1,Wed May 21 00:00:00 EDT 2018
2796 BRANCH ST,SACRAMENTO,95815,CA,2,1,Wed May 21 00:00:00 EDT 2018
2805 JANETTE WAY,SACRAMENTO,95815,CA,2,1,Wed May 21 00:00:00 EDT 2018
6001 MCMAHON DR,SACRAMENTO,95824,CA,2,1,,Wed May 21 00:00:00 EDT 2018
5828 PEPPERMILL CT,SACRAMENTO,95841,CA,3,1,Wed May 21 00:00:00 EDT 2018
```

## XML Configuration

To demonstrate a use of the **streaming** property, the following Mule flow streams a CSV file and transforms it to JSON.

```
<flow name="dw-streamingFlow" >
  <scheduler doc:name="Scheduler" >
    <scheduling-strategy >
      <fixed-frequency frequency="1" timeUnit="MINUTES"/>
    </scheduling-strategy>
  </scheduler>
  <file:read
    path="${app.home}/input.csv"
    config-ref="File_Config"
    outputMimeType="application/csv; streaming=true; header=true"/>
  <ee:transform doc:name="Transform Message" >
    <ee:message >
      <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
payload map ((row) -> {
  zipcode: row.zip
})]]></ee:set-payload>
    </ee:message>
  </ee:transform>
  <file:write doc:name="Write"
    config-ref="File_Config1"
    path="/path/to/output/file/output.json"/>
  <logger level="INFO" doc:name="Logger" message="#[payload]"/>
</flow>
```

- The example configures the Read operation (`<file:read/>`) to stream the CSV input by setting `outputMimeType="application/csv; streaming=true"`. The input CSV file is located in the project directory, `src/main/resources`, which is the location of  `${app.home}`.
- The DataWeave script in the **Transform Message** component uses the `map` function to iterate over each row in the CSV payload and select the value of each field in the `zip` column.
- The Write operation returns a file, `output.json`, which contains the result of the transformation.
- The Logger prints the same output payload that you see in `output.json`.

## Output

The CSV streaming example produces the following output.

```
[
  {
    "zipcode": "95838"
  },
  {
    "zipcode": "95823"
  },
  {
    "zipcode": "95815"
  },
  {
    "zipcode": "95815"
  },
  {
    "zipcode": "95824"
  },
  {
    "zipcode": "95841"
  }
]
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
bodyStartLineNumber	Number	0	Line number on which the body starts.
escape	String	\	Character to use for escaping special characters, such as separators or quotes.
header	Boolean	true	<p>Indicates whether a CSV header is present.</p> <ul style="list-style-type: none"> <li>If <code>header=true</code>, you can access the fields within the input by name, for example, <code>payload.userName</code>.</li> <li>If <code>header=false</code>, you must access the fields by index, referencing the entry first and the field next, for example, <code>payload[107][2]</code>.</li> </ul> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
headerLineNum er	Number	0	Line number on which the CSV header is located.
ignoreEmptyLin e	Boolean	true	Indicates whether to ignore an empty line.  Valid values are <code>true</code> or <code>false</code> .
quote	String	"	Character to use for quotes.
separator	String	,	Character that separates one field from another field.
streaming	Boolean	false	Streams input when set to <code>true</code> . Use only if entries are accessed sequentially. The input must be a top-level array. See the <a href="#">streaming example</a> , and see <a href="#">DataWeave Readers</a> .  Valid values are <code>true</code> or <code>false</code> .

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bodyStartLineN umber	Number	0	Line number on which the body starts.
bufferSize	Number	8192	Size of the buffer writer. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
encoding	String	null	The encoding to use for the output, such as UTF-8.
escape	String	\	Character to use for escaping special characters, such as separators or quotes.

Parameter	Type	Default	Description
header	Boolean	true	<p>Indicates whether a CSV header is present.</p> <ul style="list-style-type: none"> <li>If <code>header=true</code>, you can access the fields within the input by name, for example, <code>payload.userName</code>.</li> <li>If <code>header=false</code>, you must access the fields by index, referencing the entry first and the field next, for example, <code>payload[107][2]</code>.</li> </ul> <p>Valid values are <code>true</code> or <code>false</code>.</p>
headerLineNumber	Number	0	Line number on which the CSV header is located.
ignoreEmptyLine	Boolean	true	<p>Indicates whether to ignore an empty line.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
lineSeparator	String	New Line	Line separator to use when writing CSV, for example, " <code>\r\n</code> ". By default, DataWeave uses the system line separator.
quote	String	"	Character to use for quotes.
quoteHeader	Boolean	false	<p>Quotes header values when set to <code>true</code>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
quoteValues	Boolean	false	<p>Quotes every value when set to <code>true</code>, including values that contain special characters.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
separator	String	,	Character that separates one field from another field.

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>*/csv</code>

## DataWeave Format (dw)

MIME type: `application/dw`

ID: `dw`

The DataWeave (dw) format is the canonical format for all transformations. This format can help you understand how input data is interpreted before it is transformed to a new format.



This format is intended to help you debug the results of DataWeave transformations. It is significantly slower than other formats. It is not recommended to be used in production applications because it can impact the performance.

### Example: Express XML in the DataWeave (dw) Format

The following example shows how to transform XML input to the DataWeave (dw) format.

#### Input

The example uses the following XML snippet as input.

```
<employees>
  <employee>
    <firstname>Mariano</firstname>
    <lastname>DeAchaval</lastname>
  </employee>
  <employee>
    <firstname>Leandro</firstname>
    <lastname>Shokida</lastname>
  </employee>
</employees>
```

#### Output

*Output: in DataWeave Format*

```
{
  employees: {
    employee: {
      firstname: "Mariano",
      lastname: "DeAchaval"
    },
    employee: {
      firstname: "Leandro",
      lastname: "Shokida"
    }
  }
} as Object {encoding: "UTF-8", mimeType: "text/xml"}
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
externalResources	Boolean	false	Enables the <code>readUrl</code> to read external entities. Valid values are <code>true</code> or <code>false</code> .
javaModule	Boolean	false	Enables Java module functions to load. Valid values are <code>true</code> or <code>false</code> .
onlyData	Boolean	false	Handles only data and not other types of content, such as functions, when set to <code>true</code> . The DataWeave parser runs faster in the <code>onlyData</code> mode. Valid values are <code>true</code> or <code>false</code> .
privileges	String	''	Accepts a comma-separated list of privileges to use in the format, such as ' <code>Resources,Properties</code> '.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed. Valid values are <code>true</code> or <code>false</code> .
ignoreSchema	Boolean	false	Ignores the schema when set to <code>true</code> . Valid values are <code>true</code> or <code>false</code> .
indent	String	''	String to use for indenting.

Parameter	Type	Default	Description
maxCollectionSize	Number	-1	Maximum number of elements allowed in an array or an object. -1 indicates no limitation.
onlyData	Boolean	true	<p>Handles only data and not other types of content, such as functions, when set to true. The DataWeave parser runs faster in the <code>onlyData</code> mode.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

## Supported MIME Types

This format supports the following MIME types.

MIME Type
*/dw

## Excel Format (XLSX)

MIME type: `application/xlsx`

ID: `excel`

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.



Only `.xlsx` files are supported (Excel 2007). `.xls` files are not supported by Mule.

The DataWeave reader for Excel input supports the following parsing strategies:

- In-Memory
- Streaming

To understand the parsing strategies that DataWeave readers and writers can apply to this format, see [DataWeave Parsing Strategies](#).

## Excel Type Mapping

The following table shows how Excel types map to DataWeave types.

Excel Type	DataWeave Type
String	String
Numeric	Number
Boolean	Boolean

Excel Type	DataWeave Type
Data	Date

## Examples

The following examples show uses of the Excel format.

- [Example: Represent Excel in the DataWeave \(dw\) Format](#)
- [Example: Output an Excel Table](#)
- [Example: Stream Excel Input](#)

### Example: Represent Excel in the DataWeave (dw) Format

This example shows how DataWeave represents an Excel workbook.

#### Input

The Excel workbook ([Sheet1](#)) serves as an input payload for the DataWeave source.

*Sheet1:*

.	A	B
1	Id	Name
2	123	George
3	456	Lucas

#### Source

The DataWeave script transforms the Excel input payload to the DataWeave (dw) format and MIME type.

```
%dw 2.0
output application/dw
---
payload
```

#### Output

The DataWeave output looks like this. You can select values the same way you select values in other objects.

```
{
  "Sheet1": [
    {
      "A": 123,
      "B": "George"
    },
    {
      "A": 456,
      "B": "Lucas"
    }
  ]
}
```

#### **Example: Output an Excel Table**

The following DataWeave script outputs an Excel table with the header and fields.

The body of this DataWeave script is a DataWeave object that defines the content of the Excel sheet. The name of the sheet, **Sheet1**, is the key of this object. The value is an array of objects. Each object in the array contains a collection of key-value pairs. The keys in each pair are treated as header values for the spreadsheet. The values in each pair are treated as data values for a row in the sheet.

The output directive indicates that the output is the Excel format and MIME type. The **header=true** setting indicates that the output includes the header values.

```
%dw 2.0
output application/xlsx header=true
---
{
  Sheet1: [
    {
      Id: 123,
      Name: George
    },
    {
      Id: 456,
      Name: Lucas
    }
  ]
}
```

For another example, see [Look Up Data in an Excel \(XLSX\) File \(Mule\)](#).

#### **Example: Stream Excel Input**

By default, the Excel reader stores input data from an entire file in-memory if the file is 1.5MB or less. If the file is larger than 1.5 MB, the process writes the data to disk. For very large files, you can improve the performance of the reader by setting a **streaming** property to **true**.

The following Configuration XML for a Mule application streams an Excel file and transforms it to JSON.

```
<http:listener-config
    name="HTTP_Listener_config"
    doc:name="HTTP Listener config" >
    <http:listener-connection host="0.0.0.0" port="8081" />
</http:listener-config>
<flow name="streaming_flow" >
    <http:listener
        doc:name="Listener"
        config-ref="HTTP_Listener_config"
        path="/"
        outputMimeType="application/xlsx; streaming=true"/>
    <ee:transform doc:name="Transform Message" >
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
payload."Sheet Name" map ((row) -> {
    foo: row.a,
    bar: row.b
})]]></ee:set-payload>
        </ee:message>
    </ee:transform>
</flow>
```

The example:

- Configures the HTTP listener to stream the XLSX input by setting `outputMimeType="application/xlsx; streaming=true"`. In the Studio UI, you can use the **MIME Type** on the listener to `application/xlsx` and the **Parameters** for the MIME Type to **Key streaming** and **Value true**.
- Uses a DataWeave script in the **Transform Message** component to iterate over each row in the XLSX payload (an XLSX sheet called `"Sheet Name"`) and select the values of each cell in the row (using `row.a`, `row.b`). It assumes columns named `a` and `b` and maps the values from each row in those columns into `foo` and `bar`, respectively.

## Output

The following image shows the Excel table output.

	A	B	C	D
1	Id	Name		
2	123	George		
3	456	Lucas		
4				
5				
6				

## Limitations

- Macros are currently not supported.
- Charts are ignored.
- Pivot tables are not supported.
- Formatting is currently not supported.

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
header	Boolean	true	Indicates whether the first line of the output contains header field names.  Valid values are <code>true</code> or <code>false</code> .
ignoreEmptyLine	Boolean	true	Ignores an empty line by default.  Valid values are <code>true</code> or <code>false</code> .
streaming	Boolean	false	Streams input when set to <code>true</code> . Use only if entries are accessed sequentially. The input must be a top-level array. See the <a href="#">streaming example</a> , and see <a href="#">DataWeave Readers</a> .  Valid values are <code>true</code> or <code>false</code> .
tableLimit	String	'Unbounded'	Position of the last column in each row. Accepts a pattern <code>&lt;Column&gt;</code> (for example, <code>'A'</code> or <code>'AB'</code> ), the value <code>'HeaderSize'</code> , which uses the location of the last header, or <code>'Unbounded'</code> , which consumes each row.
tableOffset	String	null	Sets the position of the first cell. Accepts the pattern <code>&lt;Column&gt;&lt;Row&gt;</code> , for example, <code>A1</code> or <code>B3</code> .
zipBombCheck	Boolean	true	Turns off the zip bomb (decompression bomb) check when set to <code>false</code> .  Valid values are <code>true</code> or <code>false</code> .

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
header	Boolean	true	Indicates whether the first line of the output contains header field names.  Valid values are <code>true</code> or <code>false</code> .
ignoreEmptyLine	Boolean	true	Ignores an empty line by default.  Valid values are <code>true</code> or <code>false</code> .
tableOffset	String	null	Sets the position of the first cell. Accepts the pattern <code>&lt;Column&gt;&lt;Row&gt;</code> , for example, <code>A1</code> or <code>B3</code> .
zipBombCheck	Boolean	true	Turns off the zip bomb (decompression bomb) check when set to <code>false</code> .  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

This format supports the following MIME types.

MIME Type
application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
application/xlsx

## Fixed Width Format

MIME Type: `application/flatfile`

ID: `flatfile`

Fixed width types are technically considered a type of Flat File format, but when selecting this option, the Transform component offers you settings that are better tailored to the needs of this format.

 Fixed width in DataWeave supports files of up to 15 MB, and the memory requirement is roughly 40 to 1. For example, a 1-MB file requires up to 40 MB of memory to process, so it's important to consider this memory requirement in conjunction with your TPS needs for large fixed width files. This is not an exact figure; the value might vary according to the complexity of the mapping instructions.

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>allowLenientWidthBinaryNotEndElement</code>	Boolean	<code>false</code>	<p>When the schema contains elements of type Binary or Packed, the <code>lenient</code> option does not allow short records, regardless of the last element's format type. When you set this property to <code>true</code>, the validation applies only to cases in which the record ends with type Binary or Packed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>enforceRequires</code>	Boolean	<code>false</code>	<p>Produces an error when set to <code>true</code> if a required value is missing.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>missingValues</code>	<code>String</code>	<code>null</code>	<p>Fill character used to represent missing values. To activate a non-default setting, set the <code>useMissCharAsDefaultForFill</code> property to <code>true</code>, and use one of the following values to <code>missingValues</code>:</p> <ul style="list-style-type: none"> <li>• <code>none</code> (for the reader) or <code>NONE</code> (for the writer): Treats all data as values.</li> <li>• <code>spaces</code> (for the reader) or <code>SPACES</code> (for the writer): Interprets a field consisting of only spaces as a missing value. Default for flat file and fixed-width formats.</li> <li>• <code>zeroes</code> (for the reader) or <code>ZEROES</code> (for the writer): Interprets numeric fields consisting of only <code>0</code> characters and character fields consisting of only spaces as missing values.</li> <li>• <code>nulls</code> (for the reader) or <code>NULLS</code> (for the writer): Interprets a field consisting only of <code>0</code> bytes as a missing value. Default for COBOL copybook schema.</li> </ul>
<code>notTruncateDependingOnSubjectNotPresent</code>	<code>Boolean</code>	<code>false</code>	<p>Fills the entire group when the DEPENDING ON subject is not present.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>recordParsing</code>	<code>String</code>	<code>'strict'</code>	<p>Specifies the expected type of separation between lines or records:</p> <ul style="list-style-type: none"> <li>• <code>strict</code>: Line break is expected at exact end of each record. <code>strict</code> is the default.</li> <li>• <code>lenient</code>: Line break is used, but records can be shorter or longer than the schema specifies. Do not use <code>lenient</code> if your payload lacks line breaks. The other options to <code>recordParsing</code> support records that lack line breaks.</li> <li>• <code>noTerminator</code>: Records follow one another with no separation. This option is preferred for fixed-length records that lack a line break.</li> <li>• <code>singleRecord</code>: The entire input is a single record.</li> </ul> <p>Note that schemas with type <code>Binary</code> or <code>Packed</code> don't allow for line break detection, so setting <code>recordParsing</code> to <code>lenient</code> only allows long records to be handled, not short ones. These schemas also currently only work with certain single-byte character encodings (so not with UTF-8 or any multibyte format).</p>
<code>retainEmptyStringFieldsOnParsing</code>	<code>Boolean</code>	<code>false</code>	<p>Allow parsing behavior to keep missing string value fields with a default value in the output map</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>schemaPath</code> (Required)	<code>String</code>	<code>null</code>	Path to the schema definition. Specifies the location in your local disk of the schema file that parses your input.
<code>segmentIdent</code>	<code>String</code>	<code>null</code>	Segment identifier in the schema for fixed-width or COBOL copybook schemas. Required when parsing a single segment or record definition if the schema includes multiple segment definitions.

Parameter	Type	Default	Description
structureIdent	String	null	Structure identifier in the schema for flat file schemas. Required when parsing a structure definition if the schema includes multiple structure definitions.
truncateDependingOn	Boolean	false	For COBOL copybook, truncates DEPENDING ON values to the length used.  Valid values are <code>true</code> or <code>false</code> .
useMissCharAsDefaultForFill	Boolean	false	By default, the flat file reader and writer use spaces for missing characters and ignore the setting of the <code>missingValues</code> property. When you set this property to <code>true</code> , DataWeave honors the setting of the <code>missingValues</code> property. Introduced in DataWeave 2.3 (2.3.0-20210823) for the September 2021 release of Mule 4.3.0-20210823.  Valid values are <code>true</code> or <code>false</code> .
zonedDecimalStrict	Boolean	false	For COBOL copybook, uses the 'strict' ASCII form of sign encoding for zoned decimal values.  Valid values are <code>true</code> or <code>false</code> .

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
encoding	String	null	The encoding to use for the output, such as UTF-8.

Parameter	Type	Default	Description
<code>enforceRequires</code>	<code>Boolean</code>	<code>false</code>	<p>Produces an error when set to <code>true</code> if a required value is missing.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>missingValues</code>	<code>String</code>	<code>null</code>	<p>Fill character used to represent missing values. To activate a non-default setting, set the <code>useMissCharAsDefaultForFill</code> property to <code>true</code>, and use one of the following values to <code>missingValues</code>:</p> <ul style="list-style-type: none"> <li>• <code>none</code> (for the reader) or <code>NONE</code> (for the writer): Treats all data as values.</li> <li>• <code>spaces</code> (for the reader) or <code>SPACES</code> (for the writer): Interprets a field consisting of only spaces as a missing value. Default for flat file and fixed-width formats.</li> <li>• <code>zeroes</code> (for the reader) or <code>ZEROES</code> (for the writer): Interprets numeric fields consisting of only <code>0</code> characters and character fields consisting of only spaces as missing values.</li> <li>• <code>nulls</code> (for the reader) or <code>NULLS</code> (for the writer): Interprets a field consisting only of <code>0</code> bytes as a missing value. Default for COBOL copybook schema.</li> </ul>
<code>notTruncateDependingOnSubjectNotPresent</code>	<code>Boolean</code>	<code>false</code>	<p>Fills the entire group when the DEPENDING ON subject is not present.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>recordTerminator</code>	<code>String</code>	<code>null</code>	<p>Line break for a record separator. Valid values: <code>lf</code>, <code>cr</code>, <code>crlf</code>, <code>none</code>. Note that in Mule versions 4.0.4 and later, this is only used as a separator when there are multiple records. Values translate directly to character codes (<code>none</code> leaves no termination on each record).</p>
<code>schemaPath</code> (Required)	<code>String</code>	<code>null</code>	<p>Path to the schema definition. Specifies the location in your local disk of the schema file that parses your input.</p>

Parameter	Type	Default	Description
segmentIdent	String	null	Segment identifier in the schema for fixed-width or COBOL copybook schemas. Required when parsing a single segment or record definition if the schema includes multiple segment definitions.
structureIdent	String	null	Structure identifier in the schema for flat file schemas. Required when parsing a structure definition if the schema includes multiple structure definitions.
trimValues	Boolean	false	Trim values that are longer than the width of a field.  Valid values are <code>true</code> or <code>false</code> .
truncateDependingOn	Boolean	false	For COBOL copybook, truncates DEPENDING ON values to the length used.  Valid values are <code>true</code> or <code>false</code> .
useMissCharAsDefaultForFill	Boolean	false	By default, the flat file reader and writer use spaces for missing characters and ignore the setting of the <code>missingValues</code> property. When you set this property to <code>true</code> , DataWeave honors the setting of the <code>missingValues</code> property. Introduced in DataWeave 2.3 (2.3.0-20210823) for the September 2021 release of Mule 4.3.0-20210823.  Valid values are <code>true</code> or <code>false</code> .
zonedDecimalStrict	Boolean	false	For COBOL copybook, uses the 'strict' ASCII form of sign encoding for zoned decimal values.  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types (for Fixed Width)

The Fixed Width format supports the following MIME types.

MIME Type
<code>*/flatfile</code>

## Flat File Format

MIME type: `application/flatfile`

ID: `flatfile`

The Flat File format supports multiple types of fixed width records within a single message. The schema structure allows you to define how different record types are distinguished, and how the records are logically grouped.



Flat File in DataWeave supports files of up to 15 MB, and the memory requirement is roughly 40 to 1. For example, a 1-MB file requires up to 40 MB of memory to process, so it's important to consider this memory requirement in conjunction with your TPS needs for large flat files. This is not an exact figure; the value might vary according to the complexity of the mapping instructions.

### Example: Specify a Flat File Schema

This example shows a DataWeave script that outputs data in the Flat File format. Notice that it uses the `schemaPath` and `structureIndent` writer properties.

```
%dw 2.0
output application/flatfile schemaPath="src/main/resources/test-data/QBReqRsp.esl",
structureIndent="QBResponse"
---
payload
```

### Configuration Properties

DataWeave supports the following configuration properties for this format.

#### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>allowLenientWithBinaryNotEndElement</code>	Boolean	<code>false</code>	<p>When the schema contains elements of type Binary or Packed, the <code>lenient</code> option does not allow short records, regardless of the last element's format type. When you set this property to <code>true</code>, the validation applies only to cases in which the record ends with type Binary or Packed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>enforceRequires</code>	<code>Boolean</code>	<code>false</code>	<p>Produces an error when set to <code>true</code> if a required value is missing.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>missingValues</code>	<code>String</code>	<code>null</code>	<p>Fill character used to represent missing values. To activate a non-default setting, set the <code>useMissCharAsDefaultForFill</code> property to <code>true</code>, and use one of the following values to <code>missingValues</code>:</p> <ul style="list-style-type: none"> <li>• <code>none</code> (for the reader) or <code>NONE</code> (for the writer): Treats all data as values.</li> <li>• <code>spaces</code> (for the reader) or <code>SPACES</code> (for the writer): Interprets a field consisting of only spaces as a missing value. Default for flat file and fixed-width formats.</li> <li>• <code>zeroes</code> (for the reader) or <code>ZEROES</code> (for the writer): Interprets numeric fields consisting of only <code>0</code> characters and character fields consisting of only spaces as missing values.</li> <li>• <code>nulls</code> (for the reader) or <code>NULLS</code> (for the writer): Interprets a field consisting only of <code>0</code> bytes as a missing value. Default for COBOL copybook schema.</li> </ul> <p>Valid values are <code>none</code> or <code>spaces</code> or <code>zeroes</code> or <code>nulls</code>.</p>
<code>notTruncateDependingOnSubjectNotPresent</code>	<code>Boolean</code>	<code>false</code>	<p>Fills the entire group when the DEPENDING ON subject is not present.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>recordParsing</code>	<code>String</code>	<code>'strict'</code>	<p>Specifies the expected type of separation between lines or records:</p> <ul style="list-style-type: none"> <li>• <code>strict</code>: Line break is expected at exact end of each record. <code>strict</code> is the default.</li> <li>• <code>lenient</code>: Line break is used, but records can be shorter or longer than the schema specifies.</li> <li>• <code>noTerminator</code>: Records follow one another with no separation.</li> <li>• <code>singleRecord</code>: The entire input is a single record.</li> </ul> <p>Note that schemas with type <code>Binary</code> or <code>Packed</code> don't allow for line break detection, so setting <code>recordParsing</code> to <code>lenient</code> only allows long records to be handled, not short ones. These schemas also currently only work with certain single-byte character encodings (so not with UTF-8 or any multibyte format).</p> <p>Valid values are <code>strict</code> or <code>lenient</code> or <code>noTerminator</code> or <code>singleRecord</code>.</p>
<code>retainEmptyStringFieldsOnParsing</code>	<code>Boolean</code>	<code>false</code>	<p>Allow parsing behavior to keep missing string value fields with a default value in the output map</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>schemaPath</code> (Required)	<code>String</code>	<code>null</code>	Path to the schema definition. Specifies the location in your local disk of the schema file that parses your input.
<code>segmentIdent</code>	<code>String</code>	<code>null</code>	Segment identifier in the schema for fixed-width or COBOL copybook schemas. Required when parsing a single segment or record definition if the schema includes multiple segment definitions.
<code>structureIdent</code>	<code>String</code>	<code>null</code>	Structure identifier in the schema for flat file schemas. Required when parsing a structure definition if the schema includes multiple structure definitions.

Parameter	Type	Default	Description
truncateDependingOn	Boolean	false	<p>For COBOL copybook, truncates DEPENDING ON values to the length used.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
useMissCharAsDefaultForFill	Boolean	false	<p>By default, the flat file reader and writer use spaces for missing characters and ignore the setting of the <code>missingValues</code> property. When you set this property to <code>true</code>, DataWeave honors the setting of the <code>missingValues</code> property.</p> <p>Introduced in DataWeave 2.3 (2.3.0-20210823) for the September 2021 release of Mule 4.3.0-20210823.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
zonedDecimalStrict	Boolean	false	<p>For COBOL copybook, uses the 'strict' ASCII form of sign encoding for zoned decimal values.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	<p>Size of the buffer writer, in bytes. The value must be greater than 8.</p>
deferred	Boolean	false	<p>Generates the output as a data stream when set to <code>true</code>, and defers the script's execution until the generated content is consumed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
encoding	String	null	<p>The encoding to use for the output, such as UTF-8.</p>
enforceRequires	Boolean	false	<p>Produces an error when set to <code>true</code> if a required value is missing.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

Parameter	Type	Default	Description
<code>missingValues</code>	<code>String</code>	<code>null</code>	<p>Fill character used to represent missing values. To activate a non-default setting, set the <code>useMissCharAsDefaultForFill</code> property to <code>true</code>, and use one of the following values to <code>missingValues</code>:</p> <ul style="list-style-type: none"> <li>• <code>NONE</code>: Write nothing for missing values.</li> <li>• <code>spaces</code> (for the reader) or <code>SPACES</code> (for the writer): Interprets a field consisting of only spaces as a missing value. Default for flat file and fixed-width formats.</li> <li>• <code>zeroes</code> (for the reader) or <code>ZEROES</code> (for the writer): Interprets numeric fields consisting of only <code>0</code> characters and character fields consisting of only spaces as missing values.</li> <li>• <code>nulls</code> (for the reader) or <code>NULLS</code> (for the writer): Interprets a field consisting only of <code>0</code> bytes as a missing value. Default for COBOL copybook schema.</li> </ul> <p>Valid values are <code>none</code> or <code>spaces</code> or <code>zeroes</code> or <code>nulls</code>.</p>
<code>notTruncateDependingOnSubjectNotPresent</code>	<code>Boolean</code>	<code>false</code>	<p>Fills the entire group when the DEPENDING ON subject is not present.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
<code>recordTerminator</code>	<code>String</code>	<code>null</code>	<p>Line break for a record separator. Valid values: <code>lf</code>, <code>cr</code>, <code>crlf</code>, <code>none</code>. DataWeave uses this property as a separator only when there are multiple records. Values translate directly to character codes, and <code>none</code> leaves no termination on each record.</p>
<code>schemaPath</code> (Required)	<code>String</code>	<code>null</code>	<p>Path to the schema definition. Specifies the location in your local disk of the schema file that parses your input.</p>
<code>segmentIdent</code>	<code>String</code>	<code>null</code>	<p>Segment identifier in the schema for fixed-width or COBOL copybook schemas. Required when parsing a single segment or record definition if the schema includes multiple segment definitions.</p>

Parameter	Type	Default	Description
structureIdent	String	null	Structure identifier in the schema for flat file schemas. Required when parsing a structure definition if the schema includes multiple structure definitions.
trimValues	Boolean	false	Trim values that are longer than the width of a field.  Valid values are <code>true</code> or <code>false</code> .
truncateDependingOn	Boolean	false	For COBOL copybook, truncates DEPENDING ON values to the length used.  Valid values are <code>true</code> or <code>false</code> .
useMissCharAsDefaultForFill	Boolean	false	By default, the flat file reader and writer use spaces for missing characters and ignore the setting of the <code>missingValues</code> property. When you set this property to <code>true</code> , DataWeave honors the setting of the <code>missingValues</code> property. Introduced in DataWeave 2.3 (2.3.0-20210823) for the September 2021 release of Mule 4.3.0-20210823.  Valid values are <code>true</code> or <code>false</code> .
zonedDecimalStrict	Boolean	false	For COBOL copybook, uses the 'strict' ASCII form of sign encoding for zoned decimal values.  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>*/flatfile</code>

## Java Format

MIME type: `application/java`

ID: `java`

For the Java data format, DataWeave attempts to map any Java value to a DataWeave value, most often by matching the semantics of DataWeave and Java.

## Java Value Mapping

The following table maps Java classes to DataWeave types.

Java Class	DataWeave
<code>java.lang.String</code>	<code>String</code>
<code>Enum</code>	<code>String</code>
<code>Class</code>	<code>String</code>
<code>UUID</code>	<code>String</code>
<code>CharSequence</code>	<code>String</code>
<code>Char</code>	<code>String</code>
<code>java.sql.Clob</code>	<code>String</code>
<code>java.io.Reader</code>	<code>String</code>
<code>int</code>	<code>Number</code>
<code>long</code>	<code>Number</code>
<code>double</code>	<code>Number</code>
<code>short</code>	<code>Number</code>
<code>BigInteger</code>	<code>Number</code>
<code>BigDecimal</code>	<code>Number</code>
<code>AtomicInteger</code>	<code>Number</code>
<code>AtomicLong</code>	<code>Number</code>
<code>boolean</code>	<code>Boolean</code>
<code>AtomicBoolean</code>	<code>Boolean</code>
<code>java.util.Collection</code>	<code>Array</code>
<code>java.lang.Iterable</code>	<code>Array</code>
<code>java.util.Iterator</code>	<code>Array</code>
<code>java.util.Map</code>	<code>Object</code>
<code>java.util.Optional</code>	<code>Null or Value</code>
<code>java.util.OptionalInt</code>	<code>Null or Number</code>
<code>java.util.OptionalDouble</code>	<code>Null or Number</code>
<code>java.util.OptionalLong</code>	<code>Null or Number</code>
<code>byte[]</code>	<code>Binary</code>
<code>java.lang.InputStream</code>	<code>Binary</code>
<code>java.nio.ByteBuffer</code>	<code>Binary</code>
<code>byte</code>	<code>Binary</code>
<code>java.lang.File</code>	<code>Binary</code>
<code>java.time.Instant</code>	<code>LocalDateTime</code>
<code>java.time.LocalDateTime</code>	<code>LocalDateTime</code>
<code>java.sql.Timestamp</code>	<code>LocalDateTime</code>
<code>java.sql.Date</code>	<code>LocalDateTime</code>

Java Class	DataWeave
<code>java.util.Date</code>	<code>LocalDateTime</code>
<code>java.time.ZonedDateTime</code>	<code>DateTime</code>
<code>java.time.LocalTime</code>	<code>LocalTime</code>
<code>java.time.OffsetTime</code>	<code>Time</code>
<code>java.time.LocalDate</code>	<code>Date</code>
<code>java.time.ZoneOffset</code>	<code>TimeZone</code>
<code>java.util.Calendar</code>	<code>DateTime</code>
<code>java.util.XMLGregorianCalendar</code>	<code>DateTime</code>
<code>*[]</code>	<code>Array</code>



If a Java class is not present in the table, DataWeave treats it as a `JavaBean` and maps it as an `Object` type. In this case, DataWeave takes all properties from the Java getters.

## Metadata

DataWeave supports the `^class` metadata for the Java format.

All Java objects are associated with a class. DataWeave maintains this association by mapping the class to a Metadata property.

`payload.^class` returns the name of the class of the `payload`.

The Java writer uses the metadata property to discover the Java class to which the DataWeave value maps. That is, DataWeave uses the metadata property to determine which Java class is to be created from a given DataWeave value. For example, the expression `output application/java --- now() as DateTime {class: "java.util.Calendar"}` creates a `java.util.Calendar`.

## Enum Custom Type

The Java format supports the `Enum` custom type. To put a Java `enum` value into a `java.util.Map`, the DataWeave Java module defines a custom type called `Enum`. This custom type enables you to handle a given string as the name of a specified `enum` type. You must use the `class` metadata property when specifying the Java class name of the enum, for example: `{gender: "Female" as Enum {class: "org.mycompany.Gender"}}`

## Examples

The following examples show uses of the Java format.

- [Example: Access Values of Java Properties](#)
- [Example: Create a Customer Object](#)
- [Example: Use a Generic](#)

## Class Definitions

The Java examples use the following class definitions:

- `User`
- `Customer` (which extends `User`)

*User class:*

```
class User {  
    private String name;  
    private String lastName;  
  
    public User(){  
    }  
  
    public User(String name, String lastName){  
        this.name = name;  
        this.lastName = lastName;  
    }  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public void setLastName(String lastName){  
        this.lastName = lastName;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public String getLastName(){  
        return lastName;  
    }  
}
```

The following class extends `User`.

*Customer class:*

```
import java.util.Calendar;
class Customer extends User {
    private Calendar expirationDate;
    private User salesRepr;

    public User(){
    }

    public User(String name, String lastName,Calendar expirationDate){
        super(name,lastName);
        this.expirationDate = expirationDate;
    }

    public void setSalesRepr(User salesRepr){
        this.salesRepr = salesRepr;
    }

    public User getSalesRepr(){
        return this.salesRepr;
    }

    public void setExpirationDate(Calendar expirationDate){
        this.expirationDate = expirationDate;
    }

    public Calendar getExpirationDate(){
        return this.expirationDate;
    }
}
```

#### **Example: Access Values of Java Properties**

This example shows how to access the values of Java properties.

#### **Input**

The **User** values serves as the input payload to the DataWeave script.

```
new User("Leandro", "Shokida")
```

#### **Source**

The DataWeave scripts transforms the input value to JSON.

```
output application/json
---
{
  a: payload.name,
  b: payload.lastName
}
```

## Output

The output is a JSON object that contains key-value pairs. The values are the `name` and `lastName` values from the input `User` object.

```
{
  "a": "Leandro",
  "b": "Shokida"
}
```

## Example: Create a Customer Object

This example shows how to create an instance of a `Customer` class. The script outputs the object in the JSON format and MIME type.

Notice that it is not necessary to specify the class of inner properties because their class is inferred from the parent class definition.

```
output application/json
---
{
  name: "Tomo",
  lastName: "Chibana",
  expirationDate: now(),
  salesRepr: {
    name: "Mariano",
    lastName: "de Achaval",
  }
} as Object {class: "Customer"}
```

## Example: Use a Generic

This example relies on generic support in the class name to create a `java.util.ArrayList` of `User` objects.

Note that you do not need to use a generic to specify the class in each instance. The class is taken from the generic in the list.

```

output application/json
---
[{
    name: "Tomo",
    lastName: "Chibana"
},
{
    name: "Ana",
    lastName: "Felissati"
},
{
    name: "Leandro",
    lastName: "Shokida"
}
] as Array {class: "java.util.ArrayList<User>"}

```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

There are no reader properties for this format.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
<code>duplicateKeyAsArray</code>	<code>Boolean</code>	<code>false</code>	Converts the values of duplicate keys in an object to a single array of values to the duplicated key.  Valid values are <code>true</code> or <code>false</code> .
<code>writeAttributes</code>	<code>Boolean</code>	<code>false</code>	Converts attributes of a key into child key-value pairs of that key. The attribute key name starts with <code>@</code> .  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>*/java</code>

## JSON Format

MIME type: `application/json`

ID: `json`

In the JSON data format, values map one-to-one with DataWeave values. JSON supports `String`, `Boolean`, `Number`, `Null`, `Object`, and `Array` types. DataWeave supports each of these values natively.

The DataWeave reader for JSON input supports the following parsing strategies:

- Indexed
- In-Memory
- Streaming

To understand the parsing strategies that DataWeave readers and writers can apply to this format, see [DataWeave Parsing Strategies](#).

## Examples

The following examples show uses of the JSON format:

- [Example: Represent JSON in the DataWeave \(dw\) Format](#)
- [Example: Convert Repeated XML Elements to JSON](#)
- [Example: Stream JSON Data](#)

### Example: Represent JSON in the DataWeave (dw) Format

This example shows how JSON input is represented in the DataWeave (dw) format.

#### Input

The following JSON object serves as the input payload to the DataWeave source.

```
{  
  "name": "Matias",  
  "age": 8,  
  "male": true,  
  "kids": null,  
  "brothers": ["Pedro", "Sara"]  
}
```

#### Source

The DataWeave script transforms the JSON input payload to the DataWeave (dw) format and MIME type.

```
output application/dw
---
payload
```

## Output

Notice that only the keys of the JSON input and the `application/dw` output differ. The JSON keys are surrounded by quotation marks. These DataWeave (dw) keys do not require quotation marks. See [Valid Keys](#) for details.

```
{
  name: "Matias",
  age: 8,
  male: true,
  kids: null,
  brothers: [
    "Pedro",
    "Sara"
  ]
}
```

## Example: Convert Repeated XML Elements to JSON

This example shows how to convert repeated XML elements to JSON.

Note that XML encodes collections using repeated (unbounded) elements. DataWeave represents unbounded elements by repeating the same key.

## Input

The following XML data contains repeating child elements with the key `name`. This XML serves as the input payload to the DataWeave source.

```
<?xml version='1.0' encoding='UTF-8'?>
<friends>
  <name>Mariano</name>
  <name>Shoki</name>
  <name>Tomo</name>
  <name>Ana</name>
</friends>
```

## Source

The following DataWeave script takes the XML input as its payload and outputs that payload in the JSON format.

The script selects the `name` elements from the XML input and uses the DataWeave `map` function to map the values of those elements into an array of objects. Each object in the array takes the form {

`name : item },` where `item` is the value of the `name` element. The entire array serves as the value of the `friends` key.

```
%dw 2.0
output application/json
---
friends: payload.friends.*name map (
    (item, index) -> {name: item}
)
```

## Output

The DataWeave script outputs the following JSON object.

```
{
  "friends": [
    {
      "name": "Mariano"
    },
    {
      "name": "Shoki"
    },
    {
      "name": "Tomo"
    },
    {
      "name": "Ana"
    }
  ]
}
```

## Example: Stream JSON Data

To demonstrate streaming, the following example streams a JSON file by reading each element in an array one at a time.

## Input

The JSON input payload serves as input for the XML configuration.

*JSON Input (truncated):*

```
{ "myJsonExample" : [  
    {  
        "name" : "Shoki",  
        "zipcode": "95838"  
    },  
    {  
        "name" : "Leandro",  
        "zipcode": "95823"  
    },  
    {  
        "name" : "Mariano",  
        "zipcode": "95815"  
    },  
    {  
        "name" : "Cristian",  
        "zipcode": "95815"  
    },  
    {  
        "name" : "Kevin",  
        "zipcode": "95824"  
    },  
    {  
        "name" : "Stanley",  
        "zipcode": "95841"  
    }  
]
```

## XML Configuration

The following XML configuration streams JSON input.

```

<file:config name="File_Config" doc:name="File Config" />
<flow name="dw-streaming-jsonFlow" >
    <scheduler doc:name="Scheduler" >
        <scheduling-strategy >
            <fixed-frequency frequency="1" timeUnit="MINUTES"/>
        </scheduling-strategy>
    </scheduler>
    <file:read doc:name="Read"
        config-ref="File_Config"
        path="${app.home}/myjsonarray.json"
        outputMimeType="application/json; streaming=true"/>
    <ee:transform doc:name="Transform Message" >
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
payload.myJsonExample map ((element) -> {
returnedElement : element.zipcode
})]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <file:write doc:name="Write"
        path="/path/to/output/file/output.json"
        config-ref="File_Config1"/>
    <logger level="INFO" doc:name="Logger" message="#[payload]"/>
</flow>

```

- The streaming example configures the File Read operation to stream the JSON input by setting `outputMimeType="application/json; streaming=true"`. In the Studio UI, you can set the **MIME Type** on the listener to `application/json` and the **Parameters** for the MIME Type to **Key streaming** and **Value true**.
- The DataWeave script in the **Transform Message** component iterates over the array in the input payload and selects its `zipcode` values.
- The Write operation returns a file, `output.json`, which contains the result of the transformation.
- The Logger prints the same output payload that you see in `output.json`.

## Output

The JSON streaming example produces a JSON array of objects.

```
[
  {
    "returnedElement": "95838"
  },
  {
    "returnedElement": "95823"
  },
  {
    "returnedElement": "95815"
  },
  {
    "returnedElement": "95815"
  },
  {
    "returnedElement": "95824"
  },
  {
    "returnedElement": "95841"
  }
]
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>streaming</code>	Boolean	<code>false</code>	<p>Streams input when set to <code>true</code>. Use only if entries are accessed sequentially. The input must be a top-level array. See the <a href="#">streaming example</a>, and see <a href="#">DataWeave Readers</a>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
<code>bufferSize</code>	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.

Parameter	Type	Default	Description
deferred	Boolean	false	<p>Generates the output as a data stream when set to <code>true</code>, and defers the script's execution until the generated content is consumed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
duplicateKeyAsArray	Boolean	false	<p>Converts the values of duplicate keys in an object to a single array of values to the duplicated key.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
encoding	String	'UTF-8'	The encoding to use for the output, such as UTF-8.
indent	Boolean	true	<p>Write indented output for better readability by default, or compress output into a single line when set to <code>false</code>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
skipNullOn	String	null	<p>Skips <code>null</code> values in the specified data structure. By default, DataWeave does not skip the values.</p> <ul style="list-style-type: none"> <li>• <code>arrays</code> + Ignore and omit <code>null</code> values inside arrays from the JSON output, for example, with <code>output application/json skipNullOn="arrays"</code>.</li> <li>• <code>objects</code> + Ignore key-value pairs that have <code>null</code> as the value, for example, with <code>output application/json skipNullOn="objects"</code>.</li> <li>• <code>everywhere</code> + Apply <code>skipNullOn</code> to arrays and objects, for example, <code>output application/json skipNullOn="everywhere"</code>.</li> </ul> <p>Valid values are <code>arrays</code> or <code>objects</code> or <code>everywhere</code>.</p>
writeAttributes	Boolean	false	<p>Converts attributes of a key into child key-value pairs of that key. The attribute key name starts with <code>@</code>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>*/json</code>
<code>/+json</code>

## Multipart Format

MIME type: `multipart/form-data`

ID: `multipart`

DataWeave supports Multipart subtypes, in particular `form-data`. These formats enable you to handle several different data parts in a single payload, regardless of the format each part has. To distinguish the beginning and end of a part, a boundary is used and metadata for each part can be added through headers.

DataWeave represents a Multipart document with the given `Object` structure:

```
type Multipart = {
  preamble?: String,
  parts: {
    _: MultipartPart
  }
}
type MultipartPart = {
  headers?: {
    "Content-Disposition"?:
      {
        name: String,
        filename?: String
      },
    "Content-Type"?:
      String
  },
  content: Any
}
```

## Examples

The following examples show uses of the Multipart format.

- [Example: Multipart](#)
- [Example: Access and Transform Data from Parts](#)
- [Example: Generate Multipart Content](#)
- [Example: Iterate through Multipart payload](#)
- [Example: Using ^mediaType and ^mimeType Metadata Selectors](#)

### Example: Multipart

This example shows how DataWeave reads simple Multipart content.

## Input

The Multipart input serves as the payload to the DataWeave source.

```
--34b21
Content-Disposition: form-data; name="text"
Content-Type: text/plain
Book
--34b21
Content-Disposition: form-data; name="file1"; filename="a.json"
Content-Type: application/json
{
  "title": "Java 8 in Action",
  "author": "Mario Fusco",
  "year": 2014
}
--34b21
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html

<title> Available for download! </title>
--34b21--
```

## Source

The DataWeave script transforms the Multipart input payload to the DataWeave (dw) format.

```
%dw 2.0
output application/dw
---
payload
```

## Output

The output shows how the DataWeave (dw) format represents the Multipart input. Note that the `raw` and `content` values are shortened for brevity. The full values are longer.

```
{
  parts: {
    text: {
      headers: {
        "Content-Disposition": {
          name: "text",
          subtype: "form-data"
        },
        "Content-Type": "text/plain"
```

```

    },
    content: "Book" as String {
        raw: "Qm9vaw==" as Binary {
            base: "64"
        },
        encoding: null,
        mediaType: "text/plain",
        mimeType: "text/plain"
    }
},
file1: {
    headers: {
        "Content-Disposition": {
            name: "file1",
            filename: "a.json",
            subtype: "form-data"
        },
        "Content-Type": "application/json"
    },
    content: {
        title: "Java 8 in Action",
        author: "Mario Fusco",
        year: 2014
    } as Object {
        raw: "ewogICJ0aXRsZSI6ICJKYXZhI...==" as Binary {
            base: "64"
        },
        encoding: null,
        mediaType: "application/json",
        mimeType: "application/json"
    }
},
file2: {
    headers: {
        "Content-Disposition": {
            name: "file2",
            filename: "a.html",
            subtype: "form-data"
        },
        "Content-Type": "text/html"
    },
    content: "PCFET0NUWVBFIGh0bWw+Cjx0aXRsZT4KI...==" as Binary {
        base: "64"
    }
}
}
}
----
```

#### Example: Access and Transform Data from Parts

Within a DataWeave script, you can access and transform data from any of the parts by selecting the `parts` element. Navigation can be array-based or key-based when parts feature a name to reference them by. The part's data can be accessed through the `content` keyword, while headers can be accessed through the `headers` keyword.

## Input

This example serves as input to separate DataWeave scripts. It shows a raw `multipart/form-data` payload with a `34b21` boundary consisting of 3 parts:

- a `text/plain` one named `text`
- an `application/json` file (`a.json`) named `file1`
- a `text/html` file (`a.html`) named `file2`

*Raw Multipart Data:*

```
--34b21
Content-Disposition: form-data; name="text"
Content-Type: text/plain

Book
--34b21
Content-Disposition: form-data; name="file1"; filename="a.json"
Content-Type: application/json

{
  "title": "Java 8 in Action",
  "author": "Mario Fusco",
  "year": 2014
}
--34b21
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html

<!DOCTYPE html>
<title>
  Available for download!
</title>
--34b21--
```

## Source

The following DataWeave script uses the raw `multipart/form-data` payload as input to produce `Book:a.json`.

## *Reading Multipart Content:*

```
%dw 2.0
output text/plain
---
payload.parts.text.content ++ ':' ++ payload.parts[1].headers.'Content-
Disposition'.filename
```

## **Example: Generate Multipart Content**

You can generate multipart content that DataWeave uses to build an object with a list of parts, each containing its headers and content. The following DataWeave script produces the raw multipart data (previously analyzed) if the HTML data is available in the payload.



To provide a consistent output in the DataWeave documentation, the example hardcodes the **boundary** property. However, you do not need to hardcode this property in most cases because DataWeave generates a random boundary and adds it to the output MIME type.

## Writing Multipart Content:

```
%dw 2.0
output multipart/form-data boundary='34b21'
---
{
  parts : {
    text : {
      headers : {
        "Content-Type": "text/plain"
      },
      content : "Book"
    },
    file1 : {
      headers : {
        "Content-Disposition" : {
          "name": "file1",
          "filename": "a.json"
        },
        "Content-Type" : "application/json"
      },
      content : {
        title: "Java 8 in Action",
        author: "Mario Fusco",
        year: 2014
      }
    },
    file2 : {
      headers : {
        "Content-Disposition" : {
          "filename": "a.html"
        },
        "Content-Type" : payload.^mimeType
      },
      content : payload
    }
  }
}
```

Notice that the key determines the part's name if the name is not explicitly provided in the **Content-Disposition** header, and note that DataWeave can handle content from supported formats, as well as references to unsupported ones, such as HTML.

The following example shows how to send the current payload as a file part in a multipart form-data. You need to use a DataWeave script similar to the following:

```
%dw 2.0
input payload application/xml
output multipart/form-data boundary="----myboundary----"
---
{
  parts: {
    filePart: {
      headers: {
        "Content-Disposition": {
          "name": "<NAME>",
          "filename": "<FILE_NAME>.<FILE_EXTENSION>"
        },
        "Content-Type": "<FILE_CONTENT-TYPE>"
      },
      content: payload
    },
    otherJsonPart: {
      content: '{"name": "sampleFile.xml", "sampleFile": {"type": "sample"} }'
    }
  }
}
```

- **boundary** sets the boundary value, for example "----myboundary----".
- **<NAME>** is the tag name of your file, for example "sampleFile".
- **<FILE\_NAME>.<FILE\_EXTENSION>** is the filename to populate the payload content, for example "sampleFile.xml".
- **<FILE\_CONTENT-TYPE>** matches the content of the file you send, for example "application/xml".
- The content of the file is the current payload.

This example contains two parts in the multipart which are the file and the **otherJsonPart**. In case you want to add more parts, add them under the **parts:{}** section with the following format:

```
<PART_NAME>: {
  content: <PART_CONTENT>
}
```

- **<PART\_NAME>** is the name of the part without quotes, for example **json**.
- **<PART\_CONTENT>** is the content for that part, for example '**{"name": "sampleFile.xml", "sampleFile": {"type": "sample"} }**'.

Based on the following XML input:

```

<Customers>
  <Customer>
    <Number>1</Number>
    <FirstName>Fred</FirstName>
    <LastName>Landis</LastName>
    <Address>
      <Street>Oakstreet</Street>
      <City>Boston</City>
      <ZIP>23320</ZIP>
      <State>MA</State>
    </Address>
  </Customer>
  <Customer>
    <Number>2</Number>
    <FirstName>Michelle</FirstName>
    <LastName>Butler</LastName>
    <Address>
      <Street>First Avenue</Street>
      <City>San-Francisco</City>
      <ZIP>44324</ZIP>
      <State>CA</State>
    </Address>
  </Customer>
  <Customer>
    <Number>3</Number>
    <FirstName>Ted</FirstName>
    <LastName>Little</LastName>
    <Address>
      <Street>Long Way</Street>
      <City>Los-Angeles</City>
      <ZIP>34424</ZIP>
      <State>CA</State>
    </Address>
  </Customer>
</Customers>

```

The DataWeave script is:

```
%dw 2.0
output multipart/form-data boundary="----myboundary----"
---
{
  parts: {
    filePart: {
      headers: {
        "Content-Disposition": {
          "name": "test",
          "filename": "sampleFile.xml"
        },
        "Content-Type": "application/xml"
      },
      content: payload
    },
    otherJsonPart: {
      content: '{"name": "sampleFile.xml", "sampleFile": {"type": "sample"} }'
    }
  }
}
```

The output multipart is:

```

-----myboundary---
Content-Type: application/xml
Content-Disposition: form-data; name="test"; filename="sampleFile.xml"

<Customers>
  <Customer>
    <Number>1</Number>
    <FirstName>Fred</FirstName>
    <LastName>Landis</LastName>
    <Address>
      <Street>Oakstreet</Street>
      <City>Boston</City>
      <ZIP>23320</ZIP>
      <State>MA</State>
    </Address>
  </Customer>
  <Customer>
    <Number>2</Number>
    <FirstName>Michelle</FirstName>
    <LastName>Butler</LastName>
    <Address>
      <Street>First Avenue</Street>
      <City>San-Francisco</City>
      <ZIP>44324</ZIP>
      <State>CA</State>
    </Address>
  </Customer>
  <Customer>
    <Number>3</Number>
    <FirstName>Ted</FirstName>
    <LastName>Little</LastName>
    <Address>
      <Street>Long Way</Street>
      <City>Los-Angeles</City>
      <ZIP>34424</ZIP>
      <State>CA</State>
    </Address>
  </Customer>
</Customers>
-----myboundary---
Content-Disposition: form-data; name="otherJsonPart"

{"name": "sampleFile.xml", "sampleFile": {"type": "sample"} }
-----myboundary-----

```

#### **Example: Iterate through Multipart payload**

This example shows how DataWeave iterates through all the parts of the payload and processes the each part individually.

## Source

```
%dw 2.0
output application/json
var multi = '--34b21
Content-Disposition: form-data; name="text"
Content-Type: text/plain

Book
--34b21
Content-Disposition: form-data; name="file1"; filename="a.json"
Content-Type: application/json

{
  "title": "Java 8 in Action",
  "author": "Mario Fusco",
  "year": 2014
}
--34b21
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html

<!DOCTYPE html>
<title>
Available for download!
</title>
--34b21--'
var parsed = read(multi, "multipart/form-data", {boundary:'34b21'})
---
parsed.parts mapObject ((value, key, index) -> {(index): key})
```

## Output

```
{
  "0": "text",
  "1": "file1",
  "2": "file2"
}
```

### Example: Using ^mediaType and ^MimeType Metadata Selectors

The `^MimeType` selector returns the MIME type of a value without parameters, for example, `application/json`. The `^mediaType` selector, by contrast, provides the MIME type and any other parameters related to it, for example, `application/json;charset=UTF-16`.

The following example shows how to transform multipart inputs into multipart outputs with a different format. The DataWeave script uses both the `^MimeType` and `^mediaType` selectors to generate the new multipart content.

## Input

The multipart input has more than one Content-Type. In this case `application/json` and `application/csv` including a non-standard separator:

```
-----049709565842371278701691
Content-Type: application/json
Content-Disposition: form-data; name="order"

{
  "name": "order1",
  "id": 123,
  "content": "orderContent"
}
-----049709565842371278701691
Content-Type: application/csv;separator=,
Content-Disposition: form-data; name="partner"

partnerName.id
mulesoft.1
-----049709565842371278701691--
```

## Source

The DataWeave script transforms the multipart input into multipart output by defining `output multipart/form-data` MIME type. The multipart output data contains two parts:

- `orderAck`

Contains a text with an acknowledgement of the receipt of an order. The Content-Type is `text/plain`. The `^mimeType` selector becomes useful to log the Content-Type as part of the message.`

- `partner`

Contains the original partner itself. Notice that this time, the Content-Type is `payload.parts..^mediaType`. The `^mediaType` selector includes also the separator, that parses the `application/csv` content.

```
%dw 2.0
input payload multipart boundary='boundary1'
output multipart boundary='test2'
---
{
  parts : {
    orderAck : {
      headers : {
        "Content-Type": "text/plain"
      },
      content : "Order payload of type " ++ payload.parts.order.content.^mimeType ++ " and id " ++ (payload.parts.order.content.id as String) ++ " received. Receipt available."
    },
    partner: {
      headers : {
        "Content-Type": payload.parts.partner.content.^mediaType
      },
      content : payload.parts.partner.content
    }
  }
}
```

## Output

```
--test2
Content-Type: text/plain
Content-Disposition: form-data; name="orderAck"

Order payload of type application/json and id 123 received. Receipt available.
--test2
Content-Type: application/csv;separator=,
Content-Disposition: form-data; name="partner"

partnerName.id
mulesoft.1
--test2--
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>boundary</code>	<code>String</code>	<code>null</code>	The multipart <code>boundary</code> value, a string to delimit parts.
<code>defaultContentType</code>	<code>String</code>	<code>'application/octet-stream'</code>	Sets the default Content-Type to use on parts of the <code>multipart/*</code> format. When set, this property takes precedence over the setting for the <a href="#">system property</a> <code>com.mulesoft.dw.multipart.defaultContentType</code> .  <i>Introduced in DataWeave 2.3 (2.3.0-20210720) for the August 2021 release of Mule 4.3.0-20210719.</i>

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
<code>boundary</code>	<code>String</code>	<code>null</code>	The multipart <code>boundary</code> value, a string to delimit parts.
<code>bufferSize</code>	<code>Number</code>	<code>8192</code>	Size of the buffer writer, in bytes. The value must be greater than 8.
<code>deferred</code>	<code>Boolean</code>	<code>false</code>	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>multipart/*</code>

## New Line Delimited (ndjson) Format

MIME type: `application/x-ndjson`

ID: `ndjson`

DataWeave represents the Newline Delimited JSON format (ndjson) as an array of objects. Each line of the ndjson format is mapped to one object in the array.

The following parser strategies are supported by the ndjson reader:

- In-memory
- Streaming

For details, see [DataWeave Readers](#).

## Examples

The following examples show uses of the ndjson format.

- [Example](#)
- [Example](#)

### Example

This example shows how DataWeave represents a simple ndjson input.

#### Input

```
{"name": "Leandro", "lastName": "Shokida"}  
 {"name": "Mariano", "lastName": "De Achaval"}
```

#### Source

The DataWeave script transforms the ndjson input to the DataWeave (dw) format and MIME type.

```
%dw 2.0  
output application/dw  
---  
payload
```

#### Output

The DataWeave (dw) format outputs the ndjson input into an array of comma-separated objects.

```
[  
 {"name": "Leandro", "lastName": "Shokida"},  
 {"name": "Mariano", "lastName": "De Achaval"}  
]
```

### Example

This example shows that the ndjson reader ignores all lines of ndjson data that are invalid if `skipInvalid=true`.

#### Input

The input to the DataWeave source includes valid and invalid lines of ndjson data. Assume that the input is from a file `myInput.ndjson`.

`myInput.ndjson:`

```
{"name": "Christian"
{"name": "Mariano"
{"name": "Tomo"
 {"name": "Shoki"}
```

## Source

The DataWeave script inputs the contents of the input file `myInput.ndjson`, applies the `skipInvalid=true` reader property, and transforms the input to the JSON format and MIME type.

```
%dw 2.0
var myInput = readUrl('classpath://myInput.ndjson', 'application/x-ndjson',
{skipInvalid=true})
output application/json
---
myInput
```

## Output

The JSON output is an array of the valid objects from the ndjson input.

```
[ 
  {
    "name": "Mariano"
  },
  {
    "name": "Shoki"
  }
]
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>ignoreEmptyLine</code>	Boolean	<code>true</code>	Ignores empty lines. Valid values are <code>true</code> or <code>false</code> .

Parameter	Type	Default	Description
skipInvalid	Boolean	false	<p>Skips invalid records and ignores values that are not valid in this format.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	<p>Size of the buffer writer, in bytes. The value must be greater than 8.</p>
deferred	Boolean	false	<p>Generates the output as a data stream when set to <code>true</code>, and defers the script's execution until the generated content is consumed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
encoding	String	'UTF-8'	<p>Encoding that the writer uses for output. Defaults to "UTF-8".</p>
skipNullOn	String	null	<p>Skip <code>null</code> values. By default, DataWeave does not skip.</p> <ul style="list-style-type: none"> <li> <b>arrays</b>            Ignore and omit <code>null</code> values inside arrays from the JSON output, for example, with <code>output application/x-ndjson skipNullOn="arrays"</code>.         </li> <li> <b>objects</b>            Ignore key-value pairs that have <code>null</code> as the value, for example, with <code>output application/x-ndjson skipNullOn="objects"</code>.         </li> <li> <b>everywhere</b>            Apply <code>skipNullOn</code> to arrays and objects, for example, <code>output application/x-ndjson skipNullOn="everywhere"</code>.         </li> </ul> <p>Valid values are <code>arrays</code> or <code>objects</code> or <code>everywhere</code>.</p>

Parameter	Type	Default	Description
<code>writeAttributeS</code>	<code>Boolean</code>	<code>false</code>	Converts attributes of a key into child key-value pairs of that key. The attribute key name starts with <code>@</code> .  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>application/x-ndjson</code>
<code>application/x-ldjson</code>

## Protobuf Format

MIME type: `application/protobuf`

ID: `protobuf`

Protobuf is a data format that can be mapped to DataWeave values natively. From a user perspective, always specify the location of the descriptor file and the fully qualified name of the message for DataWeave to read or write. When specifying the configuration properties, use the `descriptorUrl` and the `messageType` properties, respectively.

### Example: Specify the Descriptor when Reading a Protobuf Message

The following example shows how to specify the descriptor location and the message type to parse a Protobuf message and transform it into JSON.

#### Schema

The following schema specifies the protocol used in this example.

```
syntax = "proto3";

package examples.descriptor;

message MyMessage {
    int32 myInt = 3;
    bool myBool = 13;
    string myString = 23;
}
```

#### Input

The Protobuf message serves as input payload to the DataWeave source. It contains a `myInt` field with the value `42`, a `myBool` field with the value `false`, and a `myString` field with the value `DW + Proto`. We omit showing the Protobuf messages since their representation is not user-friendly.

## Source

The DataWeave script reads a Protobuf message that has an `int` field, a `bool` field, and a `string` field, and transforms the input into JSON format. `messageType` points to the fully qualified name of the message being read, in this case `examples.descriptor.MyMessage`. `descriptorUrl` points to the descriptor, a compiled version of the `.proto` schema previously presented.

```
%dw 2.0
output application/json
input payload application/x-protobuf
messageType='examples.descriptor.MyMessage',descriptorUrl="descriptors/examples.dsc"
---
payload
```

## Output

The DataWeave script outputs the following JSON object containing the three values.

```
{
  "myInt": 42.0,
  "myBool": false,
  "myString": "DW <3 Proto"
}
```

## Protobuf Features

Protobuf supports the following features which are not common in other formats.

### Enumerations

[Protobuf enumerations](#), or enums, are read as a `String` with a particular schema that specifies the enum index. When parsing a `proto` message, the schema is used to extract the label of the enum value. If many labels share the same index, the first one is used. If the index does not have a matching label, the special label `"-UNRECOGNIZED"` is used. When writing a `proto` message, the schema specifies the protocol used to get the index corresponding to the given label.

### Example: Use DataWeave to Write Protobuf Enumerations

The following example shows how to generate a `proto` message with some enum values, given a particular schema.

### Schema

The following schema specifies the protocol used in this example.

```
syntax = "proto3";

package examples.enumerations;

message Langs {
    enum Languages {
        DataWeave = 0;
        Scala = 2;
        Java = 343049039;
    }

    Languages okayLanguage = 10;
    Languages bestLanguage = 11;
}
```

## Source

The DataWeave script outputs a Protobuf message containing `okayLanguage` and `bestLanguage` values.

```
%dw 2.0
output application/x-protobuf
messageType='examples.enumerations.Langs',descriptorUrl="descriptors/examples.dsc"
---
{
    okayLanguage : "Scala",
    bestLanguage : "DataWeave"
}
```

## Output

The DataWeave script outputs a Protobuf message with two fields specified with the encoded values.

### Example: Use DataWeave to Read Protobuf Enumerations

The following example shows how to read a `proto` message with some enum values, given a particular schema, and what to expect when the value present in the message is not specified in the schema.

## Schema

The following schema specifies the protocol used in this example.

```

syntax = "proto3";

package examples.enumerations;

message Langs {
    enum Languages {
        DataWeave = 0;
        Scala = 2;
        Java = 343049039;
    }

    Languages okayLanguage = 10;
    Languages bestLanguage = 11;
}

```

## Input

The Protobuf message which serves as input payload to the DataWeave source is not shown since it's binary. It contains a `okayLanguage` field with enum index 3, an index not specified on the schema, while the `bestLanguage` field has the expected value.

## Source

The DataWeave script just reads a Protobuf message and outputs it as a DataWeave output.

```

%dw 2.0
input payload application/x-protobuf
messageType='examples.enumerations.Langs',descriptorUrl="descriptors/examples.dsc"
output application/dw
---
payload

```

## Output

The DataWeave script shows how the Protobuf message input is represented in the DataWeave (`dw`) format.

```
{
    okayLanguage: "-UNRECOGNIZED" as String {protobufEnumIndex: 3},
    bestLanguage: "DataWeave" as String {protobufEnumIndex: 0},
}
```

## Oneof

`Oneof` fields, a Protobuf particularity, are represented as regular fields on DataWeave. When writing a `proto` message, a particular schema specifies what the DataWeave script needs to validate.

If more than one of the fields defined is present, the script fails.

#### Example: An Invalid Attempt to Write Two Exclusive Fields

The following example shows what happens when you try to write two exclusive fields according to the schema.

#### Schema

The following schema specifies the protocol used in this example.

```
syntax = "proto3";

package examples.oneof;

message ThisOrThat {
    oneof thisOrThat {
        bool this = 2;
        bool that = 4;
    }
}
```

#### Source

The DataWeave script outputs a Protobuf message containing both **this** and **that** fields set.

```
%dw 2.0
output application/x-protobuf
messageType='examples.oneof.ThisOrThat',descriptorUrl="descriptors/examples.dsc"
---
{
    this: true,
    that: false,
}
```

#### Output

The DataWeave script outputs a **ProtoBufWritingException** specifying which **oneof** is being wrongly used.

#### Repeated Fields

Since DataWeave admits repeated fields, Protobuf repeated fields are matched to DataWeave repeated fields, and vice versa. Note that the DataWeave object being written has to match the schema being used. If the schema does not specify a field as repeated and the DataWeave script has that field more than once, the script fails.

#### Example: Transform a JSON Array to a Protobuf Repeated Field

The following example shows how to generate a `proto` message with a repeated field obtained from a JSON array.

## Schema

The following schema specifies the protocol used in this example.

```
syntax = "proto3";

package examples.repeated;

message People {
    repeated string names = 1;
}
```

## Input

The JSON input serves as the payload to the DataWeave source.

```
{
  "names": [
    "Mariano",
    "Shoki",
    "Tomo",
    "Ana"
  ]
}
```

## Source

The DataWeave script uses the `reduce` function to generate a new field for each name in the array.

```
%dw 2.0
output application/x-protobuf
messageType='examples.repeated.People',descriptorUrl="descriptors/examples.dsc"
---
reduce(payload.names, (item, acc = {}) -> acc ++ {names: item})
```

## Output

The DataWeave script outputs a Protobuf message with the repeated field containing all the names from the JSON array.

### Example: Transform a Protobuf Repeated Field to a JSON Array

The following example shows how to transform a `proto` message with a repeated field to a JSON array.

## Schema

The following schema specifies the protocol used in this example.

```
syntax = "proto3";

package examples.repeated;

message People {
    repeated string names = 1;
}
```

## Source

The DataWeave script outputs a JSON message containing an array of names.

```
%dw 2.0
input payload application/x-protobuf
messageType='examples.repeated.People',descriptorUrl="descriptors/examples.dsc"
output application/json
---
names: payload.people.*names
```

## Output

The DataWeave script outputs a JSON message containing an array of names.

```
{
  "names": [
    "Mariano",
    "Shoki",
    "Tomo",
    "Ana"
  ]
}
```

## Unknowns

Protobuf offers capabilities for forward and backward compatibility of protocols. In order to achieve this, readers and writers accept [unknown fields](#) on messages. DataWeave adapts to this functionality by using certain key names.

When reading a Protobuf message, if a field not present in the schema is found, it is read into something similar to `"-35": 111111 as Number {wireType: "Varint"}, where "-35" means that the field index is 35, and wireType: "Varint" specifies the wire type the field has in the message. The wireType can be "Varint", "64Bit", "LengthDelimited", "Group", or "32Bit".`

## Semantic Parsing (Or Commonly Used Message Types)

Protobuf offers a collection of [commonly used message types](#). DataWeave parses some of these as the value they represent instead of as the underlying message. For example, a `google.protobuf.Duration` is read into DataWeave as a `Period`, while a `google.protobuf.NullValue` is read as `Null`.

The following table describes the correspondence between Protobuf types and DataWeave types.

Protobuf type	DataWeave type
BoolValue	Boolean
BytesValue	Binary
DoubleValue	Number
Duration	a Duration Period
Empty	{}
FloatValue	Number
Int32Value	Number
Int64Value	Number
ListValue	Array
NullValue	Null
StringValue	String
Struct	Object
Timestamp	LocalDateTime
UInt32Value	Number
UInt64Value	Number
Value	ProtoBufValue

Where:

```
type ProtoBufValue = Null
| Number
| String
| Boolean
| { _: ProtoBufValue }
| Array<ProtoBufValue>
```

## Maps

Protobuf maps enable you to have a structure without a predefined set of keys, but with every field sharing the same value type. A `map<keyType, valueType>` is mapped to a DataWeave object with the keys represented as strings and the values mapped to their corresponding value. When writing a `proto` message, the key is casted to the `keyType` specified on the descriptor. If it's not possible to

execute the cast, the script fails.

## Compiling Schemas into Descriptors

DataWeave is not able to directly use `*.proto` files and expects an already compiled version called descriptor. Generate descriptors by using the `protoc` compiler as in `protoc --descriptor_set_out=./out.dsc file1.proto file2.proto ...`, where `out.dsc` is the output file for the descriptor (the one that DataWeave expects on the `descriptorUrl` property), while `file1.proto` and `file2.proto` are the actual protocol specifications that the descriptor needs to compile.

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
<code>bufferSize</code>	<code>Number</code>	<code>8192</code>	Size of the buffer writer. The value must be greater than 8.
<code>deferred</code>	<code>Boolean</code>	<code>false</code>	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
<code>descriptorUrl</code> (Required)	<code>String</code>	<code>''</code>	The URL for the ProtoBuf descriptor. Valid values are <code>classpath://</code> , <code>file://</code> , or <code>http://</code> .
<code>messageType</code> (Required)	<code>String</code>	<code>null</code>	The message type's full name taken from the given descriptor, including the package where it's located.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
<code>bufferSize</code>	<code>Number</code>	<code>8192</code>	Size of the buffer writer. The value must be greater than 8.

Parameter	Type	Default	Description
<code>deferred</code>	<code>Boolean</code>	<code>false</code>	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
<code>descriptorUrl</code> (Required)	<code>String</code>	<code>''</code>	The URL for the ProtoBuf descriptor. Valid values are <code>classpath://</code> , <code>file://</code> , or <code>http://</code> .
<code>messageType</code> (Required)	<code>String</code>	<code>null</code>	The message type's full name taken from the given descriptor, including the package where it's located.

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>application/protobuf</code>
<code>application/x-protobuf</code>

## Text Java Properties Format

MIME type: `text/x-java-properties`

ID: `properties`

The Text Java Properties format parses any Java properties file. This format represents simple key-value pairs. DataWeave represents these pairs as an object with string values.

### Example: Represent a properties File in the DataWeave Format (dw)

This example shows how DataWeave represents a properties file.

#### Input

The following `text/x-java-properties` data is from a properties file. The file contains key-value pairs that provide host and port values. This content serves as the input payload to the DataWeave script.

```
host=localhost
port=1234
```

#### Source

The DataWeave script transforms the `text/x-java-properties` input payload to the DataWeave (dw)

format and MIME type. It returns a string.

```
%dw 2.0
output application/dw
---
payload
```

## Output

The output is an object in the DataWeave (dw) format. The object contains a collection of key-value pairs that match the `text/x-java-properties` input. Notice that the output wraps the values in quotation marks.

```
{
  host: "localhost",
  port: "1234"
}
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

There are no reader properties for this format.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
<code>bufferSize</code>	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
<code>deferred</code>	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
<code>encoding</code>	String	null	The encoding to use for the output, such as UTF-8.

## Supported MIME Types

This format supports the following MIME types.

## MIME Type

\*`/x-java-properties`  
\*`/properties`

## Text Plain Format

MIME type: `text/plain`

ID: `text`

The Text Plain format represents text as a string.

Note that DataWeave parses, encodes, and stores this format into RAM memory.

### Example

This example shows how DataWeave represents Text Plain data.

#### Input

The Plain Text data serves as the input payload to the DataWeave source.

```
This is text plain
```

#### Source

The DataWeave script transforms the Text Plain input payload to the DataWeave (dw) format and MIME type. It returns a string.

```
output application/dw
---
payload
```

#### Output

Because the DataWeave (dw) output is a string, it is wrapped in quotation marks.

```
"This is text plain"
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

There are no reader properties for this format.

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
encoding	String	null	The encoding to use for the output, such as UTF-8.

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>text/plain</code>

## URL Encoded Format

MIME type: `application/x-www-form-urlencoded`

ID: `urlencoded`

URL encoded data represents a collection of key-value pairs. DataWeave represents these values as an object in which each value is a string.

## Examples

The following examples show uses of the URL Encoded format:

- [Example: Represent URL Encoded Data in the DataWeave \(dw\) Format](#)
- [Example: Generate URL Encoded Data from a JSON Object](#)
- [Example: Generate URL Encoded Data from a Collection of Key-Value Pairs](#)
- [Example: Transform URL Encoded Data to the Text Plain Format](#)

### Example: Represent URL Encoded Data in the DataWeave (dw) Format

This example shows how DataWeave represents simple URL encoded data.

#### Input

The URL encoded data serves as input payload to the DataWeave source.

```
name=Mariano&lastName=de+Achaval
```

## Source

The DataWeave script transforms the URL encoded input payload to the DataWeave (dw) format and MIME type.

```
%dw 2.0
output application/dw
---
payload
```

## Output

The output is a DataWeave object that contains a collection of key-value pairs from the input.

```
{
  "name": "Mariano",
  "lastName": "de Achaval"
}
```

## Example: Generate URL Encoded Data from a JSON Object

This example shows how to generate URL Encoded data.

### Input

The JSON object serves as the input payload to the DataWeave source.

```
{
  "name": "Mariano"
}
```

## Source

The DataWeave script selects the value of the `name` key in the input payload and transforms all values of the object in the body of the script to the `urlencoded` format.

```
%dw 2.0
output urlencoded
---
{
    name: payload.name,
    age: 37
}
```

## Output

The output is URL encoded data that is constructed from the key-value pairs in the body of the DataWeave script.

```
name=Mariano&age=37
```

### Example: Generate URL Encoded Data from a Collection of Key-Value Pairs

This DataWeave script produces the URL encoded output.

## Source

The DataWeave script transforms the format of the JSON input found within the body of the script into the URL Encoded format.

```
%dw 2.0
output application/x-www-form-urlencoded
---
{
    "key" : "value",
    "key 1": "@here",
    "key" : "other value",
    "key 2%": null
}
```

## Output

The output shows the input data in the URL Encoded format.

```
key=value&key+1=%40here&key=other+value&key+2%25
```

### Example: Transform URL Encoded Data to the Text Plain Format

This example transforms URL Encoded input into the Text Plain format and MIME type.

## Source

The DataWeave script reads the URL Encoded data and concatenates values of the selected keys.

```
%dw 2.0
var myData = read('key=value&key+1=%40here&key=other+value&key+2%25', 'application/x-www-form-urlencoded')
output text/plain
---
myData.*key[0] ++ myData.'key 1'
```

## Output

The output is in Text Plain format.

```
value@here
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

There are no reader properties for this format.

### Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
<code>bufferSize</code>	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
<code>deferred</code>	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
<code>encoding</code>	String	null	The encoding to use for the output, such as UTF-8.

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>application/x-www-form-urlencoded</code>

## XML Format

MIME type: `application/xml`

ID: `xml`

The XML data structure is mapped to DataWeave objects that can contain other objects, strings, or `null` values. XML uses unbounded elements to represent collections, which are mapped to repeated keys in DataWeave objects. In addition, DataWeave natively supports `namespaces`, `CData` and `xsi:types`.

The DataWeave reader for XML input supports the following parsing strategies:

- Indexed
- In-Memory
- Streaming

To understand the parsing strategies that DataWeave readers and writers can apply to this format, see [DataWeave Parsing Strategies](#).

### CData Custom Type

`CData` is a custom DataWeave data type for XML that is used to identify a character data (CDATA) block. The `CData` type makes the XML writer wrap content inside a `CDATA` block or to check for an input string inside a `CDATA` block. In DataWeave, `CData` inherits from the type `String`.

### DocType Custom Type

`DocType` is a custom DataWeave data type for XML that is used to identify a doctype declaration (DTD).

### Read and Write DTDs

*Introduced in DataWeave 2.5.0. Supported by Mule 4.5.0 and later.*

DataWeave can read and write doctype declarations in XML files.

To read a doctype declarations, set the system property `com.mulesoft.dw.xml_reader.parseDtd`.

During the reading phase, DataWeave parses the doctype declarations and stores the content as metadata of the root element. The `DocType` value is stored in the `docType` variable. You can use the metadata selector (`^`) to extract the value of that variable with an expression like this one:

- `payload.^docType`

See examples [Example: Read DTD value](#), [Example: Write a DTD Value](#) and [Example: Transform DTD Value to a String Representation](#).

You can use the [metadata selector](#) to access it.

## NOTE

DTDs are disabled by default even if their declarations are read or written. For details, see XML reader property `supportDtd` in [Reader Properties](#).

## Examples

The following examples show uses of the XML format.

- [Example: Stream Input XML Data](#)
- [Example: Null or Empty String in XML](#)
- [Example: Output `null` Values for Missing XML Values](#)
- [Example: Output `null` Values for Missing XML Values](#)
- [Example: Represent XML Attributes in the DataWeave \(dw\) Format](#)
- [Example: Represent XML Namespaces in the DataWeave \(dw\) Format](#)
- [Example: Create a CDATA Element](#)
- [Example: Check for CDATA in a `String`](#)
- [Example: Use the `inlineCloseOn` Writer Property](#)
- [Example: Transforms Repeated JSON Keys to Repeated XML Elements](#)
- [Example: Transform XML Elements to JSON and Replace Characters](#)
- [Example: Read DTD value](#)
- [Example: Write a DTD Value](#)
- [Example: Transform DTD Value to a String Representation](#)

Refer to [DataWeave Formats](#) for more detail on available reader and writer properties for various data formats.

### Example: Stream Input XML Data

This example shows how to set up XML streaming, which requires you to specify the following reader properties:

- `streaming=true`
- `collectionPath="root.repeated"`

The `collectionPath` setting selects the elements to stream.

When streaming, the XML parser can start processing content without having all the XML content.

### Input

The following XML serves as the input payload to the DataWeave source. Assume that it is the content of an XML file `myXML.xml`.

## myXML.xml

```
<root>
  <text>
    Text
  </text>
  <repeated>
    <user>
      <name>Mariano</name>
      <lastName>de Achaval</lastName>
      <age>36</age>
    </user>
    <user>
      <lastName>Shokida</lastName>
      <name>Leandro</name>
      <age>30</age>
    </user>
    <user>
      <age>29</age>
      <name>Ana</name>
      <lastName>Felissati</lastName>
    </user>
    <user>
      <age>29</age>
      <lastName>Chibana</lastName>
      <name>Christian</name>
    </user>
  </repeated>
</root>
```

## Source

The reader property settings in the DataWeave script tell the XML reader to stream the input and process the repeated keys. The script uses the DataWeave `map` function to iterate over the repeated keys.

```
%dw 2.0
var myInput  readUrl('classpath://myXML.xml', 'application/xml', {streaming:true,
collectionPath: "root.repeated"})
output application/dw
---
myInput.root.repeated.*user map {
  n: $.name,
  l: $.lastName,
  a: $.age
}
```

## Output

The script transforms the mapped input XML to the DataWeave (dw) format and MIME type.

```
[  
  {  
    "n": "Mariano",  
    "l": "de Achaval",  
    "a": "36"  
  },  
  {  
    "n": "Leandro",  
    "l": "Shokida",  
    "a": "30"  
  },  
  {  
    "n": "Ana",  
    "l": "Felissati",  
    "a": "29"  
  },  
  {  
    "n": "Christian",  
    "l": "Chibana",  
    "a": "29"  
  }  
]
```

## Example: Null or Empty String in XML

Because there is no standard way to represent a `null` value in XML, the reader maps the value to `null` when the `nil` attribute is set to `true`.

## Input

The following XML serves as the input payload to the DataWeave source. Notice the `nil` setting in `<xsi:nil="true"/>`.

```
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  <author xsi:nil="true"/>  
</book>
```

## Source

The DataWeave script transforms the input XML to the JSON format and MIME type.

```
%dw 2.0
output application/json
---
payload
```

## Output

The output is in the JSON format. Notice that the `nil` value in the input is transformed to `null`.

```
{
  "book": {
    "author": null
  }
}
```

## Example: Output `null` Values for Missing XML Values

The XML reader property `nullValueOn` accepts the value `blank` (the default) or `empty`.

This example uses the `nullValueOn` default, so it maps the values of the `title` and `author` elements to `null`.

## Input

The following XML serves as input to the DataWeave source. Notice that the `title` and `author` elements lack values.

Assume that this input is content within the file, `myInput.xml`.

*myXML.xml content:*

```
<book>
  <author></author>
  <title>

  </title>
</book>
```

## Source

The DataWeave script transforms the XML input to JSON. Note that it explicitly sets the `nullValueOn` default, `blank`, for demonstration purposes.

```
%dw 2.0
var myInput readUrl('classpath://myXML.xml', 'application/xml', {nullValueOn:
"blank"})
output application/json
---
myInput
```

## Output

The `title` and `author` keys in the JSON output are assigned `null` values.

```
{
  "book": {
    "author": null,
    "title": null
  }
}
```

## Example: Output `null` Values for Missing XML Values

The XML reader property `nullValueOn` accepts the value `blank` (the default) or `empty`.

The example maps the value of the `title` element to a `String` and value of the `author` element to `null` because the XML reader property `nullValueOn` is set to `empty`.

## Input

The following XML serves as input to the DataWeave source. Notice that the `title` and `author` elements lack values. The difference between the two elements is the space between the starting and ending tags. The tags of the `title` element are separated by line breaks (the hidden characters, `\n`), while the tags of the `author` element are not separated by any characters.

Assume that this input is content within the file, `myInput.xml`.

*myXML.xml content:*

```
<book>
  <author></author>
  <title>

  </title>
</book>
```

## Source

The DataWeave script uses a DataWeave variable to input the content of `myXML.xml` and applies the `nullValueOn: "empty"` to it. The script transforms the XML input to JSON.

```
%dw 2.0
var myInput readUrl('classpath://myXML.xml', 'application/xml', {nullValueOn:
"empty"})
output application/json
---
myInput
```

## Output

The JSON output maps the value of the `author` element to `null` and value of the `title` element to the `String` value "`\n\n`", which is for the new line characters.

```
{
  "book": {
    "author": null,
    "title": "\n\n"
  }
}
```

## Example: Represent XML Attributes in the DataWeave (dw) Format

This example maps XML attributes to a canonical DataWeave representation, the `application/dw` format and MIME type.

### Input

The XML serves as the input payload to the DataWeave source. Notice that the input contains XML attributes.

```
<users>
  <company>MuleSoft</company>
  <user name="Leandro" lastName="Shokida"/>
  <user name="Mariano" lastName="Achaval"/>
</users>
```

### Source

The DataWeave script transforms the XML input payload to the DataWeave (dw) format and MIME type.

```
%dw 2.0
output application/dw
---
payload
```

## Output

The output shows how the DataWeave (dw) format represents the XML input. Notice how the attributes from the XML input and the empty values are represented.

```
{  
  users: {  
    company: "MuleSoft",  
    user @name: "Leandro",lastName: "Shokida": "",  
    user @name: "Mariano",lastName: "Achaval": ""  
  }  
}
```

#### Example: Represent XML Namespaces in the DataWeave (dw) Format

This example maps XML namespaces to a canonical DataWeave representation, the [application/dw](#) format and MIME type.

#### Input

The XML serves as the input payload to the DataWeave source. Notice that the input contains XML namespaces.

```
<root>  
  <h:table xmlns:h="http://www.w3.org/TR/html4/">  
    <h:tr>  
      <h:td>Apples</h:td>  
      <h:td>Bananas</h:td>  
    </h:tr>  
  </h:table>  
  
  <f:table xmlns:f="https://www.w3schools.com/furniture">  
    <f:name>African Coffee Table</f:name>  
    <f:width>80</f:width>  
    <f:length>120</f:length>  
  </f:table>  
</root>
```

#### Source

The DataWeave script transforms the XML input payload to the DataWeave (dw) format and MIME type.

```
%dw 2.0  
output application/dw  
---  
payload
```

#### Output

The output shows how the DataWeave (dw) format represents the XML input. Notice how the namespaces from the XML are represented.

```
ns h http://www.w3.org/TR/html4/
ns f https://www.w3schools.com/furniture
---
{
  root: {
    h#table: {
      h#tr: {
        h#td: "Apples",
        h#td: "Bananas"
      }
    },
    f#table: {
      f#name: "African Coffee Table",
      f#width: "80",
      f#length: "120"
    }
  }
}
```

### Example: Create a CDATA Element

This example shows how to use the `CData` type to create a CDATA element in the XML output.

#### Source

The body of the DataWeave script coerces the `String` value to the `CData` type.

```
%dw 2.0
output application/xml
---
{
  test: "A text <a>" as CData
}
```

#### Output

The output encloses the input `String` value in a CDATA block, which contains the special characters, `<` and `>`, from the input.

```
<?xml version='1.0' encoding='UTF-8'?>
<test><![CDATA[A text <a>]]></test>
```

### Example: Check for CDATA in a `String`

This example indicates whether a given `String` value is CDATA.

## Input

The XML serves as the input payload to the DataWeave source. Notice that the `test` element contains a CDATA block.

```
<?xml version='1.0' encoding='UTF-8'?>
<test><![CDATA[A text <a>]]></test>
```

## Source

The DataWeave script uses the `is CData` expression to determine whether the `String` value is CDATA.

```
%dw 2.0
output application/json
---
{
    test: payload.test is CDATA
}
```

## Output

The JSON output contains the value `true`, which indicates tha the input `String` value is CDATA.

```
{
    "test": true
}
```

## Example: Use the `inlineCloseOn` Writer Property

This example uses the `inlineCloseOn` writer property with the value `none` to act on the key-value pairs from the input.

## Source

The DataWeave script transforms the body content to XML. Notice that values of the `emptyElement` keys are `null`.

```
%dw 2.0
output application/xml inlineCloseOn="none"
---
{
  someXml: {
    parentElement: {
      emptyElement1: null,
      emptyElement2: null,
      emptyElement3: null
    }
  }
}
```

## Output

The `emptyElement` elements are empty. They do not contain the value `null`.

```
<?xml version='1.0' encoding='UTF-8'?>
<someXml>
  <parentElement>
    <emptyElement1></emptyElement1>
    <emptyElement2></emptyElement2>
    <emptyElement3></emptyElement3>
  </parentElement>
</someXml>
```

## Example: Transforms Repeated JSON Keys to Repeated XML Elements

XML encodes collections using repeated (unbounded) elements. DataWeave represents unbounded elements by repeating the same key.

This example shows how to convert the repeated keys in a JSON array of objects into repeated XML elements.

## Input

The JSON input serves as the payload to the DataWeave source. Notice that the `name` keys in the array are repeated.

```
{
  "friends": [
    {"name": "Mariano"},
    {"name": "Shoki"},
    {"name": "Tomo"},
    {"name": "Ana"}
  ]
}
```

## Source

The DataWeave script selects the value of the `friends` key.

```
%dw 2.0
output application/xml
---
friends: {
    (payload.friends)
}
```

## Output

The output represents the `name` keys as XML elements.

```
<?xml version='1.0' encoding='UTF-8'?>
<friends>
    <name>Mariano</name>
    <name>Shoki</name>
    <name>Tomo</name>
    <name>Ana</name>
</friends>
```

See also, [Example: Outputting Self-closing XML Tags](#).

## Example: Transform XML Elements to JSON and Replace Characters

This example iterates over an XML file that contains details of employees, such as the `Id`, `Name`, and `Address`, and converts the file into JSON format. The DataWeave script uses the `replace` function to iterate over each `Address` element and replace the characters `-` and `/` with blank space.

## Input

The XML input serves as the payload to the DataWeave source. Notice that the `Address` element contains `-` and `/` characters.

```

<?xml version='1.0' encoding='UTF-8'?>
<root>
  <employees>
    <Id>1</Id>
    <Name>Mule</Name>
    <Address>MuleSoft Avenue - 123</Address>
  </employees>
  <employees>
    <Id>2</Id>
    <Name>Max</Name>
    <Address>MuleSoft Avenue-456/5/e</Address>
  </employees>
</root>

```

## Source

The following DataWeave script iterates over the payload elements and performs a mapping to an object. The instruction `payload01.Address replace /([\\-\\,\\/] )/ with " "` replaces the - and / characters with blank spaces.

```

%dw 2.0
output application/json
---
payload.root.*employees map ((payload01 , indexOfPayload01) ->
{
  Id: payload01.Id as String,
  Name: payload01.Name as String,
  Address: payload01.Address replace /([\\-\\,\\/] )/ with " "
}
)

```

## Output

The output represents the transformed XML elements into JSON.

```
[
  {
    "Id": "1",
    "Name": "Mule",
    "Address": "MuleSoft Avenue 123"
  },
  {
    "Id": "2",
    "Name": "Max",
    "Address": "MuleSoft Avenue 456 5 e"
  }
]
```

### Example: Read DTD value

The following DataWeave script transforms the XML input payload that contains a `DocType` value into JSON. Notice that extracts the value of that doctype directive using the expression `payload.^docType`.

#### Input

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cXML SYSTEM "http://xml.cxml.org/schemas/cXML/1.2.014/cXML.dtd">
<cXML xml:lang="en-US" payloadID="1615232861.000506@prd327utl1.int.couphost.com"
      timestamp="2021-03-08T11:47:42-08:00">
  <Header>
    <From>
      <Credential domain="NetworkID">
        <Identity>TestIdentity</Identity>
      </Credential>
    </From>
    <To>
      <Credential domain="TEST">
        <Identity>Identity</Identity>
      </Credential>
    </To>
  </Header>
</cXML>
```

#### Source

```
%dw 2.0
output application/json
---
{
  header: {
    senderId: payload.cXML.Header.From.Credential.Identity,
    receiverId: payload.cXML.Header.To.Credential.Identity,
    docType: payload.^docType
  }
}
```

#### Output

```
{
  "header": {
    "senderId": "Identity",
    "receiverId": "TestIdentity",
    "docType": {
      "rootName": "cXML",
      "systemId": "http://xml.cxml.org/schemas/cXML/1.2.014/cXML.dtd"
    }
  }
}
```

### Example: Write a DTD Value

This example shows how to use the `docType` metadata to create a `DocType` value in the XML output.

#### Source

```
%dw 2.0
output application/xml
ns xml http://www.w3.org/XML/1998/namespace

var cXmlObj = {
  cXML @(payloadID: "9949494", xml#lang: "en-US", timestamp: "2002-02-04T18:39:09-08:00"): {
    Header: {
      From: {
        Credential @(domain: "NetworkId"): {
          Identity: "Identity"
        }
      }
    }
  }
} as Object {
  docType: { rootName: "cXML", systemId: "http://xml.cXML.org/schemas/cXML/1.2.014/cXML.dtd" }
}
---
cXmlObj
```

#### Output

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE cXML SYSTEM "http://xml.cXML.org/schemas/cXML/1.2.014/cXML.dtd">
<cXML payloadID="9949494" xmlns:xml="http://www.w3.org/XML/1998/namespace"
      xml:lang="en-US" timestamp="2002-02-04T18:39:09-08:00">
  <Header>
    <From>
      <Credential domain="NetworkId">
        <Identity>Identity</Identity>
      </Credential>
    </From>
  </Header>
</cXML>

```

#### Example: Transform DTD Value to a String Representation

This example uses [Dtd Module \(dw::xml::Dtd\)](#) to transform a `DocType` value that includes a `systemId` to a string representation.

#### Input

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cXML SYSTEM "http://xml.cxml.org/schemas/cXML/1.2.014/cXML.dtd">
<cXML xml:lang="en-US" payloadID="1615232861.000506@prd327utl1.int.couphost.com"
      timestamp="2021-03-08T11:47:42-08:00">
  <Header>
    <From>
      <Credential domain="NetworkID">
        <Identity>Identity</Identity>
      </Credential>
    </From>
    <To>
      <Credential domain="TEST">
        <Identity>Identity</Identity>
      </Credential>
    </To>
  </Header>
</cXML>

```

#### Source

```

%dw 2.0
output application/json
import * from dw::xml::Dtd
---
docTypeAsString(payload.^docType)

```

#### Output

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
collectionPath	String	null	Sets the path to the location in the document where the collection is located. Accepts a path expression that identifies the location of the elements to stream.
externalEntities	Boolean	false	Indicates whether to process external entities. Disabled by default to avoid XML External Entity (XXE) attacks.  Valid values are <code>true</code> or <code>false</code> .
indexedReader	Boolean	true	Uses the indexed reader by default when reaching the threshold. Supports US-ASCII, UTF-8 and ISO-8859-1 encodings only. For other encodings, DataWeave uses the in-memory reader.  Valid values are <code>true</code> or <code>false</code> .
maxAttributeSize	Number	-1	Sets the maximum number of characters accepted in an XML attribute. <i>Available since Mule 4.2.1.</i>
maxEntityCount	Number	1	Sets the maximum number of entity expansions. The limit helps avoid Billion Laughs attacks.
nullValueOn	String	'blank'	Indicates whether to read an element with empty or blank text as a <code>null</code> value.  Valid values are <code>empty</code> or <code>none</code> or <code>blank</code> .
optimizeFor	String	'speed'	Configures the type of optimization for the XML parser to use.  Valid values are <code>speed</code> or <code>memory</code> .

Parameter	Type	Default	Description
streaming	Boolean	false	<p>Streams input when set to <code>true</code>. Use only if entries are accessed sequentially. The input must be a top-level array. See the <a href="#">streaming example</a>, and see <a href="#">DataWeave Readers</a>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
supportDtd	Boolean	false	<p>Enable or disable DTD support. Disabling skips (and does not process) internal and external subsets. You can also enable this property by setting the Mule system property <code>com.mulesoft.dw.xml.supportDTD</code>. Note that the default for this property changed from <code>true</code> to <code>false</code> in Mule version 4.3.0-20210601, which includes the June 2021 patch of DataWeave version 2.3.0.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer. The value must be greater than 8.
defaultNamespace	String	null	Specifies the default namespaces of the output XML.
deferred	Boolean	false	<p>Generates the output as a data stream when set to <code>true</code>, and defers the script's execution until the generated content is consumed.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
doubleQuoteInDeclaration	Boolean	false	<p>Escapes double quotes in the XML declaration when set to <code>true</code>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>
encoding	String	null	The encoding to use for the output, such as UTF-8.

Parameter	Type	Default	Description
escapeCR	Boolean	false	Escapes CR characters when set to true. Valid values are true or false.
escapeGT	Boolean	false	Escapes '>' characters when set to true. Valid values are true or false.
indent	Boolean	true	Write indented output for better readability by default, or compress output into a single line when set to false. Valid values are true or false.
inlineCloseOn	String	'empty'	Write an inline close tag, or explicitly open and close tags when the value is null. Valid values are empty or none.
onInvalidChar	String	null	Valid values are base64 or ignore or none.
skipNullOn	String	null	Skips null values in the specified data structure. By default, DataWeave does not skip the values. <ul style="list-style-type: none"> <li>elements + Ignore and omit null elements inside XML output, for example, with <code>output application/xml skipNullOn="arrays"</code>.</li> <li>attributes ` + Ignore and omit 'null attributes inside XML, for example, with <code>output application/xml skipNullOn="objects"</code>.</li> <li>everywhere + Apply skipNullOn to elements and attributes, for example, <code>output application/xml skipNullOn="everywhere"</code>.</li> </ul> Valid values are elements or attributes or everywhere.
writeDeclaration	Boolean	true	Writes the XML header declaration when set to true. Valid values are true or false.

Parameter	Type	Default	Description
<code>writeDeclaredNamespaces</code>	<code>String</code>	<code>null</code>	<p>Marks the namespaces to declare in the root element of the XML:</p> <ul style="list-style-type: none"> <li>• <code>All</code>: Write all declared namespaces in the root element.</li> <li>• <code>ids:&lt;comma separated namespace id&gt;</code>: Write only the specified namespaces.</li> <li>• <code>regex:&lt;regex&gt;</code>: Write only the matching namespaces.</li> </ul>
<code>writeNilOnNull</code>	<code>Boolean</code>	<code>false</code>	<p>Writes the <code>nil</code> attribute for a <code>null</code> value when this property is set to <code>true</code>.</p> <p>Valid values are <code>true</code> or <code>false</code>.</p>

## Supported MIME Types

This format supports the following MIME types.

MIME Type
<code>*/*xml</code>
<code>/+xml</code>

## YAML Format

MIME type: `application/yaml`

ID: `yaml`

Values in the YAML data format map one-to-one with DataWeave values. DataWeave natively supports all of the following YAML types:

- `String`
- `Boolean`
- `Number`
- `Nil`
- `Mapping`
- `Sequences`

## Example: Represent YAML in the DataWeave Format (dw)

This example shows how DataWeave represents YAML values.

### Input

The following YAML snippet serves as the input payload for the DataWeave source in this example.

```
american:
- Boston Red Sox
- Detroit Tigers
- New York Yankees
national:
- New York Mets
- Chicago Cubs
- Atlanta Braves
```

## Source

The DataWeave script transforms the YAML encoded input payload to the DataWeave (dw) format and MIME type.

```
%dw 2.0
output application/dw
---
payload
```

## Output

The following output shows how the YAML input is represented in the DataWeave (**dw**) format.

```
{
  "american": [
    "Boston Red Sox",
    "Detroit Tigers",
    "New York Yankees"
  ],
  "national": [
    "New York Mets",
    "Chicago Cubs",
    "Atlanta Braves"
  ]
}
```

## Configuration Properties

DataWeave supports the following configuration properties for this format.

### Reader Properties

This format accepts properties that provide instructions for reading input data.

Parameter	Type	Default	Description
maxEntityCount	Number	1	Sets the maximum number of entity expansions. The limit helps avoid Billion Laughs attacks.

## Writer Properties

This format accepts properties that provide instructions for writing output data.

Parameter	Type	Default	Description
bufferSize	Number	8192	Size of the buffer writer, in bytes. The value must be greater than 8.
deferred	Boolean	false	Generates the output as a data stream when set to <code>true</code> , and defers the script's execution until the generated content is consumed.  Valid values are <code>true</code> or <code>false</code> .
encoding	String	'UTF-8'	The encoding to use for the output, such as UTF-8.
skipNullOn	String	null	Skips <code>null</code> values in the specified data structure. By default, DataWeave does not skip the values. <ul style="list-style-type: none"> <li><code>arrays</code> + Ignore and omit <code>null</code> values inside arrays from the YAML output, for example, with <code>output application/yaml skipNullOn="arrays"</code>.</li> <li><code>objects</code> + Ignore key-value pairs that have <code>null</code> as the value, for example, with <code>output application/yaml skipNullOn="objects"</code>.</li> <li><code>everywhere</code> + Apply <code>skipNullOn</code> to arrays and objects, for example, <code>output application/yaml skipNullOn="everywhere"</code>.</li> </ul> Valid values are <code>arrays</code> or <code>objects</code> or <code>everywhere</code> .
writeDeclaration	Boolean	true	Indicates whether to write the header declaration or not.  Valid values are <code>true</code> or <code>false</code> .

## Supported MIME Types

This format supports the following MIME types.

<b>MIME Type</b>
application/yaml
text/yaml
application/x-yaml
text/x-yaml

## Streaming in DataWeave

DataWeave supports end-to-end streaming through a flow in a Mule application. Streaming speeds the processing of large documents without overloading memory.

DataWeave processes streamed data as its bytes arrive instead of scanning the entire document to index it. When in deferred mode, DataWeave can also pass streamed output data directly to a message processor without saving it to the disk. This behavior enables DataWeave and Mule to process data faster and consume fewer resources than the default processes for reading and writing data.

To stream successfully, it is important to understand the following:

- The basic unit of the stream is specific to the data format.  
The unit is a record in a CSV document, an element of an array in a JSON document, or a collection in an XML document.
- Streaming accesses each unit of the stream sequentially.  
Streaming does not support random access to a document.

## Enabling Streaming

Streaming is not enabled by default. You can use two configuration properties to stream data in a supported data format:

- **streaming** property, for reading source data as a stream
- **deferred** writer property, for passing an output stream directly to the next message processor in a flow

For DataWeave to read source data as a stream, you must set the **streaming** reader property to **true** on the data source. Within a Mule application, you append this setting to the value of the **MIME Type** property **outputMimeType** or **contentType**. You can set the property in any connector operation or Mule component that generates data, such as an HTTP Listener operation, HTTP Request operation, On New or Updated File operation, or a Set Payload component. DataWeave reads data as streamed data from the point in the application where you set the streaming property through all downstream components and connector operations that contain DataWeave expressions and scripts.

```
<flow name="dw-streaming-example" >
  <http:listener doc:name="Listener"
    outputMimeType="application/json; streaming=true"
    config-ref="HTTP_Listener_config" path="/input"/>
</flow>
```

Notice that `streaming=true` is part of the `outputMimeType` value. Many other Mule components, such as the File and FTP components, also support the MIME Type setting.

To pass the streamed output to the next message processor, you can use the `output` directive in a DataWeave script, for example:

```
output application/json deferred=true
```

For more detail on use of the `deferred` property, see [Streaming Output](#).

## Streaming CSV

CSV is simplest format for streaming because of its structure. Each row below the CSV header is a streamable record. The following CSV example consists of records that contain `name`, `lastName`, and `age` values:

```
name,lastName,age
mariano,achaval,37
leandro,shokida,30
pedro,achaval,4
christian,chibana,25
sara,achaval,2
matias,achaval,8
```

To stream this CSV example, the following script selects values from each record. It uses the `map` function to iterate over each record in the document.

```
payload map (record) ->
{
  FullName: record.lastName ++ "," ++ record.name,
  Age: record.age
}
```

Although streaming does not support random access to the entire document, a DataWeave script can access data randomly *within each record* because each record is loaded into memory. For example, the expression `record.lastName "," record.name`, can access a `lastName` value before it accesses a `name` value even though the order of values is reversed in the input.

However, streaming does not work in the following script. The script requires random access to the

entire document to return the elements in a different order than they are given in the input.

```
[payload[-2], payload[-1], payload[3]]
```

## Streaming JSON

The unit of a JSON stream is each element in an array.

Note that DataWeave 2.2.0 support for JSON streaming for Mule 4.2 was limited by the requirement that the root be an array. DataWeave support in Mule 4.3 includes streaming to arrays that are not at the root of the input.

```
{
    "name" : "Mariano",
    "lastName": "Achaval",
    "family": [
        {"name": "Sara", "age": 2},
        {"name": "Pedro", "age": 4},
        {"name": "Matias", "age": 8}
    ],
    "age": 37
}
```

In this example, DataWeave can stream `payload.family` and perform random access within each element of that array. However, DataWeave cannot randomly access the container object. For example, it is not possible to stream `{ a: payload.age , b: payload.family}` because `age` follows `family`, and DataWeave cannot go backwards.

## Streaming XML

XML is more complicated than JSON because there are no arrays in the document.

To enable XML streaming, DataWeave provides the following reader property to define the location in the document to stream:

- `collectionPath`

For example, assume the following XML input:

```

<order>
  <header>
    <date>Wed Nov 15 13:45:28 EST 2006</date>
    <customer number="123123">Joe</customer>
  </header>
  <order-items>
    <order-item id="31">
      <product>111</product>
      <quantity>2</quantity>
      <price>8.90</price>
    </order-item>
    <order-item id="31">
      <product>222</product>
      <quantity>7</quantity>
      <price>5.20</price>
    </order-item>
    <order-item id="31">
      <product>111</product>
      <quantity>2</quantity>
      <price>8.90</price>
    </order-item>
    <order-item id="31">
      <product>222</product>
      <quantity>7</quantity>
      <price>5.20</price>
    </order-item>
    <order-item id="31">
      <product>222</product>
      <quantity>7</quantity>
      <price>5.20</price>
    </order-item>
  </order-items>
</order>

```

Given this XML source data, you can set the unit of the stream to `<order-item/>` by setting `collectionPath=order.order-items` in the `outputMimeType` value:

```

<flow name="dw-streaming-example" >
  <http:listener doc:name="Listener"
    outputMimeType="application/xml; collectionpath=order.order-items;
    streaming=true"
    config-ref="HTTP_Listener_config" path="/input"/>
</flow>

```

Note that you need to set both `streaming=true` and the `collectionPath` value. If either is missing, DataWeave will not stream the content. The following DataWeave script streams the XML input using each `<order-items/>` element as the streamable unit.

```
%dw 2.0
output application/xml
---
{
    salesorder: {
        itemList: payload.order."order-items".*"order-item" map {
            ("i_" ++ $$) : {
                id: $.@id,
                productId: $.product,
                quantity: $.quantity,
                price: $.price
            }
        }
    }
}
```

The script produces the following XML output:

```

<?xml version='1.0' encoding='UTF-8'?>
<salesorder>
  <itemList>
    <i_0>
      <id>31</id>
      <quantity>2</quantity>
      <productId>111</productId>
      <price>8.90</price>
    </i_0>
  </itemList>
  <itemList>
    <i_1>
      <id>31</id>
      <quantity>7</quantity>
      <productId>222</productId>
      <price>5.20</price>
    </i_1>
  </itemList>
  <itemList>
    <i_2>
      <id>31</id>
      <quantity>2</quantity>
      <productId>111</productId>
      <price>8.90</price>
    </i_2>
  </itemList>
  <itemList>
    <i_3>
      <id>31</id>
      <quantity>7</quantity>
      <productId>222</productId>
      <price>5.20</price>
    </i_3>
  </itemList>
  <itemList>
    <i_4>
      <id>31</id>
      <quantity>7</quantity>
      <productId>222</productId>
      <price>5.20</price>
    </i_4>
  </itemList>
</salesorder>

```

## Validate a Script for Streamed Data (Experimental Feature)

To check that your code can process an input stream successfully, DataWeave provides the following *advanced, experimental* annotation and a related directive:

- **@StreamCapable()**

Use this annotation to validate whether the script can sequentially access a variable (typically the **payload** variable).

- **input** directive:

The **@StreamCapable()** annotation requires the use of an input directive in the DataWeave script that identifies the MIME type of the data source, for example, **input payload application/xml**.

The DataWeave validator (which is triggered by the **@StreamCapable** annotation in the script) checks a script against the following criteria:

- The variable is referenced only once.
- No index selector is set for negative access, such as **[-1]**.
- No reference to the variable is found in a nested lambda.

If all criteria are met, the selected data is streamable.

The following example validates successfully. The script is designed to act on the **JSON input** from the **JSON streaming** section:

```
%dw 2.0

@StreamCapable()
input payload application/json
output application/json
---
payload.family filter (member) -> member.age > 3
```

The script successfully validates and returns the following output:

```
[ 
  {
    "name": "Pedro",
    "age": 4
  },
  {
    "name": "Matias",
    "age": 8
  }
]
```

## Validation Failures

If any of the criteria that the validator checks is false, the validation fails.

Before proceeding, note that validation can fail in some cases when streaming works. If you write a script in a way that sequentially accesses the input variable in a given data source, streaming works, but that script might not work in all cases. For example, JSON does not place a restriction on

the order of the keys in an object. If the keys in some JSON documents arrive in a different order than the script expects, streaming will fail in those cases. The annotation processor follows the rules of the format and cannot assume that the keys always arrive in the same order.

#### Error: Variable Is Referenced More Than Once

Validation fails if a script attempts to reference the same variable more than once.

The following script is designed to act on the [JSON input](#) from the [JSON streaming](#) section. Validation fails because the script attempts to reference the [payload](#) variable more than once:

```
%dw 2.0

@StreamCapable()
input payload application/json
output application/json
---
{
    family: payload.family filter (member) -> member.age > 3,
    name: payload.name
}
```

The script fails with the following error:

```
4| input payload application/json streaming=true
      ^^^^^^
Parameter `payload` is not stream capable.
Reasons:
- Variable payload is referenced more than once. Locations:
-----
8|     family: payload.family filter (member) -> member.age > 3,
      ^^^^^^
-----
9|     name: payload.name
      ^^^^^^ at
4| input payload application/json streaming=true
```

#### Error: Wrong Scope Reference

Validation fails if a script attempts to reference a variable from a scope that is different from the scope in which the variable is defined.

The following script fails because the [payload](#) variable is referenced from within the lambda expression [\[1,2,3\] map \(\(item, index\) -> payload\)](#). Even if the expression is [\[1\] map \(\(item, index\) -> payload](#), streaming fails because [payload](#) is in the wrong scope.

```
%dw 2.0

@StreamCapable()
input payload application/json
output application/json
---
[1,2,3] map ((item, index) -> payload)
```

The example fails with the following error:

```
4| input payload application/json
   ^^^^^^
Parameter 'payload' is not stream capable.
Reasons:
- Variable payload is referenced in a different scope from where it was defined.
Locations:
-----
9| [1,2,3] map ((item, index) -> payload)
   ^^^^^^^^^^^^^^^^^^^^^ at
4| input payload application/json
```

## Streaming Output

After processing streamed data, you can stream the output directly to another message processor. To facilitate this behavior, use the **deferred** writer property in the output directive of the DataWeave script, for example, **output application/json deferred=true**.

### NOTE

Exceptions are not handled when you set **deferred = true**. For example, you can see this behavior in Studio when a flow throws an exception. If you are running an application in Studio debug mode and an exception occurs in a Transform Message component when **deferred = true**, the console logs the error, but the flow does not stop at the Tranform Message component.

Building on the example in [JSON Streaming](#), the following flow uses a DataWeave script to filter streamed input and then streams the output directly to a Write operation:

```

<flow name="dw-streamingexample">
    <file:listener doc:name="On New or Updated File"
        config-ref="File_Config" directory="/Users/me/testing/json" recursive="false"
        outputMimeType="application/json;
            streaming=true">
        <scheduling-strategy>
            <fixed-frequency timeUnit="SECONDS" />
        </scheduling-strategy>
        <file:matcher />
    </file:listener>
    <ee:transform doc:name="Transform Message">
        <ee:message>
            <ee:set-payload><![CDATA[%dw 2.0

@StreamCapable()
input payload application/json
output application/json deferred = true
---

{
    family: payload.family filter (member) -> member.age > 1
}]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <file:write doc:name="Write"
        config-ref="File_Config2"
        path="/Users/me/testing/output.json"/>
</flow>

```

The flow provides the following configuration:

1. The listener (`<file:listener>`) uses `streaming=true` to stream the incoming JSON data.
2. The DataWeave script in `<ee:transform/>` filters records in the streamed data and uses the `deferred = true` property to stream the resulting records directly to the next processor in the flow.
3. The next component in the flow, `<file:write/>`, receives the filtered stream directly and writes the records to a file.

## See Also

- [CSV Format](#)
- [JSON Format](#)
- [XML Format](#)
- [Supported DataWeave Formats](#)

# Indexed Readers in DataWeave

In DataWeave, some readers of input data (such as XML, JSON, and CSV) support an indexed strategy to avoid loading the whole document in memory while still allowing random access just like in the in-memory strategy. DataWeave uses indexed readers when the input is larger than a certain configurable threshold or if the `indexedReader` setting is set to `true`.

The trade-off for managing files that wouldn't fit in memory is to write the files to a temporary file in disk and index the document content beforehand, which also takes time and disk space.

Indexed readers can process input files up to 20 GB. For bigger files use the streaming strategy, which is faster. Despite this mode being more limited in functionality, there's no maximum input size limitation for streaming readers.

The actual max input file size supported is difficult to estimate because it depends on the content of the input.

The following list shows the limits on different parts of the input file:

- Max nesting depth: [4096](#)
- Max individual value size: [4 GB](#)
- Tokens: [2<sup>32</sup>-1](#) (approximately 4 billion)  
A token is either a key/value pair, object-start or array-start, that limits the max input file size.  
The amount of the max input file size is irrespective of the size of each token in the file. The bigger your average value length, the bigger your input file size can be because what is important is the amount of tokens, not their size.

The following examples show input file size samples:

In this example, the max input file size is between 40 GB and 50 GB:

```
[  
 {  
   "name": "Mariano",  
   "lastName": "Achaval"  
 },  
 ...  
 ]
```

The following example is similar to the previous one but minified where the max file size is approximately 30 GB:

```
[{"name": "Mariano", "lastName": "Achaval"}, ...]
```

The following example shows a minified array of numbers [1](#). DataWeave supports only approximately 8 GB of such file:

```
[1,1,1,1,1,1,...]
```

## Large Strings Management

When processing a **String** with a size larger than 1.5 MB, DataWeave automatically splits the value in chunks to avoid out-of-memory issues. This feature works only with **JSON** and **XML** input data.

You can configure the threshold that DataWeave uses to determine when to process a string using this splitting strategy by modifying the following system property:

```
com.mulesoft.dw.max_memory_allocation
```

Note that using this feature has a negative impact on performance due to splitting strings and accessing them through local storage. If your environment has enough resources to load all content in memory, you can also disable this string management feature completely by setting the value of the following system property to **false**:

```
com.mulesoft.dw.buffered_char_sequence.enabled
```

## See Also

- [CSV Format](#)
- [JSON Format](#)
- [XML Format](#)
- [Supported DataWeave Formats](#)

## Flatfile Schemas

DataWeave can process several different types of data. For most of these types, you can import a schema that describes the input structure in order to have access to valuable metadata at design time. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

DataWeave uses a YAML format called FFD (for Flat File Definition) to represent flat file schemas. The FFD format is flexible enough to support a range of use cases, but is based around the concepts of elements, composites, segments, groups, and structures.

Schemas must be written in Flat File Schema Language, and by convention use a **.ffd** extension. This language is very similar to EDI Schema Language (ESL), which is also accepted by Anypoint Studio.

In DataWeave, you can bind your input or output to a flat file schema through a property.

Note that if you intend to use a simple fixed-width format, you can set up your data type directly through the Transform component using the **Fixed Width** type. Creating an instance of this type in Studio will automatically generate a matching schema definition.

## Types of Components in a Schema

Here are the different types of components that make up a flat file schema, going from the most elementary to the more complex. Elements and Segments are always required, while Composites, Groups, and Structures might be needed depending on the [top-level structure of your document](#).

- Element - An element is a basic data item, which has an associated type and fixed width, along with formatting options for how the data value is read and written.
- Composite - (Optional) A group of elements. It can also include other child composites.
- Segment - A line of data, or record, made up of any number of elements and/or composites that might be repeated.
- Group - (Optional) Several segments grouped together. It can also include other child groups.
- Structure - A hierarchical organization of segments, which requires that the segments have unique identifier codes as part of their data.

## Top-level structure of an FFD Document

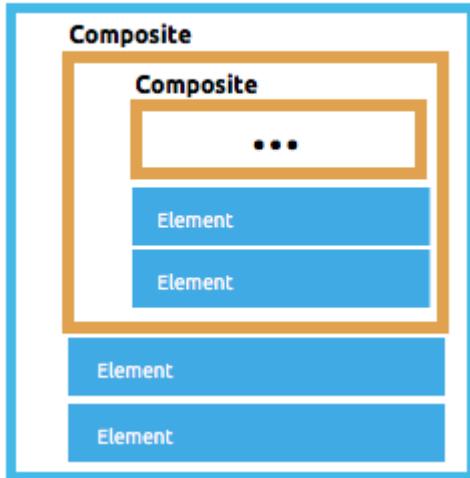
The top-level definition in an FFD document starts with the form of the schema. In this case it must always be "FLATFILE", "COPYBOOK", or "FIXEDWIDTH". The differences between these forms are minor and mostly relate to how they are handled in Studio. The rest depends on the form of definitions present in the file, with the following alternatives:

- Single segment - Segment information directly at the top level (including a **values** key giving the element and/or composite details of the segment).
- Multiple segments - A **segments** key with multiple child objects, each defining one segment.
- Multiple structures - A **structures** key with multiple child objects, each defining one structure.

### Single Segment

If you are only working with one type of record, you only need to have a segment definition for that record type in your FFD.

## Segment



This is a simple schema with only one segment:

```
form: FLATFILE
id: 'RQH'
name: Request Header Record
values:
- { name: 'Organization Code', type: String, length: 10 }
- { name: 'File Creation Date', type: Date, length: 8 }
- { name: 'File Creation Time', type: Time, length: 4 }
```

The example above simply defines a list of elements within a single segment. It has no composites to group them into.

Here is another example:

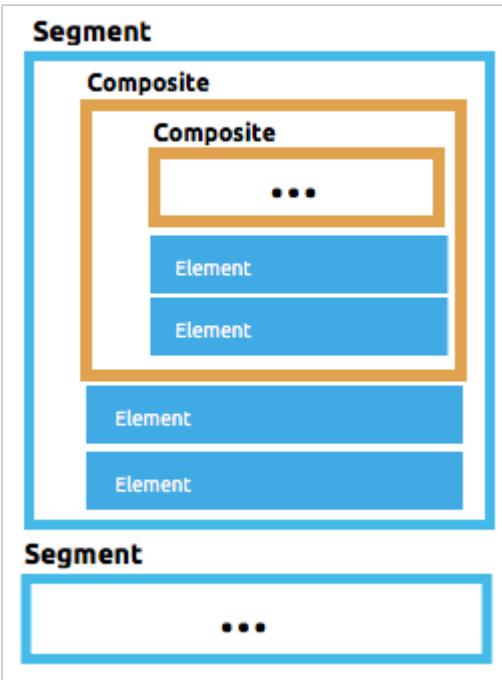
```
form: FIXEDWIDTH
name: my-flat-file
values:
- { name: 'Row-id', type: String, length: 2 }
- { name: 'Total', type: Decimal, length: 11 }
- { name: 'Module', type: String, length: 8 }
- { name: 'Cost', type: Decimal, length: 8, format: { implicit: 2 } }
- { name: 'Program-id', type: String, length: 8 }
- { name: 'user-id', type: String, length: 8 }
- { name: 'return-sign', type: String, length: 1' }
```

Note that simplified forms are only for convenience. You can use the [segments](#) key even if you only have a single child segment definition.

## Multiple Segments

If you are working with multiple types of records in the same transformation, you need to use a structure definition that controls how these different records are combined.

## Segments



In most cases, you need to define a way to distinguish between the different types of records. You do this by identifying `tagValue` fields as part of the record definitions. The parser will check the content of the fields of each record to identify the record type, then see how it fits into the defined structure.

This is an example of a complete structure schema with several component record types:

```
form: FIXEDWIDTH
structures:
- id: 'BatchReq'
  name: Batch Request
  data:
    - { idRef: 'RQH' }
    - groupId: 'Batch'
      count: '>1'
      items:
        - { idRef: 'BCH' }
        - { idRef: 'TDR', count: '>1' }
        - { idRef: 'BCF' }
    - { idRef: 'RQF' }
segments:
- id: 'RQH'
```

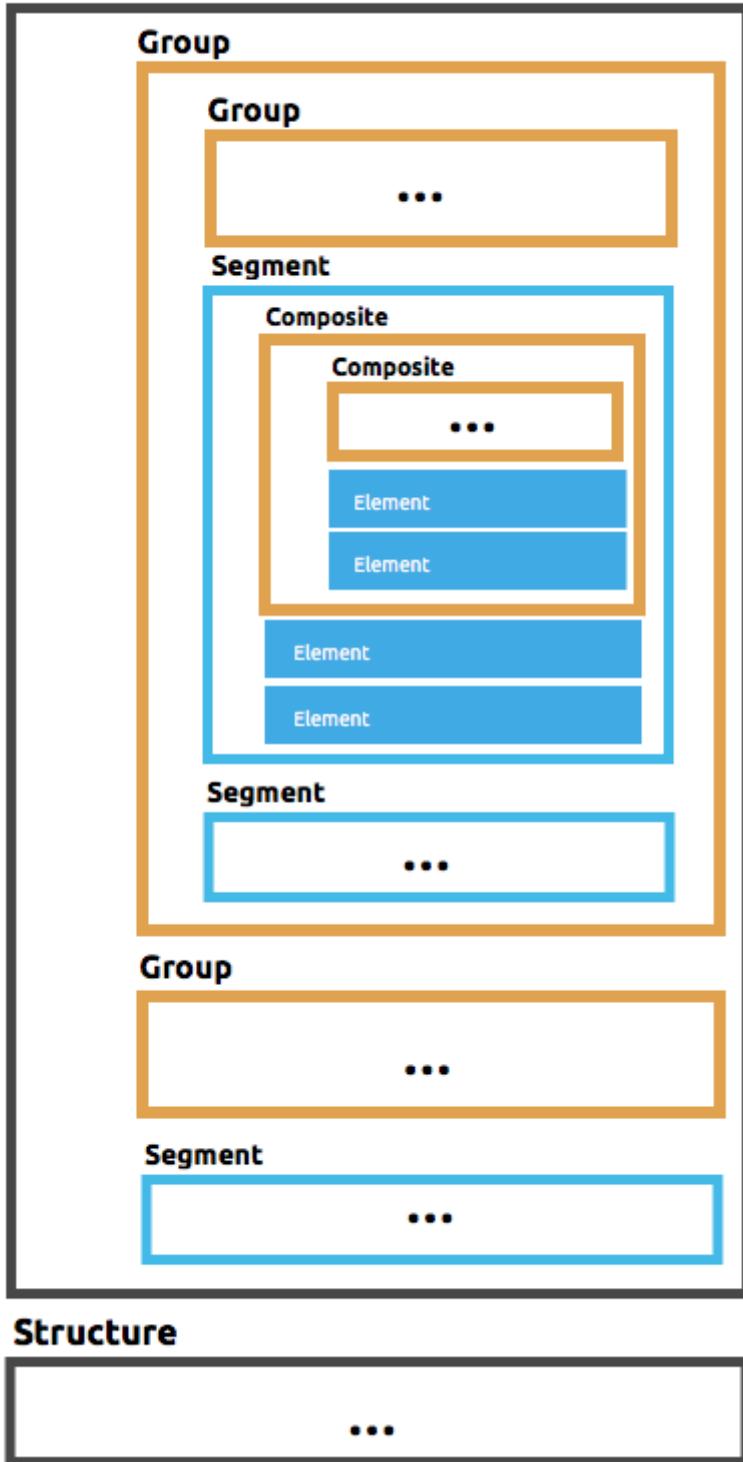
Note that this example is not complete. It needs to define each of the referenced segments at the end. See [Referenced versus Inlined Definitions](#) to understand how these segments are being referenced in this example.

## Multiple Structures

If you have multiple structures or segment definitions in an FFD, when you apply your schema to an metadata description on a Transform component, you need to specify which one you want to use.

## Structures:

### Structure



A top-level structure of a schema with multiple structures might look like this:

```

form: FIXEDWIDTH
structures:
- id: 'BatchReq'
  name: Batch Request
  data:
    - { idRef: 'RQH' }
    - groupId: 'Batch'
      usage: 0
      count: '>1'
      items:
        - { idRef: 'BCH' }
        - { idRef: 'TDR', count: '>1' }
        - { idRef: 'BCF' }
    - { idRef: 'RQF' }
- id: 'BatchRsp'
  name: Batch Response
  data:
    - { idRef: 'RSH' }
    - groupId: 'Batch'
      usage: 0
      count: '>1'
      items:
        - { idRef: 'BCH' }
        - { idRef: 'TDR', count: '>1' }
        - { idRef: 'BCF' }
    - { idRef: 'RSF' }
segments:
- id: 'RQH'
...

```

The example defines two different structures, the `BatchReq` structure and the `BatchRsp` structure. Each of these structures uses a particular sequence of segments and groups of segments. The group `Batch` is repeated in both structures. A Batch group is composed of a single BCH line, multiple TDR lines, and a single BCF line.

Note that this example is not complete. It needs to define each of the referenced segments at the end. See [Referenced versus Inlined Definitions](#) to understand how these segments are referenced in this example.

## Element Definitions

Element definitions are the basic building blocks of application data, consisting of basic key-value pairs for standard characteristics. Flat file schemas generally use inline element definitions, where each element is defined at the point it is used within a segment or composite structure, but you can also define elements separately and reference them as needed. Here are several element definitions defined for use by reference:

```

- { id: 'OrgCode', name: 'Organization Code', type: String, length: 10 }
- { id: 'CreateDate', name: 'File Creation Date', type: Date, length: 8 }
- { id: 'CreateTime', name: 'File Creation Time', type: Time, length: 4 }
- { id: 'BatchTransCount', name: 'Batch Transaction Count', type: Integer, format: {
justify: zeroes }, length: 6 }
- { id: 'BatchTransAmount', name: 'Batch Transaction Amount', type: Integer, format: {
justify: zeroes }, length: 10 }

```

The supplied **id** value is used as the **idRef** value when referencing one of these definitions as part of a segment or composite. Note that if you are defining elements inline within a segment definition (as opposed to defining them at the end of the document and referencing them), the **id** field is not required.

Element definitions might have the following attributes, classified by Form as applying to inline definitions, referenced definitions (as in the above example), or references to definitions:

*Table 1. Attributes*

Name	Description	Form
<b>count</b>	Number of occurrences (optional, default is 1)	Inline or reference
<b>id</b>	Element identifier	Referenced definition
<b>idRef</b>	Element identifier	Reference
<b>name</b>	Element name (optional)	All
<b>type</b>	Value type code, as listed below	Inline, or referenced definition
<b>format</b>	Type-specific formatting information	Inline, or referenced definition
<b>length</b>	Number of character positions for value	Inline, or referenced definition
<b>tagValue</b>	Value for this element used to identify a segment (see <a href="#">the Full Schema Example</a> )	Inline, or referenced definition

The allowed types for defining an element are:

*Table 2. Types*

Name	Description
Binary	Binary value (COBOL format, 2, 4, or 8 bytes)
Boolean	Boolean value
Date	Unzoned date value with year, month, and day components (which might not all be shown in text form)
DateTim e	Unzoned date/time value with year, month, day, hour, minute, second, and millisecond components (which might not all be shown in text form)

Name	Description
Decimal	Decimal number value, which might or might not include an explicit decimal point in text form
Integer	Integer number value
Packed	Packed decimal representation of a decimal number value (COBOL format)
String	String value
Time	Unzoned time value with hour, minute, second, and millisecond components (which might not all be shown in text form)
Zoned	Zoned decimal (COBOL format)

Value types support a range of format options that affect the text form of the values. These are the main options, along with the types to which they apply:

*Table 3. Format Options*

Key	Description	Applies to
digits	Number of digits allowed	Binary
implicit	Implicit number of decimal digits (used for fixed-point values with no decimal in text form)	Binary, Decimal, Packed, Zoned
justify	Justification in field (LEFT, RIGHT, NONE, or ZEROES, the last only for numbers)	All except Binary and Packed
pattern	For numeric values, the <code>java.text.DecimalFormat</code> pattern for parsing and writing. For date/time values, the <code>java.time.format.DateTimeFormatter</code> pattern.	Date, DateTime, Decimal, Integer, Time
sign	Sign usage for numeric values (UNSIGNED, NEGATIVE_ONLY, OPTIONAL, ALWAYS_LEFT, ALWAYS_RIGHT)	Decimal, Integer, Zoned
signed	Signed versus unsigned flag	Binary, Packed

## Composite Definitions

Composites serve to reference a list of elements that are typically presented together. For example, `firstName` and `lastName` can be bundled together into a single composite because they are likely to be referred to as a group. Grouping elements into a composite also allows the list to be repeated.

Composite definitions are very similar to segment definitions, composed of some key-value pairs for standard characteristics along with lists of values. Composites might include both references to elements or other nested composites and inlined definitions. This is a example of a composite definition:

```

- id: 'DateTime'
  name: 'Date/Time pair'
  values:
    - { name: 'File Creation Date', type: Date, length: 8 }
    - { name: 'File Creation Time', type: Time, length: 4 }

```

Composite definitions might have the following attributes:

Name	Description	Form
<code>controlVal</code>	Value from containing level giving actual number of occurrences (only used with <code>count</code> != 1)	Inline definition, or on reference
<code>count</code>	Number (or maximum number, if <code>controlVal</code> is used) of occurrences (optional, default is 1)	Inline definition, or on reference
<code>id</code>	Composite identifier for references	Referenced definition
<code>name</code>	Composite name (optional)	Inline or referenced definition
<code>values</code>	List of elements and composites within the composite	Inline or referenced definition

The values list takes the same form as the values list in a segment definition.

## Segment Definitions

A segment describes a type of record in your data. Segments are mainly composed of references or direct definitions of elements and composites, together with some key-value pairs that describe the segment. In a somewhat complex schema, you might have a structure that contains two different segments, where one of these describes the fields that go in the single header of a bill of materials (such as date and person), while the other segment describes the recurring fields that go into each of the actual items in the bill of materials.

This is a sample segment definition that includes one simple element and a composite with two elements within:

```

- id: 'RQH'
  name: Request Header Record
  values:
    - { name: 'Organization Code', type: String, length: 10 }
    - id: 'DateTime'
      name: 'Date/Time pair'
      values:
        - { name: 'File Creation Date', type: Date, length: 8 }
        - { name: 'File Creation Time', type: Time, length: 4 }

```

Segment definitions might include the following attributes:

Section	Description
<code>id</code>	Segment identifier (unused for inline definitions, required for <a href="#">referenced definitions</a> )
<code>name</code>	Segment name (optional)
<code>values</code>	List of elements and composites within the segment (either inlined, or <a href="#">references</a> )

## Structure Definitions

Structure definitions are composed of a list of references to segments and group definitions, as well as a set of key-value pairs for standard characteristics. Segments may be further organized into groups consisting of a potentially repeated sequence of segments.

Here's a sample structure definition again:

```
form: FIXEDWIDTH
structures:
- id: 'BatchReq'
  name: Batch Request
  data:
    - { idRef: 'RQH' }
    - groupId: 'Batch'
      count: '>1'
      items:
        - { idRef: 'BCH' }
        - { idRef: 'TDR', count: '>1' }
        - { idRef: 'BCF' }
    - { idRef: 'RQF' }

segments:
- id: 'RQH'
```

This example includes references to two segments at the top level (**RQH** and **RQF**), as well as a group definition **Batch** that includes references to other segments (**BCH**, **TDR** and **BCF**). Note that for this structure to work, each of the **referenced segments** needs to be defined. See [Referenced versus Inlined Definitions](#) to understand how segments are referenced in this example.

A structure definition can contain the following attributes:

Structure Key/Section	Description
<code>id</code>	Structure identifier
<code>name</code>	Structure name (optional)
<code>data</code>	List of segments (and groups) in the structure

Each item in a segment list is either a segment reference (or inline definition) or a group definition (always inline).

## Group Definitions

A group definition can have the following attributes:

Value	Description
groupId	The group identifier
usage	Usage code, which might be M for Mandatory, O for Optional, or U for Unused (optional, defaults to M)
count	Maximum repetition count value, which might be a number or the special value >1, meaning any number of repeats (optional, count value of 1 is used if not specified)
items	List of segments (and potentially nested groups) making up the group

## Referenced Versus Inlined Definitions

Besides the choice of top-level form, you also have choices when it comes to representing the components of a structure, segment, or composite. You can define the component segments, composites, and elements inline at the point of use, or you can define them in a table and reference them from anywhere. Inlining definitions is simpler and more compact, but the table form allows definitions to be reused. Table form examples must include an `id` value, and each reference to that definition uses an `idRef`. This example shows how this applies to the segments making up a structure:

```

form: FIXEDWIDTH
structures:
- id: 'BatchReq'
  name: Batch Request
  data:
    - { idRef: 'RQH' }
    - { idRef: 'RQF' }
segments:
- id: 'RQH'
  name: "Request File Header Record"
  values:
    - { idref: createDate }
    - { idref: createTime }
    - { idref: fileId }
    - { idref: currency }
- id: 'RQF'
  name: "Request File Footer Record"
  values:
    - { idref: batchCount }
    - { idref: transactionCount }
    - { idref: transactionAmount }
    - { idref: debitCredit }
    - { idref: fileId }
elements:
- { id: createDate, type: Date, length: 8 }
- { id: createTime, type: Time, length: 4 }
- { id: fileId, type: String, length: 10 }
- { id: currency, type: String, length: 3 }
- { id: batchCount, type: Integer, format: { justify: zeroes }, length: 4 }
- { id: transactionCount, type: Integer, format: { justify: zeroes }, length: 6 }
- { id: transactionAmount, type: Integer, format: { justify: zeroes }, length: 12 }
- { id: debitCredit, type: String, length: 2 }

```

In the above example, the `BatchReq` structure references segments in the `data` definition section. The segments are each then defined in the `segments` section at the top level of the schema, and these in turn reference elements that are later defined in the `elements` section.

An inlined definition of the same structure looks like this:

```

form: FIXEDWIDTH
structures:
- id: 'BatchReq'
  name: Batch Request
  data:
    - { idRef: 'RQH' }
    - { idRef: 'RQF' }
segments:
- id: 'RQH'
  name: "Request File Header Record"
  values:
    - { name: 'File Creation Date', type: Date, length: 8 }
    - { name: 'File Creation Time', type: Time, length: 4 }
    - { name: 'Unique File Identifier', type: String, length: 10 }
    - { name: 'Currency', type: String, length: 3 }
- id: 'RQF'
  name: "Request File Footer Record"
  values:
    - { name: 'File Batch Count', type: Integer, format: { justify: zeroes }, length: 4 }
    - { name: 'File Transaction Count', type: Integer, format: { justify: zeroes }, length: 6 }
    - { name: 'File Transaction Amount', type: Integer, format: { justify: zeroes }, length: 12 }
    - { name: 'Type', type: String, length: 2 }
    - { name: 'Unique File Identifier', type: String, length: 10 }

```

## Full Example Schema

```

form: FLATFILE
structures:
- id: 'BatchReq'
  name: Batch Request
  data:
    - { idRef: 'RQH' }
    - groupId: 'Batch'
      count: '>1'
      items:
        - { idRef: 'BCH' }
        - { idRef: 'TDR', count: '>1' }
        - { idRef: 'BCF' }
    - { idRef: 'RQF' }
segments:
- id: 'RQH'
  name: "Request File Header Record"
  values:
    - { name: 'Record Type', type: String, length: 3, tagValue: 'RQH' }
    - { name: 'File Creation Date', type: Date, length: 8 }
    - { name: 'File Creation Time', type: Time, length: 4 }

```

```

- { name: 'Unique File Identifier', type: String, length: 10 }
- { name: 'Currency', type: String, length: 3 }
- id: 'BCH'
  name: "Batch Header Record"
  values:
    - { name: 'Record Type', type: String, length: 3, tagValue: 'BAT' }
    - { name: 'Sequence Number', type: Integer, format: { justify: zeroes }, length: 6 }
    - { name: 'Batch Function', type: String, length: 1, tagValue: 'H' }
    - { name: 'Company Name', type: String, length: 30 }
    - { name: 'Unique Batch Identifier', type: String, length: 10 }
- id: 'TDR'
  name: "Transaction Detail Record"
  values:
    - { name: 'Record Type', type: String, length: 3, tagValue: 'BAT' }
    - { name: 'Sequence Number', type: Integer, format: { justify: zeroes }, length: 6 }
    - { name: 'Batch Function', type: String, length: 1, tagValue: 'D' }
    - { name: 'Account Number', type: String, length: 10 }
    - { name: 'Amount', type: Integer, format: { justify: zeroes }, length: 10 }
    - { name: 'Type', type: String, length: 2 }
- id: 'BCF'
  name: "Batch Footer Record"
  values:
    - { name: 'Record Type', type: String, length: 3, tagValue: 'BAT' }
    - { name: 'Sequence Number', type: Integer, format: { justify: zeroes }, length: 6 }
    - { name: 'Batch Function', type: String, length: 1, tagValue: 'T' }
    - { name: 'Batch Transaction Amount', type: Integer, format: { justify: zeroes },
length: 10 }
      - { name: 'Type', type: String, length: 2 }
      - { name: 'Batch Transaction Count', type: Integer, format: { justify: zeroes },
length: 6 }
      - { name: 'Unique Batch Identifier', type: String, length: 10 }
- id: 'RQF'
  name: "Request File Footer Record"
  values:
    - { name: 'Record Type', type: String, length: 3, tagValue: 'RQF' }
    - { name: 'File Batch Count', type: Integer, format: { justify: zeroes }, length: 4
}
    - { name: 'File Transaction Count', type: Integer, format: { justify: zeroes },
length: 6 }
    - { name: 'File Transaction Amount', type: Integer, format: { justify: zeroes },
length: 12 }
      - { name: 'Type', type: String, length: 2 }
      - { name: 'Unique File Identifier', type: String, length: 10 }

```

This example contains a single [structure](#) named 'BatchReq' with 5 components [segments](#), using a doubly-nested structure of file and batch data for the segments. Each batch contains repeating detail records. All element definitions are in-lined.

The [BatchReq](#) structure definition requires that the data will consist of:

- A single record that corresponds to the segment [RQH](#)

- One or more records that correspond to the segment **BCH**
- For each **BCH** record, one or more **TDR** records giving details of a particular transaction
- For each **BCH** record, a **BCF** record following any contained **TDR** records
- A final, single record that corresponds to the segment **RQF**

For this example every record starts with a three-character Record Type field with a specified **tagValue**. In the case of the batch records, the record type is further specified by a Batch Function **tagValue**.

This is a sample of data matching the schema example:

```
RQH201809011010A00000001USD
BAT00001HACME RESEARCH           A00000001
BAT00002D0123456789000032876CR
BAT00003D0123456788000087326CR
BAT00004T0000120202CR000002A00000001
BAT00005HAJAX EXPLOSIVES         A00000002
BAT00006D1234567890000003582DB
BAT00007D1234567891000000256CR
BAT00008T000003326DB000002A00000002
RQF00020000800000116876CRA00000001
```

The lines in the example match the defined structure as listed below:

- 1 **RQH** (Request File Header Record) identified by the "RQH" value in the first three characters
- 2 **BCH** (Batch Header Record) identified by the "BAT" value in the first three characters combined with the 'H' character in position 10
- 3-4 **TDR** (Transaction Detail Record) identified by the "BAT" value in the first three characters combined with the 'D' character in position 10
- 5 **BCF** (Batch Footer Record) identified by the "BAT" value in the first three characters combined with the 'T' character in position 10
- 6 **BCH** (Batch Header Record) identified by the "BAT" value in the first three characters combined with the 'H' character in position 10
- 7-8 **TDR** (Transaction Detail Record) identified by the "BAT" value in the first three characters combined with the 'D' character in position 10
- 9 **BCF** (Batch Footer Record) identified by the "BAT" value in the first three characters combined with the 'T' character in position 10
- 10 **RQF** (Request File Footer Record) identified by the "RQF" value in the first three characters

**tagValue** fields provide a lot of flexibility. The above example shows using a single **tagValue** for some record types, while adding a second **tagValue** for others, but you can also use completely different fields (or even disjoint sets of fields) for a **tagValue**, as long as you provide enough details for the parser to distinguish between the different types of records.

Note that older versions of the documentation showed a different way of distinguishing records

based on tag values, using `tagStart` and `tagLength` values for the structure and `tag` values for the segments. This method of distinguishing segments is much more limited than the `tagValue` approach, and is now deprecated.

## See Also

[\[dataweave-formats:::format\\_flat\\_file\]](#)

# Type System

2.x versions of DataWeave support a type system. To take advantage of the type-checking that the type system executes, you need to provide constraint expressions for variables and functions described in this section. For information about how values for these types are defined, refer to [Value Constructs for Types](#).

A type system defines a set of constraints to a set of constructs, such as:

- Variables
- Function parameters

These constraints are used in the type-checking phase, when DataWeave ensures that the values assigned to variables or the arguments for a function call respect its constraints, for example:

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/json

var userName: String = "John"
```

The variable definition for `userName` has the constraint of assigning a value of type `String`. If some other value type is assigned, a type-checking error is raised.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/json

fun toUser(id: Number, userName: String): String = "you called the function toUser!"
```

The constraints to call the function `toUser` are that the first argument has to be of type `Number`, the second one of type `String` (to be coercible to those types). Another constraint is that the result of the function has to be of type `String`. This constraint isn't used in the call for `toUser`; it's applied to its definition to validate that the function body generates the proper type.

```
toUser(123, "John")
toUser("123", true)
```

Although you might expect the second call to fail, autocoercion enables the call to be successful, and both calls work. In the case of the second function call, there is autocoercion of the `String` value `"123"` to the `Number` value `123`, and the `Boolean` value `true` to the `String` value `"true"`, which makes it possible to call the function.

An example of a call that throws a type-checking error is `toUser("a 12", "John")`, because the `String` value `"a 12"` isn't coercible to a `Number` type.

DataWeave also uses a global type inference algorithm to validate your code even if no types are specified as constraints.



Although the use of constraints is optional, they can be useful in big scripts or multiple scripts. The type system helps you to avoid bugs in your DataWeave logic because it causes the type-checking algorithm to run. Additionally, when you write your script in an IDE and define these constraints, the tooling helps you to find type checking errors, instead of when the script is running and fails.

The types supported by the DataWeave type system are divided into three categories:

- Simple Types
- Composite Types
- Complex Types

## Simple Types

Simple types represent values such as strings, Booleans, and so on. These values are atomic; they are not composed of other values. These are the simple types:

- `String`
- `Boolean`
- `Number`
- `Regex`
- `Null`
- Temporal: `Date`, `DateTime`, `LocalDateTime`, `LocalTime`, `Time`, `Period`

## Null

`Null` is a type of only one value, the value `null`, which means that `null` cannot be assigned to the type `String` or any other type except for the type `Null`.

For example, the `repeat` function in the `String` module has the signature `repeat(String, Number): String`, which means that the function accepts as a first parameter only a value of type `String`, and as a second parameter only a value of type `Number`. The function returns a value of type `String`.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
repeat("a", 3)
```

This returns "aaa", but the following example throws an error because the value `null` cannot be assigned to the parameter that expects a value of type `String`:

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
repeat(null, 3)
```

In the following example, the script assigns to the first parameter a value that is the type `Null` (the type of this value is `Type<Null>`). It returns "NullNullNull" because autocoercion coerces the value to the type `String ("Null")`.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
repeat(Null, 3)
```

Some functions have an overloaded signature (the function has multiple definitions for different parameter types) that let you call it with `null`.

For example, the `isNumeric` function from `dw::core::Strings` has the signature `isNumeric(String): Boolean`, which receives a string and successfully returns a result if that string is numeric, but the function is also defined as `isNumeric(Null): Boolean`.

So, if you call the function `isNumeric` with `null` (`isNumeric(null)`), the function does not throw an exception. At runtime, the function that is dispatching algorithm selects the correct function based on the types of the values you use. The function returns `false` because of the definition of `isNumeric` for `null` as the argument.

## Composite Types

A composite type contains other values. The composite types are `Array`, `Object` and `Function`.

### Array Type

An array is of the type `Array<T>` in which `T` is a type parameter that defines the type of the elements inside the array. For example, the syntax to define a variable with the type `Array` of `Number` is:

```
var idsList: Array<Number> = [1, 22, 333, 4444]
```

The variable `idsList` passes the type-checking phase only when the array that is assigned to it only contains values of the `Number` type.

## Object Type

You can define an `Object` type in two ways: `Object` or `{}`. Both define an open object and don't specify any constraint to their key-value pairs. See also [Close and Open Objects](#).

The syntax for defining an `Object` type with a set of constraints for its key-value pairs is very similar to the syntax for defining the value of an `Object` type:

```
{
  keyName @(attrsName: AttrType): valueType,
  ...
}
```

For example, the syntax to define a `User` type with `firstName String`, `lastName String`, and `age Number` is:

*DataWeave Script:*

```
%dw 2.0
output application/json

type User = {
  firstName: String,
  lastName: String,
  age: Number
}
```

You can apply modifiers to each key-value pair to specify when a field is always present or when a field can be repeated.

For repeated fields use `*`. For example, the following type allows you to have multiple `lastName` fields:

*DataWeave Script:*

```
%dw 2.0
output application/json

type User = {
    firstName: String,
    lastName*: String,
    age: Number
}
```

For conditional fields, use `?`. For example, the following type allows the `age` field to be present or not:

*DataWeave Script:*

```
%dw 2.0
output application/json

type User = {
    firstName: String,
    lastName*: String,
    age?: Number
}
```

## Close and Open Objects

`Object` type can be either closed or open. A *closed* object accepts an object value only if there are no fields except the key-value pairs specified by the type.

An *open* object only put constraints on the fields that are declared in the type, and if they are all verified, then the type accepts the value. The type accepts objects that have other fields apart of the ones mentioned in the declaration. All `object` types are open unless specified.

The following example succeeds even if `age` is not in the type declaration because the type of the variable `user` is an open object:

*DataWeave Script:*

```
%dw 2.0
output application/json

var user: {firstName: String, lastName: String} = {firstName: "John", lastName: "Smith", age: 34}
```

The syntax to specify a closed object is `{| |}`. The following example makes the `User` type support only the keys `firstName` and `lastName`. The script throws an exception because the `age` field is not accepted.

*DataWeave Script:*

```
%dw 2.0
output application/json

var user: {|firstName: String, lastName: String|} = {name: "John", lastName: "Smith",
age: 34}
```

## Function Type

DataWeave is a functional language, and functions are considered first class citizens, which means that functions are values with associated types.

The syntax to define a function type is:

```
(paramType: ParamType,...) -> ReturnType
```

For example, if you want to define in a constraint a **Function** type that has a parameter of type **String**, a second parameter of type **Number**, and a **Boolean** return value, the correct syntax is:

```
(paramA: String, paramB: Number) -> Boolean
```

In the following example, you define a function that receives another function as an argument:

*DataWeave Script:*

```
%dw 2.0
output application/json

fun applyIDsChange(ids: Array<Number>, changeTo: (Number) -> Number): Array<Number> =
???
```

If you call the function **applyIDsChange** with a function that does not match the **changeTo** constraint **(Number) → Number**, DataWeave throws a type-checking error.

For example, the following call works because **abs** is a function that takes a value of the **Number** type and returns a **Number**, which matches the **changeTo** parameter:

```
applyIDsChange([1,-6, 3, -8], abs)
```

But the following call fails because **sum** is a function that takes an **Array** and returns a **Number**, so it does not match the parameter constraint:

```
applyIDsChange([1,-6, 3, -8], sum)
```

## Complex Types

Complex types include the `Any` and `Nothing` types, the `Union` type, the `Intersection` type, and `Literal` types, each of which is named intuitively.

### Any and Nothing

In some cases, you cannot enforce a restriction because the type accepts all the value:

- `Any` type accepts all possible values.
- `Nothing` type accepts no value, but it can be assigned to all the types. This type is not frequently used explicitly, but it is used by the type inference algorithm.

### Union Type

The `Union` type is used to compose types. The syntax to define a `Union` type is:

```
TypeA | TypeB | ...
```

The following example defines a variable with a constraint that accepts a value of type `String` or `Number`:

```
var age: String | Number = if (payload.allStrings) "32" else 32
```

A common pattern is to use `Null` type with a `Union` type to specify that a type accepts `null` as a value. In the following example, the function `parseEmail` allows inputs of type `String` or `Null`. In this case, you can have a object with a payload optional field `email` and you call the function with `payload.email`.

```
fun parseEmail(email: String | Null) = "Code that handles email being of type String or Null"
```

### Intersection Type

*Introduced in DataWeave 2.3.0. Supported by Mule 4.3 and later.*

The `Intersection` type intersects `Object` types. In this case, the intersection works as the concatenation (`++`) of object types.

The syntax for the `Intersection` type is:

```
TypeA & TypeB & ...
```

In the following example, the `Intersection` concatenates the two `Object` types, resulting in the type `{name: String, lastName: String}`. The type is an open object that can accept additional key-value pairs. The variable accepts the value assigned to it:

```
var a: {name: String} & {lastName: String} = {name: "John", lastName: "Smith", age: 34}
```

In the case of closed objects, it returns the concatenation of the object types but results in a closed object.

In the following example, the intersection results in the type `{|name: String, lastName: String|}`, which throws an exception because a closed object does not accept an `Object` value if there are additional fields in the object (field `age`):

```
var a: {|name: String|} & {|lastName: String|} = {name: "John", lastName: "Smith", age: 34}
```

## Literal Types

*Introduced in DataWeave 2.3.0. Supported by Mule 4.3 and later.*

A literal type represents exactly one value. For example, the `String` value `"foo"` can be represented with the type `"foo"`.

The following literal types are included to the type system:

- `String` literal types
- `Number` literal types
- `Boolean` literal types

You can use literal types with `Union` types to declare a type as a finite set of allowed values. For example, the following type declarations are aliases of `Union` and literal types:

```
type Weekdays = "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday"
type Days = Weekdays | "Saturday" | "Sunday"
```

In the following example, the variable accepts only a value of the literal types `404` and `500`. The type system ensures before runtime that the variable can be only one of those literal values:

*DataWeave Script:*

```
%dw 2.0
output application/json

var errorStatusCode: 404 | 500 = payload.statusCode match {
    case "error 404" -> 404
    case "error 500" -> 500
}
```

Function overloading enables you to define different behaviors based on the input argument's value:

## DataWeave Script:

```
%dw 2.0
import * from dw::core::Strings
output application/json

fun errorHandling(errorCode: 404 | 405, response): String = "Code for error 4XX
handling here!"
fun errorHandling(errorCode: 500 | 501, response): String = "Code for error 5XX
handling here!"
---
errorHandling(payload.statusCode, payload)
```

At runtime, the function-dispatching algorithm selects the correct function based on the type of the value of `payload.statusCode`. It is not necessary to check the value of an argument with an `if` statement to change the behavior of the function.

## Type Parameters

A type parameter enables you to write a function or data type generically so that you can handle values identically, without depending on the type of the values.

For example, you can define a function that returns the first element of an array without losing its type information. In the example, the variable definitions `firstString:String` and `firstNumber:Number` specify the type of the item in the array that the `head` function takes as input.

```
%dw 2.0
fun head<ItemType>(elements: Array<ItemType>): ItemType = elements[0]!

var firstString:String = head(["DataWeave", "Java", "Scala", "Haskell"]) ++ "
Rules!!!"
var firstNumber:Number = head([1,2,3])
output application/json
---
firstNumber
```

In the body of the script, `firstNumber` returns the number `1`. If you replace `firstNumber` with `firstString`, the script returns the string `"DataWeave Rules!!!"`.

## Binding Type Parameters

You can bind type parameters to require a value of a specific type. Appending `{name: "DataWeave"}` in the following example is valid because `{name: String}` in the function definition indicates that the value of the `name` key must be a string.

```
%dw 2.0

fun addName<User <: {}>(user: User): User & {name: String} = user ++ {name: "DataWeave"}

var myUser: {name: String, developers: Number} = addName({developers: 3})
```

## Parameterizing Type Definitions

You can use type parameters to parameterize type definitions. For example, assume that you define a data structure that models file content, one for text files and the other for binary files. You can use type parameters to construct new types that dispatch the correct implementation.

```
%dw 2.0

type FileData<Content> = {
    data: Content,
    name: String
}

fun read(file: FileData<String>) = "This is a text file with data " ++ file.data
fun read(file: FileData<Binary>) = "This is a binary file with data " ++ (file.data as String {base: "64"})

---
{
    binary: read({data: "Hello World" as Binary, name: "myFile.bin"}),
    text: read({data: "Hello World", name: "myFile.txt"})
}
```

## Reuse Types

You can declare and use types within the same DataWeave script (see [Type System](#)), or you can reuse types from different sources by loading them with a specific module loader. DataWeave currently supports loading types from:

- [Reusing Types from DataWeave Modules](#)
- [Reusing Types from a JSON Schema](#)
- [Reusing an XML Schema](#)
- [Reusing Types from Java Classes](#)

Module loaders are used to load files as DataWeave modules, in which different parts of those files are loaded as DataWeave functions, variables, types, and namespace definitions. Using a prefix such as `java!` or `xmllschema!` in the import directive tells DataWeave which module loader to use when loading the Java class or the XML schema as a DataWeave module.

## Reusing Types from DataWeave Modules

To reuse types declared in other DataWeave modules, use the `import` directive:

*DataWeave Module ([dw/Weather.dwL](#)):*

```
%dw 2.0
type Weather = "cloudy" | "sunny" | "rainy" | "stormy"
type DetailedWeather = {temperature: Number, weather: Weather}
```

You can use those imported types as any other type in DataWeave, as constraints for variables and functions or for pattern matching and checking types at runtime:

*DataWeave Script:*

```
%dw 2.0
output json
import * from dw::Weather

fun weatherMessage(todayWeather: Weather) =
    "The weather for today is: " ++ todayWeather
var worstPossibleWeather: DetailedWeather = {temperature: -10, weather: "stormy"}
---
{
    a: weatherMessage("sunny"),
    b: worstPossibleWeather.weather is Weather
}
```

### See Also

- [Create Custom Modules and Mappings](#)

## Reusing Types from a JSON Schema

JSON Schema is a standard that provides a format for the JSON data required for a given application and how to interact with it. Many existing data formats or types are expressed using this standard. In particular, REST APIs described by RAML files are likely to have their types defined in JSON schemas.

To reuse predefined types, you can load the JSON schema into DataWeave to make the types structure available for use in your scripts.

The JSON schema loader parses schemas using the [JSON Schema Draft 7](#).

### Import a Type from a JSON Schema

The following example shows a JSON schema ([Person.json](#)) in the resources directory:

JSON Schema ([example/schema/Person.json](#)):

```
{  
  "$id": "https://example.com/person.schema.json",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "Person",  
  "type": "object",  
  "properties": {  
    "firstName": {  
      "type": "string",  
      "description": "The person's first name."  
    },  
    "lastName": {  
      "type": "string",  
      "description": "The person's last name."  
    },  
    "age": {  
      "description": "Age in years, which must be equal to or greater than zero.",  
      "type": "integer",  
      "minimum": 0  
    }  
  },  
  "required": ["firstName", "lastName"]  
}
```

The following example shows how to import the type using the JSON schema loader:

*DataWeave Script Header:*

```
import * from jsonschema!example::schema::Person
```

## Import Syntax

To import JSON schema types, use the following syntax, where:

- **typeToImport**: You can use `*` alone to import all types defined in the schema or you can import a single type from the schema using, for example, `Root`. You can also import JSON schema types with a different name, for example, `Root as Person`, which enables you to reference the type with that name in the script.
- **pathToJsonSchema**: To specify the path to the schema file, replace the file separators with `::` and remove the `.json` extension from the file name. For example, if the path to the schema is `example/schema/Person.json`, use `example::schema::Person`.

```
import _typesToImport_ from jsonschema!_pathToJsonSchema_
```

The following example shows how to import a type:

```
import * from jsonschema!example::schema::Person
```

## Use Your Types in a DataWeave Script

Including the import directive from the above example in the script header loads all the existing types in the JSON schema. In `import * from jsonschema!example::schema::Person`, the only existing type is the `Root` type, specified at the root which describes an Object that can have three properties (`firstName`, `lastName`, and `age`). This is equivalent to declaring the following type in your DataWeave script:

*DataWeave Script:*

```
%dw 2.0
type Root = { firstName: String, lastName: String, age?: Number }
```

Notice that `age` is the only optional field, as indicated by the `?`.

You can use the type to determine if a value follows the structure defined by the JSON schema. The following example outputs the value `true` because the object contains the required fields, `firstName` and `lastName`:

*DataWeave Script:*

```
%dw 2.0
import * from jsonschema!jsonschema!example::schema::Person
---
{
  firstName: "John",
  lastName: "Doe"
} is Root
```

*Output:*

```
"true"
```

The following example outputs the value `false` because the object does not contain the required fields, `firstName` and `lastName`:

*DataWeave Script:*

```
%dw 2.0
import * from jsonschema!jsonschema!example::schema::Person
---
{
  firstName: "John",
  age: 20
} is Root
```

*Output:*

```
"false"
```

## Use Definitions inside Schemas

DataWave loads the definitions inside JSON schemas when the module is imported:

*JSON Schema ([example/schema/Account.json](#)):*

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": {
          "type": "string"
        },
        "city": {
          "type": "string"
        },
        "state": {
          "type": "string"
        }
      },
      "required": [
        "street_address",
        "city",
        "state"
      ]
    },
    "person": {
      "type": "object",
      "properties": {
        "firstName": {
          "type": "string",
          "description": "The person's first name."
        },
        "lastName": {
          "type": "string",
          "description": "The person's last name."
        },
        "age": {
          "description": "Age in years, which must be equal to or greater than zero.",
          "type": "integer",
          "minimum": 0
        }
      },
      "required": [
        "age"
      ]
    }
  }
}
```

```
        "firstName",
        "lastName"
    ]
}
},
"type": "object",
"properties": {
    "person": {
        "$ref": "#/definitions/person"
    },
    "address": {
        "$ref": "#/definitions/address"
    }
}
}
```

You can import the types from that schema with the following directive:

```
import * from jsonschema!example::schema::Account
```

The types defined in the schema have the same effect as declaring the following types:

```
type Root = { address?: address, person?: person }

type address = { street_address: String, city: String, state: String }

type person = { firstName: String, lastName: String, age?: Number }
```

Use the import directive to import a single type from the schema:

```
import address from jsonschema!example::schema::Account
```

To avoid type-name collision, you can use the `as` keyword to change the imported type name to another name:

```
import address as Account_Address from jsonschema!example::schema::Account
```

## Facilitate Usage with Pattern Matching

You can use the types imported from the JSON schemas to take a specific action when an input is of a certain type. These imported types are useful for pattern matching.

The following example outputs the value `person` because the object is importing that type definition:

*DataWeave Script:*

```
%dw 2.0
import * from jsonschema!example::schema::Account
---
{
  firstName: "John",
  lastName: "Doe"
} match {
  case is person -> "PERSON"
  case is address -> "ADDRESS"
  else -> "NO TYPE MATCHED"
}
```

*Output:*

```
"PERSON"
```

The following example outputs the value `address` because the object is importing that type definition:

*DataWeave Script:*

```
%dw 2.0
import * from jsonschema!example::schema::Account
---
{
  street_address: "742 Evergreen Terrace",
  city: "Gotham",
  State: "OH"
} match {
  case is person -> "PERSON"
  case is address -> "ADDRESS"
  else -> "NO TYPE MATCHED"
}
```

*Output:*

```
"ADDRESS"
```

## Reusing an XML Schema

You can import XML schema files (`.xsd`) in your DataWeave script as modules by using the `xmlschema!` module loader. This loader enables you to use types that are declared in your schema in DataWeave directly. DataWeave loads your XSD file and translates declarations in your file into DataWeave type directives that you can access in the same way as types from any other DataWeave module. Use the directives to build new types, type-check your variables, match patterns, or declare new functions that use types. DataWeave places no restrictions on how to use these types.

The following example shows how to look for an XSD file in the path `org/weave/myfolder/User.xsd` in your DataWeave script:

```
import * from xschema!org::weave::myfolder::User
```

## Import Syntax

To import the XML schema types, use the following syntax, where:

- **typeToImport**: You can use `*` to import all types defined in the schema, or to import a single type from the schema, for example, `Root`. You can also import XML schema types with a different name, for example, `Root as Person`. This way you can reference the type with that name in the script.
- **pathToXsdSchemaFile**: To specify the path to the schema file, replace the file separators with `::` and remove the `.xsd` extension from the file name. For example, if the path to the schema is `example/schema/Person.xsd`, use `example::schema::Person`.

```
import _typesToImport_ from xschema!_pathToXsdSchemaFile_
```

The following example shows how to import a type:

```
import * from xschema!example::schema::Person
```

## Use Your Types in a DataWeave Script

This example shows how to use types to import XML schema files into your DataWeave script.

XSD File ([User.xsd](#)):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xss:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xss="http://www.w3.org/2001/XMLSchema">
    <xss:element name="Root">
        <xss:complexType>
            <xss:choice>
                <xss:element name="employee" type="employee"/>
                <xss:element name="member" type="member"/>
            </xss:choice>
            <xss:attribute name="id" use="required"/>
        </xss:complexType>
    </xss:element>
    <xss:complexType name="employee">
        <xss:sequence>
            <xss:element name="name" type="xs:string"/>
        </xss:sequence>
    </xss:complexType>

    <xss:complexType name="member">
        <xss:sequence>
            <xss:element name="id" type="xs:string"/>
        </xss:sequence>
    </xss:complexType>
</xss:schema>
```

DataWeave Script:

```
%dw 2.0
import * from xmlschema!org::weave::myfolder::User

{
    Root @(id:"test") :{
        employee: {
            "name": "Mariano"
        }
    } match {
        case is Root -> true
        else -> false
    }
}
```

## From the XML Model to DataWeave Types

Because the XML schema type system translates to DataWeave type directives and the two type models are different, the resulting DataWeave type might not represent exactly the same XML model. The following files illustrate how XSD translates to DataWeave code. These files are

generated dynamically and are not visible, though they are useful for learning how to consume types from generated XSD files.

### Built-in XML Schema Types

DataWeave supports the following XSD types where `xs` is the XMLSchema default namespace:

XSD Type	Dataweave Type
<code>xs:number</code>	Number
<code>xs:int</code>	Number
<code>xs:integer</code>	Number
<code>xs:long</code>	Number
<code>xs:short</code>	Number
<code>xs:byte</code>	Number
<code>xs:double</code>	Number
<code>xs:decimal</code>	Number
<code>xs:unsignedLong</code>	Number
<code>xs:unsignedInt</code>	Number
<code>xs:unsignedShort</code>	Number
<code>xs:unsignedByte</code>	Number
<code>xs:positiveInteger</code>	Number
<code>xs:negativeInteger</code>	Number
<code>xs:nonNegativeInteger</code>	Number
<code>xs:nonPositiveInteger</code>	Number
<code>xs:hexBinary</code>	Binary
<code>xs:base64Binary</code>	Binary
<code>xs:boolean</code>	Boolean
<code>xs:any</code>	String
<code>xs:string</code>	String
<code>xs:normalizedString</code>	String
<code>xs:datetime</code>	DateTime
<code>xs:time</code>	Time
<code>xs:date</code>	LocalDateTime

XSD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsschema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xselement name="Root">
    <xsccomplexType>
      <xsallo>
        <xselement name="string" type="xss:string"/>
        <xselement name="normalizedString" type="xss:normalizedString"/>

        <xselement name="integer" type="xss:integer"/>
        <xselement name="negativeInteger" type="xss:negativeInteger"/>
        <xselement name="int" type="xss:int"/>
        <xselement name="unsignedInt" type="xss:unsignedInt"/>
        <xselement name="unsignedLong" type="xss:unsignedLong"/>
        <xselement name="unsignedShort" type="xss:unsignedShort"/>
        <xselement name="float" type="xss:float"/>
        <xselement name="double" type="xss:double"/>

        <xselement name="date" type="xss:date"/>
        <xselement name="dateTime" type="xss:dateTime"/>
        <xselement name="time" type="xss:time"/>

        <xselement name="boolean" type="xss:boolean"/>

        <xselement name="hexBinary" type="xss:hexBinary"/>
        <xselement name="base64Binary" type="xss:base64Binary"/>
        <xselement name="decimal" type="xss:decimal"/>

      </xsallo>
    </xsccomplexType>
  </xselement>
</xsschema>
```

## Generated DataWeave Module:

```
%dw 2.0
```

```
type RootElementDefinition = {|
    Root: { string: String, normalizedString: String,
    integer: Number, negativeInteger: Number, int: Number, unsignedInt: Number,
    unsignedLong: Number, unsignedShort: Number, float: Number, double: Number, date:
    LocalDateTime, dateTime: DateTime, time: Time, boolean: Boolean, hexBinary: Binary,
    base64Binary: Binary, decimal: Number } |}
```

```
type Root = RootElementDefinition
```

## Root Elements Declaration

Each root element (`<xs:element>`) declared in your XSD file generates the following:

- A DataWeave `type` directive with the element name followed by the `ElementDefinition` extension
- A `type` directive named `Root` as a `Union` type that consists of all the root elements in the XSD.

The following example shows how to declare root elements:

XSD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="Root">
        <xs:complexType>
            <xs:sequence>
                <xs:element type="xs:string" name="color" default="Red"/>
                <xs:element type="xs:integer" name="age" default="12"/>
            </xs:sequence>
            <xs:attribute type="xs:string" name="type" default="ALBA"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="AnotherRoot">
        <xs:complexType>
            <xs:sequence>
                <xs:element type="xs:string" name="name" default="Red"/>
            </xs:sequence>
            <xs:attribute type="xs:string" name="type"/>
        </xs:complexType>
    </xs:element>
</xsschema>
```

## *Generated DataWeave Module:*

```
%dw 2.0

type AnotherRootElementDefinition = {|
    AnotherRoot @("type"? String): {- name: String}
    -} |}

type RootElementDefinition = {|
    Root @("type"? String): {- color: String, age: Number}
    -} |}

type Root = AnotherRootElementDefinition | RootElementDefinition
```

## Complex Types

Named complex types declared as top-level elements on the schema which are referenced in your schema or are available for reuse on other structures generate their own type directive. These type directive names include **complexType** name followed by the **Definition** extension.

The following example shows how to declare complex types:

XSD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xss:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xss="http://www.w3.org/2001/XMLSchema">
    <xss:element name="Root">
        <xss:complexType>
            <xss:choice>
                <xss:element name="employee" type="employee"/>
                <xss:element name="member" type="member"/>
            </xss:choice>
            <xss:attribute name="id" default="foo"/>
        </xss:complexType>
    </xss:element>
    <xss:complexType name="employee">
        <xss:sequence>
            <xss:element name="name" type="xs:string"/>
        </xss:sequence>
    </xss:complexType>

    <xss:complexType name="member">
        <xss:sequence>
            <xss:element name="id" type="xs:string"/>
        </xss:sequence>
    </xss:complexType>
</xss:schema>
```

*Generated DataWeave Module:*

```
%dw 2.0

type EmployeeDefinition = { employee: {- name: String -} }

type MemberDefinition = { member: {- id: String -} }

type RootElementDefinition = {|| Root @id?: String): {|| employee:
EmployeeDefinition.employee ||} | {|| member: MemberDefinition.member ||} ||}

type Root = RootElementDefinition
```

**Imports/Includes**

The use of imports or includes in an XSD schema file results in a DataWeave module which contains an **import** directive that uses the XML schema module loader to point to the schema file located at the specified **schemaLocation**. This location must be declared relative to the main schema. Notice how the following examples use [Selecting Types](#):

XSD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:common="http://common.com/COMMON">

    <xsd:import schemaLocation="common.xsd" namespace="http://common.com/COMMON" />

    <xsd:element name="Root" type="common:RequestHeader"/>

</xsd:schema>
```

### *Generated DataWeave Module:*

```
%dw 2.0

ns ns0 http://common.com/COMMON

import
xmlschema!org::mule::weave::v2::module::xmlschema::moduleloader::imports::common

type RootElementDefinition = {|
    Root:
    common::RequestHeaderDefinition.ns0#RequestHeader
|}

type Root = RootElementDefinition
```

### **Choice**

You can map this XML schema structure to [UnionType](#) in DataWeave:

XSD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xss:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xss:element name="Root">
        <xss:complexType>
            <xss:choice>
                <xss:element name="employee" type="employee"/>
                <xss:element name="member" type="member"/>
            </xss:choice>
        </xss:complexType>
    </xss:element>
    <xss:complexType name="employee">
        <xss:sequence>
            <xss:element name="name" type="xs:string"/>
        </xss:sequence>
    </xss:complexType>
    <xss:complexType name="member">
        <xss:sequence>
            <xss:element name="id" type="xs:string"/>
        </xss:sequence>
    </xss:complexType>
</xss:schema>
```

*Generated DataWeave Module:*

```
%dw 2.0

type EmployeeDefinition = { employee: {- name: String -} }

type MemberDefinition = { member: {- id: String -} }

type RootElementDefinition = {|| Root: {|| employee: EmployeeDefinition.employee ||} || member: MemberDefinition.member || ||}

type Root = RootElementDefinition
```

## Namespaces

Namespaces in your XML schema file translate directly to DataWeave type directives.

The following example shows how to set the specific **targetNamespace** attribute where your elements exist. This namespace, which translates to DataWeave, is required to select and navigate these types.

XSD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xss:schema attributeFormDefault="unqualified"
targetNamespace="http://NamespaceTest.com/CommonTypes" elementFormDefault="qualified"
xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:element name="Root">
    <xss:complexType>
      <xss:all>
        <xss:element name="string" type="xss:string"/>
        <xss:element name="normalizedString" type="xss:normalizedString"/>
        <xss:element name="integer" type="xss:integer"/>
        <xss:element name="negativeInteger" type="xss:negativeInteger"/>
        <xss:element name="int" type="xss:int"/>
        <xss:element name="unsignedInt" type="xss:unsignedInt"/>
        <xss:element name="unsignedLong" type="xss:unsignedLong"/>
        <xss:element name="unsignedShort" type="xss:unsignedShort"/>
        <xss:element name="float" type="xss:float"/>
        <xss:element name="double" type="xss:double"/>
        <xss:element name="date" type="xss:date"/>
        <xss:element name="dateTime" type="xss:dateTime"/>
        <xss:element name="time" type="xss:time"/>
        <xss:element name="boolean" type="xss:boolean"/>
        <xss:element name="hexBinary" type="xss:hexBinary"/>
        <xss:element name="base64Binary" type="xss:base64Binary"/>
        <xss:element name="decimal" type="xss:decimal"/>
      </xss:all>
    </xss:complexType>
  </xss:element>
</xss:schema>
```

*Generated DataWeave Module:*

```
%dw 2.0

ns ns0 http://NamespaceTest.com/CommonTypes

type RootElementDefinition = {|
  ns0#Root: { ns0#string: String, ns0#normalizedString: String,
  ns0#integer: Number, ns0#negativeInteger: Number, ns0#int: Number,
  ns0#unsignedInt: Number, ns0#unsignedLong: Number, ns0#unsignedShort: Number,
  ns0#float: Number, ns0#double: Number, ns0#date: LocalDateTime, ns0#dateTime: DateTime,
  ns0#time: Time, ns0#boolean: Boolean, ns0#hexBinary: Binary,
  ns0#base64Binary: Binary, ns0#decimal: Number } |}

type Root = RootElementDefinition
```

## Reusing Types from Java Classes

Use the import directive with the `java!` prefix when loading a Java class. The prefix `java!` tells DataWeave to use the Java module loader. After the module loads, you can call constructors, variables, and functions (see [Call Java Methods \(Mule\)](#)). The class type is also available. DataWeave exposes the type of the Java class with the type name `Class`.

Java classes are mapped to the DataWeave types listed in [Java Value Mapping](#). If a Java class is not specified in the table, DataWeave treats the class as `JavaBean` and maps it to a DataWeave `Object` type. The keys in that object match the names of the properties in the Java class and their values correspond to the DataWeave type that matches the property class. The following example shows how DataWeave takes all properties from the Java getters:

Product Class:

```
package org.mycompany;

public class Product {
    private String name;
    private int price;

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public int getPrice() {return price;}
    public void setPrice(int price) {this.price = price;}
}
```

Loading the type for this Java class has the same effect as declaring this DataWeave type:

```
type Class = { name?: String | Null, price?: Number | Null }
```

The following example uses the type `Product::Class` as any other type in DataWeave. Creating a DataWeave object by calling the function `Product::new` (which calls the constructor of the Java class) also matches the type `Product::Class`. Notice that the import directive lacks `from` and instead uses the type `Class` prefix `Product::`. The prefix enables more declarative use of the type and avoids collisions with any other imported `Class` types:

*DataWeave Script:*

```
%dw 2.0
output json
import java!org::mycompany::Product

var aBook: Product::Class = {name: "Learn DW", price: 123}
fun description(p: Product::Class) = "The product: $(p.name), costs: $(p.price)"

type Order = {
    product: Product::Class,
    date: LocalDateTime
}

---
{

    a: Product::new("DW lang", 321) is Product::Class,
    b: description(aBook),
    c: {product: aBook, date: |2022-12-18T14:00:00|} is Order
}
```

*Output:*

```
{
    a: true,
    b: "The product: Learn DW, costs: 123",
    c: true
}
```

## Value Constructs for Types

DataWeave represents data using values, each of which has a data type associated with it. There are many types, such as strings, arrays, Booleans, numbers, objects, dates, times, and others. Each type supports several ways of creating its values. This topic explores many of the ways you can create them. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

The types that DataWeave provide are bundled into modules that also contain the related functions.

Types in the `dw::Core` module (Core types) are available without having to import the Core module. Other modules need to be imported for their functions and types to be available.

It is important for you to know the values that are possible for each DataWeave type, how to create those values, and the common patterns used:

- Conditional elements that use `if` expressions for [arrays](#) and [objects](#).
- [String Interpolation](#)
- [Date Decomposition](#) for accessing different parts of a date

- [Date and Time Formats](#)
- [Dynamic Keys](#) and [Dynamic Elements](#), which allow you to access parts of an object dynamically
- Use of [Regex \(dw::Core Type\)](#) expressions

## Array (dw::Core Type)

An array can hold elements of any supported type. Here is an example of an array:

*Example: DataWeave Array*

```
%dw 2.0
output application/json
var x = "words"
---
[ "My", "three", x ]
```

## Conditional Elements

Arrays can define elements that appear (or not) based on a condition.

Conditional elements take the form `(value) if condition`, for example:

*Example: if Condition*

```
%dw 2.0
output application/json
---
[(1) if true, (2) if false]
```

*Output*

```
[1]
```

## Boolean (dw::Core Type)

A [Boolean](#) is defined by the keywords `true` and `false`.

## CData (dw::Core Type)

XML defines a custom type named [CData](#), which inherits from and extends [String](#). It 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.

## *Transform*

```
%dw 2.0
output application/xml encoding="UTF-8"
---
{
  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>
```

# Date and Time (dw::Core Types)

Dates in DataWeave follow the [ISO-8601 standard](#) and literals are defined between | characters.

The language has the following native date types:

- **Date**
- **DateTime**
- **LocalDateTime**
- **LocalTime**
- **Period**
- **Time**
- **TimeZone**

## **Date**

A **Date** represented by **Year**, **Month**, and **Day**, specified as |uuuu-MM-dd|. The **Date** type has no time component.

### *Example*

```
|2003-10-01|
```

## **DateTime**

A Date and Time within a TimeZone. It is the conjunction of **Date** + **Time** + **TimeZone**.

*Example*

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

## LocalDateTime

A **DateTime** in the current **TimeZone**.

*Example*

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

## LocalTime

A **Time** in the current **TimeZone**.

## Period

**Period** represents an amount of time. The type takes the following form:

- **P[n]Y[n]M[n]DT[n]H[n]M[n]S**
- **P<date>T<time>**

Where the [n] is replaced by the value for each of the date and time elements that follow the [n].

**Period** has two subcategories:

- **DatePeriod**

A date-based amount of time, for example, 1 year, 2 months, 3 days. Because these expressions are date-based, they cannot be converted to milliseconds. For example, a month does not specify whether it is a month of 31 days or not, nor does a year specify whether it is a leap year. The data expresses just an amount of date. In DataWeave, the syntax is **|P1Y|**.

- **Duration**

A time-based amount of time, for example, 1 second, 2 hours, or 2 days, each of which can be represented in seconds or milliseconds. In DataWeave, the syntax is **|PT1H|**.

**P** is the duration designator placed at the start of the duration representation.

- **Y** is the year designator (e.g. **|P1Y|**)
- **M** is the month designator (e.g. **|P1M|**)
- **D** is the day designator (e.g. **|P1D|**)

**T** is the time designator that precedes the time components of the representation.

- **H** is the hour designator (e.g. **|PT1H|**)
- **M** is the minute designator (e.g. **|PT1M|**)

- **S** is the second designator (e.g. `|PT1S|`)

**DatePeriod** is useful for date manipulation, such as addition or subtraction.

The following example shows how to subtract one year from a date using **DatePeriod**. The return type is a new **Date** value:

#### Source

```
output application/dw
---
|2003-10-01| - |P1Y|
```

#### Output

```
|2002-10-01|
```

**Duration** is useful as a result of a **Date** to **Date** subtraction so the amount of time between those two dates can be inferred.

The following example shows how subtracting one **Date** value from another returns a **Duration** value:

#### Source

```
output application/dw
---
|2003-11-01| - |2003-10-01|
```

#### Output

```
|PT744H|
```

### Period Coercion

Because a **DatePeriod** value is date-based, it can be coerced to **Number** date-based units, such as **years** or **months**.

The following example shows how to coerce a **DatePeriod** value to a **Number** value by using different units:

#### Source

```
output application/json
---
{
  years: |P1Y12M| as Number {unit: "years"},
  months: |P8Y12M| as Number {unit: "months"}
}
```

## *Output*

```
{  
  "years": 2,  
  "months": 108  
}
```

## **Duration Coercion**

Because **Duration** is time-based, this type can be coerced to a **Number** with different time-based units, such as **nanos**, **milliseconds**, **seconds**, **hours**, and **days**.

The following example shows how to coerce a **Duration** to a **Number** by using different units:

## *Source*

```
output application/json  
var period = (|2010-12-10T12:10:12| - |2010-09-09T10:02:10|)  
---  
{  
  nanos: period as Number {unit: "nanos"},  
  millis: period as Number {unit: "milliseconds"},  
  seconds: period as Number {unit: "seconds"},  
  hours: period as Number {unit: "hours"},  
  days: period as Number {unit: "days"}  
}
```

## *Output*

```
{  
  "nanos": 7956482000000000,  
  "millis": 7956482000,  
  "seconds": 7956482,  
  "hours": 2210,  
  "days": 92  
}
```

## **Deconstruct**

A **Duration** can be deconstructed into **hours**, **minutes** and **secs**.

The following example shows how to deconstruct a **Duration** value:

## Source

```
%dw 2.0
output application/json
var period = (|2010-12-10T12:10:12| - |2010-12-10T10:02:10|)
---
{
    hours: period.hours,
    minutes: period.minutes,
    secs: period.secs,
}
```

## Output

```
{
    "hours": 2,
    "minutes": 8,
    "secs": 2
}
```

## Time

A time in a specific **TimeZone**, specified as **|HH:mm:ss.SSS|**.

### Example

```
|23:59:56|
```

## TimeZone

The **Time** relative to Greenwich Mean Time (GMT). A **TimeZone** must include a **+** or a **-**. For example, **|03:00|** is a time, while **|+03:00|** is a **TimeZone**.

### Example

```
|-08:00|
```

## Date Decomposition

To access the different parts of the date, special selectors must be used.

## Transform

```
%dw 2.0
output application/json
var myDate = |2003-10-01T23:57:59.700-03:00|
---
{
    year: myDate.year,
    month: myDate.month,
    day: myDate.day,
    hour: myDate.hour,
    minutes: myDate.minutes,
    seconds: myDate.seconds,
    milliseconds: myDate.milliseconds,
    nanoseconds: myDate.nanoseconds,
    quarter: myDate.quarter,
    dayOfWeek: myDate.dayOfWeek,
    dayOfYear: myDate.dayOfYear,
    offsetSeconds: myDate.offsetSeconds
}
```

## Output

```
{
    "year": 2003,
    "month": 10,
    "day": 1,
    "hour": 23,
    "minutes": 57,
    "seconds": 59,
    "milliseconds": 700,
    "nanoseconds": 700000000,
    "quarter": 4,
    "dayOfWeek": 3,
    "dayOfYear": 274,
    "offsetSeconds": -10800
}
```

## Date and Time Formats

To enable you to format dates and times, DataWeave supports formatting characters, such as the **u** (for the year), **M**, and **d** in the date format **uuuu-MM-dd**. These characters are based on the Java 8 **java.time.format** package.

For examples, see [Format Dates and Times](#).

## Timezone IDs

In addition to accepting time zone values, such as **-07:00**, DataWeave also accepts IDs such as **America/Buenos\_Aires**, **America/Los\_Angeles**, **Asia/Tokyo**, and **GMT**.

For examples and a list of supported IDs, see [Time Zone IDs](#) and [Change a Time Zone](#).

## Enum (dw::Core Type)

This type is based on the [Enum Java class](#). It must always be used with the `class` property, specifying the full Java class name of the class, as shown in this example.

*Transform*

```
%dw 2.0
output application/java
---
"Male" as Enum {class: "com.acme.GenderEnum"}
```

## Iterator (dw::Core Type)

The [Iterator](#) type is based on the [Iterator Java class](#), that iterates through arrays. [Iterator](#) contains a collection and includes methods to iterate through and filter it.

Note that like the Java class, the iterator is designed to be consumed only once. For example, if a logger consumes the value, the value is no longer readable by subsequent elements in the flow.

## Null (dw::Core Type)

DataWeave defines a [Null](#) data type for `null` values. It is important to note that `null` is the value, while [Null](#) is the type. DataWeave functions do not recognize [Null](#) as a `null` value.

You can determine whether a DataWeave function accepts `null` values by looking in the DataWeave reference documentation for a function signature that accepts a Null type. For example, the [flatten\(@StreamCapable value: Null\): Null](#) function has two function signatures, one of which is [flatten\(Null\): Null](#). The signature indicates that `flatten` can accept `null` values. The `++` (concatenate) function, which lacks a signature for handling value so the [Null](#) type, does not accept `null`.

- `flatten([[1],2,[null],null])` returns `[1,2,null,null]`
- `"a" ++ null` returns the error `Unable to call '+' with ('String', 'Null')`.

## Number (dw::Core Type)

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 does not compromise precision.

DataWeave provides a mechanism for formatting numeric values and for coercing dates and strings to numbers. The language also provides operators that act on numeric values and includes many functions that take numeric values as arguments.

- [dataweave-types-coercion:::pdf](#) shows how to format a number.

- [mapObject](#) provides an example that coerces a string to a number and formats the number.
- [Period Coercion](#) and [Duration Coercion](#) include date coercion examples.
- [Operators](#) describes operators that act on numeric values.
- [avg](#) is one of many DataWeave functions that accepts a numeric value. The [Number](#) type appears in its function signature, `avg(Array<Number>): Number`, which indicates that the function accepts an array of numeric values and returns a numeric value.
- The [Numbers](#) module provides helper functions that work on numbers.

## Object (dw::Core Type)

Represents any object as a collection of [Key:value](#) pairs, where a [Key](#) is composed of a [Name](#) and [Attributes](#).

The [Name](#) type is composed of a [String](#) as the local name and the [Namespace](#). [Attributes](#) is composed of an Array of [Name:value](#) pairs. Note that the [Key](#) is not a [String](#), so it is not possible to compare keys. However, you can get the local name by performing an `as String` type coercion on any value of type [Key](#).

*Example*

```
%dw 2.0
output application/json
---
{
  name: "Annie"
}
```

## Single Value Objects

If an object has only one [key:value](#) pair, the enclosing curly braces `{ }` are not required:

*Example*

```
%dw 2.0
output application/json
---
name: "Annie"
```

## Conditional Elements

Objects can define conditional key-value pairs based on a conditional expression. Conditional elements have the form `(key:value) if condition`.

```
%dw 2.0
output application/xml encoding="UTF-8"
---
file: {
  name: "transform",
  (extension: "zip") if 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 Keys

To specify a key through an expression, you need to wrap the expression in parentheses.

### *Transform*

```
%dw 2.0
output application/json
var dynamicKey = "language"
---
{
  (dynamicKey): "Data Weave"
}
```

### *Output*

```
{
  "language": "Data Weave"
}
```

## Dynamic Elements

Dynamic elements allow you to add the result of an expression as **key:value** pairs of an object. That expression must be either an **object** or an **array of objects**.

### Transform

```
%dw 2.0
output application/json
var x = [
  {b: "b"},
  {c: "c", d: "d"}
]
var y = {e: "e"}
---
{
  a: "a",
  (x),
  (y)
}
```

The *evaluation parentheses* around the variables in the body of the script above ((**x**) and (**y**)) enable the object constructor curly braces ({ }) surrounding the body of the script to act on the values of the variables. Specifically, the object constructor curly braces extract the key-value pairs from the values of **x** and **y** and convert them into a collection of key-value pairs within an object. Without the parentheses (for example, if you use **x** instead of (**x**)), the script returns an error.

The object constructor curly braces require a data construct that contains one or more key-value pairs, specifically an object, such as { "a": "one", "b": "two"}, or an array of objects, such as [{"a": "one"}, {"b": "two"}] so that it can produce a valid object. Because the first element within the object constructor curly braces is the key-value pair **a: "a"**, the object constructor curly braces leave the pair as the first element in the output object, without any modifications. However, the object constructor curly braces extract the outer-level key-value pairs in the evaluated expressions (**x**) and (**y**) and append those key-value pairs to the output object.

### Output

```
{
  "a": "a",
  "b": "b",
  "c": "c",
  "d": "d",
  "e": "e"
}
```

## Conditional XML Attributes

You might want your output to only include certain XML attributes based on a condition. Conditional elements have the form **(key:value) if condition**

## Transform

```
%dw 2.0
output application/xml
---
{
  name @(
    (company: "Acme") if false,
    (transform: "Anything") if true
  ): "DataWeave"
}
```

## Output

```
<?xml version='1.0' encoding='US-ASCII'?>
<name transform="Anything">DataWeave</name>
```

## Dynamic XML Attributes

You might want to include a changing set of key:value pairs in a specific location as XML attributes.

## Input

```
{
  "company": "Mule",
  "product": "DataWeave"
}
```

## Transform

```
%dw 2.0
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>
```

## Regex (dw::Core Type)

Regular Expressions are defined between `/`. For example `/\d+/` 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.

## String (dw::Core Type)

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

```
{  
    doubleQuoted: "Hello",  
    singleQuoted: 'Hello',  
}
```

### Escaping Special Characters

Use the backslash (\) to escape special characters in an input string:

- `$`: You need to escape any use of `$` in a string. Otherwise, DataWeave treats the `$` as an unnamed parameter for a function and returns the error **Unable to resolve reference of \$..**
- `"`: For a string that is surrounded by double quotes, you need to escape any double quote that is part of the string, for example, `"a\"bcdef"`. Here, the second double-quote is part of the string that starts with an `a` and ends with `f`.
- `'`: For a string that is surrounded by single quotes, you need to escape any single quote that is part of the string: for example, `'abcd\'e"f'`. In this example, the second single quote is part of the string that starts with an `a` and ends with `f`. Notice that you do not need to escape the double quote in this case.
- `\``: For a string that is surrounded by backticks, you must escape any backtick that is part of the string: for example, ``abc\`def``.
- `\`: Because the backslash is the character you use to escape other special characters, you need to escape it with a separate backslash to use it in a string, for example, `\\\`.
- `\n`: For inserting a new line.
- `\t`: For inserting a tab.
- `\u`: For inserting a unicode character, such as `\u25c4`.

### String Interpolation

String interpolation allows you to embed variables or expressions directly in a string. The expression need to be enclosed `($( <expression> ))`, and it should always return String type or something that is coercible to String.

```
%dw 2.0  
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"  
}
```

## **TryResult (dw::Runtime Type)**

Evaluates the delegate and returns an object with the result or an error message. See the [try](#) example. A successful **TryResult** contains the **result** field and a **success** value of **true**. An unsuccessful **TryResult** contains the **error** field and a **success** value of **false**.

### *Definition*

```
{  
  success: Boolean,  
  result?: T,  
  error?: {  
    kind: String,  
    message: String,  
    stack?: Array<String>,  
    location?: String  
  }  
}
```

## **URI Types (dw::core::URL)**

Functions in the URI function module can return a URI data type.

### *Definition:*

```
{  
  isValid: Boolean,  
  host?: String,  
  authority?: String,  
  fragment?: String,  
  path?: String,  
  port?: Number,  
  query?: String,  
  scheme?: String,  
  user?: String,  
  isAbsolute?: Boolean,  
  isOpaque?: Boolean  
}
```

The URI type consists of the following fields:

- **isValid**: Boolean that indicates whether the URI is valid. Invalid characters include <, >, and

blank spaces.

- **host**: String representing the host name (for example, `my.company.com` from `http://my.company.com:8080/hello`)
- **authority**: String representing the authority, which includes the host and port (for example, `my.company.com:8080` from `http://my.company.com:8080/hello`). Returns the port only if it is explicitly specified in the URI.
- **fragment**: String representing the subordinate resource after the `#` in the URI (for example, `footer` in the URI "`https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#footer`").
- **path**: String representing a path following the host or authority (for example, `/hello` in `http://my.company.com:8080/hello`).
- **port**: Number representing an explicit port in the URI (for example, `8080` in `http://my.company.com:8080/hello`). If no port is specified (for example, `http://my.company.com/hello`), the value of the `port` field is `null`.
- **query**: String identifying a query portion of a URI (for example, `field=value` in '`http://my.company.com:1234/hello/?field=value`').
- **scheme**: String identifying the URI scheme, such as `https` or `http`, which appears before a colon (`:`).
- **user**: String representing user information in a URI (for example, the `myname` in `http://myname@www.mycompany.com`).
- **isAbsolute**: Boolean value of `true` if the URI contains a scheme, `false` if the URI does not (for example, a relative URI such as `/path/to/somewhere`).
- **isOpaque**: Boolean value of `true` if the URI lacks a scheme followed by forward slashes (`/`). `mailto:somebody@somewhere.com` returns `true`.

The following DataWeave script uses the `parseURI` function to return and access values to fields from a URI type. Note that fields with `null` values (for example, the URI defined by `uriDataTypeEx`) are not returned unless you explicitly select them.

#### *DataWeave Script*

```
%dw 2.0
import * from dw::core::URL
var uriDataTypeEx = "https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#footer"
var queryStringsEx = 'http://my.company.com:1234/hello/?field=value'
output application/json
---
{
    'myUriDataTypeEx': parseURI(uriDataTypeEx),
    // queryStringsEx has a query string:
    'myQueryStringsEx': parseURI(queryStringsEx).query,
    // uriDataTypeEx lacks a query string:
    'myQueryStringNullEx': parseURI(uriDataTypeEx).query,
    // The URI includes the port number:
    'myAuthorityEx': parseURI('http://localhost:8080/test').authority
}
```

## Output

```
{  
    "myUriDataTypeEx": {  
        "isValid": true,  
        "raw": "https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#footer",  
        "host": "en.wikipedia.org",  
        "authority": "en.wikipedia.org",  
        "fragment": "footer",  
        "path": "/wiki/Uniform_Resource_Identifier",  
        "scheme": "https",  
        "isAbsolute": true,  
        "isOpaque": false  
    },  
    "myQueryStringsEx": "field=value",  
    "myQueryStringNullEx": null,  
    "myAuthorityEx": "localhost:8080"  
}
```

## DataWeave Type References

You can find more data type documentation in the reference pages for the DataWeave function modules:

- [Core Types](#)
- [Coercions Types](#)
- [DataFormat Types](#)
- [Dates Types](#)
- [Diff Types \(dw::core::Diff\)](#)
- [Mule Types](#)
- [Multipart Types](#)
- [Runtime Types](#)
- [Timer Types \(dw::core::Timer\)](#)
- [Tree Types](#)
- [Types Types](#)
- [URL Types](#)
- [Values Types](#)

## See Also

- [Scripts](#)
- [Anypoint Exchange \(List of Projects that use DataWeave\)](#)

# Type Coercion with DataWeave

In DataWeave, types can be coerced from one type to other using the `as` operator. Type coercion takes place at runtime. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

Note that when you provide an operator with properties that do not match the expected types, DataWeave automatically attempts to coerce the provided property to the required type.

## Defining DataWeave Types For Type Coercion

The DataWeave example defines the type `Currency` using the `String` type, formats the value with the Java DecimalFormat pattern (`##`), and then uses `as` to coerce the `price` values to the `Currency` type.

### *Input*

```
<items>
  <item>
    <price>22.30</price>
  </item>
  <item>
    <price>20.31</price>
  </item>
</items>
```

### *DataWeave*

```
%dw 2.0
output application/json
type Currency = String { format: "\$\#,###.00"}
---
books: payload.items.*item map
  book:
    price: $.price as Currency
```

## Output

```
{  
  "books": [  
    {  
      "book": {  
        "price": "22.30"  
      }  
    },  
    {  
      "book": {  
        "price": "20.31"  
      }  
    }  
  ]  
}
```

## Type Coercion Table

This table shows the possible combinations and the properties from the schema that are used in the transformation.

Source	Target	Property
Range	Array	
Number	Binary	
String	Binary	
String	Boolean	
Number	DateTime	unit
LocalDateTime	DateTime	
String	DateTime	format / locale
DateTime	LocalDate	
LocalDateTime	LocalDate	
String	LocalDate	format / locale
DateTime	LocalDateTime	
String	LocalDateTime	format / locale
DateTime	LocalTime	
LocalDateTime	LocalTime	
Time	LocalTime	
String	LocalTime	format / locale
DateTime	Number	unit
String	Number	format / locale
String	Period	
String	Regex	

Source	Target	Property
DateTime	String	format / locale
LocalDateTime	String	format / locale
LocalTime	String	format / locale
LocalDate	String	format / locale
Time	String	format / locale
Period	String	
TimeZone	String	
Number	String	format / locale
Boolean	String	
Range	String	Returns a string with all the values of the range using , as the separator
Type	String	
DateTime	Time	
LocalDateTime	Time	
LocalTime	Time	
String	Time	format
DateTime	TimeZone	
Time	TimeZone`	
String	TimeZone	

## Properties for Type Coercion

Property	Description
class	Accepts Java classes for Object types.
format	Accepts Java <code>DecimalFormat</code> patterns to format numbers and dates.
locale	Accepts Java locales. A Java <code>Locale</code> object represents a region (geographical, political, or cultural).
mode	<p>Parsing mode for date and time values. Valid values: <code>SMART</code>, <code>STRICT</code>, <code>LENIENT</code></p> <p>Note that <code>LENIENT</code> and <code>SMART</code> autocorrect invalid dates in different ways, but <code>STRICT</code> returns an error on the date. See <a href="#">examples</a> that use <code>mode</code>.</p>
unit	Value can be <code>milliseconds</code> or <code>seconds</code> . These are used for Number to DateTime conversions.

The following examples show uses of the `mode` property. Notice that `LENIENT` and `SMART` return valid dates when they receive an invalid date such as `02/31/2020`, but the resulting dates differ. `STRICT` returns an error on an invalid date. The examples use `uuuu` to represent the year instead of `yyyy`.

*DataWeave script:*

```
%dw 2.0
output application/json
---
{
    examples : {

        badDateWithLenient: '02/31/2020' as Date {mode: "LENIENT", format: 'MM/dd/uuuu'},
        badDateWithSmart: '02/31/2020' as Date {mode: "SMART", format: 'MM/dd/uuuu'}
    }
}
```

*Output:*

```
%dw 2.0
output application/json
---
{
    "examples": {
        "badDateWithLenient": "03/02/2020",
        "badDateWithSmart": "02/29/2020"
    }
}
```

Using `STRICT` on an invalid date returns an error. The following example is an error returned by `badDateWithStrict: '02/31/2020' as Date {mode: "STRICT", format: 'MM/dd/uuuu'}`:

```
"Cannot coerce String (02/31/2020) to Date, caused by: Text '02/31/2020' could not be
parsed: Invalid date 'FEBRUARY 31'

9|   badDateWithSmart:
  '02/31/2020' as Date {mode: "STRICT", format: 'MM/dd/uuuu'}
  ^^^^^^^^^^^^^^^^^^
```

The following example converts two strings into a date format and concatenates the result. The first string represents the date, and the second string represents the time. The transformation uses `as` to coerce the first string to `LocalDateTime`, and then to a `String` with the specified format. The transformation also uses `++` to concatenate the result.

*DataWeave script:*

```
%dw 2.0
output application/json
var s1= 20201228 // (uuuuMMdd),
var s2= 1608 //(HHMM)
---
(s1 ++ s2)
as LocalDateTime {format:"uuuuMMddHHmm"}
as String {format:"MM-dd-uuuu HH:mm:ss"}
```

*Output:*

```
"12-28-2020 16:08:00"
```

The following example also converts a string into a date format. First, the transformation uses `as` to coerce the string into a `LocalDateTime` type, and then it coerces the result into a `String` type with the specified output format.

*DataWeave script:*

```
%dw 2.0
output application/json
---
"8/30/2020 4:00:13 PM"
    as LocalDateTime {format: "M/dd/uuuu h:mm:ss a"}
    as String {format: "MM/dd/uuuu"}
```

*Output:*

```
"08/30/2020"
```

The following example converts a string to a number with decimal representation. First, you transform the string value `"22"` to `Number` and then to `String` again because `format` adds zeros when converting from a number to a string. If the input data is the number `22`, it is not necessary to perform the first `Number` conversion. The latter `Number` conversion makes the output a number with the decimal representation.

*DataWeave script:*

```
%dw 2.0
output application/json
---
{
    data: "22" as Number as String {format: ".00"} as Number
}
```

*Output:*

```
{  
    "data": 22.00  
}
```

The following example uses the `locale` property to format number and date values. First, you coerce a number value into `String` type to the specified output format that uses the Java DecimalFormat pattern (`##`) and also the `locale` property `en` (English) or `es` (Spanish). The `locale: "en"` property, formats the output number decimal representation using a `,`, while the `locale: "es"` property, formats the output using a `,`.

Then, you coerce a date value into `Date` type and then to `String` type with the output format `dd-MMM-yy` and the `locale` property `en` (English), or `es` (Spanish). The `locale: "en"` property, formats the month `MMM` in English, while the `locale: "es"` property formats the month in Spanish.

*DataWeave script:*

```
%dw 2.0  
output application/json  
---  
{  
    enNumber: 12.3 as String {format: "##.##", locale: "en"},  
    esNumber: 12.3 as String {format: "##.##", locale: "es"},  
    esDate: "2020-12-31" as Date as String {format: "dd-MMM-yy", locale: "es"},  
    enDate: "2020-12-31" as Date as String {format: "dd-MMM-yy", locale: "en"}  
}
```

*Output:*

```
{  
    "enNumber": "12.3",  
    "esNumber": "12,3",  
    "esDate": "31-dic.-20",  
    "enDate": "31-Dec-20"  
}
```

## See Also

- [Format Dates and Times](#)
- [Supported Data Formats](#)

## Variables

DataWeave is a functional programming language in which variables behave just like functions. DataWeave uses eager evaluation for variables and function parameters. In addition, DataWeave variables are immutable. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other

Mule versions, you can use the version selector in the DataWeave table of contents.

A variable is assigned a value, which is either a constant (such as `var msg = "hello"`) or a lambda expression such as `( ) -> "hello"`.



DataWeave 1.0 variables are described in the DataWeave 1.2 documentation for Mule 3.

## DataWeave Variable Assignment

Here are some examples of DataWeave 2.0 variable assignments:

*Example: Variable Assignment in a DataWeave Script*

```
%dw 2.0
output application/json
var msg = "Hello"

var msg2 = (x = "ignore") -> "hello"

var toUpper = (aString) -> upper(aString)

var combined = (function, msg="universe") -> function(msg ++ " WORLD")
---
[
    msg: msg,
    msg2: msg2(),
    toUpper: toUpper(msg),
    combined: combined(toUpper, "hello"),
    combined2: combined(((x) -> lower(x) ++ " Today"), msg)
]
```

The previous script results in this output:

## Example: Output

```
[  
  {  
    "msg": "Hello"  
  },  
  {  
    "msg2": "hello"  
  },  
  {  
    "toUpperCase": "HELLO"  
  },  
  {  
    "combined": "HELLO WORLD"  
  },  
  {  
    "combined2": "hello world Today"  
  }  
]
```

In these examples:

- The `msg` var is assigned to a constant.
- `msg2` is assigned to an expression that maps to a constant, so `msg2()` behaves the same way as `msg`, without the `()`.
- The `toUpperCase` variable maps an input parameter to the DataWeave `upper` function. So it is used like a more traditional function in other languages as `toUpperCase(aString)`.
- The `combined` variable accepts a function as an input parameter, then applies that function name to the second `msg` argument. The second `msg` argument is optional because a default value `universe` is specified.

The result of the expression `combined(toUpper, " world")` is to apply the `toUpper` function defined earlier in the header to the `msg` parameter value `world`. The supplied second argument `world` overrides the `msg="universe"` default value, so the result is `HELLO WORLD`. The expression `combined(toUpper)`, which omits the second argument, uses the default `msg` value to return `"combined": "UNIVERSE WORLD"`.

The `combined2: combined(x) → lower(x) ++ " Today", msg`) defines the lambda expression that you can reuse in different contexts. In the example, `combined2` key defines the expression “in place” at the time the function is called.

Note that these types of “in place” expressions are called closures in languages like JavaScript.

DataWeave includes syntactic sugar to make it easier to access lambda expressions that are assigned to a variable as functions. To do this, you replace the `var` directive with the `fun` directive, and replace the arrow `→` in the lambda expression with an equal sign (`=`). You also move the lambda expression arguments next to the function name so the syntax looks like a function declaration in other procedural programming languages. This enables you to use variables as if they were

functions. So the previous example can be equivalently written as:

```
%dw 2.0
output application/json
var msg = "Hello"

var toUpper = (aString) -> upper(aString)
var toLower = (aString) -> lower(aString)

fun msg2(optParm = "ignore") = "hello"

fun toTitle(text: String) = toLower(text[0]) ++ toUpper(text[1 to -1])

fun combined(function, msg="universe") = function(msg ++ " world")
---
[
  msg: msg,
  msg2: msg2(),
  toUpper: toTitle(msg),
  combined: combined(toUpper, msg),
  combined2: combined((x) -> lower(x) ++ " today", msg)
]
```

This example produces the same result:

```
[
  {
    "msg": "Hello"
  },
  {
    "msg2": "hello"
  },
  {
    "toUpper": "hELLO"
  },
  {
    "combined": "HELLO WORLD"
  },
  {
    "combined2": "hello world today"
  }
]
```

The important distinction in DataWeave is that functions are variables, and every function is just a syntactical renaming of the underlying `var` syntax, which allows you to pass function names or lambda expressions as arguments to other functions. The `fun` syntax allows you to access the powerful functional programming aspects of DataWeave while also being able to write simpler expressions as function calls you might be more familiar with.

Also notice that DataWeave variables (and functions) can specify any number of optional arguments by providing default values, so long as all those optional arguments are last in the argument list.

## Example: Declare an Array Variable and Transform its Elements

Variables are immutable in DataWeave, however you can iterate over an array variable and apply a transformation to its elements, which can then be assigned to new variables. The following example declares two array variables, iterates over them to transform their elements, and then declares an object containing two new arrays with the transformed elements:

DataWeave Source:

```
%dw 2.0
import * from dw::core::Strings
output application/json
var numberArray = [1,2,3,4,5,6]
var stringArray = ["max", "astro", "einstein"]
---
{
    "numberArrayUpdated" : numberArray map ((value, index) -> value * value),
    "stringArrayUpdated" : stringArray map ((value, index) -> index ++ " - " ++
capitalise(value))
}
```

Output:

```
{
    "numberArrayUpdated": [ 1, 4, 9, 16, 25, 36 ],
    "stringArrayUpdated": [ "0 - Max", "1 - Astro", "2 - Einstein" ]
}
```

## Naming Rules for Variables

You can declare any variable name that is a valid identifier. See [Rules for Declaring Valid Identifiers](#) for more details.

## Variable Scopes

You can initialize and use both global and local variables in DataWeave scripts.

- Global variables are initialized in the header of the DataWeave script and can be referenced by name from anywhere in the body of a DataWeave script.

The header of a DataWeave script accepts a `var` directive that initializes a variable, for example: `var language='Español'`. You can declare multiple global variables on separate lines in the header.

- Local variables are initialized in the body of the DataWeave script and can be referenced by name only from within the scope of the expression where they are initialized.

The syntax for initializing a local variable looks like this:

```
do {
    [type <name> = <type expression>]*
    [var <name> = <expression>]*
    [fun <name>(<params>*) = <expression>]*
    [ns <name> <nsUri>]*
    ---
    <body>
}
```

Note that DataWeave variables cannot be reassigned. They are also distinct from variables that are part of the Mule message (such as target variables). DataWeave variables do not persist beyond the scope of the script in which they are initialized.

## Example: Global DataWeave Variables

This example defines a `do` operation that initializes a `language` variable with the constant value `Español` and returns that constant value. The body of the DataWeave script calls the `do` operation using the variable `myVar`, which populates the `<language/>` element in the XML output with the value `Español`.

*Transform*

```
%dw 2.0
output application/xml
var myVar = do {
    var language = "Español"
    ---
    language
}
---
{
    document: {
        language: myVar,
        text: "Hola mundo"
    }
}
```

## *Output*

```
<?xml version='1.0' encoding='UTF-8'?>
<document>
  <language>Español</language>
  <text>Hola mundo</text>
</document>
```

## **Example: Local DataWeave Variables**

To initialize local variables, you can use either literal expressions, variable reference expressions, or functional expressions. These expressions can reference any other local variables within their scope or any input or global variables.

As a best practice, declare local variables within `do` scopes (see [Flow Control in DataWeave](#)). The alternative method of declaring a scope with the `using` keyword is no longer recommended and available only to provide compatibility.

You can only reference a local variable by name from within the scope of the expression that initializes it. The declaration can be prepended to any literal expression. The literal delimits the scope of the variable, so you cannot reference any variable outside of its scope.

The examples that follow show initialization of local variables.

### *Example: Scoped to a Simple Value*

The following example sets the local variable `myVar` in a `do` scope. It calls `do` from the body of the DataWeave script:

```
%dw 2.0
output application/json
---
do {
  var myVar = 2
  ---
  3 + myVar
}
```

The result is `5`.

### *Example: Scoped to an Array Literal*

The following example sets the local variable `myVar` in a `do` scope and uses it to set the value of the second element in the array to `2`:

```
%dw 2.0
output application/json
---
do {
    var myVar = 2
    ---
    [1, myVar, 3]
}
```

The result is [ 1, 2, 3]

*Example: Scoped to the Object literal*

In the following example, references to all the variables are valid. `fn` and `ln` are defined and called within the `do` scope. The global `myVar` variable is also accessible from that scope.

```
%dw 2.0
var myVar = 1234
var myDo = do {
    var fn = "Annie"
    var ln = "Point"
    ---
    {
        id : myVar,
        firstname : fn,
        lastname : ln
    }
}
output application/xml
---
{ person : myDo }
```

```
<?xml version='1.0' encoding='UTF-8'?>
<person>
    <id>1234</id>
    <firstname>Annie</firstname>
    <lastname>Point</lastname>
</person>
```

*Example: Invalid Reference That Is Outside the Scope*

The following example produces an error because `fromDoScope` is referenced from outside the scope of `do`. As a consequence, the concatenation operation (`++`) cannot append the name-value pair to the collection.

```
%dw 2.0
var myVar = 1234
var myDo = do {
    var fn = "Annie"
    var ln = "Point"
    var fromDoScope = "Platform"
    ---
    {
        id : myVar,
        firstname : fn,
        lastname : ln
    }
}
output application/xml
---
{
    person : myDo ++ { "outsideDoScope" : fromDoScope }
}
```

The invalid example returns this error: `Unable to resolve reference of fromDoScope.`

*Example: Reference That Is Inside a Function*

The following example passes the string `HELLO` to the `test` function defined in the header. The `do` scope accepts the string, converts it to lowercase, and then concatenates that string to the `suffix` variable, which is also defined in the header of the scope.

```
%dw 2.0
fun test(param1: String) = do {
    var suffix = "123"
    fun innerTest(str: String) = lower(str)
    ---
    innerTest(param1 ++ suffix)
}
output application/json
---
test("HELLO")
```

The result is `"hello123"`.

## Variables Assigned as Lambda Expressions

When a variable is assigned a lambda expression, it behaves just like a function. Like a function, the lambda expression for a variable assignment can include parameters that can be used in the right-hand side expression, such as `(string) → upper(string)`, which will convert any string to uppercase.

Other variables can also be passed as arguments to another variable's lambda expression. This example shows how the `toUpperCase` variable's lambda expression can be passed by name into the

`addSuffix` variable's lambda expression.

```
%dw 2.0
output application/json
var toUpper = (aString) -> upper(aString)

var addSuffix = (msg, aFunction, suffix) -> aFunction(msg ++ suffix)
---
addSuffix("hello", toUpper, " world") //result is "HELLO WORLD"
```

## @Lazy() Annotation

When you are writing a DataWeave script in Studio, you might see the internal annotation `@Lazy()` in autocompletion suggestions. This annotation is not intended for general use and does not improve the performance of your scripts.

## Predefined Variables

DataWeave expressions accept variables that can retrieve values from a variety of Mule Runtime engine objects. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

Commonly used variables include `attributes`, `payload`, and `vars`, the main parts of the Mule event.

Variables	Description	Fields and Examples
<code>app</code>	The Mule artifact in context.	<ul style="list-style-type: none"><li>• Fields:<ul style="list-style-type: none"><li>◦ <code>encoding</code>: Encoding used by the Mule message processor.</li><li>◦ <code>name</code>: Name of your project and Mule app.</li><li>◦ <code>standalone</code>: Returns <code>true</code> or <code>false</code>.</li><li>◦ <code>workDir</code>: Working directory for the Mule app. For Studio, this is an Eclipse plugin directory.</li><li>◦ <code>registry</code>: Map of Mule registry entries.</li></ul></li><li>• Example: <code>#[app.encoding]</code> might return <code>UTF-8</code>.</li></ul>

Variables	Description	Fields and Examples
<code>attributes</code>	Attributes (metadata) of a Mule Message object ( <code>message</code> ).	<ul style="list-style-type: none"> <li>Fields: Depends on the message type, such as HTTP request attributes.</li> <li>Example: <code>#[attributes]</code> returns message attributes. For an HTTP request, <code>#[attributes.header]</code> returns HTTP header metadata, and <code>#[attributes.version]</code> might return <code>HTTP/1.1</code>.</li> </ul>
<code>authentication</code>	Provides access to the authentication information.	<ul style="list-style-type: none"> <li>Fields: <ul style="list-style-type: none"> <li><code>principal</code>: The main part of the authentication, such as the user or subject.</li> <li><code>credentials</code>: The credentials for the authentication.</li> <li><code>properties</code>: A map of properties for the authentication.</li> </ul> </li> <li>Example: <code>#[authentication.principal]</code></li> </ul>
<code>correlationId</code>	The <code>correlationId</code> of the message being processed.	<ul style="list-style-type: none"> <li>Fields: No fields.</li> <li>Example: <code>#[correlationId]</code> might return <code>0-f77404d0-e699-11e7-a217-38c9864c2f8f</code>.</li> </ul>
<code>dataType</code>	Data type of the message payload.	<ul style="list-style-type: none"> <li>Fields: No fields.</li> <li>Example: <code>#[dataType]</code> might return <code>SimpleDataType</code>.</li> </ul>

Variables	Description	Fields and Examples
<code>error</code>	Error associated with a Mule message object.	<ul style="list-style-type: none"> <li>• Fields: <ul style="list-style-type: none"> <li>◦ <code>description</code>: Concise description of the error.</li> <li>◦ <code>detailedDescription</code>: Can provide a more thorough description.</li> <li>◦ <code>failingComponent</code>: The component where the error occurred.</li> <li>◦ <code>errorType</code>: The type of error. The <code>errorType</code> is composed of <code>identifier</code> and <code>namespace</code>.</li> <li>◦ <code>cause</code>: The internal Java <code>Throwable</code> that caused the error.</li> <li>◦ <code>errorMessage</code>: An optional Mule Message provided by the failing component (used by components like HTTP when a response is invalid).</li> <li>◦ <code>childErrors</code>: An optional list of internal errors (provided by components that aggregate errors, such as scatter-gather).</li> </ul> </li> <li>• Example: <code>#[\$error.description]</code></li> </ul>
<code>flow</code>	Deprecated: The name of the current flow. Because flow names are static, this field is deprecated and only available through the Logger component.	<ul style="list-style-type: none"> <li>• Fields: <ul style="list-style-type: none"> <li>◦ <code>name</code></li> </ul> </li> <li>• Example: <code>#[\$flow.name]</code> returns <code>testFlow</code> in the Studio console when executed within a Flow with the name <code>testFlow</code>.</li> </ul>
<code>message</code>	Package ( <code>payload</code> and <code>attributes</code> ) being processed.	<ul style="list-style-type: none"> <li>• Fields: <ul style="list-style-type: none"> <li>◦ <code>payload</code>: The message payload, or <code>null</code> if the payload is null.</li> <li>◦ <code>dataType</code>: The message payload's data type.</li> <li>◦ <code>attributes</code>: The message attributes, or <code>null</code> if the attributes are null.</li> </ul> </li> <li>• Example: <code>#[\$message]</code></li> </ul>

Variables	Description	Fields and Examples
<code>mule</code>	The Mule instance on which the application is currently running.	<ul style="list-style-type: none"> <li>• Fields: <ul style="list-style-type: none"> <li>◦ <code>clusterId</code>: The ID of the cluster when the Mule runtime is part of a High Availability cluster.</li> <li>◦ <code>home</code>: Home directory.</li> <li>◦ <code>nodeId</code>: Cluster Node ID.</li> <li>◦ <code>version</code>: Mule version.</li> </ul> </li> <li>• Example: <code>#[mule.version]</code> might return <code>4.0.0</code>.</li> </ul>
<code>payload</code>	The body of the current Mule message object ( <code>message</code> ) being processed.	<ul style="list-style-type: none"> <li>• Fields: Depends on the current payload.</li> <li>• Example: <code>#[payload]</code> returns the body of the message.</li> </ul>

Variables	Description	Fields and Examples
<code>server</code>	<p>The operating system on which the Mule instance is running. Exposes information about both the physical server and the JVM on which Mule runs.</p> <ul style="list-style-type: none"> <li>Note that the ability to select system properties, like <code>user.name</code>, depends on the behavior of the JVM on your operating system.</li> </ul>	<ul style="list-style-type: none"> <li>Fields: <ul style="list-style-type: none"> <li><code>env</code>: Map of operating system environment variables.</li> <li><code>fileSeparator</code>: Character that separates components of a file path, which is <code>/</code> on UNIX and <code>\</code> on Windows.</li> <li><code>host</code>: Fully qualified domain name for the server.</li> <li><code>ip</code>: IP address of the server.</li> <li><code>locale</code>: Default locale (<code>java.util.Locale</code>) of the JRE. Can be used language (<code>locale.language</code>), country (<code>locale.country</code>).</li> <li><code>javaVendor</code>: JRE vendor name</li> <li><code>javaVersion</code>: JRE version</li> <li><code>osArch</code>: Operating system architecture.</li> <li><code>osName</code>: Operating system name.</li> <li><code>osVersion</code>: Operating system version.</li> <li><code>systemProperties</code>: Map of Java system properties.</li> <li><code>timeZone</code>: Default time zone (<code>java.util.TimeZone</code>) of the JRE.</li> <li><code>tmpDir</code>: Temporary directory for use by the JRE.</li> <li><code>userDir</code>: User directory.</li> <li><code>userHome</code>: User home directory.</li> <li><code>userName</code>: User name.</li> </ul> </li> <li>Example: <code>#[server.osName]</code> might return <code>Mac OS X</code>.</li> </ul>
<code>vars</code>	All variables currently set on the current Mule event being processed.	<ul style="list-style-type: none"> <li>Fields: No fields.</li> <li>Example: <code>#[vars.myVar]</code> returns the value of <code>myVar</code>.</li> </ul>

# Flow Control in DataWeave

You can use the following operators within any DataWeave expression:

- `do`
- `if else`
- `else if`

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

## do

A `do` statement creates a scope in which new variables, functions, annotations, or namespaces can be declared and used. The syntax is similar to a mapping in that it is composed of a header and body separated by `---`. Its header is where all the declarations are defined, and its body is the result of the expression.

This example uses `do` to return the string "`DataWeave`" when `myfun()` is called from the main body of the script.

*DataWeave Script:*

```
%dw 2.0
output application/json
fun myfun() = do {
    var name = "DataWeave"
    ---
    name
}
---
{ result: myfun() }
```

This example uses `do` to return the string "`DataWeave`" when the variable `myVar` is referenced from the main body of the script.

*DataWeave Script:*

```
%dw 2.0
output application/json
var myVar = do {
    var name = "DataWeave"
    ---
    name
}
---
{ result: myVar }
```

Both scripts produce this output:

*Output:*

```
{  
  "result": "DataWeave"  
}
```

The next example uses `do` to prepend the string `"Foo"` to a string (`" Bar"`) that is passed to the `test(p: String)` function.

*DataWeave Script:*

```
%dw 2.0  
output application/json  
fun test(p: String) = do {  
  var a = "Foo" ++ p  
  ---  
  a  
}  
---  
{ result: test(" Bar") }
```

*Output:*

```
{  
  "result": "Foo Bar"  
}
```

See also, [Examples: Local DataWeave Variables](#).

## `if else`

An `if` statement evaluates a conditional expression and returns the value under the `if` only if the conditional expression returns `true`. Otherwise, it returns the expression under `else`. Every `if` expression must have a matching `else` expression.

The following example uses the input `{ country : "FRANCE" }`, which is defined by the `myVar` variable in the header:

*DataWeave Script:*

```
%dw 2.0  
var myVar = { country : "FRANCE" }  
output application/json  
---  
if (myVar.country == "USA")  
  { currency: "USD" }  
else { currency: "EUR" }
```

*Output:*

```
{  
  "currency": "EUR"  
}
```

You can use the if-else construct on any condition that evaluates to `true` or `false`, including mathematical, logical, equality, and relational statements. The condition can act on any valid input.

The next DataWeave script applies `if else` statements to the result of the following conditional statements:

- A mathematical operation in `ex1`

The `if else` statement returns the Boolean value `true` if the operation `1 + 1 == 55` is true and `false` if not.

- An equality operation in `ex2`

The `if else` statement returns `1` if the value of the specified index is `1` or a string if the value is not `1`.

- An `isEmpty` function in `ex3`

The `if else` statement returns the string `"ID is empty"` or `"ID is not empty"` depending on whether `aRecord.bookId` contains a value.

- A mapping in `ex4` that iterates over `firstInput`

The `if else` statement returns the value of the `bookId` as a Number if the value is equal to `101`. It returns the specified string if the value is not equal to `101`.

## DataWeave Script:

```
%dw 2.0
var aRecord =
[
    "bookId":"101",
    "title":"world history",
    "price":"19.99"
]
output application/xml
---
{ examples:
  {
    ex1 : if (1 + 1 == 55) true
          else false,
    ex2 : if ([1,2,3,4][1] == 1) 1
          else "value of index 1 is not 1",
    ex3 : if (isEmpty(aRecord.bookId)) "ID is empty"
          else "ID is not empty",
    ex4 : aRecord.bookId map (idValue) ->
      if (idValue as Number == 101) idValue as Number
      else "not 101"
  }
}
```

## Output:

```
<?xml version='1.0' encoding='UTF-8'?>
<examples>
  <ex1>false</ex1>
  <ex2>value of index 1 is not 1</ex2>
  <ex3>ID is not empty</ex3>
  <ex4>101</ex4>
</examples>
```

Additional examples are available in [DataWeave Operators](#).

## else if

You can chain several `else` expressions together within an if-else construct by incorporating `else if`. The following example uses the input `var myVar = { country : "UK" }`, which is defined by the `myVar` variable in the header.

*DataWeave Script:*

```
%dw 2.0
var myVar = { country : "UK" }
output application/json
---
if (myVar.country == "USA")
    { currency: "USD" }
else if (myVar.country == "UK")
    { currency: "GBP" }
else { currency: "EUR" }
```

*Output*

```
{
  "currency": "GBP"
}
```

The following example is similar but takes an array as input instead of an object. The body of the script uses **if** **else** and **else if** statements within a **do** operation to populate the value of the **hello** variable.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
["Argentina", "USA", "Brazil"] map (country) -> do {
  var hello = if(country == "Argentina") "Hola"
  else if(country == "USA") "Hello"
  else if(country == "Brazil") "Ola"
  else "Sorry! We don't know ${country}'s language."
  ---
  "${hello} DataWeave"
}
```

*Output:*

```
[
  "Hola DataWeave",
  "Hello DataWeave",
  "Ola DataWeave"
]
```

## See Also

- [DataWeave Types](#)
- [Selectors](#)

# Pattern Matching in DataWeave

The `match` statement behaves like a `match` or `switch` statement in other languages, like Java or C++, and routes an input expression to a particular output expression based on some conditions. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

*match Statement Syntax:*

```
inputExpression match {  
    case <condition> -> <routing expression>  
    case <condition> -> <routing expression>  
    else -> <default routing expression>  
}
```

As in other languages, the DataWeave `match` statement provides a compact way to organize multiple, chained `if-else` statements. A `match` expression consists of a list of `case` statements that optionally end with an `else` statement. Each `case` statement consists of a conditional selector expression that must evaluate to `true` or `false`. This conditional expression corresponds to the condition expressed by an `if` statement, followed by a corresponding DataWeave routing expression that can include the `match` statement's input expression.

Each `case` statement can be an `if` statement, or the `case` statement can use other pattern-matching shortcuts to define the `case` statement's condition. If a `case` statement is `false`, its routing expression is ignored.

DataWeave supports four different types of patterns for a `case` statement's condition:

- [Expressions](#)
- [Literal values](#)
- [Data types](#)
- [Regular expressions](#)

Each case can be named or unnamed. If the case is named, the name stores the input expression as a local variable that can be used both in that case statement's conditional expression and in the corresponding routing expression.

*match Statement Structure*

```
value match {  
    case (<name>:) <condition> -> <routing expression>  
    case (<name>:) <condition> -> <routing expression>  
    else -> <when none of them matched>  
}
```

As this example shows, the expression returns the results of the first matching `case` statement:

### *match Statement Example*

```
%dw 2.0
output application/json
---
"hello world" match {
    case word matches /(hello)\s\w+/ -> word[1] as String ++ " was matched"
    case literalMatch: "hello world" -> upper(literalMatch)
    case hasOne if( hasOne is Object and hasOne.three? ) -> hasOne.three
    else -> $
}
```

### *Output*

```
"hello was matched"
```

Notice that you can refer to the input expression ("hello world") through each `case` statement's local variable name (that is, `word` or `literalMatch`), while the `else` statement can refer to the input expression using the default parameter name `$`, an unnamed parameter.

To name the input expression in the `else` statement, you can replace the `else` statement with a `case` statement that is always `true`:

### *match Statement Example*

```
%dw 2.0
output application/json
---
"hello world" match {
    case word matches /(hello)\s\w+/ -> word[1] ++ " was matched"
    case literalMatch: "hello world" -> upper(literalMatch)
    case last if(true) -> last
}
```

For use cases in which the input is of type `String`, you can also use the `match function` in the `dw::Core` module to test the string against a regex pattern. The function returns an array with the strings that match the entire regex and any capture groups.

For use cases that require a Boolean result based on whether a string matches a regex pattern, you can use the `matches function` in the `dw::Core` module instead of using pattern matching or the module's `match` function.

## **Pattern Matching to Literal Values**

Matches when the evaluated value equals a simple literal value.

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

## DataWeave Script

```
%dw 2.0
var myInput = {
    "string": "Emiliano"
}
output application/json
---
{
    a: myInput.string match {
        case "Emiliano" -> true
        case "Mariano" -> false
    },
    b: myInput.string match {
        case str: "Emiliano" -> { "matches": true, value: str }
        case str: "Mariano" -> { "matches": false, value: str }
    }
}
```

## Output

```
{
    "a": true,
    "b": {
        "matches": true,
        "value": "Emiliano"
    }
}
```

## Pattern Matching on Expressions

Matches when a given expression returns **true**.

In this example, the first field matches the value of `myInput.string` against two alternatives and conditionally appends a different string to it. The second field evaluates if the value in `myInput.number` is greater than (`>`), less than (`<`), or equal to (`==`) 3 and returns the corresponding string.

## DataWeave Script

```
%dw 2.0
var myInput = {
    "string": "Emiliano",
    "number": 3.14
}
output application/json
---
{
    a: myInput.string match {
        case str if str == "Mariano" -> str ++ " de Achaval"
        case str if str == "Emiliano" -> str ++ " Lesende"
    },
    b: myInput.number match {
        case num if num == 3 -> "equal"
        case num if num > 3 -> "greater than"
        case num if num < 3 -> "less than"
    }
}
```

## Output

```
{
    "a": "Emiliano Lesende",
    "b": "greater than"
}
```

## Pattern Matching on the Data Type

Matches when the evaluated value is the specified data type.

In this example, the first field evaluates the data type of `myInput.a` and returns a string with the corresponding type name. The second field is similar but also returns the value of the `myInput.b`.

## DataWeave Script

```
%dw 2.0
var myInput =
{
  "a": "Emiliano",
  "b": 3.14
}
output application/json
---
{
  a: myInput.a match {
    case is Object -> 'OBJECT'
    case is String -> 'STRING'
    case is Number -> 'NUMBER'
    case is Boolean -> 'BOOLEAN'
    case is Array -> 'ARRAY'
    case is Null -> 'NULL'
    else -> 'ANOTHER TYPE'
  },
  b: myInput.b match {
    case y is Object -> { 'Type': { 'OBJECT' : y} }
    case y is String -> { 'Type': { 'STRING' : y} }
    case y is Number -> { 'Type': { 'NUMBER' : y} }
    case y is Boolean -> { 'Type': { 'BOOLEAN' : y} }
    case y is Array -> { 'Type': { 'ARRAY' : y} }
    case y is Null -> { 'Type': { 'NULL' : y} }
    else -> { 'Type': { 'ANOTHER TYPE' : myInput.b} }
  }
}
```

## Output

```
{
  "a": "STRING",
  "b": {
    "Type": {
      "NUMBER": 3.14
    }
  }
}
```

## Pattern Matching on Regular Expressions

Matches when the evaluated value fits a given regular expression (regex), specifically a regex "flavor" supported by Java. In this example, the input variable (`myInput`) includes an array of strings. The script uses the `map` function to iterate through the array. It evaluates each element against a Java regex and outputs an object based on matches to the input.

## DataWeave Script

```
%dw 2.0
var myInput = {
    "phones": [
        "+1 (415) 229-2009",
        "(647) 456-7008"
    ]
}
output application/json
---
{
    a: myInput.phones map ($ match {
        case phone matches /\+(\d+)\s\(((\d+)\)\)\s(\d+\-\d+)/ -> { country: phone[1]}
        case phone matches /\((\d+)\)\s(\d+\-\d+)/ -> { area: phone[1], number: phone[2]}
    }),
    b: myInput.phones map ($ match {
        case phone matches /\+(\d+)\s\(((\d+)\)\)\s(\d+\-\d+)/ -> { country: phone[1], area: phone[2], number: phone[3] }
        case phone matches /\((\d+)\)\s(\d+\-\d+)/ -> { area: phone[1], number: phone[2] }
    })
}
```

## Output

```
{
    "a": [
        {
            "country": "1"
        },
        {
            "area": "647",
            "number": "456-7008"
        }
    ],
    "b": [
        {
            "country": "1",
            "area": "415",
            "number": "229-2009"
        },
        {
            "area": "647",
            "number": "456-7008"
        }
    ]
}
```

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Quickstart](#)

# External Functions Available to DataWeave

In addition to the built-in DataWeave functions, Mule Runtime and some modules, connectors, and Core components provide functions that you can use in DataWeave scripts. The functions are specific to the modules, connectors, and components that provide them. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

## Mule Runtime Functions

These functions are injected by the Mule Runtime:

- `p`
- `lookup`
- `causedBy`

Starting in Mule 4.1.4, DataWeave incorporated the Mule Runtime functions into its Mule function module.

MuleSoft recommends that you start using the `Mule` namespace when using these functions in Mule apps that are running on Mule Runtime 4.1.4 or later. To use them, you simply prepend the namespace to the function name, for example, `Mule:::p('http:port')`, instead of `p('http:port')`.



In addition to its continued support for the use of Mule Runtime functions in your DataWeave scripts and mappings, the Mule module also enables you to use Mule Runtime functions in your [custom DataWeave modules](#).

Mule apps running on Mule Runtime versions *prior* to version 4.1.4 cannot use the `Mule` namespace or use Mule Runtime functions in custom modules, only in DataWeave scripts and mappings.

For DataWeave runtime properties, see the DataWeave functions `props` and `prop`.

### Accessing Properties (p Function)

The `p` function provides access to properties, whether these are:

- Mule property placeholders
- System properties

- Environment properties

For more on configuring properties and how they are handled, see [Configure Properties](#).

The following example logs the value of the property `http.port`.

*Example: Property Function Usage for Mule 4.1.4 and Later*

```
<flow name="simple">
  <logger level="INFO" doc:name="Logger"
    message="#[Mule:::p('http.port')]" />
</flow>
```

Note that you can also use the function when defining a DataWeave variable or function. This example uses the `p` function in the `myVar` definition. You can do the same thing with pre-4.1.4 runtimes simply by omitting the `Mule:::` namespace.

```
<flow name="simple">
  <logger level="INFO" doc:name="Logger"
    message="#[var myVar = Mule:::p('http.port') --- myVar]" />
</flow>
```

*Example: Property Function Usage for Mule 4.1.3 and Earlier*

```
<flow name="simple">
  <logger message="#[p('http.port')]" />
</flow>
```

See also, the `p` function introduced in the Mule function module in Mule 4.2.

## Execute a Flow (lookup Function)

Similar to the Flow Reference component, the `lookup` function enables you to execute another flow within your app and to retrieve the resulting payload. It takes the flow's name and an input payload as parameters. For example, `lookup("anotherFlow", payload)` executes a flow named `anotherFlow`.

The function executes the specified flow using the current attributes, variables, and any error, but it only passes in the payload without any attributes or variables. Similarly, the called flow will only return its payload.

Note that `lookup` function does not support calling subflows.

### Parameters

Name	Description
<code>flowName</code>	A string that identifies the target flow.
<code>payload</code>	The payload to send to the target flow, which can be any ( <code>Any</code> ) type.

Name	Description
<code>timeoutMillis</code>	Optional. Timeout (in milliseconds) for the execution of the target flow. Defaults to <code>2000</code> milliseconds (2 seconds) if the thread that is executing is <code>CPU_LIGHT</code> or <code>CPU_INTENSIVE</code> , or 1 minute when executing from other threads. If the lookup takes more time than the specified <code>timeoutMillis</code> value, an error is raised.

This example shows XML for two flows. The `lookup` function in `flow1` executes `flow2` and passes the object `{test:'hello'}` as its payload to `flow2`. The Set Payload component (`<set-payload/>`) in `flow2` then concatenates the value of `{test:'hello'}` with the string `world` to output and log `hello world`.

*Example: Using the `lookup` Function in Mule 4.1.4 or Later*

```

<flow name="flow1">
  <http:listener doc:name="Listener" config-ref="HTTP_Listener_config"
    path="/source"/>
  <ee:transform doc:name="Transform Message" >
    <ee:message >
      <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
Mule:::lookup('flow2', {test:'hello'})]]></ee:set-payload>
      </ee:message>
    </ee:transform>
  </flow>
<flow name="flow2" >
  <set-payload value='#[payload.test ++ "world"]' doc:name="Set Payload" />
  <logger level="INFO" doc:name="Logger" message='#[payload]'/>
</flow>

```

*Example: Using the `lookup` Function in Mule 4.1.3 and Earlier*

```

<flow name="flow1">
  <http:listener doc:name="Listener" config-ref="HTTP_Listener_config"
    path="/source"/>
  <ee:transform doc:name="Transform Message" >
    <ee:message >
      <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
lookup('flow2', {test:'hello'})]]></ee:set-payload>
      </ee:message>
    </ee:transform>
  </flow>
<flow name="flow2" >
  <set-payload value='#[payload.test ++ "world"]' doc:name="Set Payload" />
  <logger level="INFO" doc:name="Logger" message='#[payload]'/>
</flow>

```

## Output

```
hello world
```

Always keep in mind that a functional language like DataWeave expects the invocation of the `lookup` function to *not* have side effects. As such, the internal workings of the DataWeave engine might cause a `lookup` function to be invoked in parallel with other `lookup`'s, or not invoked at all.



MuleSoft recommends that you invoke flows with the Flow Ref (`flow-ref`) component, using the `target` attribute to put the result of the flow in a `var` and then referencing that `var` from within the DataWeave script.

## Matching Errors by Types (causedBy Function)

The `causedBy` function matches an error by its type, like an error handler does. This is useful when matching by a super type is required but specific sub-type logic is also needed or when handling a `COMPOSITE_ROUTING` error that contains child errors of different types.

The error to match against can be implicit, but the type is always a required parameter.

In the following example, a global error handler is set up to handle `SECURITY` errors in a general way, while specific actions are set up for `HTTP:UNAUTHORIZED` and `HTTP:FORBIDDEN` errors.

*Example: Error Matcher Function Usage in Mule Runtime Version 4.1.4 and Later*

```
<error-handler name="securityHandler">
    <on-error-continue type="SECURITY">
        <!-- general error handling for all SECURITY errors -->
        <choice>
            <when expression="#[error Mule::causedBy 'HTTP:UNAUTHORIZED']">
                <!-- specific error handling only for HTTP:UNAUTHORIZED errors -->
            </when>
            <when expression="#[Mule::causedBy('HTTP:FORBIDDEN')]">
                <!-- specific error handling only for HTTP:FORBIDDEN errors -->
            </when>
        </choice>
    </on-error-continue>
</error-handler>
```

```
<error-handler name="securityHandler">
  <on-error-continue type="SECURITY">
    <!-- general error handling for all SECURITY errors -->
    <choice>
      <when expression="#[error causedBy 'HTTP:UNAUTHORIZED']">
        <!-- specific error handling only for HTTP:UNAUTHORIZED errors -->
      </when>
      <when expression="#[causedBy('HTTP:FORBIDDEN')]">
        <!-- specific error handling only for HTTP:FORBIDDEN errors -->
      </when>
    </choice>
  </on-error-continue>
</error-handler>
```

Notice that the error parameter is used both explicitly and implicitly.

## Connector and Component Functions

When using connectors and components in a Mule app, you can inject functions. Unlike the Runtime functions, these functions require a namespace, which usually matches the component name.

For example, in an app using Batch you can use the following expression: `##[Batch:::isSuccessfulRecord()]`.

## See Also

[DataWeave Core Functions](#)

[DataWeave Types](#)

[Selectors](#)

## Troubleshooting

When you troubleshoot a failing script, one challenge to reproducing an error is having the same inputs the script had when it executed, especially in production environments where inputs can change unexpectedly. Therefore, it's important to capture the inputs going into a script debugging or using loggers. Often, the failures occur because the input coming from another component upstream is not valid. You can find here a listing of the most common DataWeave errors and how to overcome them.

## Dump Input Context and the Script into a Folder

In Mule 4.2.1, DataWeave introduced an experimental feature that enables you to dump the input context and the failing script into a folder so that you can track the failing script along with the data that makes the script fail. This tool is particularly useful for checking that received input data is

valid because incorrect scripts often fail when an upstream component generates invalid data.

To use this feature:

1. Set the following system property to enable the dump feature:

`-M-Dcom.mulesoft.dw.dump_files=true`

2. [Optional] Specify the path in which to generate the dump files:

`-M-Dcom.mulesoft.dw.dump_folder=<path_to_folder>`

The default directory is set in the `java.io.tmpdir` property.

## DataWeave Exceptions

The following are some common DataWeave exceptions that you can find in your dump file, along with some details to troubleshoot these errors.

### Incorrect Arguments

When a function is called with the incorrect kind of argument, it throws the exception, `org.mule.weave.v2.exception.UnsupportedTypeCoercionException`. Causes of this exception include:

- Function Does Not Accept Null Argument
- MIME Type Is Not Set

#### Function Does Not Accept Null Argument

The most common cause of this exception occurs when one of the arguments is `Null` and the function does not accept `Null` as an argument. This issue results in an error message similar to the following:

You called the function '++' with these arguments:

```
1: Null (null)
2: String (" A text")
```

But it expects one of these combinations:

```
(Array, Array)
(Date, Time)
(Date, LocalTime)
(Date, TimeZone)
(LocalDateTime, TimeZone)
(LocalTime, Date)
(LocalTime, TimeZone)
(Object, Object)
(String, String)
(Time, Date)
(TimeZone, LocalDateTime)
(TimeZone, Date)
(TimeZone, LocalTime)
```

```
1| payload.message ++ " A text"
^~~~~~
```

Trace:

```
at ++ (line: 1, column: 1)
at main (line: 1, column: 17)
```

One way to resolve this issue uses the `default` operator. For example, using `payload.message default "" ++ " A text"` appends empty text when the message is null.

#### MIME Type Is Not Set

When the MIME type is not set, you receive an error message similar to the following:

You called the function 'Value Selector' with these arguments:

```
1: String ("{ \"message\": 123}")
2: Name ("message")
```

But it expects one of these combinations:

```
(Array, Name)
(Array, String)
(Date, Name)
(DateTime, Name)
(LocalDateTime, Name)
(LocalTime, Name)
(Object, Name)
(Object, String)
(Period, Name)
(Time, Name)
```

```
1| payload.message
 ^^^^^^^^^^^^^^
```

Trace:

```
at main (line: 1, column: 1)
```

When no MIME type is set on the payload, the MIME type defaults to `application/java`, and the content is handled as a `String` instead of a JSON object.

## Reader Properties Not Working In a Mule Application

In a Mule application, the `input` directive to a DataWeave script does not work. Unlike Mule runtime, a standalone DataWeave runtime, such as the one in the [DataWeave Playground](#), can process a valid MIME type set through the `input` directive in the same DataWeave script. To input reader properties to a script in a Mule application, configure the `outputMimeType` attribute for the data source to produce the same results. See [Using Reader and Writer Properties](#) for further details.

## Stack Overflow

When a function recurses too deeply, an error like the following one is thrown:

```
Stack Overflow. Max stack is 256
```

You can configure the maximum stack size by using the property `com.mulesoft.dw.stacksize`.

## No space left on device

To handle large payloads, DataWeave generates data that is handled in memory unless the payload exceeds a configurable limit. If the payload exceeds the limit, the data is stored on disk as output, input, and buffer files in a temporary directory. See [Memory Management](#) for further details.

When the streams that reference the files are closed, the files are released. Closure normally occurs

when flows complete their execution, so many buffer files in the temporary folder can remain in use during long-running and concurrent executions.

To avoid the exception `No space left on device`, try to provide more resources to run the application, or determine whether you can reduce the application's resource consumption.

Sometimes bugs prevent the release of the files even after the flow has completed. If the latest Mule release does not fix this issue, please report the issue to the MuleSoft support team.

## Closed Streams

Input data that reaches DataWeave is usually a stream. Mule handles the stream and adds auto-closing and repeatable capabilities to it by generating “cursors” over the stream. Sometimes a stream closes prematurely, before it reaches the DataWeave script, and it can be difficult to understand what closed it. In these circumstances, the script fails with an error similar to `Cannot open a new cursor on a closed stream`.

If this error occurs in latest Mule update, please report it to the MuleSoft support team. It is not possible to work around the issue, but you can use the `com.mulesoft.dw.track.cursor.close` system property to determine which component closed the stream prematurely. With the property set, the error shows the stack trace from the moment that the stream was closed, which points to the component that triggered the closure.

## Output Mismatch When Undefined

Unlike transformations, DataWeave expressions do not require you to define an output format because DataWeave can infer the output based on the expression and the variables you use. Occasionally, the inference process results in a mismatch between the inferred type and the expected type. To resolve this issue, you must make the output explicit. Common examples of this situation occur when:

- [Extract Data from XML](#)
- [Handle Multipart Entries](#)
- [Manipulate Text Data](#)

### Extract Data from XML

When extracting a String, for example, from an XML payload with the expression `payload.order.product.model`, DataWeave infers an XML output based on the payload format. In such cases, an error similar to the following one occurs:

"Trying to output non-whitespace characters outside main element tree (in prolog or epilog), while writing Xml at ." evaluating expression: "payload.order.product.model".

For such an error, you must make the output format explicit, for example: `output text/plain --- payload.order.product.model`.

### Handle Multipart Entries

A common inference error occurs with multipart data, which has a very specific structure. Consider a multipart payload and the expression `dw::core::Objects::keySet(payload.parts)`. Without an explicit output format, DataWeave must infer, based on the payload type, that you intend to output multipart content. In this case, an error similar to the following is thrown:

```
"Expecting type is {
  preamble?: String,
  parts: {
    _: {
      headers: Object,
      content: Any
    }
  }
} but got Array, while writing MultiPart.
Trace:
  at main (Unknown)" evaluating expression:
"dw::core::Objects::keySet(payload.parts)".
```

To resolve this issue, you must define an output format, for example: `output application/json --- dw::core::Objects::keySet(payload.parts)`

### Manipulate Text Data

You can use text data to create a more complex object. However, if you do not define the output format for the input text data, DataWeave infers that it must use the plain text writer for the output. The expression `payload splitBy ' '`, for example, will fail with an error similar to:

```
"Text plain writer is unable to write Array.
Reason:
Cannot coerce Array
(org.mule.weave.v2.model.values.ArrayValue$ArraySeqArrayValue@1331b353) to String
Trace:
  at main (Unknown), while writing TextPlain.
Trace:
  at main (Unknown)" evaluating expression: "payload splitBy ' '".
```

Making the output explicit solves the issue: `output application/java --- payload splitBy ' '`

### Encoding Issues

Encoding issues can occur because of a mismatch between encodings used to read and write a file or when the encoding to write some text does not support some of the characters. Common examples include the following:

- [Incorrect Encoding for the Reader](#)
- [Encoding Used by the Writer Does Not Support Some Characters](#)
- [Multipart/Form-Data Reader Does Not Support UTF-8 Characters by Default](#)

## Incorrect Encoding for the Reader

In Mule, when you use components or connector operations that provide a [MIME Type](#) configuration (such as [Set Payload](#), [File Read](#), [HTTP Listener](#)), check that the encoding you set corresponds to the encoding used to write the payload. DataWeave reads the encoding that you set in the [MIME Type](#) configuration.

If you do not set the MIME type, DataWeave uses the default encoding provided by Mule through the [mule.encoding](#) system property.

## Encoding Used by the Writer Does Not Support Some Characters

Check that your writer encoding supports the characters that you are trying to write.

For example, writing the text "𠀀𠁇𠁈𠁉①" with the encoding [sjis](#) outputs "???.?????" because many of the input characters are not supported in that encoding (unlike UTF-8, for example).

```
%dw 2.0
output application/json encoding="sjis"
---
"𠀀𠁇𠁈𠁉①"
```

## Multipart/Form-Data Reader Does Not Support UTF-8 Characters by Default

If you use UTF-8 characters in the multipart filenames, the non-ASCII characters in the names are corrupted.

Set the following system property to enable support for UTF-8 in multipart: [-M-Dmail.mime.allowutf8=true](#)

For example, posting a multipart payload with `filename=ä\ufffd\ufffd.txt` without setting the [allowutf8](#) property to [true](#) produces the following unreadable text for the [filename](#) field:

```
{
  "Content-Disposition": {
    "name": "file",
    "filename": "ä\ufffd\ufffd.txt",
    "subtype": "form-data"
  },
  "Content-Type": "text/plain"
}
```

## DataWeave Warnings

The following are some common DataWeave warnings that appear in your dump file, along with some details to address these warnings.

### End of Input Was Reached

This warning appears because DataWeave treats the Unicode non-character `U+FFFF` as a special character that indicates the end of input.

To avoid this warning, replace the special character in the raw payload and then proceed as usual.

```
read(payload.^raw replace "\uFFFF" with "NonChar")
```

## Low Performance Issues

### application/dw Format

Using `output application/dw` format can impact the performance of transformations. This format is intended to help you debug the results of DataWeave transformations. It is significantly slower than other formats, so avoid using this format in production applications.

### multipart/form-data Content Parsing

For `multipart/form-data` inputs, accessing content of a large part can cause performance degradation, especially for high-memory formats like `application/xlsx`, because the interpreter attempts to parse the content of the part for further querying and analysis.

You can use the `^raw` selector to avoid parsing binary contents of parts.

The following example uses the `^raw` selector on specific operations over each part.

#### Input

```
--myboundary
Content-Disposition: form-data; name="file1"; filename="a.json"
Content-Type: application/json

{
  "title": "Java 8 in Action",
  "author": "Mario Fusco",
  "year": 2014
}
--myboundary
Content-Disposition: form-data; name="file2"; filename="a.xml"
Content-Type: application/xml

<doc>
  <title> Available for download! </title>
  <content> Really large content </content>
</doc>
--myboundary--
```

#### Source

```
%dw 2.0
input payload multipart boundary='myboundary'
output application/json
---
payload.parts mapObject ((value, key, index) ->
{
  (key): {
    fileName: payload.parts[index].headers.'Content-Disposition'.filename,
    rawContent: payload.parts[index].content.^raw
  }
})
```

## Define Functions

You can define your own DataWeave functions using the `fun` declaration in the header of a DataWeave script. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

### DataWeave Function Definition Syntax

To define a function in DataWeave use the following syntax:

```
fun myFunction(param1, param2, ...) = <code to execute>
```

- The `fun` keyword starts the definition of a function.
- `myFunction` is the name you define for the function.

Function names must be valid identifiers. For additional details about valid identifiers, see [Rules for Declaring Valid Identifiers](#).

- `(param1, param2, ..., paramn)` represents the parameters that your function accepts.

You can specify from zero to any number of parameters, separated by commas (,) and enclosed in parentheses.

- The `=` sign marks the beginning of the code block to execute when the function is called.
- `<code to execute>` represents the actual code that you define for your function.

### Example Function Definitions

The following example defines a DataWeave function that accepts a single string argument, calls the `upper` function using the received string argument, and then outputs the result:

*Example: Simple Function Definition*

```
%dw 2.0
output application/json
fun toUpper(aString) = upper(aString)
---
toUpper("hello")
```

*Output*

```
"HELLO"
```

The next example shows how the argument to a DataWeave function can be any DataWeave expression:

*Example: Using Expressions as Arguments*

```
%dw 2.0
output application/json
fun toUpper(aString) = upper(aString)
---
toUpper("h" ++ "el" ++ lower("LO") )
```

*Output*

```
"HELLO"
```

The following example tests the type of the argument passed to the function by performing [pattern matching](#) with the built-in [match](#) operation:

*Example: Function that Uses Pattern Matching*

```
%dw 2.0
output application/json
fun toUpper(aString)
= aString match {
  case is String -> upper(aString)
  else -> null
}
---
toUpper("h" ++ "el" ++ lower("LO") )
```

*Output*

```
"HELLO"
```

This example creates a function that reformats a numeric string into a common phone number

format:

*Example: toPhoneFormat() Function*

```
%dw 2.0
output application/json
fun toPhoneFormat(str: String) =
  "(" ++ str[0 to 2] ++ ") " ++ str[3 to 5] ++ "-" ++ str[6 to 9]
---
toPhoneFormat("1234567890")
```

*Output*

```
"(123) 456-7890"
```

## Type Constraint Definition

You can define type constraint expressions when you declare functions to ensure that the arguments for a function call respect the defined type constraint.

To define type constraints for functions, use the following syntax:

```
fun myFunction(param1: Type, param2: Type): ResultType = <code to execute>
```

*Example: Function with Type Constraints*

```
%dw 2.0
output application/json
fun toUser(id: Number, userName: String): String = "you called the function toUser!"
```

In this example, the type constraints define that the function `toUser` accepts only a first argument of type `Number` and a second argument of type `String`.

Also, the function specifies a constraint for the result to be of type `String`. This return type constraint does not affect the function call; rather, this constraint applies to the function code to validate that the function generates a result of the defined type.

You can find more details about the DataWeave type system in <https://docs.mulesoft.com/dataweave/latest/dataweave-type-system>.

## Optional Parameters Definition

You can define optional parameters by assigning a default value to them. The following example shows a function that defines one parameter that is mandatory (`name`) and a second parameter that is optional (`countryCode`) and has a default value:

### *Example: Function with Optional Parameters*

```
%dw 2.0
output application/json
fun createUserData(name, countryCode = "US") = { "User name" : name, "Location":countryCode}
---
[createUserData("Cristian", "AR"), createUserData("Max The Mule")]
```

### *Output*

```
[
{
  "User name": "Cristian",
  "Location": "AR"
},
{
  "User name": "Max The Mule",
  "Location": "US"
}]
```

### **Optional Parameter Order**

DataWeave enables you to define optional parameters at the beginning or at the end of the parameter definition:

### *Example: Functions with Optional Parameters*

```
%dw 2.0
output application/json
fun optionalParamsLast(a, b = 2, c = 3)
fun optionalParamsFirst(a = 1, b = 2, c)
```

When you call a function, the arguments are assigned from left to right. However, if you define optional parameters first, then the arguments are assigned from right to left. In addition, if all the parameters in a function are optional, the assignment is also from right to left:

*Example: Optional Parameter Assignment*

```
%dw 2.0
output application/json
fun optionalParamsLast(param1, param2 = 2, param3 = 3) =
    { "param1": param1, "param2": param2, "param3": param3}
fun optionalParamsFirst(param1 = 1, param2 = 2, param3) =
    { "param1": param1, "param2": param2, "param3": param3}
fun allParametersOptional(param1 = 1, param2 = 2, param3 = 3) =
    { "param1": param1, "param2": param2, "param3": param3}
---
{
    "optionalParamsLast(A)":optionalParamsLast('A'),
    "optionalParamsLast(A, B)":optionalParamsLast('A', 'B'),
    "optionalParamsLast(A, B, C)":optionalParamsLast('A', 'B', 'C'),
    "optionalParamsFirst(A)":optionalParamsFirst('A'),
    "optionalParamsFirst(A, B)":optionalParamsFirst('A', 'B'),
    "optionalParamsFirst(A, B, C)":optionalParamsFirst('A', 'B', 'C'),
    "allParametersOptional(A)":allParametersOptional('A'),
    "allParametersOptional(A, B)":allParametersOptional('A', 'B'),
    "allParametersOptional(A, B, C)":allParametersOptional('A', 'B', 'C')
}
```

*Output*

```
{  
    "optionalParamsLast(A)": {  
        "param1": "A",  
        "param2": 2,  
        "param3": 3  
    },  
    "optionalParamsLast(A, B)": {  
        "param1": "A",  
        "param2": "B",  
        "param3": 3  
    },  
    "optionalParamsLast(A, B, C)": {  
        "param1": "A",  
        "param2": "B",  
        "param3": "C"  
    },  
    "optionalParamsFirst(A)": {  
        "param1": 1,  
        "param2": 2,  
        "param3": "A"  
    },  
    "optionalParamsFirst(A, B)": {  
        "param1": 1,  
        "param2": "A",  
        "param3": "B"  
    },  
    "optionalParamsFirst(A, B, C)": {  
        "param1": "A",  
        "param2": "B",  
        "param3": "C"  
    },  
    "allParametersOptional(A)": {  
        "param1": 1,  
        "param2": 2,  
        "param3": "A"  
    },  
    "allParametersOptional(A, B)": {  
        "param1": 1,  
        "param2": "A",  
        "param3": "B"  
    },  
    "allParametersOptional(A, B, C)": {  
        "param1": "A",  
        "param2": "B",  
        "param3": "C"  
    }  
}
```

## Type Parameters Definition

DataWeave function definitions support the use of type parameters, which are similar to generics in some programming languages.

*Example: Function that Uses Type Parameters*

```
%dw 2.0
output application/json

fun toArray<T>(x: T): Array<T> = [x]
---
toArray(2)
```

DataWeave also supports adding type parameter constraints, which limit accepted inputs for a function.

*Example Function that Uses Type Parameters with Constraints*

```
%dw 2.0
output application/json

fun getName<T <: { name: String }>(x: T): String = x.name
---
{
  name: getName({ name: "Andrés" })
  //name: getName({ age: 20 }) invalid call as the parameter does not fulfill the
constraint
}
```

## Function Overloading

DataWeave enables you to create multiple functions with the same name but different parameters. This feature is useful for defining different behaviors based on the arguments of a function call.

Parameters in overloaded functions differ in number or type. To understand more about defining type constraints on function parameters, see [Type System](#).

DataWeave uses the first function it finds that accepts the arguments of the function call, based on the order in which the functions are declared in the script.

### Example: Function that Uses Function Overloading

```
%dw 2.0
output application/json

fun toUpper(a: String) = upper(a)
fun toUpper(a: Any) = null
fun toUpper(a: String, b: Number) = upper(a) ++ b as String
---
toUpper("hi!")
```

The argument of the function call `toUpper("hi!")` matches the types String and Any, so it is possible to call the function with the first two definitions. However, DataWeave executes the function `fun toUpper(a: String) = upper(a)` because that function is defined before the one that uses the type Any.

*Output of toUpper("hi!")*

```
"HI!"
```

*Output of toUpper(true)*

```
null
```

*Output of toUpper("age: ", 26)*

```
"AGE: 26"
```

## See Also

[Create Custom Modules and Mappings](#)

[DataWeave Reference](#)

## Create Custom Modules and Mappings

In addition to using the built-in DataWeave function modules (such as `dw::Core` and `dw::Crypto`), you can also create and use custom modules and mapping files. The examples demonstrate common data extraction and transformation approaches. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

You write modules and mapping files in a DataWeave Language (`.dwl`) file and import into your Mule app through DataWeave scripts in Mule components. Both modules and mapping files are useful when you need to reuse the same functionality or feature over and over again.

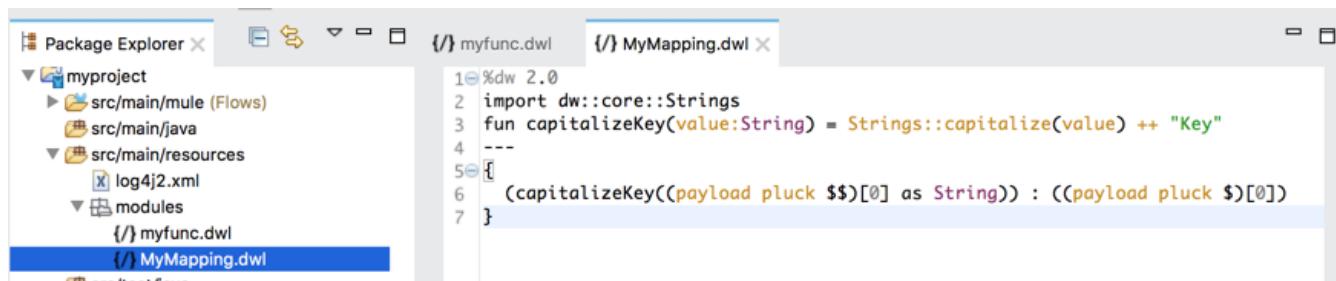
- Custom modules can define functions, variables, types, and namespaces. You can import these modules into a DataWeave script to use the features.
- Custom mapping files are a type of module that contains a complete DataWeave script that you can import and use in another DataWeave script or reference in a Mule component.

Fields in many Mule connectors and components accept DataWeave expressions and scripts.

Note that if you want to import and use a built-in DataWeave function module, and not a custom one, see [DataWeave Function Reference](#).

## Creating and Using DataWeave Mapping Files

You can store a DataWeave transformation in a `.dwl` mapping file (mapping module), then import the file into another DataWeave script. Mapping files can be executed through the Transform Message component, or you can import them into another mapping and execute them through the `main` function.



```
%dw 2.0
import dw::core::Strings
fun capitalizeKey(value:String) = Strings::capitalize(value) ++ "Key"
---
main {
    (capitalizeKey((payload pluck $$)[0] as String)) : ((payload pluck $)[0])
}
```

Figure 3. Example: DataWeave Mapping File in a Studio Project

1. In your Studio project, set up a subfolder and file for your mapping module:
  - You can create a subfolder in `src/main/resources` by navigating to New → Folder → [your\_project] → `src/main/resources`, then adding a folder named `modules`.
  - You can create a new file for your module in that folder by navigating to New → File → [your\_project] → `src/main/resources/modules`, then adding a DWL (DataWeave language) file such as `MyMapping.dwl`.

Saving the module within `src/main/resources` makes it accessible for use in any DataWeave script within the Mule app in that project.

2. Create your function in your mapping file, for example:

*Example: Mapping File Content*

```
%dw 2.0
import dw::core::Strings
fun capitalizeKey(value:String) = Strings::capitalize(value) ++ "Key"
---
payload mapObject ((value, key) ->
{
    (capitalizeKey(key as String)) : value
})
)
```

3. Save your DWL function module file.

### Using a Mapping File in a DataWeave Script

To use a mapping file, you need to import it into a DataWeave script and use the `main` function to access the body of the script in the mapping file.

Assume that you have created the `MyMapping.dwl` file in `/src/main/resources/modules` that contains this script.

To import and use the body expression from the `MyMapping.dwl` file (above) in DataWeave Mapping file, you need to do this:

- Specify the `import` directive in the header.
- Invoke the `MyMapping::main` function. The function expects an input that follows the structure of the input that the mapping file uses. For example, the body of `MyMapping.dwl` expects an object of the form `{"key" : "value"}`.

*Example: Importing and Using the Mapping in a DataWeave Script*

```
%dw 2.0
import modules::MyMapping
output application/json
---
MyMapping::main(payload: { "user" : "bar" })
```

Here is the result:

#### Output

```
{
  "UserKey": "bar"
```

Even though the `capitalizeKey` function is private, it is still used through the `main` function call, and the DataWeave mapping file is also able to import and reuse the `dw::core::Strings` module.

## Creating and Using a Custom Module

The steps for creating a custom DataWeave module are almost identical to the steps for creating a custom mapping file. The only difference is the contents of the `.dw1` file. Unlike a typical DataWeave script or mapping file, a custom DataWeave module cannot contain an `output` directive, body expression, or the separator (---) between header and body sections. (For guidance with mappings, see [Creating and Using DataWeave Mapping Files](#).)

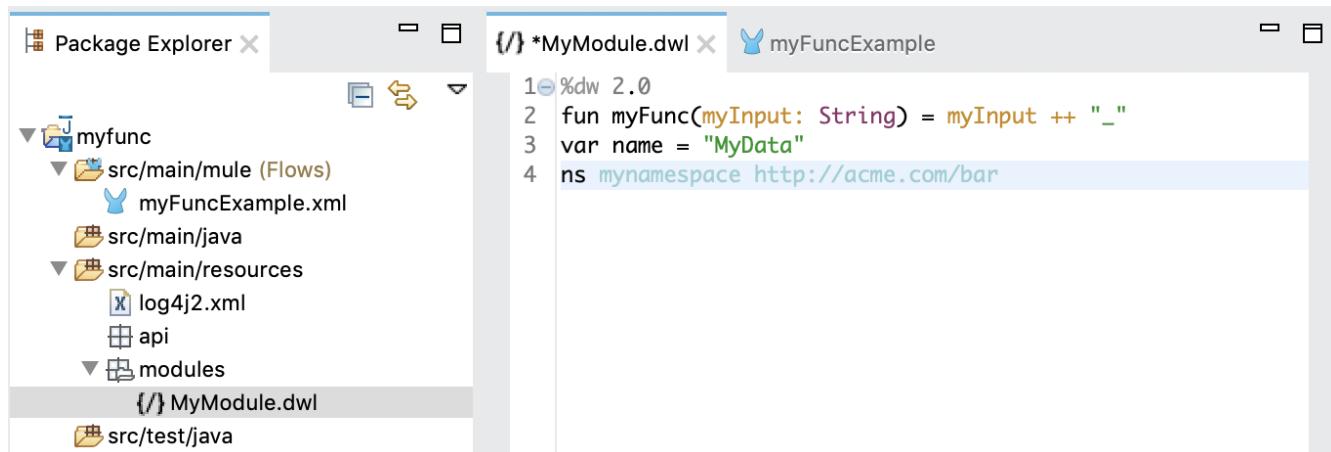


Figure 4. Example: Custom Module in a Studio Project

A custom module file can only contain `var`, `fun`, `type`, and `ns` declarations, for example:

### Example: Custom DataWeave Module

```
%dw 2.0
var name = "MyData"
fun myFunc(myInput: String) = myInput ++ "_"
type User = {
    name: String,
    lastname: String
}
ns mynamespace http://acme.com/bar
```

When you import a custom module into another DataWeave script, any functions, variables, types, and namespaces defined in the module become available for use in the DataWeave body. In the next example, a DataWeave script:

- Imports the module `MyModule` through the `import` directive in the header. In this case, the imported module is stored in a Studio project path `src/main/resources/modules/MyModule.dw1`
- Calls a function in `MyModule` by using `MyModule::myFunc("dataweave")`.

### Example: Importing and Using a Custom DataWeave Module

```
%dw 2.0
import modules::MyModule
output application/json
---
MyModule::myFunc("dataweave") ++ "name"
```

There are several ways to import a module or elements in it:

- Import the module, for example: `import modules::MyModule`. In this case, you must include the name of the module when you call the element (here, a function) in it, for example: `MyModule::myFunc`.
- Import all elements from the module, for example: `import * from modules::MyModule`. In this case, you do not need to include the name of the module when you call the element. For example: `myFunc("dataweave") ++ "name"` works.
- Import specific elements from a module, for example: `import myFunc from modules::MyModule`. In this case, you do not need to include the name of the module when you call the element. For example: `myFunc("dataweave") ++ "name"` works. You can import multiple elements from the module like this, for example: `import myFunc someOtherFunction from modules::MyModule` (assuming both `myFunc` and `someOtherFunction` are defined in the module).

#### *Output*

```
"dataweave_name"
```

## Assigning a Local Alias for an Imported Element

To avoid name clashes, you can use `as` to assign an alias for a custom module or its elements when you import the module into a DataWeave script.

Assume that you define a custom module like this one in the file `MyModule.dwl`:

#### *Example: Custom Module*

```
%dw 2.0
fun myFunc(name:String) = name ++ "_"
var myVar = "Test"
```

When you import the custom module into a DataWeave script, you can create aliases to elements in the custom module, for example:

#### *Example: Applying an Alias to Imported Elements*

```
%dw 2.0
import myFunc as appendDash, myVar as weaveName from modules::MyModule
var myVar = "Mapping"
output application/json
---
appendDash("dataweave") ++ weaveName ++ "_" ++ myVar
```

The script returns `"dataweave_Test_Mapping"`.

You can create an alias to the imported module, for example:

### *Example: Applying an Alias to an Imported Module*

```
%dw 2.0
import modules::MyModule as WeaveMod
output application/json
---
WeaveMod::myFunc("dataweave")
```

## Referencing a DWL File

You can use DWL files directly in Mule connectors and components.

See [dwl File](#) for details.

## Writing Documentation for Custom DataWeave Modules

DataWeave supports use of the AsciiDoc text format for documenting your functions, types, annotations, and the modules that contain them.

### Documenting Your DataWeave Functions

DataWeave defines a multi-section template for documenting function code. The template provides sections for the following:

- A description of the function.
- Parameter descriptions in an AsciiDoc table format.
- Example code that includes sections for any input, the DataWeave script, and the resulting output.

To define parameter and example sections below the function description, the template uses AsciiDoc heading grammar:

- **== Parameters**
- **== Example**
  - **==== Source**
  - **==== Input**
  - **==== Output**

The following function template provides guidance on documenting the function. As the template notes, some sections are optional.

```

/**
 * %Replace with your function description%
 *
 *
 * %Add additional information to your function description% (optional section)
 *
 * === Parameters (optional section)
 *
 * [%header, cols="1,1,3"]
 * |===
 * | Name | Type | Description
 * | %`The parameter name`% | %`The parameter type`% | %The parameter description%
 * (one row per param)
 * |===
 *
 * === Example (optional section)
 *
 * %The example description% (optional)
 *
 * === Source (optional section)
 *
 * [source,%The language%,linenums] (optional)
 * ----
 * YOUR CODE
 * ----
 *
 * === Input (optional section)
 *
 * The input description (optional)
 *
 * [source,%The language%,linenums] (optional)
 * ----
 * YOUR CODE
 * ----
 *
 * === Output (optional section)
 *
 * %The output description% (optional)
 *
 * [source,%The language%,linenums] (optional)
 * ----
 * YOUR CODE
 * ----
 */

```

## Description Section

This section provides a description of the function. The section starts at the top of the comments and extends to the line before the first section heading. If there is no section header, the description extends to the end of the comments for the function.

A description consists of the following parts:

- Short description: The first paragraph, which provides the primary description of the function.
- Long description: Any additional information about the function. This optional description begins two line breaks below the short description.

See the [example](#) that includes a description with sections for the parameters and examples.

Each part is useful for auto-generated documentation.

### Parameters Section

The optional `==== Parameters` section describes parameters of the function.

The template uses AsciiDoc table format to document each parameter in a table row.

```
[%header, cols="1,1,3"]
=====
| Name | Type | Description
| yourParameter1 | The parameter type | Your description here.
| yourParameter2 | The parameter type | Your description here.
=====
```

### Example Section

You can use zero or more `==== Example` sections to provide any examples needed to illustrate how your function works.

This section contains the following optional subsections:

- `==== Input` for input to the DataWeave script.
- `==== Source` for the DataWeave script.
- `==== Output` for output generated by the script.

All subsections follow the same template, which includes an optional description and an optional code section:

```
Your section description

[source,%The language%,linenums]
-----
YOUR CODE
-----
```

The following example shows the [log](#) function documentation.

```
/**
```

```

* Without changing the value of the input, `log` returns the input as a system
* log. So this makes it very simple to debug your code, because any expression or
subexpression can be wrapped
* with *log* and the result will be printed out without modifying the result of the
expression.
* The output is going to be printed in application/dw format.
* (1)
*
* The prefix parameter is optional and allows to easily find the log output.
*
*
* Use this function to help with debugging DataWeave scripts. A Mule app
* outputs the results through the 'DefaultLoggingService', which you can see
* in the Studio console.
*
* === Parameters
*
* [%header, cols="1,13"]
* |===
* | Name | Type | Description
* | `prefix` | `String` | An optional string that typically describes the log.
* | `value` | `T` | The value to log.
* |===
*
* === Example
*
* This example logs the specified message. Note that the 'DefaultLoggingService'
* in a Mule app that is running in Studio returns the message
* 'WARNING - "Houston, we have a problem,"' adding the dash '-' between the
* prefix and value. The Logger component's 'LoggerMessageProcessor' returns
* the input string '"Houston, we have a problem."', without the 'WARNING' prefix.
*
* ===== Source
*
* [source,DataWeave,linenums]
* ----
* %dw 2.0
* output application/json
* ---
* log("WARNING", "Houston, we have a problem")
* ----
*
* ===== Output
*
* 'Console Output'
*
* [source,XML,linenums]
* ----
* "WARNING - Houston, we have a problem"
* ----
*

```

```

* `Expression Output`
*
* [source,XML,linenums]
* ----
* "Houston, we have a problem"
* ----
*
* === Example
*
* This example shows how to log the result of expression `myUser.user` without
modifying the
* original expression `myUser.user.friend.name`.
*
* ===== Source
*
* [source,DataWeave,linenums]
* ----
* %dw 2.0
* output application/json
*
* var myUser = {user: {friend: {name: "Shoki"}, id: 1, name: "Tomo"}, accountId:
"leansh" }
* ---
* log("User", myUser.user).friend.name
* ----
*
* ===== Output
*
* `Console output`
*
* [source,console,linenums]
* ----
* User - {
*   friend: {
*     name: "Shoki"
*   },
*   id: 1,
*   name: "Tomo"
* }
* ----
*
* `Expression Output`
*
* [source,DataWeave,linenums]
* ----
* "Shoki"
* ----
*/

```

## Documenting Your DataWeave Modules

The DataWeave module description does not follow a structured template.

The only requirement is to place the documentation above the DataWeave version tag.

The following example documents the `dw::Core` module:

```
/**  
 * This module contains core DataWeave functions for data transformations.  
 * It is automatically imported into any DataWeave script. For documentation  
 * on DataWeave _1.0_ functions, see  
 * https://docs.mulesoft.com/dataweave/1.2/dataweave-operators[DataWeave Operators].  
 */  
%dw 2.0
```

## Documenting Your Annotation and Types

Annotation and Types documentation do not follow a structured template.

The only requirement is to write the descriptions above the Annotation or Type definition.

The following example documents the `@StreamCapable` annotation.

```
/**  
 * Annotation that marks a parameter as stream capable, which means that this  
 * field will consume an array of objects in a forward-only manner.  
 */  
@AnnotationTarget(targets = ["Parameter", "Variable"])  
annotation StreamCapable()
```

The following example documents the `EncodingSettings` type found in the `dw::extension::DataFormat` module.

```
/**  
 * Represents encoding settings.  
 */  
@Since(version = "2.2.0")  
type EncodingSettings = {  
    /**  
     * Encoding that the writer uses for output. Defaults to "UTF-8".  
     */  
    encoding?: String {defaultValue: "UTF-8"}  
}
```

## See Also

- [Reusing Types from DataWeave Modules](#)

# Working with Functions and Lambdas in DataWeave

In DataWeave, functions and lambdas (anonymous functions) can be passed as values or be assigned to variables. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

When using lambdas within the body of a DataWeave file in conjunction with a function such as `map`, its attributes can either be explicitly named or left anonymous, in which case they can be referenced as `$`, `$$`, etc.

## Declare and Invoke a Function

You can declare a function in the header or body of a DataWeave script by using the `fun` keyword. Then you can invoke the function at any point in the body of the script.

You refer to functions using this form: `functionName()` or `functionName(arg1, arg2, argN)`

You can pass an expression in between the parentheses for each argument. Each expression between the parentheses is evaluated, and the result is passed as an argument used in the execution of the function body.

### *Input*

```
{  
    "field1": "Annie",  
    "field2": "Point",  
    "field3": "Stuff"  
}
```

### *Transform*

```
%dw 2.0  
output application/json  
fun toUser(obj) = {  
    firstName: obj.field1,  
    lastName: obj.field2  
}  
---  
{  
    "user" : toUser(payload)  
}
```

## *Output*

```
{  
  "user": {  
    "firstName": "Annie",  
    "lastName": "Point"  
  }  
}
```

## Specify Type Parameters

Starting in DataWeave syntax version 2.5, you can specify the type parameters of a function at the call site.

## *Input*

```
{  
  "measures": [1,2,4,1,5,2,3,3]  
}
```

## *Transform*

```
%dw 2.5  
output application/json  
  
fun max<T>(elems: Array<T>): T = elems reduce ((candidate: T, currentMax = elems[0])  
-> if (candidate > currentMax) candidate else currentMax)  
---  
{  
  max: max<Number>(measures)  
}
```

## *Output*

```
{  
  "max": 5  
}
```

## Assign a Lambda to a Var

You can define a function as a variable with a constant directive through `var`

### *Input*

```
{  
  "field1": "Annie",  
  "field2": "Point",  
  "field3": "Stuff"  
}
```

### *Transform*

```
%dw 2.0  
output application/json  
var toUser = (user) -> {  
  firstName: user.field1,  
  lastName: user.field2  
}  
---  
{  
  "user" : toUser(payload)  
}
```

### *Output*

```
{  
  "user": {  
    "firstName": "Annie",  
    "lastName": "Point"  
  }  
}
```

## Use Named Parameters in a Lambda

This example uses a lambda with an attribute that is explicitly named as `name`.

### *Input*

```
%dw 2.0  
output application/json  
var names = ["john", "peter", "matt"]  
---  
users: names map((name) -> upper(name))
```

### *Transform*

```
{  
  "users": ["JOHN", "PETER", "MATT"]  
}
```

## Use Anonymous Parameters in a Lambda

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

*Transform*

```
%dw 2.0
output application/json
var names = ["john", "peter", "matt"]
---
users: names map upper($)
```

*Output*

```
{
  "users": ["JOHN", "PETER", "MATT"]
}
```

## See Also

[Supported Data Formats](#)

[Scripts](#)

## Memory Management

When processing large files through DataWeave in Mule runtime engine, there are a few things you can set up to fine-tune how much memory will be used and when.

### RAM vs Disk Usage

DataWeave keeps small files in memory, but after a configurable threshold uses disk space to avoid running out of memory. The created buffer files are placed in a default temporary directory. If you want to store those files in a custom directory instead, you can specify the directory by using the `java.io.tmpdir` property.

The buffer files remain in the temporary directory until the streams that reference the files are closed. Closure normally occurs when flows complete their execution, so many buffer files in the temporary folder can remain in use during long-running and concurrent executions.

Three types of DataWeave buffer files are generated:

- `dw-buffer-output-${count}.tmp`

Used to store the output of a transformation when the result is bigger than the threshold **1572864 bytes**. To change this threshold value, add the system property `com.mulesoft.dw.max_memory_allocation` and assign it the number of bytes you want as your new threshold. Mule runtime engine deletes the file when the value is no longer referenced, JVM GC

collects it or when the Mule Event finishes executing.

- **dw-buffer-input-`#{count}`.tmp**

Used to store the input of a transformation when it is bigger than the in-memory buffer, which is **1572864 bytes** by default. This is analogous to the **dw-buffer-output-`#{count}`.tmp** file, but for input data.

- **dw-buffer-index-`#{count}`.tmp**

Used to store index information of a value being read. This file helps DataWeave access data quickly. Mule runtime engine deletes the file when the execution of the transformation ends or, in a streaming use case like the foreach loop, when the stream ends (when foreach finishes its execution).

## System properties:

Because you can define system properties in several ways, see [system properties](#) for further details.

- **com.mulesoft.dw.buffersize**

This system property determines the size in bytes of the in-memory input and output buffers used in DataWeave to keep the processed inputs and outputs. The default buffer size is 8192 bytes (8 KB). This property receives a number (size in bytes), for example: **-Dcom.mulesoft.dw.buffersize=8192**.

- **com.mulesoft.dw.directbuffer.disable**

Introduced in Mule 4.2.2, this option controls whether DataWeave uses off-heap memory (the default) or heap memory. DataWeave uses off-heap memory for internal buffering. However, this setting can cause problems on machines that have only a small amount of memory. This property receives a boolean, for example: **-Dcom.mulesoft.dw.directbuffer.disable=true**.

- **com.mulesoft.dw.memory\_pool\_size**

Since Mule 4.3.0, DataWeave buffers use off-heap memory from a pool up to a defined size and allocates the rest using heap memory. This property determines the number of slots in the pool. The default number of slots is 60. This property receives a number (slots in the pool), for example: **-Dcom.mulesoft.dw.memory\_pool\_size=60**.

- **com.mulesoft.dw.max\_memory\_allocation**

Introduced in Mule 4.3.0, this property determines the maximum amount of memory to be used before it switches to disk. If a payload exceeds this size, it is stored in **dw-buffer-input-`#{count}`.tmp** and **dw-buffer-output-`#{count}`.tmp** temporary files. It also determines the size in bytes of each slot in the pool of off-heap memory. The default size is 1572864 bytes (1.5 MB). This property receives a number (size in bytes), for example: **-Dcom.mulesoft.dw.max\_memory\_allocation=1572864**.

- **com.mulesoft.dw.buffer.memory.monitoring** (experimental)

Introduced in Mule 4.3.0, when this property is enabled, a message is logged each time a slot from the memory pool is taken or released. Note that it may be removed or change its behavior in future versions. This property receives a boolean, for example: **-Dcom.mulesoft.dw.buffer.memory.monitoring=true**.

## See Also

- [Scripts](#)

- [Selectors](#)

# Versioning Behavior in DataWeave

New minor versions of DataWeave can introduce changes that modify the way the language behaves. You can use DataWeave system properties that behave as *compatibility flags* to continue using existing scripts that depend on the previous behavior. However, each flag defines a deletion version, which identifies the version in which the flag is no longer available and the behavior that it controls defaults to the changed behavior.

Specifying a DataWeave language level indicates to the DataWeave runtime what compatibility flags are available. To upgrade and maintain backward compatibility, you can set the language level of a previous version so that the flags available on that version remain visible even if they expire in the DataWeave version to which you upgrade. The language level defaults to the running DataWeave version and has to be at most the same as the runtime version.

For Mule applications, the language level is determined by the application's `minMuleVersion` setting (see [Feature Flags Reference](#) in the Mule documentation). Given the `minMuleVersion`, the language level is computed by subtracting two major versions, as the following table shows:

minMuleVersion	Language Level
4.5	2.5
4.4	2.4
4.3	2.3

Setting the Mule version configures the application to use the corresponding DataWeave language level. This setting is independent of the DataWeave syntax version, which is set through a DataWeave script's `%dw` directive. For details, see [DataWeave Header](#). For example, the `com.mulesoft.dw.xml_reader.honourMixedContentStructure` system property is a compatibility flag for language level 2.4, when it was introduced, and it is removed in DataWeave 2.5. In DataWeave 2.5, the property is *visible only* if you set the language level to 2.4. Otherwise, the property defaults to its new behavior in 2.5.

# DataWeave Examples

The following DataWeave examples demonstrate common data extraction and transformation approaches. There are DataWeave code examples of how to transform data, and also examples of Mule applications that implement DataWeave transformations. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

## DataWeave Code Examples

These examples show how to use standalone DataWeave to extract and transform data.

Example	Description
<a href="#">Extract Data</a>	Shows common selector expressions for extracting values from a data source, such as a Mule message.
<a href="#">Select XML Elements</a>	Uses a single-value DataWeave selector (.) to extract data from XML elements.
<a href="#">Set a Default Value</a>	Shows some common ways to set default values.
<a href="#">Set Reader and Writer Configuration Properties</a>	Shows how to use and change the settings for reading and writing data formats, such as <code>application/csv</code> .
<a href="#">Perform a Basic Transformation</a>	Uses selectors to perform a basic format transformation. It does not use any functions.
<a href="#">Map Data</a>	Uses <code>map</code> to reorganize fields in a JSON array. It uses <code>as</code> to coerce a String into a Number.
<a href="#">Map and Flatten an Array</a>	Uses <code>flatMap</code> to map objects in an array and flatten the resulting array.
<a href="#">Map an Object</a>	Uses <code>mapObject</code> to go through the keys and values in each object, and set all the keys to upper case.
<a href="#">Map Objects Key</a>	Uses the <code>mapObject</code> function to iterate through an array of objects and appends a new object that matches the value of the specified criteria.
<a href="#">Map the Objects within an Array</a>	Uses the multi-value selector to pick out the keys in an object that are named "book", these are returned as an array. It then uses <code>map</code> to iterate through the array this creates.

Example	Description
<a href="#">Dynamic Map Based on a Definition</a>	Transforms the payload according to definitions sent in a variable. It defines a function that applies the logic defined in the variable, using <code>map</code> and <code>default</code> .
<a href="#">Rename Keys</a>	Renames some keys in a JSON object while retaining the names of all other keys in the output. Uses <code>mapObject</code> , <code>if</code> , <code>as</code> , and <code>and</code> .
<a href="#">Output a Field When Present</a>	Uses <code>if</code> with <code>map</code> to determine whether to include a field in the output. You might use it to exclude fields that contain sensitive data.
<a href="#">Change Format According to Type</a>	Uses <code>mapObject</code> to apply changes to the keys in an object, depending on the type of their corresponding values. Uses <code>camelize</code> , <code>capitalize</code> , and <code>pluralize</code> depending on the <code>if</code> and <code>else if</code> statement.
<a href="#">Regroup Fields</a>	Uses <code>groupBy</code> , <code>mapObject</code> , and <code>map</code> to reorganize JSON and XML fields.
<a href="#">Zip Arrays Together</a>	Uses <code>zip</code> to rearrange pairs of similar arrays so that they form a series of tuples with the matching values from each.
<a href="#">Pick Top Elements</a>	Uses <code>groupBy</code> and <code>map</code> to sort a list of candidates according to their score at a test, then it splits the array to select only the top candidates.
<a href="#">Change the Value of a Field</a>	Masks sensitive data by changing values of some keys to asterisks (***)`. Uses <code>mapObject</code> , <code>if</code> , and <code>else</code> .
<a href="#">Exclude Fields from the Output</a>	Shows how to exclude unwanted elements from the output. Uses the <code>-</code> (remove) and <code>mapObject</code> functions.
<a href="#">Use Constant Directives</a>	Defines a series of constant strings and numbers in the header, these are used to filter input and to concatenate into URLs. Uses <code>map</code> , <code>if</code> and <code>++</code> to concatenate strings.
<a href="#">Define a Custom Addition Function</a>	Defines a function that obtains totals and subtotals through the <code>accumulator</code> function. Also performs additions, subtractions, and multiplications of numeric values.
<a href="#">Define a Function That Flattens Data in a List</a>	Uses <code>reduce</code> , <code>map</code> , <code>if</code> , and <code>splitBy</code> to modify and conditionally output fields from a list.
<a href="#">Flatten Elements of Arrays</a>	Uses <code>flatten</code> , the object constructor braces <code>({})</code> , and selectors to flatten subarrays and nested elements in arrays. Uses concatenation <code>(++)</code> to combine arrays before flattening them and uses the <code>..*</code> descendants selector on a key to select values from a flattened list.

Example	Description
<a href="#">Use Regular Expressions</a>	Shows uses of regular expressions in arguments to several DataWeave functions.
<a href="#">Output self-closing XML tags</a>	Uses <code>inlineCloseOn="empty"</code> to close empty tags (for example, <code>&lt;element2/&gt;</code> ).
<a href="#">Insert an Attribute into an XML Tag</a>	Uses the <code>@(key:value)</code> syntax to create an XML attribute.
<a href="#">Remove Certain XML Attributes</a>	Defines a function that recursively checks an element and all of its children for a specific XML attribute and removes it with <code>-</code> . It also uses <code>mapObject</code> , <code>if</code> and <code>is</code> (to match types).
<a href="#">Pass XML Attributes</a>	Pass XML attributes from the input source payload to the output XML. It uses the dynamic attribute expression <code>@dynamicAttributes</code> to create the attributes of the new output tag by selecting the attributes dynamically from the input.
<a href="#">Include XML Namespaces</a>	Defines multiple XML namespaces and references these in each tag.
<a href="#">Remove Objects Containing Specified Key-Value Pairs</a>	Removes all objects that contain a set of key-value pairs from an array of objects. Uses <code>filter</code> with <code>contains</code> and a <code>not</code> operator.
<a href="#">Reference Multiple Inputs</a>	References data on the payload, a message attribute and a variable. It then processes these through <code>map</code> , <code>filter</code> and multiplications.
<a href="#">Merge Fields from Separate Objects</a>	Reference data that arrives in multiple separate payloads from one single Mule event. Filters the output based on a unique identifier. Uses <code>map</code> , <code>using</code> , <code>as</code> , and <code>filter</code> functions.
<a href="#">Parse Dates</a>	Defines a function that normalizes conflicting date formats into a single, common format. Uses <code>mapObject</code> , <code>replace</code> and <code>as</code> to coerce a data type.
<a href="#">Add and Subtract Dates</a>	Performs multiple math operations combining different types related to date and time.
<a href="#">Change a Time Zone</a>	Uses <code>&gt;&gt;</code> with a time zone ID to change a time zone and provides a list of available time zone IDs.
<a href="#">Format Dates and Times</a>	Uses formatting characters such as <code>uuuu/MM/dd</code> to change the format of dates and times. Creates a date-time format using a custom DataWeave type.

Example	Description
<a href="#">Work with Multipart Data</a>	Uses <code>mapObject</code> to iterate over a multipart payload and extract data from each part. It also uses <code>read</code> with a <code>boundary</code> value to read the multipart content.
<a href="#">Conditionally Reduce a List Via a Function</a>	Defines a function that reduces a list of elements into one field. It then calls this function only when a certain field has values. Uses <code>map</code> , <code>reduce</code> , <code>splitBy</code> , <code>if</code> and <code>++</code> to append to an array.
<a href="#">Pass Functions as Arguments</a>	Defines a function that expects to receive two inputs: a function to apply and an element to apply it on. The function is also recursively applied to the element's children. It uses <code>mapObject</code> lower is <code>if/else</code> .

## DataWeave Code in MuleSoft Applications Examples

These examples show how to use DataWeave to extract and transform data in MuleSoft applications.

Example	Description
<a href="#">Change a Script's Output Mime Type</a>	Shows how to customize the MIME type of a given format output, including an example of a JSON output with <code>application/problem+json</code> MIME type.
<a href="#">Look Up Data in an Excel (XLSX) File (Mule)</a>	Uses <code>filter</code> to return rows that contain a specified value.
<a href="#">Look Up Data in CSV File</a>	Uses <code>filter</code> with <code>map</code> to look up data in a CSV file and return a country code for each calling code found within an input array of phone numbers.
<a href="#">Decode and Encode Base64 (Mule)</a>	Converts a file stream into Base64 and converts a Base64 string into a file stream. Uses a PDF file as the input and output.
<a href="#">Call Java Methods (Mule)</a>	Calls Java methods from a Java class in a Mule project.
<a href="#">Read and Write a Flat File (Mule)</a>	Use DataWeave to read and write flat files in a Mule application.
<a href="#">Use a Reader Property through a Connector</a>	Use a reader property through a Connector operation.
<a href="#">Use Dynamic Writer Properties (Mule)</a>	Use a Mule variable as a configuration value.
<a href="#">Extract Key/Value Pairs with Pluck Function</a>	Use the DataWeave <code>pluck</code> function to extract key-value pairs from a JSON payload and store them into Mule event variables.

## See Also

- [DataWeave Language](#)
- [Scripts](#)
- [DataWeave Reference](#)

## Extract Data

DataWeave can select data from DataWeave objects and arrays, variables that store that data, and the output of DataWeave functions when that output is an array or object. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

More precisely, a DataWeave selector operates within a context, which can be a reference to the variable that stores the data, an object literal, an array literal, or the invocation of a DataWeave function. You can use selectors in Mule modules, connectors, and components that accept DataWeave expressions.

When DataWeave processes a selector, it sets a new context (or scope) for subsequent selectors, so you can navigate through the complex structures of arrays and objects using chains of selectors. The depth of the selection is limited only by the depth of the current context.

Supported variable references are [Mule Runtime variables](#), such as `payload` and `attributes`, and [DataWeave variables](#) that store arrays or objects. A simple example is `payload.myKey` where the payload is the object `{"myKey" : 1234}`, so the result is `1234`.

A selector can act on the invocation of a function, such as the DataWeave `read` function. For example, `read('{"A":"B"}','application/json')."A"` returns `"B"`.

## Use Selectors on DataWeave Arrays and Objects

In DataWeave, selectors extract values from within a DataWeave object (such as `{ myKey : "myValue" }`) or array (such as `[1,2,3,4]` or `[ { "myKey" : "1234" }, { "name" : "somebody" } ]`).

The following DataWeave script uses the single-value selector `(.)` to retrieve values from the object and array defined by the variables `myObject` and `myArray`.

*DataWeave Script:*

```
%dw 2.0
var myObject = { "myKey" : "1234", "name" : "somebody" }
var myArray = [ { "myKey" : "1234" }, { "name" : "somebody" } ]
output application/json
---
{
    selectingValueUsingKeyInObject : myObject.name,
    selectingValueUsingKeyOfObjectInArray : myArray.name,
}
```

*Output JSON:*

```
{
    "selectingValueUsingKeyInObject": "somebody",
    "selectingValueUsingKeyOfObjectInArray": [ "somebody" ]}
```

As the Output shows:

- `myObject.name` returns the value "somebody"
- `myArray.name` returns the array [ "somebody" ]

## Selector Quick Reference

### Single Value (.)

Acts on arrays and objects to return the value of a matching key. The syntax is `.myKey`. To retrieve values of duplicate keys in a DataWeave *object*, use `*`, instead. For examples, see [Single-Value Selector \(.myKey\)](#).

### Multiple Values (\*)

Acts on arrays and objects to retrieve the values of all matching keys at a single level in the hierarchy of a data structure. The syntax is `*myKey`. For the values of all duplicate keys at lower levels in the hierarchy, use the descendants selector `(..)`, instead. For examples, see [Multi-Value Selector \(.\\*\)](#).

### Key-Value Pair (8)

Acts on arrays and objects. Instead of returning the value of the DataWeave object, this selector returns the entire DataWeave object, both key and value. The syntax is `.&myKey`. For examples, see [Key-Value Pair Selector \(.&myKey\)](#).

### Descendants (..)

Acts on arrays and objects to retrieve all matching keys from arrays and objects below the given key, regardless of their location in the hierarchy. The syntax is `..myKey`. For examples, see [Descendants Selector \(..myKey\)](#).

## **Index ([])**

Returns the value at the specified index of an array. An example is the `[0]` at the end of `["a", "b", "c"][]`, which returns "a". For examples, see [Index Selector \(\[anIndex\]\)](#).

## **Range [index1 to index2]**

Returns an array with values of the selected indices. An example is the `[2 to 3]` at the end of `["a", "b", "c", "d"][]`, which returns `["c", "d"]`. For examples, see [Range selector \(anIndex to anotherIndex\)](#).

## **XML attribute (.@myKey)**

Returns the value of a selected key for an XML attribute. For an example, see [XML Attribute Selector \(@myKey\)](#).

## **Namespace Selector (myKey.#)**

Returns the `xmlns` namespace from the element that also contains the selected key. For an example, see [Namespace Selector \(#\)](#).

## **Selector Modifiers (? and !)**

? and ! check for the specified key. ? returns `true` or `false`. ! returns an error if the key is not present. For examples, see [Selector Modifiers \(!, ?\)](#).

## **Filter Selectors (myKey[?(booleanExpression)])**

Returns the selected items if the Boolean expression returns `true` and the specified key is present. It returns `null` if the expression is false or the key is not present. For examples, see [Filter Selectors \(myKey\[?\(\\$ == "aValue"\)\]\)](#).

## **Metadata Selector .^someMetadata**

Returns the value of specified metadata for a Mule payload, variable, or attribute. The selector can return the value of class (`.^class`), content length (`.^contentLength`), encoding (`.^encoding`), MIME type (`.^mimeType`), media type (`.^mediaType`), raw (`.^raw`), and custom (`.^myCustomMetadata`) metadata. For details, see [Metadata Selector \(^someMetadata\)](#).

## **Key and Selector Syntax**

In DataWeave, quotes around the name of an object's key are required in some cases but optional in others. Though a key is not an identifier, if the key name meets the criteria of a [valid identifier name](#), quotes around the key are optional. If the key does not meet this criteria, you *must* surround the key in quotes. For example, to correct the key syntax in `{ some-name : "somebody" }`, you can use `{ "some-name" : "somebody" }` or `{ 'some-name' : "somebody" }`. Without quotes, the key in `{ some-name : "somebody" }` produces the error `Invalid input '-', expected Namespace`.

If you are using a selector on a key that requires quotes, you must also surround the selector in quotes. For example, the selector `"some-name"` in `{ "some-name" : "somebody" }."some-name"` works, but the selector `some-name` in `{ "some-name" : "somebody" }.some-name` produces the error `Unable to resolve reference of: name` because the selector is not surrounded in quotes.

Quotes around a selector are optional if you are using a selector on a key that does not require the quotes. For example, all of these examples work:

- `{"name" : "somebody"}.name`
- `{"name" : "somebody"}."name"`
- `{name : "somebody"}.name`
- `{name : "somebody"}."name"`

## Single-Value Selector (`.myKey`)

`.myKey` selectors work over an object or array to return the value of a matching key.

### Single-Value Selector Over an Object

For an object, the single-value selector returns the value of the matching key. For example, in the following script, `myObject.user` returns "a".

*DataWeave Script:*

```
%dw 2.0
var myObject = { user : "a" }
output application/json
---
{ myObjectExample : myObject.user }
```

*Output JSON:*

```
{
  "myObjectExample": "a"
}
```

When operating on a DataWeave *object* (not an array), the `.` selector only returns the value of the *first* matching key, even if the object contains multiple matching keys, for example:

*DataWeave Script:*

```
%dw 2.0
var myObject = { user : "a", "user" : "b" }
output application/json
---
{ myObjectExample : myObject.user }
```

*Output JSON:*

```
{
  "myObjectExample": "a"
}
```

To return the values of multiple matching keys in cases like this, see [Multi-Value Selector \(`.\*`\)](#).

In the next example, `payload.people.person.address` returns the value of the `address` element. (It also uses the `output` directive to transform the JSON input to XML.)

*DataWeave Script:*

```
%dw 2.0
var myData = {
    "people": {
        "size" : 1,
        "person": {
            "name": "Nial",
            "address": {
                "street": {
                    "name": "Italia",
                    "number": 2164
                },
                "area": {
                    "zone": "San Isidro",
                    "name": "Martinez"
                }
            }
        }
    }
}
output application/xml
---
{ myaddresses: myData.people.person.address }
```

*Output XML:*

```
<?xml version="1.0" encoding="UTF-8"?>
<myaddresses>
    <street>
        <name>Italia</name>
        <number>2164</number>
    </street>
    <area>
        <zone>San Isidro</zone>
        <name>Martinez</name>
    </area>
</myaddresses>
```

## Single-Value Selector Over an Array

When acting on an array, the `.` selector returns an array, even if there is only one matching value. For example, `["a":"b"]. "a"` returns `["b"]`.

Note that the `.` selector acts differently on arrays than it acts on objects. Like `.*`, the `.` selector returns an array with the values of *all matching keys* at the specified level of the input array.

*DataWeave Script:*

```
%dw 2.0
var myArrayOfKeyValuePairs = [ "aString": "hello", "aNum": 2, "aString" : "world" ]
var myArrayOfObjects = [ { "aString": "hello" }, { "aNum": 2 }, { "aString" : "world" }
] ]
output application/json
---
{
    myKeyValueExample : myArrayOfKeyValuePairs.aString,
    myObjectExample :  myArrayOfObjects.aString
}
```

*Output JSON:*

```
{
    "myKeyValueExample": [ "hello", "world" ],
    "myObjectExample": [ "hello", "world" ]
}
```

In the following example, the value of the input variable, **myData**, is an array that contains two objects. The selector navigates both objects and returns the values of both **street** keys.

*DataWeave Script:*

```
%dw 2.0
var myData = {
    "people": [
        {
            "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"
                    }
                }
            }
        }
    ]
}
output application/json
---
myData.people.person.address.street
```

*Output JSON:*

```
[  
  {  
    "name": "Italia",  
    "number": 2164  
  },  
  {  
    "name": "Monroe",  
    "number": 323  
  }  
]
```

## Multi-Value Selector (`.*`)

`.*` traverses objects and arrays to select the values of all matching keys and returns matching results in an array.

### Multi-Value Selector Over an Object

`.*` returns an array with all the values whose key matches the expression.

The following example returns the values of all `user` elements from the input payload.

*DataWeave Script:*

```
%dw 2.0  
output application/json  
---  
payload.users.*user
```

*Input XML Payload:*

```
<users>  
  <user>Mariano</user>  
  <user>Martin</user>  
  <user>Leandro</user>  
</users>
```

*Output JSON:*

```
[ "Mariano", "Martin", "Leandro" ]
```

### Multi-Value Selector Over an Array

On arrays, `.*` works the same way as the single-value selector `(.)`. For example, `payload.people.person.address.*street` and the example `payload.people.person.address.street` (from [Single-Value Selector Over an Array](#)) return the same results.

*DataWeave Script:*

```
%dw 2.0
var myArrayOfKeyValuePairs = [ "aString": "hello", "aNum": 2, "aString" : "world" ]
var myArrayOfObjects = [ { "aString": "hello" }, { "aNum": 2 }, { "aString" : "world" }
] ]
output application/json
---
{
    myKeyValueExample : myArrayOfKeyValuePairs.*aString,
    myObjectExample :  myArrayOfObjects.*aString
}
```

*Output JSON:*

```
{
    "myKeyValueExample": [ "hello", "world" ],
    "myObjectExample": [ "hello", "world" ]
}
```

## Descendants Selector (`..myKey`)

The `..` selector acts on arrays and objects.

This selector applies to the context using the form `..myKey`, and it retrieves the values of all matching key-value pairs in the sub-tree under the selected context. Regardless of the hierarchical structure of the fields, the output is returned at the same level.

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.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{ names: payload.people..name }
```

*Input JSON Payload:*

```
{  
  "people": {  
    "person": {  
      "name": "Nial",  
      "address": {  
        "street": {  
          "name": "Italia",  
          "number": 2164  
        },  
        "area": {  
          "zone": "San Isidro",  
          "name": "Martinez"  
        }  
      }  
    }  
  }  
}
```

*Output JSON:*

```
{  
  "names": [  
    "Nial",  
    "Italia",  
    "Martinez"  
  ]  
}
```

## Key-Value Pair Selector (`.&myKey`)

The `&` selector acts on arrays and objects. `&` retrieves both the keys and values of all matching keys pairs in the current context. These are returned as an object, containing the retrieved keys and values.

*DataWeave Script:*

```
%dw 2.0  
output application/xml  
---  
{  
  users: payload.users.&user  
}
```

*Input XML Payload:*

```
<?xml version='1.0' encoding='US-ASCII'?>
<users>
  <user>Mariano</user>
  <user>Martin</user>
  <user>Leandro</user>
  <admin>Admin</admin>
  <admin>org_owner</admin>
</users>
```

*Output XML:*

```
<?xml version='1.0' encoding='US-ASCII'?>
<users>
  <user>Mariano</user>
  <user>Martin</user>
  <user>Leandro</user>
</users>
```

Note that unlike the multi-value selector, the output of this selector is an object, where the original keys for each value are also extracted.

### Select All the Descendant Key-Value Pairs

This example uses the `..` and `&` selectors in `myVar.people..&name` to select and return an array that contains all descendant objects from `myData` input that contain the key `name`. It also transforms the JSON input to XML output.

### DataWeave Script:

```
%dw 2.0
var myData = {
    "people": [
        {"person": {
            "name": "Nial",
            "address": {
                "street": {
                    "name": "Italia",
                    "number": 2164
                },
                "area": {
                    "zone": "San Isidro",
                    "name": "Martinez"
                }
            }
        }
    ]
}
output application/xml
---
names: {(myData.people..&name)}
```

### Output XML:

```
<?xml version='1.0' encoding='UTF-8'?>
<names>
    <name>Nial</name>
    <name>Italia</name>
    <name>Martinez</name>
</names>
```

## Index Selector ([anIndex])

The index selector returns the element at the specified position. It can be applied over an array, object, or string.

### Index Selector Over an 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. In the case of Arrays, the value of the element 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.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
payload.people[1]
```

*Input JSON Payload:*

```
{
  "people": [
    {
      "nameFirst": "Nial",
      "nameLast": "Martinez"
    },
    {
      "nameFirst": "Coty",
      "nameLast": "Belgrano"
    }
  ]
}
```

*Output JSON:*

```
{
  "nameFirst": "Coty",
  "nameLast": "Belgrano"
}
```

## Index Selector Over an Object

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

*DataWeave Script:*

```
%dw 2.0
output application/json
---
payload[1]
```

*Input JSON Payload:*

```
{
  "nameFirst": "Mark",
  "nameLast": "Nguyen"
}
```

*Output JSON:*

```
"Nguyen"
```

## Index Selector Over a String

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

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{ name: "MuleSoft"[0] }
```

*Output JSON:*

```
{ "name": "M" }
```

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.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{ name: "Emiliano"[0] }
```

*Output JSON:*

```
{ "name": "E" }
```

## Range selector ([anIndex to anotherIndex](#))

The `to` selector returns values of matching indices in an array or string. You can also use it to reverse the order of the indices in the range. The selector treats characters in the string as indices.

- Selecting an index from an array:

- `[1,2,3,4][0]` returns 1

The index of the first element in an array is always 0.

- `[1,2,3,4][3]` returns 4

## Range Selector Over an 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.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{
  slice: [0,1,2][0 to 1],
  last: [0,1,2][-1 to 0]
}
```

*Output JSON:*

```
{
  "slice": [
    0,
    1
  ],
  "last": [
    2,
    1,
    0
  ]
}
```

## Range Selector Over a 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 enables you to slice a string or invert it.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{
  slice: "DataWeave"[0 to 1],
  middle : "superfragilisticexpialadocious"[10 to 13],
  last: "DataWeave"[-1 to 0]
}
```

*Output JSON:*

```
{  
  "slice": "Da",  
  "middle": "list",  
  "last": "evaeWataD"  
}
```

## XML Attribute Selector (`.@myKey`)

`.@myKey` selects an attribute in an XML element.

Using `.@` without the key name returns an object containing the attributes as key-value pairs.

This DataWeave example reads an XML sample into a variable and uses `@` to select attributes from the XML.

*DataWeave Script:*

```
%dw 2.0  
var myVar = read('<product id="1" type="electronic">  
  <brand>SomeBrand</brand>  
</product>', 'application/xml')  
output application/json  
  
---  
{  
  item: [  
    {  
      "type" : myVar.product.@."type",  
      "name" : myVar.product.brand,  
      "attributes": myVar.product.  
    }  
  ]  
}
```

*Output JSON:*

```
{  
  "item": [  
    {  
      "type": "electronic",  
      "name": "SomeBrand",  
      "attributes": {  
        "id": "1",  
        "type": "electronic"  
      }  
    }  
  ]  
}
```

## Namespace Selector (#)

# returns the XML namespace of a selected key as plain text.

*DataWeave Script:*

```
%dw 2.0
output text/plain
---
payload.order.#
```

*Input XML Payload:*

```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:order xmlns:ns0="http://orders.company.com">
  <name>Mark</name>
  <items>42</items>
  <orderdate>2017-01-04</orderdate>
</ns0:order>
```

*Output Text:*

```
"http://orders.company.com"
```

## Selector Modifiers (!, ?)

You can check for the presence of a given key.

- ! evaluates the selection and fails with an exception message if the key is not present.
- ? returns `true` if the selected key is present, `false` if not. Note that ? is also used in [Filter Selectors](#) (`myKey[?($ == "aValue")]`).

### Assert Present Validator

! returns an error if any of the specified key is missing.

- { "name": "Annie" }.lastName! returns an error with the message, `There is no key named 'lastName'.`.
- Without the !, { "name": "Annie" }.lastName returns `null`.
- When the key is present, { "name": "Annie" }.name! the result is `"Annie"`.

### Key Present Validator

Returns `true` if the specified key is present in the object or as an attribute of an XML element.

This example returns `true` because the `name` key does exists.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
present: payload.name?
```

*Input JSON Payload:*

```
{ "name": "Annie" }
```

*Output XML:*

```
<?xml version="1.0" encoding="UTF-8"?>
<present>true</present>
```

? also works with XML attributes:

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{
  item: {
    typePresent : payload.product.@."type"?
  }
}
```

*Input XML Payload:*

```
<product id="1" type="tv">
  <brand>Samsung</brand>
</product>
```

*Output JSON:*

```
{
  "item": { "typePresent": true }
```

## Filter Selectors (`myKey[?($ == "aValue")]`)

`myKey[?($ == "aValue")]` returns only the values of matching keys within an array or object. Note that `?` is also used in [Key Present Validator](#). If no key-value pairs match, the result is `null`.

The following example inputs the array of `name` keys returned by `*.name`, then checks for `name` keys

with the value "Mariano". It filters out any values that do not match. Note that the \$ references the value of the selected key.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{ users: payload.users.*name[?($ == "Mariano")] }
```

*Input XML Payload:*

```
<users>
  <name>Mariano</name>
  <name>Luis</name>
  <name>Mariano</name>
</users>
```

*Output JSON:*

```
{
  "users": [
    "Mariano",
    "Mariano"
  ]
}
```

The following example assumes the same [input](#) and returns all the key-value pairs of the input because the expression (`1 == 1`) is true. Note that a false expression, such as (`1 == 2`), returns [null](#).

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{ users: payload.users.*name[?( 1 == 1)] }
```

*Input XML Payload:*

```
<users>
  <name>Mariano</name>
  <name>Luis</name>
  <name>Mariano</name>
</users>
```

*Output JSON:*

```
{  
  "users": [  
    "Mariano",  
    "Luis",  
    "Mariano"  
  ]  
}
```

The following example assumes the same [input](#). It uses `mapObject` to iterate over the entire input object and return matching key-value pairs, filtering out any pairs that do not match.

*DataWeave Script:*

```
%dw 2.0  
output application/json  
---  
payload mapObject { ($$) : $$[?($== "Mariano")] }
```

*Input XML Payload:*

```
<users>  
  <name>Mariano</name>  
  <name>Luis</name>  
  <name>Mariano</name>  
</users>
```

*Output JSON:*

```
{  
  "users": {  
    "name": "Mariano",  
    "name": "Mariano"  
  }  
}
```

## Metadata Selector (`.^someMetadata`)

Returns the value of specified metadata for a Mule payload, variable, or attribute. The selector can return the following metadata:

- Content length metadata: `.^contentLength` returns the content length of the value, if the value is present. For an example, see [Content Length Metadata Selector \(`.^contentLength`\)](#).
- Class metadata: `.^class` returns the class of the Plain Old Java Object (POJO). For example, `{ "string" : payload.string.^class }` might return `{ "string": "java.lang.String" }` if the input payload defines a Java string, such as `simplePojo.string = "myString"`, in a simple POJO, and `{ "date" : payload.date.^class }` might return `{ "date": "java.util.Date" }`. For an example, see

## Class Metadata Selector (`.^class`).

- Encoding metadata: `.^encoding` returns the encoding of a value. For example, `{ "myEncoding" : payload.^encoding }` might return `{"myEncoding": "UTF-8"}` for an input POJO. For an example, see [Encoding Metadata Selector \(`.^encoding`\)](#).
- Media Type Selector: `.^mediaType` returns the MIME type of a value that includes parameters, for example, `application/json; charset=UTF-16`, and the expression in the value of `{ "myMediaType" : payload.^mediaType }` might return `"myMediaType": "/; charset=UTF-8"` for an input POJO. For an example, see [Media Type Metadata Selector \(`.^mediaType`\)](#).
- MIME Type metadata: `.^mimeType` returns the MIME type (without parameters) of a value, for example, `application/json`, and `{ "myMimeType" : payload.^mimeType }` might return `{"myMimeType": "/"}` for an input POJO. For an example, see [MIME Type Metadata Selector \(`.^mimeType`\)](#).
- Raw metadata: `.^raw` returns the underlying data (typically, a binary value) of a plain old Java object (POJO). This selector is sometimes used when calculating an MD5 for hashes when checking for man-in-the-middle attacks. For examples, see [Raw Metadata Selector \(`.^raw`\)](#).
- Custom metadata: `.^myCustomMetadata` returns the value of custom metadata. For examples, see [Custom Metadata Selector \(`.^myCustomMetadata`\)](#).

## Content Length Metadata Selector (`.^contentLength`)

Returns the content length of the value, if the value is present.

In the following Mule app flow, the Logger uses `payload.^contentLength` to select the length of the string `my string`, set in the Set Payload (`set-payload`) component.

*Mule App XML in Anypoint Studio:*

```
<flow name="setpayloadobjectFlow" >
  <scheduler doc:name="Scheduler" >
    <scheduling-strategy >
      <fixed-frequency frequency="15" timeUnit="SECONDS"/>
    </scheduling-strategy>
  </scheduler>
  <!-- Set the payload to "my string". -->
  <set-payload value='"my string"' doc:name="Set Payload" />
  <!-- Select the class to which "my string" belongs. -->
  <logger level="INFO" doc:name="Logger" message="#[payload.^contentLength]"/>
</flow>
```

The Studio console output shows that the length of the input string (`my string`) is eleven (9) characters long. The length includes the blank space in the string.

*Console Output in Anypoint Studio:*

```
INFO 2019-05-07 16:59:33,690 [[MuleRuntime].cpuLight.07:  
[carets].caretsFlow.CPU_LITE @39f1dbde]  
[event: 28ce97a0-7124-11e9-acfe-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
11
```

### Class Metadata Selector (`.^class`)

Returns the class of the Plain Old Java Object (POJO). The value might result from calling a method in a Java class or have a data type (such as `String` or `DateTime`) that DataWeave treats as a Java value, for example:

- `{ "string" : payload.mystring.^class }` might return `{ "mystring": "java.lang.String" }` if the input payload defines a Java string, such as `simplePojo.string = "myString"`, in a simple POJO.
- `{ "mydate" : payload.mydate.^class }` might return `{ "mydate": "java.util.Date" }`.

In the following Mule app flow, the Logger uses `payload.^class` to select the Java class of "`my string`", set in the Set Payload (`set-payload`) component.

*Mule App XML in Anypoint Studio:*

```
<flow name="setpayloadobjectFlow" >  
  <scheduler doc:name="Scheduler" >  
    <scheduling-strategy>  
      <fixed-frequency frequency="15" timeUnit="SECONDS"/>  
    </scheduling-strategy>  
  </scheduler>  
  <!-- Set the payload to "my string". -->  
  <set-payload value='"my string"' doc:name="Set Payload" />  
  <!-- Select the class to which "my string" belongs. -->  
  <logger level="INFO" doc:name="Logger" message="#[payload.^class]" />  
</flow>
```

The Studio console output shows that the payload string belongs to the class `java.lang.String`.

*Console Output in Anypoint Studio:*

```
INFO 2019-04-20 16:10:03,075 [[MuleRuntime].cpuLight.08:  
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @6447187e]  
[event: 6da29400-63c1-11e9-98e0-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
java.lang.String
```

### Encoding Metadata Selector (`.^encoding`)

Returns the encoding of a value. For example, `{ "myEncoding" : payload.^encoding }` might return `{"myEncoding": "UTF-8"}` for an input POJO.

In the following Mule app flow, the Logger uses `payload.^encoding` to select the encoding of "my string" set in the Set Payload (`set-payload`) component. The Scheduler (`scheduler`) component is simply an event source that regularly generates a new Mule event to hold the payload set in Set Payload.

*Mule App XML in Anypoint Studio:*

```
<flow name="setpayloadobjectFlow" >
  <scheduler doc:name="Scheduler" >
    <scheduling-strategy >
      <fixed-frequency frequency="15" timeUnit="SECONDS"/>
    </scheduling-strategy>
  </scheduler>
  <!-- Set the payload to "my string". -->
  <set-payload value=""my string"" doc:name="Set Payload" />
  <!-- Select the encoding of "my string". -->
  <logger level="INFO" doc:name="Logger" message="#{payload.^encoding}"/>
</flow>
```

The Studio console output shows that the payload string has **UTF-8** encoding.

*Console Output in Anypoint Studio:*

```
INFO 2019-04-20 16:14:24,222 [[MuleRuntime].cpuLight.03:
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @62bea6a6]
[event: 0938bf70-63c2-11e9-98e0-8c8590a99d48]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
UTF-8
```

## Media Type Metadata Selector (`.^mediaType`)

Returns the MIME type of a value that includes parameters (for example, `application/json; charset=UTF-16`). The expression in the value of `{ "myMediaType" : payload.^mediaType }` might return `"myMediaType": "/; charset=UTF-8"` for an input POJO.

In the following Mule app flow, the Logger uses `payload.^mediaType` to select the media type of `2014-10-12T11:19-00:03` set in the Set Payload (`set-payload`) component.

*Mule App XML in Anypoint Studio:*

```
<flow name="setpayloadobjectFlow" >
<scheduler doc:name="Scheduler" >
    <scheduling-strategy >
        <fixed-frequency frequency="15" timeUnit="SECONDS"/>
    </scheduling-strategy>
</scheduler>
<set-payload value='#[|2014-10-12T11:11:19-00:03| as DateTime]' doc:name="Set Payload" />
    <logger level="INFO" doc:name="Logger" message="#[payload.^mediaType]"/>
</flow>
```

The Studio console output shows that the **DateTime** payload has the **application/java; charset=UTF-8** media type.

*Console Output in Anypoint Studio:*

```
INFO 2019-04-20 16:41:01,276 [[MuleRuntime].cpuLight.04:
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @7e991c71]
[event: c0e96860-63c5-11e9-bcff-8c8590a99d48]
rg.mule.runtime.core.internal.processor.LoggerMessageProcessor:
application/java; charset=UTF-8
```

In the following Mule app flow, the Loggers use **payload.^mediaType** to select a string "**my string**", then to select a string that is set within an **fx** expression (**#["my string as String type" as String]**) in the Set Payload (**set-payload**) component.

*Mule App XML in Anypoint Studio:*

```
<flow name="setpayloadobjectFlow" >
<scheduler doc:name="Scheduler" >
    <scheduling-strategy >
        <fixed-frequency frequency="15" timeUnit="SECONDS"/>
    </scheduling-strategy>
</scheduler>
<!-- Set the payload to "my string". --&gt;
&lt;set-payload value='"my string"' doc:name="Set Payload" /&gt;
<!-- Select the media type of "my string". --&gt;
&lt;logger level="INFO" doc:name="Logger" message='#[payload.^mediaType]'/&gt;
<!-- Set the payload using the fx expression "my string" as String. --&gt;
&lt;set-payload value='#[("my string as String type" as String)' doc:name="Set Payload" /&gt;
<!-- Select the media type of a Java string. --&gt;
&lt;logger level="INFO" doc:name="Logger" message='#[payload.^mediaType]'/&gt;
&lt;/flow&gt;</pre>
```

The Studio console output shows that the simple string has the media type **/**, while the string that is set in the **fx** expression has the media type **application/java; charset=UTF-8**.

*Console Output in Anypoint Studio:*

```
INFO 2019-04-20 16:52:50,801 [[MuleRuntime].cpuLight.01:  
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @5d914abe]  
[event: 68121cd0-63c7-11e9-bcff-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
/*  
  
INFO 2019-04-20 16:52:51,085 [[MuleRuntime].cpuLight.01:  
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @5d914abe]  
[event: 68121cd0-63c7-11e9-bcff-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
application/java; charset=UTF-8
```

### MIME Type Metadata Selector (`.^MimeType`)

Returns the MIME type (without parameters) of a value, for example, `application/json`, and `{ "myMimeType" : payload.^MimeType }` might return `{ "myMediaType": "/" }` for an input POJO.

In the following Mule app flow, the Loggers use `payload.^MimeType` to select a string "`my string`", then to select a string that is set within an **fx** expression (`#["my string as String type" as String]`) in the Set Payload (**set-payload**) component.

*Mule App XML in Anypoint Studio:*

```
<flow name="setpayloadobjectFlow" >  
  <scheduler doc:name="Scheduler" >  
    <scheduling-strategy >  
      <fixed-frequency frequency="15" timeUnit="SECONDS"/>  
    </scheduling-strategy>  
  </scheduler>  
  <!-- Set the payload to "my string". -->  
  <set-payload value='"my string"' doc:name="Set Payload" />  
  <!-- Select the MIME type of "my string". -->  
  <logger level="INFO" doc:name="Logger" message='#[payload.^MimeType]'/>  
  <!-- Set the payload using the fx expression "my string" as String. -->  
  <set-payload value='#[("my string as String type" as String)' doc:name="Set Payload"  
  />  
  <!-- Select the MIME type of a Java string. -->  
  <logger level="INFO" doc:name="Logger" message='#[payload.^MimeType]'/>  
</flow>
```

The Studio console output shows that the simple string has the MIME type `/`, while the string that is set in the **fx** expression has the MIME type `application/java`.

*Console Output in Anypoint Studio:*

```
INFO 2019-04-20 17:02:07,762 [[MuleRuntime].cpuLight.06:  
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @2d6f64b9]  
[event: b4097b00-63c8-11e9-bcff-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
/*  
  
INFO 2019-04-20 17:02:08,029 [[MuleRuntime].cpuLight.06:  
[setpayloadobject].setpayloadobjectFlow.CPU_LITE @2d6f64b9]  
[event: b4097b00-63c8-11e9-bcff-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
application/java
```

### Raw Metadata Selector (.^raw)

Returns the underlying binary value of a POJO. This selector is sometimes used when calculating an MD5 or some other cryptographic hash function to check for man-in-the-middle (MITM) attacks.

The following example uses the Set Payload component ([set-payload](#)) to produce a binary value, then uses the Transform Message component ([ee:transform](#)) component to return raw data for the MD5 ([MD5\(payload.^raw\)](#)) of the binary value. The Logger component ([logger](#)) is also set to write the raw data to the Studio console. For comparison, the second Logger returns the typical [payload](#) in a standard JSON format.

*Mule App XML in Anypoint Studio:*

```
<flow name="rawcaret2Flow" >  
  <scheduler doc:name="Scheduler" >  
    <scheduling-strategy >  
      <fixed-frequency frequency="30" timeUnit="SECONDS"/>  
    </scheduling-strategy>  
  </scheduler>  
  <set-payload value='#[{"id": "1234-5678-9123"} as Binary]' doc:name="Set Payload" />  
  <ee:transform doc:name="Transform Message" >  
    <ee:message >  
      <ee:set-payload ><![CDATA[%dw 2.0  
import * from dw::Crypto  
output application/json  
---  
{ "myRawData" : MD5(payload.^raw) }]]></ee:set-payload>  
    </ee:message>  
  </ee:transform>  
  <logger level="INFO" doc:name="Logger" message="#[payload.^raw]" />  
  <logger level="INFO" doc:name="Logger" message="#[payload]" />  
</flow>
```

Notice that instead of producing standard JSON output, the raw output in the Logger message surrounds the entire payload in double-quotes and inserts new line characters ([\n](#)) for each new

line.

*Console Output in Anypoint Studio:*

```
INFO 2019-04-22 14:10:14,537 [[MuleRuntime].cpuLight.08:  
[rawcaret2].rawcaret2Flow.CPU_LITE @764a5a61]  
[event: 058f6a90-6543-11e9-9d99-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
"\n  \"myRawData\": \"5403e5a202c594871d59898b13054be5\"\n"  
  
INFO 2019-04-22 14:10:14,540 [[MuleRuntime].cpuLight.08:  
[rawcaret2].rawcaret2Flow.CPU_LITE @764a5a61]  
[event: 058f6a90-6543-11e9-9d99-8c8590a99d48]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:  
{ "myRawData": "5403e5a202c594871d59898b13054be5" }
```

The following example uses the HTTP Listener source ([listener](#)) to get the XML payload received via a POST request, then uses the Transform Message component ([ee:transform](#)) to get the encoding value of the XML payload, by returning the raw data ([\(\(payload.^raw as String\) scan /encoding='\(\[A-z0-9-\]+'\)/\)](#)). The Logger component ([logger](#)) returns the [payload](#) in a JAVA format.

*Mule App XML in Anypoint Studio:*

```
<http:listener-config name="HTTP_Listener_config" >  
  <http:listener-connection host="0.0.0.0" port="8081" />  
</http:listener-config>  
<flow name="test-flow">  
  <http:listener path="/test" config-ref="HTTP_Listener_config"/>  
  <ee:transform>  
    <ee:message>  
      <ee:set-payload ><![CDATA[%dw 2.0  
output application/java  
---  
((payload.^raw as String) scan /encoding='([A-z0-9-]+')/)[0][1]]></ee:set-payload>  
      </ee:message>  
    </ee:transform>  
    <logger level="INFO" message="#{payload}" />  
</flow>
```

Using your preferred REST client or API testing tool, send a POST request with the XML body:

```
<?xml version='1.0' encoding='ISO-8859-1'?>  
<test>  
</test>
```

The Logger message returns the extracted encoding value [ISO-8859-1](#):

*Console Output in Anypoint Studio:*

```
INFO 2021-05-12 12:25:41,618 [[MuleRuntime].uber.03: [xmlmodule].Flow.CPU_INTENSIVE  
@de48fc8] [processor: Flow/processors/2; event: 4fe7f6a0-b336-11eb-909c-f01898ad2638]  
org.mule.runtime.core.internal.processor.LoggerMessageProcessor: ISO-8859-1
```

### Custom Metadata Selector (`.^myCustomMetadata`)

Returns the value of custom metadata. Metadata can be associated with any value by using the `as` operator.

The following example uses `userName.^myCustomMetadata` to return the value of custom metadata that is defined as a variable (named `userName`) in the header of the script as a DataWeave script. For comparison, the example also returns the value of `userName`.

*DataWeave Script:*

```
%dw 2.0  
output application/json  
var userName = "DataWeave" as String {myCustomMetadata: "customMetadataValue"}  
---  
  
{  
  "valueOfVariableMetaData" : userName.^myCustomMetadata,  
  "valueOfVariable" : userName,  
}
```

The output of the script is "`customMetadataValue`" for the value of the custom metadata and "`DataWeave`" for value of the `userName` variable.

*Output JSON:*

```
{  
  "valueOfVariableMetaData": "customMetadataValue",  
  "valueOfVariable": "DataWeave"  
}
```

## See Also

- [Selectors](#)
- [DataWeave Quickstart](#)
- [DataWeave Types](#)
- [DataWeave Cookbook](#)

## Select XML Elements

DataWeave provides a number of [selectors](#) for traversing the structure of input data and returning

matching values. The following examples use a single-value DataWeave selector (`.`) to extract data from XML elements.

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The XML markup language requires a single root element. In the following example, the selector on the root (`language`) is able to extract the child element because the child has no siblings.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
payload.language
```

*Input XML Payload:*

```
<language>
  <name>DataWeave</name>
</language>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<name>DataWeave</name>
```

To return all subelements of the root element in a valid XML structure with a single root, the following script uses a DataWeave key to specify a new element name (`myroot`) and selects the root (`root`) to return its value. A script that fails to return a single XML root produces the following error: `Trying to output non-whitespace characters outside main element tree (in prolog or epilog)…`.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
{ myroot: payload.root }
```

*Input XML Payload:*

```
<root>
  <element>
    <subelement1>SE1</subelement1>
  </element>
  <element>E2</element>
</root>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<myroot>
  <element>
    <subelement1>SE1</subelement1>
  </element>
  <element>E2</element>
</myroot>
```

The following example uses the name of the child element (`name`) with the single-value selector to select its value. Notice how `payload.language.name` navigates to the child element. The script also provides a root element for the value so that DataWeave can construct valid XML.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
{ newname : payload.language.name }
```

*Input XML Payload:*

```
<language>
  <name>DataWeave</name>
  <version>2.0</version>
</language>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<newname>DataWeave</newname>
```

The following example uses an index to return the value of the second child element from the input XML. Notice that DataWeave treats child elements as indices of an array, so it can select the second child by using the index `language[1]`. The result is a single XML element.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
{ version : payload.language[1] }
```

*Input XML Payload:*

```
<language>
  <name>DataWeave</name>
  <version>2.0</version>
</language>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<version>2.0</version>
```

If multiple child elements of the input have the same name, use the index to select the value of the child. The following example selects a subelement of a child element.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
{ mysubelement : payload.root[0].subelement1 }
```

*Input XML Payload:*

```
<root>
  <element>
    <subelement1>SE1</subelement1>
  </element>
  <element>E2</element>
</root>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<mysubelement>SE1</mysubelement>
```

## Set Default Values

Use one of the following methods to set default values when a payload value is absent or when the value is `null`:

- Using the `default` keyword
- Setting the default in an `if-else` or `else-if` statement
- Using `else` when pattern matching

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

## Example of Using the Keyword `default` to Set a Default Value

Consider an application that expects a JSON input with fields `id` and `name` to do a transformation. You can configure default values for these fields in case the fields are not present, or their value is `null`. For example:

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{
  "userId": payload.id default "0000",
  "userName": payload.name default "Undefined"
}
```

If the application receives a JSON message with values set for the `id` and `name` fields, then the DataWeave transformation matches field `id` to `userId` and field `name` to `userName`.

*Input JSON Payload:*

```
{
  "id": "123",
  "name": "Max the Mule"
}
```

*Output JSON:*

```
{
  "userId": "123",
  "userName": "Max the Mule"
}
```

However, if the application receives a JSON message without the expected fields, or the fields have `null` values, then the transformation uses the configured default values for fields `userId` and `userName`.

*Input JSON Payload:*

```
{  
  "id": null  
}
```

*Output JSON:*

```
{  
  "userId": "0000",  
  "userName": "Undefined"  
}
```

## Example of Using if-else and else-if Statements to Set Default Values

Another method for providing a default value is using `if-else` and `else-if`.

The following example sets **United States** as the default value for `userLocation` if it is not present, or its value is `null`, in the JSON input message:

*DataWeave Script:*

```
%dw 2.0  
output application/json  
---  
if (payload.location != null) {  
  "userLocation" : payload.location  
} else {  
  "userLocation" : "United States"  
}
```

If the application receives a JSON message with a value set for the `location` field, then the DataWeave transformation matches field `location` to `userLocation`.

*Input JSON Payload:*

```
{  
  "location": "Argentina"  
}
```

*Output JSON:*

```
{  
  "userLocation": "Argentina"  
}
```

However, if the application receives a JSON message without the expected field, or the field is `null`, then the transformation uses the configured default value for field `userLocation`.

*Input JSON Payload:*

```
{}
```

*Output JSON:*

```
{
  "userLocation": "United States"
}
```

## Example of Using Matching Patterns to Set Default Values

In [pattern-matching scripts](#), DataWeave `case` statements end with an `else` expression that can serve as the default to return if all preceding `case` expressions return `false`.

*DataWeave Script:*

```
%dw 2.0
var myVar = "someString"
output application/json
---
myVar match {
  case myVarOne if (myVar == "some") -> ("some" ++ "is myVar")
  case myVarOne if (myVar == "strings") -> ("strings" ++ "is myVar")
  else -> myVar ++ " is myVar"
}
```

*Output JSON:*

```
"someString is myVar"
```

## Set Reader and Writer Configuration Properties

DataWeave provides configuration properties for data formats, such as JSON ([application/json](#)), XML ([application/xml](#)), and ([application/csv](#)). The properties change the behavior of DataWeave readers and writers for those formats. For example, the default separator for a CSV reader is a comma (,). You can use the format's `separator` property to specify a different separator for CSV content.

Refer to [DataWeave Formats](#) for more details on available reader and writer properties for various data formats.

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

## Use a Writer Property in an Output Directive

The following example shows how to append writer properties to the DataWeave `output` directive. The script uses `indent = false` to compress the JSON output into a single line.

*DataWeave Script:*

```
%dw 2.0
output application/json indent = false
---
{
    hello : "world",
    bello : "world",
    mello : "world"
}
```

*Output JSON:*

```
{"hello": "world", "bello": "world", "mello": "world"}
```

The following examples also append writer configuration properties to the `output` directive:

- [Avro example](#) that uses `schemaUrl`
- [Excel \(XLSX\) example](#) that uses `header=true`
- [Flat File example](#) that sets a `schemaPath`
- [XML example](#) that uses `inlineCloseOn="empty"` to close any empty XML elements

## Use Reader and Writer Properties in DataWeave Functions

The DataWeave `read`, `readUrl`, and `write` functions accept one or more comma-separated property configurations within curly braces.

In the header of the following script, the value of `myVar` is a `read` function that inputs an XML sample with an empty child element (`<ex1></ex1>`). The function passes an XML reader property `{nullValueOn: "empty"}` that converts the value of the empty element to `null`.

In the body of the script, a `write` function accepts the value of `myVar` as input. The function passes the JSON writer properties `{skipNullOn:"objects", writeAttributes:true}` to skip the object with the `null` value (`<ex1>null</ex1>`) and to write the attribute and value of `<ex3 a='greeting'>hello</ex3>`.

*DataWeave Script:*

```
%dw 2.0
var myVar = read("<greeting><ex1></ex1><ex2>hello</ex2><ex3
a='greeting'>hello</ex3></greeting>", "application/xml", {nullValueOn: "empty"})
output application/json with binary
---
write(myVar.greeting, "application/json", {skipNullOn:"objects",
writeAttributes:true})
```

*Output JSON:*

```
{  
  "ex2": "hello",  
  "ex3": {  
    "@a": "greeting",  
    "__text": "hello"  
  }  
}
```

The following examples pass reader properties to `readUrl`:

- [XML example](#)
- [ndjson example](#)

## See Also

- [DataWeave Formats](#)
- [Core Components](#)
- [DataWeave Cookbook](#)

## Perform a Basic Transformation

These simple DataWeave examples change the XML input to JSON output. Note that more complex transformations usually require the use of the `map` or `mapObject` function. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The following DataWeave script maps the names (or keys) for the output fields to values of the input fields. The input fields are specified with selector expressions without any functions. The script also changes the order and names of some of the fields.

*DataWeave Script:*

```
%dw 2.0  
output application/json  
---  
{  
  address1: payload.order.buyer.address,  
  city: payload.order.buyer.city,  
  country: payload.order.buyer.nationality,  
  email: payload.order.buyer.email,  
  name: payload.order.buyer.name,  
  postalCode: payload.order.buyer.postCode,  
  stateOrProvince: payload.order.buyer.state  
}
```

*Input XML Payload:*

```
<?xml version='1.0' encoding='UTF-8'?>
<order>
  <product>
    <price>5</price>
    <model>MuleSoft Connect 2016</model>
  </product>
  <item_amount>3</item_amount>
  <payment>
    <payment-type>credit-card</payment-type>
    <currency>USD</currency>
    <installments>1</installments>
  </payment>
  <buyer>
    <email>mike@hotmail.com</email>
    <name>Michael</name>
    <address>Koala Boulevard 314</address>
    <city>San Diego</city>
    <state>CA</state>
    <postCode>1345</postCode>
    <nationality>USA</nationality>
  </buyer>
  <shop>main branch</shop>
  <salesperson>Mathew Chow</salesperson>
</order>
```

*Output JSON:*

```
{
  "address1": "Koala Boulevard 314",
  "city": "San Diego",
  "country": "USA",
  "email": "mike@hotmail.com",
  "name": "Michael",
  "postalCode": "1345",
  "stateOrProvince": "CA"
}
```

In the following example, the DataWeave script transforms the XML containing multiple XML nodes with the same key into a valid JSON creating an array instead of creating JSON nodes with the same key.

*DataWeave Script:*

```
%dw 2.0
output application/json duplicateKeyAsArray=true
---
payload
```

*Input XML Payload:*

```
<order>
  <product-lineitems>
    <product-lineitem>
      <net-price>100.0</net-price>
    </product-lineitem>
    <product-lineitem>
      <net-price>498.00</net-price>
    </product-lineitem>
  </product-lineitems>
</order>
```

*Output JSON:*

```
{
  "order": {
    "product-lineitems": {
      "product-lineitem": [
        {
          "net-price": "100.0"
        },
        {
          "net-price": "498.00"
        }
      ]
    }
  }
}
```

## Related Examples

- [Extract Data](#)
- [Map Data](#)
- [Rename Keys](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Map Data

This DataWeave example uses the DataWeave `map` function to iterate through an array of books and perform a series of tasks on each. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other

DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses these DataWeave functions:

- `map` to go through each object in the `books` array.
- `as` to coerce the price data into a Number type, which ensures that the transformation generates the correct type for each element.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
items: payload.books map (item, index) -> {
    book: item mapObject (value, key) -> {
        (upper(key)): value
    }
}
```

*Input JSON Payload:*

```
{  
  "books": [  
    {  
      "-category": "cooking",  
      "title": "Everyday Italian",  
      "author": "Giada De Laurentiis",  
      "year": "2005",  
      "price": "30.00"  
    },  
    {  
      "-category": "children",  
      "title": "Harry Potter",  
      "author": "J K. Rowling",  
      "year": "2005",  
      "price": "29.99"  
    },  
    {  
      "-category": "web",  
      "title": "XQuery Kick Start",  
      "author": [  
        "James McGovern",  
        "Per Bothner",  
        "Kurt Cagle",  
        "James Linn",  
        "Vaidyanathan Nagarajan"  
      ],  
      "year": "2003",  
      "price": "49.99"  
    },  
    {  
      "-category": "web",  
      "-cover": "paperback",  
      "title": "Learning XML",  
      "author": "Erik T. Ray",  
      "year": "2003",  
      "price": "39.95"  
    }  
  ]  
}
```

*Output JSON:*

```
{  
  "items": [  
    {  
      "book": {  
        "-CATEGORY": "cooking",  
        "TITLE": "Everyday Italian",  
        "AUTHOR": "Giada De Laurentiis",  
        "YEAR": "2005",  
        "PRICE": "30.00"  
      }  
    },  
    {  
      "book": {  
        "-CATEGORY": "children",  
        "TITLE": "Harry Potter",  
        "AUTHOR": "J K. Rowling",  
        "YEAR": "2005",  
        "PRICE": "29.99"  
      }  
    },  
    {  
      "book": {  
        "-CATEGORY": "web",  
        "TITLE": "XQuery Kick Start",  
        "AUTHOR": [  
          "James McGovern",  
          "Per Bothner",  
          "Kurt Cagle",  
          "James Linn",  
          "Vaidyanathan Nagarajan"  
        ],  
        "YEAR": "2003",  
        "PRICE": "49.99"  
      }  
    },  
    {  
      "book": {  
        "-CATEGORY": "web",  
        "-COVER": "paperback",  
        "TITLE": "Learning XML",  
        "AUTHOR": "Erik T. Ray",  
        "YEAR": "2003",  
        "PRICE": "39.95"  
      }  
    }  
  ]  
}
```



Note that when a book has multiple authors, `item.author` evaluates to the entire array of authors instead of a single name.

## Using Default Values

The following example performs the same transformation as above, but it doesn't explicitly define the properties "item" and "index". Instead, it calls them through the default names: `$` and `$$` respectively.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
items: (payload.books map {
    category: "book",
    price: $.price as Number,
    id: $$,
    properties: {
        title: $.title,
        author: $.author,
        year: $.year as Number
    }
})
```

*Input JSON Payload:*

```
{
  "books": [
    {
      "-category": "cooking",
      "title": {
        "-lang": "en",
        "#text": "Everyday Italian"
      },
      "author": "Giada De Laurentiis",
      "year": "2005",
      "price": "30.00"
    },
    {
      "-category": "children",
      "title": {
        "-lang": "en",
        "#text": "Harry Potter"
      },
      "author": "J K. Rowling",
      "year": "2005",
      "price": "29.99"
    },
    {
      "-category": "cooking",
      "title": {
        "-lang": "en",
        "#text": "The Kitchen"
      },
      "author": "Nigella Lawson",
      "year": "2003",
      "price": "25.00"
    }
  ]
}
```

```

"-category": "web",
"title": {
    "-lang": "en",
    "#text": "XQuery Kick Start"
},
"author": [
    "James McGovern",
    "Per Bothner",
    "Kurt Cagle",
    "James Linn",
    "Vaidyanathan Nagarajan"
],
"year": "2003",
"price": "49.99"
},
{
    "-category": "web",
    "-cover": "paperback",
    "title": {
        "-lang": "en",
        "#text": "Learning XML"
    },
    "author": "Erik T. Ray",
    "year": "2003",
    "price": "39.95"
}
]
}

```

*Output JSON:*

```
{
"items": [
    {
        "category": "book",
        "price": 30.00,
        "id": 0,
        "properties": {
            "title": {
                "-lang": "en",
                "#text": "Everyday Italian"
            },
            "author": "Giada De Laurentiis",
            "year": 2005
        }
    },
    {
        "category": "book",
        "price": 29.99,
        "id": 1,
        "properties": {

```

```

        "title": {
            "-lang": "en",
            "#text": "Harry Potter"
        },
        "author": "J K. Rowling",
        "year": 2005
    }
},
{
    "category": "book",
    "price": 49.99,
    "id": 2,
    "properties": {
        "title": {
            "-lang": "en",
            "#text": "XQuery Kick Start"
        },
        "author": [
            "James McGovern",
            "Per Bothner",
            "Kurt Cagle",
            "James Linn",
            "Vaidyanathan Nagarajan"
        ],
        "year": 2003
    }
},
{
    "category": "book",
    "price": 39.95,
    "id": 3,
    "properties": {
        "title": {
            "-lang": "en",
            "#text": "Learning XML"
        },
        "author": "Erik T. Ray",
        "year": 2003
    }
}
]
}

```

## Related Examples

- [Extract Data](#)
- [Perform a Basic Transformation](#)
- [Rename Keys](#)
- [Map the Objects within an Array](#)

- [Map an Object](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Map and Flatten an Array

The `flatMap` function calls `map` on an input and wraps the results in a call to `flatten`. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The `flatMap` function is useful for refactoring uses of `flatten` on the results of a call to `map`. For example, review the following DataWeave script:

*DataWeave Script:*

```
%dw 2.0
output application/json
var myData = [{name:1},{name:2},{name:3}]
fun myExternalFunction(data): Array =
    if(data.name == 1)
        []
    else if(data.name == 2)
        [{name: 3}, {name:5}]
    else
        [data]
---
//flatten(myData map ((item, index) -> myExternalFunction(item)))
myData flatMap ((item, index) -> myExternalFunction(item))
```

The header of the script creates a variable `myData` to define an array of objects, each with the key `name`. It also defines a function with a set of if-else statements that act on name-value pairs with the key `name`.

The body of the DataWeave script contains the following expressions, each of which produces the same result:

- `flatten(myData map ((item, index) → myExternalFunction(item)))`
- `myData flatMap ((item, index) → myExternalFunction(item))`

Whether you use the `flatMap` expression or explicitly use `flatten` to wrap the `map` expression, the following takes place:

1. The expression maps the items in the input array according to if-else conditions in the function

`myExternalFunction()`, which is defined in the header.

The mapping produces the following output:

```
[  
  [  
    [  
    ],  
    [  
      { "name": 3 },  
      { "name": 5 }  
    ],  
    [  
      { "name": 3 }  
    ]  
]
```

2. The expression flattens the mapped results by consolidating the elements from the subarrays into a single array, removing the parent array and eliminating the empty child array.

Flattening produces the following output:

*Output JSON:*

```
[  
  {  
    "name": 3  
  },  
  {  
    "name": 5  
  },  
  {  
    "name": 3  
  }  
]
```

## See Also

- [flatMap](#)

## Map an Object

The following DataWeave examples use the `mapObject` function to iterate through the keys and values of objects. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

### First Example

This example uses both the `map` and `mapObject` functions to iterate through the input and set all of the keys to upper case.

The example uses these DataWeave functions:

- `map` to go through the elements in the "books" array.
- `mapObject` to go through the keys and values in each of the objects of the array.
- `upper` to set each key to upper case.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
items: payload.books map (item, index) -> {
    book: item mapObject (value, key) -> {
        (upper(key)): value
    }
}
```

*Input JSON Payload:*

```
{  
  "books": [  
    {  
      "-category": "cooking",  
      "title": "Everyday Italian",  
      "author": "Giada De Laurentiis",  
      "year": "2005",  
      "price": "30.00"  
    },  
    {  
      "-category": "children",  
      "title": "Harry Potter",  
      "author": "J K. Rowling",  
      "year": "2005",  
      "price": "29.99"  
    },  
    {  
      "-category": "web",  
      "title": "XQuery Kick Start",  
      "author": [  
        "James McGovern",  
        "Per Bothner",  
        "Kurt Cagle",  
        "James Linn",  
        "Vaidyanathan Nagarajan"  
      ],  
      "year": "2003",  
      "price": "49.99"  
    },  
    {  
      "-category": "web",  
      "-cover": "paperback",  
      "title": "Learning XML",  
      "author": "Erik T. Ray",  
      "year": "2003",  
      "price": "39.95"  
    }  
  ]  
}
```

*Output JSON:*

```
{  
  "items": [  
    {  
      "book": {  
        "-CATEGORY": "cooking",  
        "TITLE": "Everyday Italian",  
        "AUTHOR": "Giada De Laurentiis",  
        "YEAR": "2005",  
        "PRICE": "30.00"  
      }  
    },  
    {  
      "book": {  
        "-CATEGORY": "children",  
        "TITLE": "Harry Potter",  
        "AUTHOR": "J K. Rowling",  
        "YEAR": "2005",  
        "PRICE": "29.99"  
      }  
    },  
    {  
      "book": {  
        "-CATEGORY": "web",  
        "TITLE": "XQuery Kick Start",  
        "AUTHOR": [  
          "James McGovern",  
          "Per Bothner",  
          "Kurt Cagle",  
          "James Linn",  
          "Vaidyanathan Nagarajan"  
        ],  
        "YEAR": "2003",  
        "PRICE": "49.99"  
      }  
    },  
    {  
      "book": {  
        "-CATEGORY": "web",  
        "-COVER": "paperback",  
        "TITLE": "Learning XML",  
        "AUTHOR": "Erik T. Ray",  
        "YEAR": "2003",  
        "PRICE": "39.95"  
      }  
    }  
  ]  
}
```

## Second Example

This example uses the `mapObject` function to iterate through the keys and values of the object that results from using `groupBy` on the payload. If some objects of the input payload have the same values in the `FirstName`, `LastName` and `Age` keys, the DataWeave script transforms those objects into a single row in a CSV file. The remaining values in the `Team Name` and `Role` keys for those objects are concatenated with `:` in the single CSV row.

*Input JSON Payload:*

```
[  
 {  
 "Sr.No.": 1,  
 "FirstName": "Charles",  
 "LastName": "Lock",  
 "Age": 40,  
 "Team Name": "Scrum team 1",  
 "Role": "developer"  
 },  
 {  
 "Sr.No.": 2,  
 "FirstName": "Josh",  
 "LastName": "Rodriguez",  
 "Age": 45,  
 "Team Name": "architecture",  
 "Role": "SA"  
 },  
 {  
 "Sr.No.": 3,  
 "FirstName": "Josh",  
 "LastName": "Rodriguez",  
 "Age": 45,  
 "Team Name": "technology",  
 "Role": "advisor"  
 },  
 {  
 "Sr.No.": 4,  
 "FirstName": "Josh",  
 "LastName": "Rodriguez",  
 "Age": 35,  
 "Team Name": "development",  
 "Role": "developer"  
 },  
 {  
 "Sr.No.": 5,  
 "FirstName": "Jane",  
 "LastName": "Rodriguez",  
 "Age": 30,  
 "Team Name": "architecture",  
 "Role": "SA"  
 },
```

```
{
  "Sr.No.": 6,
  "FirstName": "Jane",
  "LastName": "Rodriguez",
  "Age": 30,
  "Team Name": "Scrum team 1",
  "Role": "developer"
},
{
  "Sr.No.": 7,
  "FirstName": "Josh",
  "LastName": "Lee",
  "Age": 42,
  "Team Name": "Scrum team1",
  "Role": "developer"
}
]
```

In the previous input example, **Sr.No. 2** and **Sr.No. 3**, as well as **Sr.No. 5** and **Sr.No. 6**, refer to the same person, as they have the same first name, last name, and age. However, **Sr.No. 4** refers to a different person, as it has the same first and last name but a different age.

*DataWeave Script:*

```
output application/csv
---
valuesOf(
  payload
    groupBy ((item, index) -> (
      item.FirstName ++ item.LastName ++ item.Age))
    mapObject ((value, key, index) ->
      (index): {
        "Sr.No.": value."Sr.No." joinBy ":" ,
        "FirstName": value.FirstName[0],
        "LastName": value.LastName[0],
        "Age": value.Age[0],
        "Team Name": value."Team Name" joinBy ":" ,
        "Role": value.Role joinBy ":"})
    )
)
```

The DataWeave script merges the repeated values of keys **FirstName**, **LastName** and **Age** of the objects with key-value **Sr.No. 2** and **Sr.No. 3**, as well as **Sr.No. 5** and **Sr.No. 6**. These values are separated with **,** in the single CSV row. The values of **Team Name** and **Role** for those objects are concatenated with **:** in the single CSV row.

*Output CSV:*

```
Sr.No.,FirstName,LastName,Age,Team Name,Role  
1,Charles,Lock,40,Scrum team 1,developer  
2:3,Josh,Rodriguez,45,architecture:technology,SA:advisor  
4,Josh,Rodriguez,35,development,developer  
5:6,Jane,Rodriguez,30,architecture:Scrum team 1,SA:developer  
7,Josh,Lee,42,Scrum team1,developer
```

## Related Examples

- [Map Data](#)
- [Map the Objects within an Array](#)
- [Extract Data](#)
- [Perform a Basic Transformation](#)
- [Rename Keys](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Map the Objects within an Array

This DataWeave example uses the DataWeave `map` function to iterate through the object elements that match the key `book`. The input also includes the key `magazine`, which is ignored. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses these DataWeave functions:

- The multi-value selector `*book` to return an array with the elements that match the key "book".
- `map` to go through each object in the array that's returned by the multi-value selector.

*DataWeave Script:*

```
%dw 2.0
var myInputExample = {
    "inventory": {
        "book" : {
            "category": "cooking",
            "title": "Everyday Italian",
            "author": "Giada De Laurentiis",
            "year": "2005",
            "price": "30.00"
        },
        "book" :{
            "category": "children",
            "title": "Harry Potter",
            "author": "J K. Rowling",
            "year": "2005",
            "price": "29.99"
        },
        "book" :{
            "category": "web",
            "title": "Learning XML",
            "author": "Erik T. Ray",
            "year": "2003",
            "price": "39.95"
        },
        "magazine" :{
            "category": "web",
            "title": "Wired Magazine",
            "edition": "03-2017",
            "price": "15.95"
        },
        "magazine" :{
            "category": "business",
            "title": "Time Magazine",
            "edition": "04-2017",
            "price": "17.95"
        }
    }
}
output application/json
---
items: myInputExample.inventory.*book map (item, index) -> {
    "theType": "book",
    "theID": index,
    "theCategory": item.category,
    "theTitle": item.title,
    "theAuthor": item.author,
    "theYear": item.year,
    "thePrice": item.price as Number
}
```

*Output JSON:*

```
{  
  "items": [  
    {  
      "theType": "book",  
      "theID": 0,  
      "theCategory": "cooking",  
      "theTitle": "Everyday Italian",  
      "theAuthor": "Giada De Laurentiis",  
      "theYear": "2005",  
      "thePrice": 30.00  
    },  
    {  
      "theType": "book",  
      "theID": 1,  
      "theCategory": "children",  
      "theTitle": "Harry Potter",  
      "theAuthor": "J K. Rowling",  
      "theYear": "2005",  
      "thePrice": 29.99  
    },  
    {  
      "theType": "book",  
      "theID": 2,  
      "theCategory": "web",  
      "theTitle": "Learning XML",  
      "theAuthor": "Erik T. Ray",  
      "theYear": "2003",  
      "thePrice": 39.95  
    }  
  ]  
}
```

## Related Examples

- [Map Data](#)
- [Map an Object](#)
- [Exclude Field](#)
- [Rename Keys](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Map Based On an External Definition

You can create a transformation that can dynamically change what it does depending on a definition input. This DataWeave example receives both a payload input, and a variable named `mapping` that specifies how to rename each field and what default value to use in each. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses the following:

- A `map` function to go through all of the elements in the input array. Also a second `map` function to go through each field in each element.
- A custom function that applies the changes specified in the `mapping` variable.
- `default` to set a default value, that comes from the `mapping` variable.

*DataWeave Script:*

```
%dw 2.0
output application/json
var applyMapping = (in, mappingsDef) -> (
    mappingsDef map (def) -> {
        (def.target) : in[def.source] default def."default"
    }
)
---
payload.sfdc_users.*sfdc_user map (user) -> (
    applyMapping(user, vars.mappings)
)
```

*Input XML Payload:*

```
<sfdc_users>
    <sfdc_user>
        <sfdc_name>Mariano</sfdc_name>
        <sfdc_last_name>Achaval</sfdc_last_name>
        <sfdc_employee>true</sfdc_employee>
    </sfdc_user>
    <sfdc_user>
        <sfdc_name>Julian</sfdc_name>
        <sfdc_last_name>Esevich</sfdc_last_name>
        <sfdc_employee>true</sfdc_employee>
    </sfdc_user>
    <sfdc_user>
        <sfdc_name>Leandro</sfdc_name>
        <sfdc_last_name>Shokida</sfdc_last_name>
    </sfdc_user>
</sfdc_users>
```

*Input Mule Event Variable (JSON):*

```
[  
  {  
    "source": "sfdc_name",  
    "target": "name",  
    "default": "---"  
  },  
  {  
    "source": "sfdc_last_name",  
    "target": "lastName",  
    "default": "---"  
  },  
  {  
    "source": "sfdc_employee",  
    "target": "user",  
    "default": true  
  }  
]
```

*Output JSON:*

```
[  
  [  
    {"name": "Mariano"},  
    {"lastName": "Achaval"},  
    {"user": "true"}  
  ],  
  [  
    {"name": "Julian"},  
    {"lastName": "Esevich"},  
    {"user": "true"}  
  ],  
  [  
    {"name": "Leandro"},  
    {"lastName": "Shokida"},  
    {"user": true}  
  ]  
]
```

## Related Examples

- [Rename Keys](#)
- [Exclude Fields from the Output](#)
- [Output a Field When Present](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Rename Keys

This DataWeave example renames some keys in a JSON object, while retaining the names of all others in the output. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses these functions:

- `mapObject` to go through the `key:value` pairs in a JSON object.
- `if` by itself to determine when to change the name of a key.
- `if` with `and` to retain the name of all keys except the two with new names.
- `as` to coerce the type of the keys into a String.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
payload.flights map (flight) -> {
    (flight mapObject (value, key) -> {
        (emptySeats: value) if(key as String == 'availableSeats'),
        (airline: value) if(key as String == 'airlineName'),
        ((key):value) if(key as String !='availableSeats' and key as String != 'airlineName')
    })
}
```

*Input JSON Payload:*

```
{  
  "flights": [  
    {  
      "availableSeats": 45,  
      "airlineName": "Ryan Air",  
      "aircraftBrand": "Boeing",  
      "aircraftType": "737",  
      "departureDate": "12/14/2017",  
      "origin": "BCN",  
      "destination": "FCO"  
    },  
    {  
      "availableSeats": 15,  
      "airlineName": "Ryan Air",  
      "aircraftBrand": "Boeing",  
      "aircraftType": "747",  
      "departureDate": "08/03/2017",  
      "origin": "FCO",  
      "destination": "DFW"  
    }]  
}
```

*Output JSON:*

```
[  
  {  
    "emptySeats": 45,  
    "airline": "Ryan Air",  
    "aircraftBrand": "Boeing",  
    "aircraftType": "737",  
    "departureDate": "12/14/2017",  
    "origin": "BCN",  
    "destination": "FCO"  
  },  
  {  
    "emptySeats": 15,  
    "airline": "Ryan Air",  
    "aircraftBrand": "Boeing",  
    "aircraftType": "747",  
    "departureDate": "08/03/2017",  
    "origin": "FCO",  
    "destination": "DFW"  
  }]  
]
```

A more maintainable way to produce the same output uses DataWeave pattern matching.

The example uses the following functions and statements:

- `mapObject` to go through the `key:value` pairs in a JSON object.
- `match` on each key in the input.
- `case` statements to change the name of matching keys in the input.
- `else` statement to retain the name of keys that do not require a name change.

*DataWeave Script:*

```
%dw 2.0
output application/json
fun renameKey(key: Key) = key match {
    case "availableSeats" -> "emptySeats"
    case "airlineName" -> "airline"
    else -> (key)
}
---
payload.flights map (flight) ->
flight mapObject (value, key) -> {
    (renameKey(key)) : value
}
```

*Input JSON Payload:*

```
{
  "flights": [
    {
      "availableSeats": 45,
      "airlineName": "Ryan Air",
      "aircraftBrand": "Boeing",
      "aircraftType": "737",
      "departureDate": "12/14/2017",
      "origin": "BCN",
      "destination": "FCO"
    },
    {
      "availableSeats": 15,
      "airlineName": "Ryan Air",
      "aircraftBrand": "Boeing",
      "aircraftType": "747",
      "departureDate": "08/03/2017",
      "origin": "FCO",
      "destination": "DFW"
    }
  ]
}
```

*Output JSON:*

```
[  
  {  
    "emptySeats": 45,  
    "airline": "Ryan Air",  
    "aircraftBrand": "Boeing",  
    "aircraftType": "737",  
    "departureDate": "12/14/2017",  
    "origin": "BCN",  
    "destination": "FCO"  
  },  
  {  
    "emptySeats": 15,  
    "airline": "Ryan Air",  
    "aircraftBrand": "Boeing",  
    "aircraftType": "747",  
    "departureDate": "08/03/2017",  
    "origin": "FCO",  
    "destination": "DFW"  
  }  
]
```

## Related Examples

- [Output a Field When Present](#)
- [Map Based On an External Definition](#)
- [Pass Functions as Arguments](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Output a Field When Present

This DataWeave example outputs a field if it is present in the input, a JSON array. The first object in the array contains `"insurance"`, while the second does not. The XML output mirrors this structure. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses these functions:

- `map` to go through every element within the input array.
- `if($.insurance?)` to determine when to output an `insurance` field.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
users: { (payload map
    user: {
        name: $.name,
        (insurance: $.insurance) if ($.insurance?)
    } )
}
```

*Input JSON Payload:*

```
[{
  {
    "name" : "Julian",
    "gender" : "Male",
    "age" : 41,
    "insurance": "Osde"
  },
  {
    "name" : "Mariano",
    "gender" : "Male",
    "age" : 33
  }
]
```

*Output XML:*

```
<?xml version='1.0' encoding='US-ASCII'?>
<users>
  <user>
    <name>Julian</name>
    <insurance>Osde</insurance>
  </user>
  <user>
    <name>Mariano</name>
  </user>
</users>
```

## Related Examples

- [Map Data](#)
- [Exclude Fields from the Output](#)
- [Insert an Attribute into an XML Tag](#)
- [Conditionally Reduce a List Via a Function](#)

- [Change Format According to Type](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Change Format According to Type

This DataWeave example applies changes to the keys in an object, depending on the type of their corresponding values. When the element is an array, the keys are pluralized. When it's a number, they are set to camel case. In any other case they are capitalized.

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses:

- `mapObject` to go through each element in the payload.
- `if` to check that an element is of a given type.
- `camelize` from the String library to apply a camel case format to strings (all words stringed together, using upper case letters to mark separations).
- `capitalize` from the String library to apply a capitalized format to strings (all words are separate and start with an upper case letter).
- `pluralize` from the String library to change singular words into plural.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
payload mapObject ((elementValue, elementKey) -> {
    if (elementValue is Array)
        pluralize(elementKey)
    else if(elementValue is Number)
        camelize(elementKey)
    else capitalize(elementKey)) : elementValue
})
```

*Input JSON Payload:*

```
{  
    "VersionNo": 1.6,  
    "StoreOfOrigin": "SFO",  
    "Item":  
        [  
            [  
                {  
                    "ID": "34546315801",  
                    "DeliveryMethod": "AIR",  
                    "Quantity": 8  
                },  
                {  
                    "ID": "56722087289",  
                    "Boxes": 3,  
                    "DeliveryMethod": "GROUND",  
                    "Quantity": 2  
                }  
            ]  
        ]  
}
```

*Output JSON:*

```
{  
    "versionNo": 1.6,  
    "Store Of Origin": "SFO",  
    "Items": [  
        {  
            "ID": "34546315801",  
            "DeliveryMethod": "AIR",  
            "Quantity": 8  
        },  
        {  
            "ID": "56722087289",  
            "Boxes": 3,  
            "DeliveryMethod": "GROUND",  
            "Quantity": 2  
        }  
    ]  
}
```

## Related Examples

- [Output Self-closing XML tags](#)
- [Insert an Attribute into an XML Tag](#)

## See Also

- [Selectors](#)

## Regroup Fields

These DataWeave examples take input that is grouped under one field and transform it into a new structure that groups data under another field. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

Both examples use these functions:

- `groupBy` to organize the fields by `subject`
- `mapObject` and `map` to map the fields from the input to the new hierarchy.

### Example: XML to JSON

*DataWeave Script:*

```
%dw 2.0
output application/json
---
classrooms: payload..*teacher groupBy $.subject mapObject ((teacherGroup, subject) ->
{
    class: {
        name: subject,
        teachers: { (teacherGroup map {
            teacher:{ 
                name: $.name,
                lastName: $.lastName
            }
        })
    }
})
})
```

*Input XML Payload:*

```
<school>
  <teachers>
    <teacher>
      <name>Mariano</name>
      <lastName>De Achaval</lastName>
      <subject>DW</subject>
    </teacher>
    <teacher>
      <name>Emiliano</name>
      <lastName>Lesende</lastName>
      <subject>DW</subject>
    </teacher>
    <teacher>
      <name>Leandro</name>
      <lastName>Shokida</lastName>
      <subject>Scala</subject>
    </teacher>
  </teachers>
</school>
```

*Output JSON:*

```
{
  "classrooms": [
    {
      "class": {
        "name": "DW",
        "teachers": [
          {
            "teacher": {
              "name": "Mariano",
              "lastName": "De Achaval"
            }
          },
          {
            "teacher": {
              "name": "Emiliano",
              "lastName": "Lesende"
            }
          }
        ],
        "class": {
          "name": "Scala",
          "teachers": [
            {
              "teacher": {
                "name": "Leandro",
                "lastName": "Shokida"
              }
            }
          ]
        }
      }
    }
  ]
}
```

## Example: JSON to JSON

This DataWeave example changes the hierarchy of a JSON object. The output groups fields by **language** and adds a new element, **attendees**, that contains the names of attendees for each course.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{
    "langs" :
        payload.langs groupBy $.language
        mapObject ((nameGroup, language) -> {
            (language): {
                "attendees" : nameGroup map {
                    name: $.name
                }
            }
        })
}
```

*Input JSON Payload:*

```
{
    "langs": [
        {
            "name": "Alex",
            "language": "Java"
        },
        {
            "name": "Kris",
            "language": "Scala"
        },
        {
            "name": "Jorge",
            "language": "Java"
        }
    ]
}
```

*Output JSON:*

```
{  
  "langs": {  
    "Java": {  
      "attendees": [  
        {  
          "name": "Alex"  
        },  
        {  
          "name": "Jorge"  
        }  
      ]  
    },  
    "Scala": {  
      "attendees": [  
        {  
          "name": "Kris"  
        }  
      ]  
    }  
  }  
}
```

## Related Examples

- [Map Data](#)
- [Reference Multiple Inputs](#)
- [Define a Function that Flattens Data in a List](#)
- [Zip Arrays Together](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Zip Arrays Together

This DataWeave example restructures bills of materials for Ikea-style furniture. The input contains the measurements and amounts of screws in two separate arrays that run in parallel, the transformation reorders them so that the "screws" array is made up of tuples, each with a measurement and its corresponding amount. The same is applied to wooden boards: the input contains two arrays with the x and the y measurements of each; the transformation rearranges them into a series of tuples, one for each board. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2](#)

[examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

It uses these DataWeave functions:

- `map` to go through the elements in the main array.
- `zip` to rearrange pairs of long arrays, so that they're grouped by index into multiple two-element arrays.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
payload map (item, index) ->
{
    name: item.name,
    id: item.itemID,
    screws: zip(item.screws.size, item.screws.quantity),
    measurements: zip(item.measurements.x,item.measurements.y )
}
```

*Input JSON Payload:*

```
[  
  {  
    "name": "wooden-chair",  
    "itemID": "23665",  
    "screws": {  
      "size": [4, 6, 10],  
      "quantity": [15, 8, 28]  
    },  
    "measurements":  
    {"x": [25, 46, 46, 16, 150, 5, 100, 100, 8],  
     "y": [15, 4, 4, 80, 3, 4, 4, 15]  
    }  
  },  
  {  
    "name": "coffee-table",  
    "itemID": "14398",  
    "screws": {  
      "size": [3, 8, 10],  
      "quantity": [8, 12, 20]  
    },  
    "measurements":  
    {"x": [55, 48, 48, 48, 48, 30, 30, 30, 30],  
     "y": [55, 40, 40, 40, 50, 4, 4, 4, 4]  
    }  
  }  
]
```

*Output JSON:*

```
[  
  {  
    "name": "wooden-chair",  
    "id": "23665",  
    "screws": [  
      [  
        4,  
        15  
      ],  
      [  
        6,  
        8  
      ],  
      [  
        10,  
        28  
      ]  
    ],  
    "measurements": [  
      ...  
    ]  
  }  
]
```

```
[  
  25,  
  15  
,  
 [  
  46,  
  4  
,  
 [  
  46,  
  4  
,  
 [  
  16,  
  80  
,  
 [  
  150,  
  3  
,  
 [  
  5,  
  4  
,  
 [  
  100,  
  4  
,  
 [  
  100,  
  15  
,  
 ]]  
 ]]  
,  
{  
 "name": "coffee-table",  
 "id": "14398",  
 "screws": [  
  [  
    3,  
    8  
,  
  [  
    8,  
    12  
,  
  [  
    10,  
    20  
,  
  ],  
 ]],  
 },
```

```
"measurements": [
  [
    55,
    55
  ],
  [
    [
      48,
      40
    ],
    [
      [
        48,
        40
      ],
      [
        [
          48,
          40
        ],
        [
          [
            48,
            50
          ],
          [
            [
              30,
              4
            ],
            [
              [
                30,
                4
              ],
              [
                [
                  30,
                  4
                ],
                [
                  [
                    30,
                    4
                  ]
                ]
              ]
            ]
          ]
        ]
      ]
    ]
  ]
}
```

## Related Examples

- [Regroup Fields](#)
- [Perform a Basic Transformation](#)
- [Define a Custom Addition Function](#)
- [Pick Top Elements](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Pick Top Elements

This DataWeave example sorts an array of candidates by the score they got in a test, then picks only the ones with the best score, as many as there are open positions to fill. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

This example uses the following:

- `map` to go through each of the candidates in the input.
- `orderBy` to order the list of candidates according to their score.
- `[n to n]` to select only a section of the full array of candidates. As the array is ordered in ascending order, the top candidates are at the end of the array, so you must use negative indexes. With the provided input, it selects from -3 to -1, -1 being the last index in the array.

*DataWeave Script:*

```
%dw 2.0
output application/json
---
{
    TopCandidateList: (payload.candidates map ((candidate) -> {
        firstName: candidate.name,
        rank: candidate.score
    }) orderBy $.rank) [ -payload.availablePositions to -1]
}
```

*Input JSON Payload:*

```
{  
  "availablePositions": 3,  
  "candidates":  
  [  
    {  
      "name": "Gunther Govan",  
      "score": 99  
    },  
    {  
      "name": "Michael Patrick",  
      "score": 35  
    },  
    {  
      "name": "Amalia Silva",  
      "score": 96  
    },  
    {  
      "name": "Tom Mathews",  
      "score": 40  
    },  
    {  
      "name": "Simon Wilson",  
      "score": 84  
    },  
    {  
      "name": "Janet Nguyen",  
      "score": 52  
    }  
  ]  
}
```

*Output JSON:*

```
{  
  "TopCandidateList": [  
    {  
      "firstName": "Simon Wilson",  
      "rank": 84  
    },  
    {  
      "firstName": "Amalia Silva",  
      "rank": 96  
    },  
    {  
      "firstName": "Gunther Govan",  
      "rank": 99  
    }  
  ]  
}
```

## Related Examples

- [Output Self-closing XML tags](#)
- [Remove Certain XML Attributes](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Change the Value of a Field

The following DataWeave examples show how to use `update` and `mask` to change the values of some XML elements.

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

### Example: Using Update to Change Values

This example uses:

- `update` to update specified fields of the payload with new values
- single-value `(.)` and multi-value `(.*)` selectors to navigate the payload and select the fields to update

The following script shows how the `update` operator works. The example creates a new `users` list by updating each `user` in the payload with a new `user`. The transformation converts `first_name`,

`middle_name` and `last_name` to uppercase values:

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
payload update {
    case user at .users.*user -> user update {
        case .personal_information.first_name ->
upper(user.personal_information.first_name)
        case .personal_information.middle_name ->
upper(user.personal_information.middle_name)
        case .personal_information.last_name -> upper(user.personal_information.last_name)
    }
}
```

*Input XML Payload:*

```
<users>
  <user>
    <personal_information>
      <first_name>Emiliano</first_name>
      <middle_name>Romoaldo</middle_name>
      <last_name>Lesende</last_name>
      <ssn>001-08-84382</ssn>
    </personal_information>
    <login_information>
      <username>3milianno</username>
      <password>mypassword1234</password>
    </login_information>
  </user>
  <user>
    <personal_information>
      <first_name>Mariano</first_name>
      <middle_name>Toribio</middle_name>
      <last_name>de Achaval</last_name>
      <ssn>002-05-34738</ssn>
    </personal_information>
    <login_information>
      <username>machaval</username>
      <password>mypassword4321</password>
    </login_information>
  </user>
</users>
```

*Output XML:*

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user>
    <personal_information>
      <first_name>EMILIANO</first_name>
      <middle_name>ROMOALDO</middle_name>
      <last_name>LESENDE</last_name>
      <ssn>001-08-84382</ssn>
    </personal_information>
    <login_information>
      <username>3milianno</username>
      <password>mypassword1234</password>
    </login_information>
  </user>
  <user>
    <personal_information>
      <first_name>MARIANO</first_name>
      <middle_name>TORIBIO</middle_name>
      <last_name>DE ACHAVAL</last_name>
      <ssn>002-05-34738</ssn>
    </personal_information>
    <login_information>
      <username>machaval</username>
      <password>mypassword4321</password>
    </login_information>
  </user>
</users>
```

## Example: Using Mask to Change Values

DataWeave provides a simple way to mask values, without specifying the path to each field:

- **mask** updates all simple elements that match the selected name throughout the input with the specified mask.

The following example masks the **ssn** and **password** values with a set of asterisks (\*\*\*\*):

*DataWeave Script:*

```
%dw 2.0
import * from dw::util::Values
output application/xml
---
(payload mask "ssn" with "****") mask "password" with "****"
```

*Input XML Payload:*

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user>
    <personal_information>
      <first_name>EMILIANO</first_name>
      <middle_name>ROMOALDO</middle_name>
      <last_name>LESENDE</last_name>
      <ssn>001-08-84382</ssn>
    </personal_information>
    <login_information>
      <username>3milianno</username>
      <password>mypassword1234</password>
    </login_information>
  </user>
  <user>
    <personal_information>
      <first_name>MARIANO</first_name>
      <middle_name>TORIBIO</middle_name>
      <last_name>DE ACHAVAL</last_name>
      <ssn>002-05-34738</ssn>
    </personal_information>
    <login_information>
      <username>machaval</username>
      <password>mypassword4321</password>
    </login_information>
  </user>
</users>
```

*Output XML:*

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user>
    <personal_information>
      <first_name>EMILIANO</first_name>
      <middle_name>ROMOALDO</middle_name>
      <last_name>LESENDE</last_name>
      <ssn>****</ssn>
    </personal_information>
    <login_information>
      <username>3milianno</username>
      <password>****</password>
    </login_information>
  </user>
  <user>
    <personal_information>
      <first_name>MARIANO</first_name>
      <middle_name>TORIBIO</middle_name>
      <last_name>DE ACHAVAL</last_name>
      <ssn>****</ssn>
    </personal_information>
    <login_information>
      <username>machaval</username>
      <password>****</password>
    </login_information>
  </user>
</users>
```

## Related Examples

- [Perform a Basic Transformation](#)
- [Output a Field When Present](#)
- [Insert an Attribute into an XML Tag](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Exclude Fields from the Output

This DataWeave example excludes specific XML elements from the output. You might perform a task like this to remove sensitive data. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For

other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses these functions:

- `-` to remove specific **key:value** pairs (here, the `ssn` and `password` XML elements).
- `mapObject` to go through the XML elements.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
users: {
    (payload.users mapObject {
        user: {
            personal_information: $.personal_information - "ssn",
            login_information: $.login_information - "password"
        }
    })
}
```

*Input XML Payload:*

```
<users>
  <user>
    <personal_information>
      <first_name>Emiliano</first_name>
      <middle_name>Romoaldo</middle_name>
      <last_name>Lesende</last_name>
      <ssn>001-08-84382</ssn>
    </personal_information>
    <login_information>
      <username>3miliano</username>
      <password>mypassword1234</password>
    </login_information>
  </user>
  <user>
    <personal_information>
      <first_name>Mariano</first_name>
      <middle_name>Toribio</middle_name>
      <last_name>de Achaval</last_name>
      <ssn>002-05-34738</ssn>
    </personal_information>
    <login_information>
      <username>machaval</username>
      <password>mypassword4321</password>
    </login_information>
  </user>
</users>
```

*Output XML:*

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user>
    <personal_information>
      <first_name>Emiliano</first_name>
      <middle_name>Romoaldo</middle_name>
      <last_name>Lesende</last_name>
    </personal_information>
    <login_information>
      <username>3miliano</username>
    </login_information>
  </user>
  <user>
    <personal_information>
      <first_name>Mariano</first_name>
      <middle_name>Toribio</middle_name>
      <last_name>de Achaval</last_name>
    </personal_information>
    <login_information>
      <username>machaval</username>
    </login_information>
  </user>
</users>
```

## Related Examples

- [Output a Field When Present](#)
- [Change the Value of a Field](#)
- [Conditionally Reduce a List Via a Function](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Use Constant Directives

This DataWeave example builds URLs from constants that are defined as variables (`var`) in the DataWeave header. It also conditionally outputs fields if they are present in the input. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses these functions:

- `++` (concatenate) to build constant directives into URLs.

- **map** to go through the fields in the input.
- **if** to output **summary** and **brand** fields.

*DataWeave Script:*

```
%dw 2.0
var baseUrl = "http://alainn-cosmetics.cloudhub.io/api/v1.0"
var urlPage = "http://alainn-cosmetics.cloudhub.io/api/v1.0/items"
var fullUrl = "http://alainn-cosmetics.cloudhub.io/api/v1.0/items"
var pageIndex = 0
var requestedPageSize = 4
output application/json
---
using (pageSize = payload.getItemsResponse PageInfo.pageSize) {
    links: [
        {
            href: fullUrl,
            rel : "self"
        },
        {
            href: urlPage ++ "?pageIndex=" ++ (pageIndex + pageSize) ++ "&pageSize="
++ requestedPageSize,
            rel: "next"
        },
        ({
            href: urlPage ++ "?pageIndex=" ++ (pageIndex - pageSize) ++ "&pageSize="
++ requestedPageSize,
            rel: "prev"
        }) if(pageIndex > 0)
    ],
    collection: {
        size: pageSize,
        items: payload.getItemsResponse.*Item map (item) -> {
            id: item.id,
            'type': item.type,
            name: item.name,
            (summary: item.summary) if(item.summary?),
            (brand: item.brand) if(item.brand?),
            links: (item.images.*image map (image) -> {
                href: trim(image),
                rel: image.@'type'
            }) + {
                href: baseUrl ++ "/" ++ item.id,
                rel: "self"
            }
        }
    }
}
```

*Input XML Payload:*

```
<ns0:getItemsResponse xmlns:ns0="http://www.alainn.com/SOA/message/1.0">
  <ns0:PageInfo>
    <pageIndex>0</pageIndex>
    <pageSize>5</pageSize>
  </ns0:PageInfo>
  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">
    <id>B0015BYNRO</id>
    <type>Oils</type>
    <name>Now Foods LANOLIN PURE</name>
    <images>
      <image type="SwatchImage">http://ecx.images-
amazon.com/images/I/11Qoe774Q4L._SL30_.jpg
      </image>
    </images>
  </ns1:Item>
  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">
    <id>B002K8AD02</id>
    <type>Bubble Bath</type>
    <name>Deep Steep Honey Bubble Bath</name>
    <summary>Disclaimer: This website is for informational purposes only.
      Always check the actual product label in your possession for the most
      accurate ingredient information due to product changes or upgrades
      that may not yet be reflected on our web site. These statements made
      in this website have not been evaluated by the Food and Drug
      Administration. The products offered are not intended to diagnose,
      treat
    </summary>
    <images>
      <image type="SwatchImage">http://ecx.images-
amazon.com/images/I/216ytnM0eXL._SL30_.jpg
      </image>
    </images>
  </ns1:Item>
  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">
    <id>B000I206JK</id>
    <type>Oils</type>
    <name>Now Foods Castor Oil</name>
    <summary>One of the finest natural skin emollients available</summary>
    <images>
      <image type="SwatchImage">http://ecx.images-amazon.com/images/I/21Yz8q-
yQoL._SL30_.jpg
      </image>
    </images>
  </ns1:Item>
  <ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">
    <id>B003Y5XF2S</id>
    <type>Chemical Hair Dyes</type>
    <name>Manic Panic Semi-Permanent Color Cream</name>
    <summary>Ready to use, no mixing required</summary>
    <images>
      <image type="SwatchImage">http://ecx.images-
```

```

amazon.com/images/I/51A2FuX27dL._SL30_.jpg
    </image>
</images>
</ns1:Item>
<ns1:Item xmlns:ns1="http://www.alainn.com/SOA/model/1.0">
    <id>B0016BELU2</id>
    <type>Chemical Hair Dyes</type>
    <name>Herbatint Herbatint Permanent Chestnut (4n)</name>
    <images>
        <image type="SwatchImage">http://ecx.images-
amazon.com/images/I/21woUiM0BdL._SL30_.jpg
            </image>
        </images>
    </ns1:Item>
</ns0:getItemsResponse>

```

*Output JSON:*

```
{
  "links": [
    {
      "href": "http://alainn-cosmetics.cloudhub.io/api/v1.0/items",
      "rel": "self"
    },
    {
      "href": "http://alainn-cosmetics.cloudhub.io/api/v1.0/items?pageIndex=5&pageSize=4",
      "rel": "next"
    }
  ],
  "collection": {
    "size": "5",
    "items": [
      {
        "id": "B0015BYNRO",
        "type": "Oils",
        "name": "Now Foods LANOLIN PURE",
        "links": [
          {
            "href": "http://ecx.images-amazon.com/images/I/11Qoe774Q4L._SL30_.jpg",
            "rel": "SwatchImage"
          },
          {
            "href": "http://alainn-cosmetics.cloudhub.io/api/v1.0/B0015BYNRO",
            "rel": "self"
          }
        ]
      },
      {
        "id": "B002K8AD02",
        "type": "Bubble Bath",

```

```
"name": "Deep Steep Honey Bubble Bath",
"summary": "Disclaimer: This website is for informational purposes only.\nAlways check the actual product label in your possession for the most\naccurate ingredient information due to product changes or upgrades\nthat may not yet be reflected on our web site. These statements made\nin this website have not been evaluated by the Food and Drug\nAdministration. The products offered are not intended to diagnose,\ntreat\n",
"links": [
{
  "href": "http://ecx.images-amazon.com/images/I/216ytnMOeXL._SL30_.jpg",
  "rel": "SwatchImage"
},
{
  "href": "http://alainn-cosmetics.cloudhub.io/api/v1.0/B002K8AD02",
  "rel": "self"
}
],
},
{
  "id": "B000I206JK",
  "type": "Oils",
  "name": "Now Foods Castor Oil",
  "summary": "One of the finest natural skin emollients available",
  "links": [
    {
      "href": "http://ecx.images-amazon.com/images/I/21Yz8q-yQoL._SL30_.jpg",
      "rel": "SwatchImage"
    },
    {
      "href": "http://alainn-cosmetics.cloudhub.io/api/v1.0/B000I206JK",
      "rel": "self"
    }
  ]
},
{
  "id": "B003Y5XF2S",
  "type": "Chemical Hair Dyes",
  "name": "Manic Panic Semi-Permanent Color Cream",
  "summary": "Ready to use, no mixing required",
  "links": [
    {
      "href": "http://ecx.images-amazon.com/images/I/51A2FuX27dL._SL30_.jpg",
      "rel": "SwatchImage"
    },
    {
      "href": "http://alainn-cosmetics.cloudhub.io/api/v1.0/B003Y5XF2S",
      "rel": "self"
    }
  ]
},
```

```

    "id": "B0016BELU2",
    "type": "Chemical Hair Dyes",
    "name": "Herbatint Herbatint Permanent Chestnut (4n)",
    "links": [
        {
            "href": "http://ecx.images-amazon.com/images/I/21woUiM0BdL._SL30_.jpg",
            "rel": "SwatchImage"
        },
        {
            "href": "http://alainn-cosmetics.cloudbuck.io/api/v1.0/B0016BELU2",
            "rel": "self"
        }
    ]
}

```

## Related Examples

- [Define a Custom Addition Function](#)
- [Conditionally Reduce a List Via a Function](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Define a Custom Addition Function

This DataWeave example takes in a set of item prices and stock amounts and uses several functions to calculate subtotals, deduct discounts and add taxes. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

It makes use of the following:

- **map** function to go through all the items in the input.
- basic math operations like **\***, **+** and **-**.
- String concatenation with **++**.
- **as** to force string values into numbers.
- Fixed values in the header to set tax and discount amounts.
- A custom function in the header, to define once and use multiple times in the code.
- **reduce** function to aggregate the various items into a total.

*DataWeave Script:*

```
%dw 2.0
output application/json
var tax = 0.085
var discount = 0.05
fun getSubtotal (items) = items reduce ((item, accumulator = 0) ->
    accumulator + (item.unit_price * item.quantity * (1 - discount)))
---
invoice: {
    header: payload.invoice.header,
    items: { (payload.invoice.items map {
        item : {
            description: $.description,
            quantity: $.quantity,
            unit_price: $.unit_price,
            discount: (discount * 100) as Number ++ "%",
            subtotal: $.unit_price * $.quantity * (1 - discount)
        }
    }) },
    totals:
    {
        subtotal: getSubtotal(payload.invoice.items),
        tax: (tax * 100) as Number ++ "%",
        total: getSubtotal(payload.invoice.items) * (1 + tax)
    }
}
```

*Input JSON Payload:*

```
{
    "invoice": {
        "header": {
            "customer_name": "ACME, Inc.",
            "customer_state": "CA"
        },
        "items": [
            {
                "description": "Product 1",
                "quantity": "2",
                "unit_price": "10"
            },
            {
                "description": "Product 2",
                "quantity": "1",
                "unit_price": "30"
            }
        ]
    }
}
```

## Output

```
{  
  "invoice": {  
    "header": {  
      "customer_name": "ACME, Inc.",  
      "customer_state": "CA"  
    },  
    "items": {  
      "item": {  
        "description": "Product 1",  
        "quantity": "2",  
        "unit_price": "10",  
        "discount": "5%",  
        "subtotal": 19.00  
      },  
      "item": {  
        "description": "Product 2",  
        "quantity": "1",  
        "unit_price": "30",  
        "discount": "5%",  
        "subtotal": 28.50  
      }  
    },  
    "totals": {  
      "subtotal": 47.50,  
      "tax": "8.500%",  
      "total": 51.53750  
    }  
  }  
}
```

## Related Examples

- [Map Data](#)
- [Use Constant Directives](#)
- [Conditionally Reduce a List Via a Function](#)
- [Define a Function that Flattens Data in a List](#)
- [Pass Functions as Arguments](#)

## See Also

- [DataWeave Core Functions](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Define a Function that Flattens Data in a List

When presented with nested structures of data, you might need to reduce (or "flatten") the data in them to produce a simpler output. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

This DataWeave example flattens the data in the `interests` and `contenttypes` arrays with the function that is defined in the header, and it only outputs `contenttypes` when it is not empty.

The example uses these functions:

- `map` to go through a set elements in the input.
- `reduce` to flatten the arrays.
- `if` to conditionally display a field.
- `splitBy` to parse the input.

*DataWeave Script:*

```
%dw 2.0
output application/json
fun reduceMapFor(data) = data reduce ((## splitBy ":" )[0] ++ "," ++ ($ splitBy ":" )[0])
---
payload.results map() ->
{
    email: $.profile.email,
    name: $.profile.firstName,
    tags: reduceMapFor($.data.interests.tags[0]),
        (contenttypes: reduceMapFor($.data.interests.contenttypes[0]))
if(sizeOf($.data.interests.contenttypes[0]) > 0)
}
```

*Input JSON Payload:*

```
{
  "results": [
    {
      "profile": {
        "firstName": "john",
        "lastName": "doe",
        "email": "johndoe@demo.com"
      },
      "data": {
        "interests": [
          {
            "language": "English",
            "tags": [
              "digital-strategy:Digital Strategy",
              "marketing:Marketing"
            ]
          }
        ]
      }
    }
  ]
}
```

```
        "innovation:Innovation"
    ],
    "contenttypes": []
}
]
}
},
{
    "profile": {
        "firstName": "jane",
        "lastName": "doe",
        "email": "janedoe@demo.com"
    },
    "data": {
        "interests": [
            {
                "language": "English",
                "tags": [
                    "tax-reform:Tax Reform",
                    "retail-health:Retail Health"
                ],
                "contenttypes": [
                    "News",
                    "Analysis",
                    "Case studies",
                    "Press releases"
                ]
            }
        ]
    }
},
{
    "objectsCount": 2,
    "totalCount": 2,
    "statusCode": 200,
    "errorCode": 0,
    "statusReason": "OK"
}
```

## Output

```
[  
  {  
    "email": "johndoe@demo.com",  
    "name": "john",  
    "tags": "digital-strategy,innovation"  
  },  
  {  
    "email": "janedoe@demo.com",  
    "name": "jane",  
    "tags": "tax-reform,retail-health",  
    "contenttypes": "News,Analysis,Case studies,Press releases"  
  }  
]
```

## Related Examples

- [Use Constant Directives](#)
- [Conditionally Reduce a List Via a Function](#)
- [Define a Custom Addition Function](#)
- [Pass Functions as Arguments](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Flatten Elements of Arrays

DataWeave can flatten subarrays of an array and collections of key-value pairs within DataWeave objects, arrays, and subarrays. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

### Flatten Subarrays with Key-Value Pairs into an Array of Objects

This example shows how `flatten` acts on key-value pairs of the input array defined by the variable `arrayOne`. Notice that every key-value pair in the array becomes a separate DataWeave object.

The example uses this function:

- `flatten` to move the elements from the subarrays to the parent array, eliminate the subarrays, and convert all key-value pairs into a list of objects within the parent array.

*DataWeave Script:*

```
%dw 2.0
var arrayOne = [
    [
        "keyOne" : 1,
        "keyTwo" : 2
    ],
    [
        "keyThree" : 3,
        "keyFour" : 4,
        "keyFive" : 5
    ],
    "keySix" : 6
]
output application/json
---
flatten(arrayOne)
```

*Output JSON:*

```
[
{
    "keyOne": 1
},
{
    "keyTwo": 2
},
{
    "keyThree": 3
},
{
    "keyFour": 4
},
{
    "keyFive": 5
},
{
    "keySix": 6
}]
```

## Flatten Combined Arrays

Like the previous DataWeave example, the following returns an array that combines and flattens the elements from two arrays. This second example applies the selector `fruit` to select only the values of the key `fruit` and to exclude other values in the array.

This example uses:

- `++` to combine the objects of two arrays (`arrayOne` and `arrayTwo`) into a single array.
- `flatten` to flatten the combined array into a list of key-value pairs.

*DataWeave Script:*

```
%dw 2.0
var arrayOne = [
    [
        "fruit" : "orange",
        "fruit" : "apple"
    ],
    [
        "fruit" : "grape",
        "notfruit" : "something else"
    ]
]
var arrayTwo = [
    [
        { "fruit" : "kiwi" }
    ],
    "fruit" : "strawberry",
    "fruit" : "plum",
    { "fruit" : "banana" },
    "notfruit" : "something else"
]
output application/json
---
flatten(arrayOne ++ arrayTwo)
```

*Output JSON:*

```
[
    { "fruit": "orange" },
    { "fruit": "apple" },
    { "fruit": "grape" },
    { "notfruit": "something else" },
    { "fruit": "kiwi" },
    { "fruit": "strawberry" },
    { "fruit": "plum" },
    { "fruit": "banana" },
    { "notfruit": "something else" }
]
```

The only difference between the [previous example](#) and the following example is the addition of the `.fruit` selector to the body expression to select all the `fruit` values and to exclude values of the `notfruit` keys from the output array.

*DataWeave Script:*

```
%dw 2.0
var arrayOne = [
    [
        "fruit" : "orange",
        "fruit" : "apple"
    ],
    [
        "fruit" : "grape",
        "notfruit" : "something else"
    ]
]
var arrayTwo = [
    [
        { "fruit" : "kiwi" }
    ],
    "fruit" : "strawberry",
    "fruit" : "plum",
    { "fruit" : "banana" },
    "notfruit" : "something else"
]
output application/json
---
flatten(arrayOne ++ arrayTwo).fruit
```

*Output JSON:*

```
[
    "orange",
    "apple",
    "grape",
    "kiwi",
    "strawberry",
    "plum",
    "banana"
]
```

The next example uses `..*fruit` as a selector in the body expression instead of `.fruit` to return all nested `fruit` values in the flattened array. It also flattens three combined arrays instead of two.

*DataWeave Script:*

```
%dw 2.0
var arrayOne = [
    [
        "fruit" : "orange",
        "fruit" : "apple"
    ],
    [
        ..*fruit
    ]
]
```

```

        "fruit" : "grape",
        "notfruit" : "something else"
    ]
]
var arrayTwo = [
[
    { "fruit" : "kiwi" },
],
"fruit" : "strawberry",
"fruit" : "plum",
{ "fruit" : "banana" },
"notfruit" : "something else"
]
var arrayThree = [
    { parentOne :
        [
            { child :
                [
                    { grandchild :
                        {
                            "fruit" : "watermelon",
                            "notfruit" : "something else"
                        }
                    },
                    {
                        fruit : "cantaloupe",
                        "notfruit" : "something else"
                    }
                ]
            },
            {
                fruit : "honeydew",
                "notfruit" : "something else"
            }
        ]
    },
    { parentTwo:
        [
            [
                fruit : "cherry",
                "notfruit" : "something else"
            ]
        ]
    }
]
output application/json
---
flatten(arrayOne ++ arrayTwo ++ arrayThree)..*fruit

```

*Output JSON:*

```
[  
  "orange",  
  "apple",  
  "grape",  
  "kiwi",  
  "strawberry",  
  "plum",  
  "banana",  
  "watermelon",  
  "cantaloupe",  
  "honeydew",  
  "cherry"  
]
```

## Flatten Subarrays into the Parent Array

This example shows how the `flatten` function acts on a variety of data types, including numbers, DataWeave objects, subarrays, a string, a `null` value, and a key-value pair. Notice that elements of the subarrays become elements of the parent array, and the subarrays are no longer present.

The example uses this function:

- `flatten` to flatten the subarrays into their parent arrays.

*DataWeave Script:*

```
%dw 2.0  
var myArray = [  
    1,  
    [2,3],  
    { a : "b"},  
    "my string",  
    [ [4,5], { c : "d"}, 6 ],  
    null,  
    "e" : "f"  
]  
output application/json  
---  
flatten(myArray)
```

*Output JSON:*

```
[  
  1,  
  2,  
  3,  
  {  
    "a": "b"  
  },  
  "my string",  
  [  
    4,  
    5  
  ],  
  {  
    "c": "d"  
  },  
  6,  
  null,  
  {  
    "e": "f"  
  }  
]
```

## See Also

- [flatten](#)
- [Define a Function That Flattens Data in a List](#)
- [Selectors](#)

## Use Regular Expressions

Several DataWeave functions accept regular expressions as arguments, which you can use to return or check for matches. You can also construct regular expressions that incorporate DataWeave expressions that include functions and variables.

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

### Return Matches from a String

This example uses regular expressions in a number of DataWeave functions to return matches to an input variable "mycompany.com".

- `contains` returns `true` based on a regular expression that matches part of the input string.
- `find` returns an array of indices that specify the matching locations in the input string. This function treats the input string as a string array.

- `match` returns an array of substrings that match the regular expression.
- `matches` returns `true` because the regular expression matches the input string exactly.
- `replace` returns a URL that changes `.com` in the input string to `.net`, based on the regular expression `\..*\m`.
- `scan` returns a subarray of comma-separated substrings that the regular expression matches.
- `splitBy` splits an input string into an array of substrings based on the `.` in the input.

*DataWeave Script:*

```
%dw 2.0
var myString = "mycompany.com"
output application/json
---
{
  "contains" : myString contains(/c.m/),
  "find" : myString find(/[m|n].|m$/),
  "match" : myString match(/([a-z]*).[a-z]*/),
  "matches" : myString matches(/([a-z]*).[a-z]*/),
  "replaceWith" : myString replace /\..*\m/ with ".net",
  "scan" : myString scan(/([a-z]*).(com)/),
  "splitBy" : myString splitBy(/[\.\]/)}
}
```

*Output JSON:*

```
{  
  "contains": true,  
  "find": [  
    [  
      0  
    ],  
    [  
      4  
    ],  
    [  
      7  
    ],  
    [  
      12  
    ]  
  ],  
  "match": [  
    "mycompany.com",  
    "mycompany"  
  ],  
  "matches": true,  
  "replaceWith": "mycompany.net",  
  "scan": [  
    [  
      "mycompany.com",  
      "mycompany",  
      "com"  
    ]  
  ],  
  "splitBy": [  
    "mycompany",  
    "com"  
  ]  
}
```

For function documentation, see:

- [contains](#)
- [find](#)
- [match](#)
- [matches](#)
- [replace](#)
- [scan](#)
- [splitby](#)

## Use DataWeave Variables and Functions in a Regular Expression

This example constructs a regular expression by using the DataWeave concatenate function (`++`) to incorporate a DataWeave variable into a regular expression. The regular expression matches "somebiz". The example uses `replace` and `with` to replace "somebiz" with "abcd".

*DataWeave Script:*

```
%dw 2.0
var myCompany = { "name" : "biz" }
var myInputA = "somebiz-98765"
output application/json
---
{
    example: myInputA replace ((("^(s.*e)" ++ myCompany.name) as Regex) with ("abcd"))
}
```

*Output JSON:*

```
{
    "example": "abcd-98765"
}
```

## See Also

- [Regex \(dw::Core Type\)](#)
- [Core Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Output self-closing XML tags

For XML, DataWeave by default outputs every value within an opening and closing tag, even if the tag contains no value (for example, `<element2></element2>`). Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

To output empty tags as a self-closing XML element (for example, `<element2/>`), you can set the `inlineCloseOn="empty"` output directive in the DataWeave header.

*DataWeave Script:*

```
%dw 2.0
output application/xml inlineCloseOn="empty"
---
payload
```

*Input JSON Payload:*

```
{  
  "customer":{  
    "userName": "John Doe",  
    "password":{},  
    "status":"active",  
    "lastLogin":{}  
  }  
}
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>  
<customer>  
  <userName>John Doe</userName>  
  <password/>  
  <status>active</status>  
  <lastLogin/>  
</customer>
```

## Related Examples

- [Insert an Attribute into an XML Tag](#)
- [Remove Certain XML Attributes](#)
- [Include XML Namespaces](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Insert an Attribute into an XML Tag

This DataWeave example uses `@(key:value)` syntax (for example, `@(loc: "US")`) to inject attributes to XML tags. The example populates the `title` element with two attributes and each of the `author` elements with one attribute. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

It uses these functions:

- `mapObject` to iterate over each of the books in the input XML, defined in the variable `myInput`.
- `map` to iterate over the `author` elements in each book.

*DataWeave Script:*

```
%dw 2.0
var myInput = read('<bookstore>
<book>
    <title>Everyday Italian</title>
    <year>2005</year>
    <price>30</price>
    <author>Giada De Laurentiis</author>
</book>
<book>
    <title>Harry Potter</title>
    <year>2005</year>
    <price>29.99</price>
    <author>J. K. Rowling</author>
</book>
<book>
    <title>XQuery Kick Start</title>
    <year>2003</year>
    <price>49.99</price>
    <author>James McGovern</author>
    <author>Per Bothner</author>
    <author>Kurt Cagle</author>
    <author>James Linn</author>
    <author>Vaidyanathan Nagarajan</author>
</book>
<book>
    <title>Learning XML</title>
    <year>2003</year>
    <price>39.95</price>
    <author>Erik T. Ray</author>
</book>
</bookstore>', 'application/xml')
output application/xml
---
bookstore: myInput.bookstore mapObject (book) -> {
    book : {
        title @({lang: "en", year: book.year}): book.title,
        price: book.price,
        (book.*author map
            author @({loc: "US"}): $)
    }
}
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<bookstore>
  <book>
    <title lang="en" year="2005">Everyday Italian</title>
    <price>30</price>
    <author loc="US">Giada De Laurentiis</author>
  </book>
  <book>
    <title lang="en" year="2005">Harry Potter</title>
    <price>29.99</price>
    <author loc="US">J. K. Rowling</author>
  </book>
  <book>
    <title lang="en" year="2003">XQuery Kick Start</title>
    <price>49.99</price>
    <author loc="US">James McGovern</author>
    <author loc="US">Per Bothner</author>
    <author loc="US">Kurt Cagle</author>
    <author loc="US">James Linn</author>
    <author loc="US">Vaidyanathan Nagarajan</author>
  </book>
  <book>
    <title lang="en" year="2003">Learning XML</title>
    <price>39.95</price>
    <author loc="US">Erik T. Ray</author>
  </book>
</bookstore>
```

## Related Examples

- [Output Self-closing XML tags](#)
- [Remove Certain XML Attributes](#)
- [Include XML Namespaces](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Remove Certain XML Attributes

You might want to remove specific attributes from within an XML tag that are known to contain sensitive data. The DataWeave example defines a function in the DataWeave header that removes a specific XML attribute from the passed element and its children. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the

DataWeave table of contents.

This example uses:

- `mapObject` to go through each element.
- `@` to refer to the attributes in an XML element.
- `if` and `else` to conditionally only list XML attributes when they are present, and also to only recursively call the function when a child element is an object.
- `is` to check if each child is of type `Object`.
- `-` to remove an element from the list of XML attributes.
- `inlineCloseOn="empty"` on the output directive, so that the output XML displays self-closing tags when there are no values.

If the parent `users` object or child `user` objects within the input contain an attribute with the key `password`, `removeAttributes` removes that attribute from the object.

*DataWeave Script:*

```
%dw 2.0
output application/xml inlineCloseOn="empty"
var removeAttribute = (element,attrName) ->
  element mapObject (value, key) -> {
    // Pass the name of the key of the input object,
    // for example, whether the key is "users" or "user",
    // and use @() to remap the XML attributes.
    (key) @(
      (
        // If the element contains attributes (key.@?),
        // remove any attribute that has the provided key,
        // "password".
        if (key.@?)
          (key.@ - attrName)
        // Otherwise, do nothing.
        else {}
      )
    ) :
    // If the value of the input object contains one or
    // more objects, apply removeAttribute to remove any
    // attribute with the key "password" from those objects.
    if (value is Object)
      removeAttribute(value, attrName)
    // Otherwise, return the value of the input object.
    else value
  }
---
removeAttribute(payload, "password")
```

*Input XML Payload:*

```
<users>
  <user username="Julian" password="1234"/>
  <user username="Mariano" password="4321"/>
</users>
```

*Output XML:*

```
<?xml version='1.0' encoding='US-ASCII'?>
<users>
  <user username="Julian"/>
  <user username="Mariano"/>
</users>
```

## Related Examples

- [Output Self-closing XML tags](#)
- [Insert an Attribute into an XML Tag](#)
- [Include XML Namespaces](#)
- [Exclude Fields from the Output](#)
- [Output a Field When Present](#)
- [Conditionally Reduce a List Via a Function](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Pass XML Attributes

DataWeave enables you to pass attributes from within an XML tag input source to the target output XML. The DataWeave example shows how to pass the XML attributes in the `NameDetail` field of the input source payload to the `TargetName` field of the XML output. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

This example uses: - `.*` multivalue selector that returns an array of `NameDetail` elements. - `map` function to iterate over the array of `NameDetail` elements of the input. - The dynamic attribute expression `@dynamicAttributes` to create the attributes of the new output tag by selecting the attributes dynamically from the input.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
{
    Result: {
        (payload.Names.*NameDetail map ( nameDetail , indexOfNameDetail ) -> {
            TargetName @((nameDetail.@)): {
                item: nameDetail.item
            }
        })
    }
}
```

*Input XML Payload:*

```
<?xml version="1.0" encoding="UTF-8"?>
<Names>
    <NameDetail NameId="11111" NameType="Person" Name="Richarson"
GivenNames="John" PreferredLanguage="English" CompanyNumber="" Gender="Male" Title="" >
        <item>doo</item>
    </NameDetail>
    <NameDetail NameId="22222" NameType="Person" Name="Richarson"
GivenNames="Susan" PreferredLanguage="Spanish" CompanyNumber="" Gender="Female"
Title="" >
        <item>dah</item>
    </NameDetail>
    <NameDetail NameId="33333" NameType="Person" Name="Knox" GivenNames="Frances"
PreferredLanguage="English" CompanyNumber="" Gender="Male" Title="" >
        <item>dab</item>
    </NameDetail>
</Names>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<Result>
  <TargetName NameId="11111" NameType="Person" Name="Richarson" GivenNames="John"
  PreferredLanguage="English" CompanyNumber="" Gender="Male" Title="">
    <item>doo</item>
  </TargetName>
  <TargetName NameId="22222" NameType="Person" Name="Richarson" GivenNames="Susan"
  PreferredLanguage="Spanish" CompanyNumber="" Gender="Female" Title="">
    <item>dah</item>
  </TargetName>
  <TargetName NameId="33333" NameType="Person" Name="Knox" GivenNames="Frances"
  PreferredLanguage="English" CompanyNumber="" Gender="Male" Title="">
    <item>dab</item>
  </TargetName>
</Result>
```

## Related Examples

- [Output Self-closing XML tags](#)
- [Insert an Attribute into an XML Tag](#)
- [Include XML Namespaces](#)
- [Exclude Fields from the Output](#)
- [Output a Field When Present](#)
- [Conditionally Reduce a List Via a Function](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Include XML Namespaces

You can define different namespaces in the header and then reference them on each tag. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

Starting in the Mule 4.2.1 release, DataWeave also supports dynamically generated namespace keys and attributes.

The following example uses:

- `ns` for namespace definitions in the header
- `@` to define other attributes in an XML element

*DataWeave Script:*

```
%dw 2.0
output application/xml
ns orders http://www.acme.com/schemas/Orders
ns stores http://www.acme.com/schemas/Stores
---
root:
  orders#orders: {
    stores#shipNodeId: "SF01",
    stores#shipNodeId @(shipsVia:"LA01"): "NY03"
  }
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<root>
  <orders:orders xmlns:orders="http://www.acme.com/schemas/Orders">
    <stores:shipNodeId
      xmlns:stores="http://www.acme.com/schemas/Stores">SF01</stores:shipNodeId>
      <stores:shipNodeId xmlns:stores="http://www.acme.com/schemas/Stores"
        shipsVia="LA01">NY03</stores:shipNodeId>
    </orders:orders>
  </root>
```

## Dynamically Generated Namespace Values

*Support starting in Mule version 4.2.1*

Using input defined in the DataWeave variables, the DataWeave functions in the following example output values for namespace keys and attributes.

*DataWeave Script:*

```
%dw 2.0
fun dynKeyNs(ns0: Namespace, value: Any) = { ns0#myDynKey: value }
fun dynAttrNs(ns0: Namespace, value: Any) = { myChildTag @(ns0#myDynAttribute: true): value }
var namespace1 = {uri: "http://acme.com", prefix: "ns0"} as Namespace
var namespace2 = {uri: "http://emca.com", prefix: "ns0"} as Namespace
output application/xml
---
root:
{
    mytagA: dynKeyNs(namespace1, "myTest1"),
    myTagB: dynKeyNs(namespace2, "myTest2"),
    myTagC: dynAttrNs(namespace1, "myTest1"),
    myTagD: dynAttrNs(namespace2, "myTest2")
}
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<root>
    <mytagA>
        <ns0:myDynKey xmlns:ns0="http://acme.com">myTest1</ns0:myDynKey>
    </mytagA>
    <myTagB>
        <ns0:myDynKey xmlns:ns0="http://emca.com">myTest2</ns0:myDynKey>
    </myTagB>
    <myTagC>
        <myChildTag xmlns:ns0="http://acme.com"
ns0:myDynAttribute="true">myTest1</myChildTag>
    </myTagC>
    <myTagD>
        <myChildTag xmlns:ns0="http://emca.com"
ns0:myDynAttribute="true">myTest2</myChildTag>
    </myTagD>
</root>
```

## Use XML Namespace with Hyphen Character

The following DataWeave script shows you how to define an XML namespace as an object, which enables you to use special characters such as hyphen (-) in the namespace definition:

*DataWeave Script:*

```
%dw 2.0
output application/xml
var myNS = {uri: "http://www.abc.com", prefix: "ns-0"} as Namespace
---
{myNS#bla : "dw"}
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<ns-0:bla xmlns:ns-0="http://www.abc.com">dw</ns-0:bla>
```

## Related Examples

- [Output Self-closing XML tags](#)
- [Insert an Attribute into an XML Tag](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

# Remove Objects Containing Specified Key-Value Pairs

This DataWeave example removes all objects that contain a set of key-value pairs from an array of objects. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

This example uses the following functions:

- **filter** iterates over the objects within the input array to return an array with the objects that return **true** as the result of an expression. Each object in the input array contains a set of key-value pairs. The expression uses the **contains** function and the **not** operator.
- **contains** identifies the key-value pairs specified in the **dropThese** variable. This variable is an array of objects containing the key-value pairs to remove.

**not** negates the result of the **contains** function to make **filter** remove the key-value pairs identified in **dropThese** from the final output. (Without **not**, the result is an array of objects containing only the *undesired* key-value pairs.)

*DataWeave Script:*

```
%dw 2.0
var dropThese = [
  {"type" : "secondary", "space" : "rgb"},
  {"type" : "primary", "space" : "cmyk"}
]
output application/json
---
payload filter (not (dropThese contains {"type": $.type, "space": $.space}))
```

*Input JSON Payload:*

```
[{"sequence": "1", "color": "red", "type": "primary", "space": "rgb"}, {"sequence": "2", "color": "green", "type": "primary", "space": "rgb"}, {"sequence": "3", "color": "blue", "type": "primary", "space": "rgb"}, {"sequence": "4", "color": "yellow", "type": "secondary", "space": "rgb"}, {"sequence": "5", "color": "magenta", "type": "secondary", "space": "rgb"}, {"sequence": "6", "color": "cyan", "type": "secondary", "space": "rgb"}, {"sequence": "7", "color": "cyan", "type": "primary", "space": "cmyk"}, {"sequence": "8", "color": "magenta", "type": "primary", "space": "cmyk"}, {"sequence": "9", "color": "yellow", "type": "primary", "space": "cmyk"}, {"sequence": "10", "color": "red", "type": "secondary", "space": "cmyk"}, {"sequence": "11", "color": "green", "type": "secondary", "space": "cmyk"}, {"sequence": "12", "color": "blue", "type": "secondary", "space": "cmyk"}]
```

*Output JSON:*

```
[  
  {  
    "sequence": "1",  
    "color": "red",  
    "type": "primary",  
    "space": "rgb"  
  },  
  {  
    "sequence": "2",  
    "color": "green",  
    "type": "primary",  
    "space": "rgb"  
  },  
  {  
    "sequence": "3",  
    "color": "blue",  
    "type": "primary",  
    "space": "rgb"  
  },  
  {  
    "sequence": "10",  
    "color": "red",  
    "type": "secondary",  
    "space": "cmyk"  
  },  
  {  
    "sequence": "11",  
    "color": "green",  
    "type": "secondary",  
    "space": "cmyk"  
  },  
  {  
    "sequence": "12",  
    "color": "blue",  
    "type": "secondary",  
    "space": "cmyk"  
  }  
]
```

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Reference Multiple Inputs

This DataWeave example takes three different input JSON files, one in the payload, another in a

variable and the third in an attribute. All of them are parts of the same Mule Event. The payload contains an array of book objects, the variable has a set of currency exchange rates, and the attribute specifies a year to be used as a query. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses the following:

- `filter` to leave out the books that are older than the date specified in the incoming attribute.
- `map` to go through each book in the incoming array. Another `map` function then goes through the currencies in the variable to calculate the book's price in each one. A third `map` function lists out all authors of a book in case there are more than one.
- `@` to define an XML attribute in the output.

*DataWeave Script:*

```
%dw 2.0
output application/xml
---
books: {
  (payload filter ($.properties.year > attributes.publishedAfter) map (item) -> {
    book @(year: item.properties.year): {
      (vars.exchangeRate.USD map {
        price @(currency: $.currency): $.ratio * item.price
      }),
      title: item.properties.title,
      authors: { (item.properties.author map {
        author: $
      }) }
    }
  } )
}
```

*Input JSON Payload:*

```
[
  {
    "type": "book",
    "price": 30,
    "properties": {
      "title": "Everyday Italian",
      "author": [
        "Giada De Laurentiis"
      ],
      "year": 2005
    }
  },
  {
```

```
[{"type": "book",  
 "price": 29.99,  
 "properties": {  
     "title": "Harry Potter",  
     "author": [  
         "J K. Rowling"  
     ],  
     "year": 2005  
 },  
,  
{  
    "type": "book",  
    "price": 41.12,  
    "properties": {  
        "title": "Mule in Action",  
        "author": [  
            "David Dossot",  
            "John D'Emic"  
        ],  
        "year": 2009  
 },  
,  
{  
    "type": "book",  
    "price": 49.99,  
    "properties": {  
        "title": "XQuery Kick Start",  
        "author": [  
            "James McGovern",  
            "Per Bothner",  
            "Kurt Cagle",  
            "James Linn",  
            "Kurt Cagle",  
            "Vaidyanathan Nagarajan"  
        ],  
        "year": 2003  
 },  
,  
{  
    "type": "book",  
    "price": 39.95,  
    "properties": {  
        "title": "Learning XML",  
        "author": [  
            "Erik T. Ray"  
        ],  
        "year": 2003  
 },  
,  
]  
]
```

*Input Attributes:*

```
{  
    "publishedAfter": 2004  
}
```

*Input Variable exchangeRate:*

```
{  
    "USD": [  
        {"currency": "EUR", "ratio": 0.92},  
        {"currency": "ARS", "ratio": 8.76},  
        {"currency": "GBP", "ratio": 0.66}  
    ]  
}
```

*Output XML:*

```
<?xml version='1.0' encoding='US-ASCII'?>  
<books>  
    <book year="2005">  
        <price currency="EUR">27.60</price>  
        <price currency="ARS">262.80</price>  
        <price currency="GBP">19.80</price>  
        <title>Everyday Italian</title>  
        <authors>  
            <author>Giada De Laurentiis</author>  
        </authors>  
    </book>  
    <book year="2005">  
        <price currency="EUR">27.5908</price>  
        <price currency="ARS">262.7124</price>  
        <price currency="GBP">19.7934</price>  
        <title>Harry Potter</title>  
        <authors>  
            <author>J. K. Rowling</author>  
        </authors>  
    </book>  
    <book year="2009">  
        <price currency="EUR">37.8304</price>  
        <price currency="ARS">360.2112</price>  
        <price currency="GBP">27.1392</price>  
        <title>Mule in Action</title>  
        <authors>  
            <author>David Dossot</author>  
            <author>John D'Emic</author>  
        </authors>  
    </book>  
</books>
```

## Related Examples

- [Extract Data](#)
- [Merge Fields from Separate Objects](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Merge Fields from Separate Objects

The DataWeave examples merge fields from separate input arrays. The first (`firstInput`) is a DataWeave variable that lists price by book ID, and the second (`secondInput`) lists authors by book ID. Each book has a unique `bookId` key. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The DataWeave scripts produce the same output. Both scripts use two `map` functions with a filter, and one of them also creates an alias for the `bookId`:

- The first `map` function iterates over the elements of the `firstInput` array. As the function evaluates each element, a second `map` function uses a filter to identify any elements in the second array (`secondInput`) that match the `filter` criteria (`secondInput filter ($.bookId contains firstInputValue.bookId)`). The filter returns an element from `secondInput` that contains a `bookId` value that matches the `bookId` value in the `firstInput` element.
- The second `map` function evaluates that filtered element and then uses `secondInputValue.author` to select and populate the value of its `"author"` field in the object `{author : secondInputValue.author}`.
- `filter` limits the scope of the second `map` function to objects in the `secondInput` that share the same `bookId`.

*DataWeave Script:*

```
%dw 2.0
var firstInput = [
  { "bookId":"101",
    "title":"world history",
    "price":"19.99"
  },
  {
    "bookId":"202",
    "title":"the great outdoors",
    "price":"15.99"
  }
]
var secondInput = [
  {
    "bookId":"101",
    "author":"john doe"
  },
  {
    "bookId":"202",
    "author":"jane doe"
  }
]
output application/json
---
firstInput map (firstInputValue) ->
{
  theId : firstInputValue.bookId as Number,
  theTitle: firstInputValue.title,
  thePrice: firstInputValue.price as Number,
  (secondInput filter ($.bookId contains firstInputValue.bookId) map
(secondInputValue) -> {
    theAuthor : secondInputValue.author
  })
}
```

*Output JSON:*

```
[  
  {  
    "theId": 101,  
    "theTitle": "world history",  
    "thePrice": 19.99,  
    "theAuthor": "john doe"  
  },  
  {  
    "theId": 202,  
    "theTitle": "the great outdoors",  
    "thePrice": 15.99,  
    "theAuthor": "jane doe"  
  }  
]
```

As the next script shows, you can also write the same script using an `id` alias (created with `using (id = firstInputValue.bookId)`). The alias replaces the selector expression, `firstInputValue.bookId`, which is longer.

*DataWeave Script:*

```
%dw 2.0
var firstInput = [
  { "bookId":"101",
    "title":"world history",
    "price":"19.99"
  },
  {
    "bookId":"202",
    "title":"the great outdoors",
    "price":"15.99"
  }
]
var secondInput = [
  {
    "bookId":"101",
    "author":"john doe"
  },
  {
    "bookId":"202",
    "author":"jane doe"
  }
]
output application/json
---
firstInput map (firstInputValue) -> using (id = firstInputValue.bookId)
{
  theValue : id as Number,
  theTitle: firstInputValue.title,
  thePrice: firstInputValue.price as Number,
  (secondInput filter ($.bookId contains id)) map (secondInputValue) -> {
    theAuthor : secondInputValue.author
  }
}
```

*Output JSON:*

```
[  
  {  
    "theValue": 101,  
    "theTitle": "world history",  
    "thePrice": 19.99,  
    "theAuthor": "john doe"  
  },  
  {  
    "theValue": 202,  
    "theTitle": "the great outdoors",  
    "thePrice": 15.99,  
    "theAuthor": "jane doe"  
  }  
]
```

Both scripts produce the same output.

## Related Examples

- [Extract Data](#)
- [Reference Multiple Inputs](#)
- [Zip Arrays Together](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Parse Dates

These DataWeave examples define a function (`fun`) in the DataWeave header to normalize date separators (/, ., and -) within different date formats so that all of them use the same separator (-). Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The examples use these functions:

- `replace` so that all dates match a single pattern.
- `mapObject` to run through the the `date` elements. The script applies the normalizing function to each `date`.
- `as` (in the second DataWeave script) to change the data type of the values to a Date type with a specific date format.

## Example: Returns Dates as String Types

*DataWeave Script:*

```
%dw 2.0
output application/xml
fun normalize(date) = (date) replace "/" with "-" replace "." with "-"
---
dates: (payload.dates mapObject {
    normalized_as_string: normalize($)
})
```

*Input XML Payload:*

```
<dates>
  <date>26-JUL-16</date>
  <date>27/JUL/16</date>
  <date>28.JUL.16</date>
</dates>
```

*Output XML:*

```
<?xml version='1.0' encoding='US-ASCII'?>
<dates>
  <normalized_as_string>26-JUL-16</normalized_as_string>
  <normalized_as_string>27-JUL-16</normalized_as_string>
  <normalized_as_string>28-JUL-16</normalized_as_string>
</dates>
```

## Example: Returns Dates as Date Types

*DataWeave Script:*

```
%dw 2.0
output application/xml
fun normalize(date) = (date) replace "/" with "-" replace "." with "-"
---
// Outputs date values as Date types in the specified format
dates: (payload.dates mapObject {
    normalized_as_date: (normalize($)) as Date {format: "d-MMM-yy"}
})
```

*Input XML Payload:*

```
<dates>
  <date>26-JUL-16</date>
  <date>27/JUL/16</date>
  <date>28.JUL.16</date>
</dates>
```

*Output XML:*

```
<?xml version='1.0' encoding='UTF-8'?>
<dates>
  <normalized_as_date>2016-07-26</normalized_as_date>
  <normalized_as_date>2016-07-27</normalized_as_date>
  <normalized_as_date>2016-07-28</normalized_as_date>
</dates>
```

## Related Examples

- [Add and Subtract Dates](#)
- [Reference Multiple Inputs](#)

## See Also

- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Add and Subtracting Dates

This DataWeave example shows multiple addition and subtraction operations that deal with date and time types. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The examples use:

- `as` function to coerce a String to Period type
- `P<date>T<time>` for the Period data type, which provides designators for years, months, days, hours, minutes, and seconds

For example, `|P2Y9M1D|` refers to a period of two years, nine months, and one day, and `|PT5H4M3S|` indicates a time period of five hours, four minutes, and three seconds.

*DataWeave Script:*

```
%dw 2.0
output application/json
var numberOfDay = 3
---
{
    oneDayBefore: |2019-10-01T23:57:59Z| - |P1D|,
    threeDaysBefore: |2019-10-01T23:57:59Z| - ("P$(numberOfDay)D" as Period),
    a: |2019-10-01| - |P1Y|,
    b: |P1Y| - |2019-10-01|,
    c: |2019-10-01T23:57:59Z| - |P1Y|,
    d: |2019-10-01T23:57:59Z| + |P1Y|,
    e: |2019-10-01T23:57:59| - |P1Y|,
    f: |PT9M| - |23:59:56|,
    g: |23:59:56| + |PT9M|,
    h: |23:59:56-03:00| - |PT9M|,
    u: |23:59:56-03:00| - |22:59:56-03:00|,
    j: |23:59:56-03:00| - |22:59:56-00:00|,
    k: |2019-10-01T23:57:59| - |P2Y9M1D| - |PT57M59S| + |PT2H|,
    l: |23:59:56| - |22:59:56|,
    o: |2019-10-01| - |2018-09-23|,
    p: |2019-10-01T23:57:59Z| - |2018-10-01T23:57:59Z|,
    q: |2019-10-01T23:57:59| - |2018-10-01T23:57:59|
}
```

*Output JSON:*

```
{
    "oneDayBefore": "2019-09-30T23:57:59Z",
    "threeDaysBefore": "2019-09-28T23:57:59Z",
    "a": "2018-10-01",
    "b": "2018-10-01",
    "c": "2018-10-01T23:57:59Z",
    "d": "2020-10-01T23:57:59Z",
    "e": "2018-10-01T23:57:59",
    "f": "23:50:56",
    "g": "00:08:56",
    "h": "23:50:56-03:00",
    "u": "PT1H",
    "j": "PT4H",
    "k": "2017-01-01T01:00:00",
    "l": "PT1H",
    "o": "PT8952H",
    "p": "PT8760H",
    "q": "PT8760H"
}
```

Note that the behavior of `o: |2019-10-01| - |2018-09-23|` changed in Mule 4.3.0. DataWeave 2.3.0, which is supported by Mule 4.3.0, returns `"o": "PT8952H"`. DataWeave 2.2.0, which is supported by

Mule 4.2.0, returns "o": "P1Y8D". To restore the 2.2.0 behavior, you can set the [-M-Dcom.mulesoft.dw.date\\_minus\\_back\\_compatibility=true](#). However, the setting does not affect the Preview window in Studio. For guidance with this setting in Studio and on-prem deployments, see [System Properties](#). For CloudHub deployments, see [Manage Properties for Applications on CloudHub](#).

## See Also

- [Period](#)
- [Parse Dates](#)
- [DataWeave Types](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

## Change a Time Zone

The following example uses the `>>` operator and a [time zone ID](#) to change the time zone. It also uses the formatting characters `uuuu-MM-dd'T'HH:mm:ss.SSS` to modify the format. The `uuuu` characters represent the year.

*DataWeave Script:*

```
%dw 2.0
output application/json
fun format(d: DateTime) = d as String {format: "yyyy-MM-dd'T'HH:mm:ss.SSS"}
---
{
    CreatedDateTime: format(|2019-02-13T13:23:00.120Z| >> "CET"),
}
```

*Output JSON:*

```
{
    "CreatedDateTime": "2019-02-13T14:23:00.120"
```

## Time Zone IDs

In addition to accepting time zone values such as `-07:00`, DataWeave accepts Java 8 [TimeZone](#) ID values, which are available through a Java method call to `ZoneId.getAvailableZoneIds()`.

To see all available values in the following table, scroll right.



Africa:	America (continued):	America (continued):	Asia (continued):	Australia:	E (continued):	J:	T:
• Africa/ Abidjan	• America/Camb ridge_Bay	• America/North_Dakota/Beulah	• Asia/Damascus	• Australia/ACT	• Europe/Amsterdam	• Jamaica	• Turkey
• Africa/ Accra	• America/Camp_o_Grande	• America/North_Dakota/Center	• Asia/Dhaka	• Australia/Adelaide	• Europe/Andorra	• Japan	U: • UCT
• Africa/ Addis_Ababa	• America/Cancun	• America/North_Dakota/New_Salem	• Asia/Dili	• Australia/Brisbane	• Europe/Astrakan	K: • Kwajalein	• US/Alaska
• Africa/ Algiers	• America/Caracas	• America/North_Dakota/Ojina ga	• Asia/Dubai	• Australia/Broken_Hill	• Europe/Athens	L: • Libya	• US/Aleutian
• Africa/ Asmara	• America/Catamarca	• America/Panama	• Asia/Dushanbe	• Australia/Canberra	• Europe/Belfast	M: • MET	• US/Central
• Africa/ Bamako	• America/Cayne	• America/Pangnirtung	• Asia/Famagusta	• Australia/Currie	• Europe/Belgrade	• MST7MDT	• US/East-Indian
• Africa/ Bangui	• America/Cayman	• America/Parariba	• Asia/Gaza	• Australia/Darwin	• Europe/Berlin	• Mexico/BajaNorte	• US/Eastern
• Africa/ Banjul	• America/Chicago	• America/Paramaribo	• Asia/Harbin	• Australia/Euclid	• Europe/Bratislava	• Mexico/BajaSur	• US/Hawaii
• Africa/ Bissau	• America/Chihua	• America/Phoenix	• Asia/Hebron	• Australia/Hobart	• Europe/Brussels	N: • NZ	• US/Indiana-Starke
• Africa/ Blantyre	• America/Phoe	• America/Phoe	• Asia/Ho_Chi_Minh	• Australia/LHI	• Europe/Bucharest	• NZ-CHAT	US/Mo
• Africa/ Brazza ville	• America/Coral	• America/Port-au-Harbo	• Asia/Hong_Kong	• Australia/Lindeman	• Europe/Budapest	P: • Navajo	untain • US/Pacific
							• PRC
							PST8PD
						T	US/Pacific-New

## Related Examples

- [Format Dates and Times](#)
- [Parse Dates](#)
- [Add and Subtracting Dates](#)

## See Also

- [Africa/](#)
- [Casablanca](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Format Dates and Times

You can use DataWeave to change the format of date and time input. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

You can combine formatting characters, such as `MM` and `dd`, to write supported date and time formats. The example uses the `as` operator to write the dates and times as a string.

```
%dw 2.0
output application/json
---
{
    formattedDate: |2020-10-01T23:57:59| as String {format: "uuuu-MM-dd"},
    formattedTime: |2020-10-01T23:57:59| as String {format: "KK:mm:ss a"},
    formattedDateTime: |2020-10-01T23:57:59| as String {format: "KK:mm:ss a, MMMM dd,
    uuuu"}
}
```

**Output:** `{"formattedDate": "2020-10-01", "formattedTime": "11:57:59 PM", "formattedDateTime": "11:57:59 PM, October 01, 2020"}`

## Create a Custom Date Format as a DataWeave Type

For multiple, similar conversions in your script, you can define a custom type as a directive in the

header and set each date to that type. Names of DataWeave type are case-sensitive.

Harare • • • • Europe Gambi  
DataWeave ScriptAmeric Americ Asia/Kr Brazil/ /Kirov er

```
%dw 2.0
output application/json
type Mydate = String { format: "uuuu/MM/dd" }
---
{
    formattedDate1: |2019-10-01T23:57:59| as Mydate,
    formattedDate2: |2020-07-06T08:53:15| as Mydate
}
```

Kampa  
Output

a/El Sa lvador	a/Santa rem	uching	.	Brazil/	Europe /Londo	Pacific/ Honolu
-------------------	----------------	--------	---	---------	------------------	--------------------

```
{  
    "formattedDate1": "2019/10/01",  
    "formattedDate2": "2020/07/06"  
}
```

# Use Date and Time Formatting Characters

The following example formats the output of the now DataWeave function to show supported letters:

```
%dw 2.0
var myDateTime = ("2020-11-10T13:44:12.283-08:00" as DateTime)
output application/json
---
{
    "dateTime" : myDateTime,
    "era-G" : myDateTime as String { format: "G"}, 
    "year-u" : myDateTime as String {format: "u"}, 
    "year-uu" : myDateTime as String {format: "uu"}, 
    //y is for use with the era (BCE or CE). Generally, use u, instead.
    "year-y" : myDateTime as String { format: "y"}, 
    "year-yy" : myDateTime as String { format: "yy"}, 
    "dayOfYear-D" : myDateTime as String { format: "D"}, 
    "monthOfYear-MMMM": myDateTime as String { format: "MMMM"}, 
    "monthOfYear-MMM": myDateTime as String { format: "MMM"}, 
    "monthOfYear-MM": myDateTime as String { format: "MM"}, 
    "monthOfYear-M": myDateTime as String { format: "M"}, 
    "monthOfYear-LL": myDateTime as String { format: "LL"}, 
    "monthOfYear-L": myDateTime as String { format: "L"}, 
    "dayOfMonth-d" : myDateTime as String {format: "d"}, 
    "quarterOfYear-qqq" : myDateTime as String {format: "qqq"}.
```

	Africa	America	Asia/Oceania	Canada	Europe	Pacific
	Africa	America	Asia/Oceania	Canada	Europe	Niue

```

"quarterOfYear-qq" : myDateTime as String {format: "qq"},  

"quarterOfYear-q" : myDateTime as String {format: "q"},  

"quarterOfYear-QQQQ" : myDateTime as String {format: "QQQQ"},  

"quarterOfYear-QQQ" : myDateTime as String {format: "QQQ"},  

"quarterOfYear-QQ" : myDateTime as String {format: "QQ"},  

"quarterOfYear-Q" : myDateTime as String {format: "Q"},  

// Understand "Y" and "YY" thoroughly before using it.  

"weekBasedYear-YY" : myDateTime as String {format: "YY"},  

"weekBasedYear-Y" : myDateTime as String {format: "Y"},  

"weekInYear-w" : myDateTime as String {format: "w"},  

"weekInMonth-W" : myDateTime as String {format: "W"},  

"dayOfWeekAbbreviatedName-E" : myDateTime as String {format: "E"},  

"dayOfWeekFullName-EEEE" : myDateTime as String {format: "EEEE"},  

"localizedDayOfWeek-eeee" : myDateTime as String {format: "eeee"},  

"localizedDayOfWeek-eee" : myDateTime as String {format: "eee"},  

"localizedDayOfWeek-ee" : myDateTime as String {format: "ee"},  

"localizedDayOfWeek-e" : myDateTime as String {format: "e"},  

"localizedDayOfWeek-cccc" : myDateTime as String {format: "cccc"},  

"localizedDayOfWeek-ccc" : myDateTime as String {format: "ccc"},  

"localizedDayOfWeek-c" : myDateTime as String {format: "c"},  

"weekOfMonth-F" : myDateTime as String {format: "F"},  

"amORpm-a" : myDateTime as String {format: "a"},  

// "h" outputs 12 o'clock as 12. Other hours match "K" output.  

"hourOfDay1to12-h" : myDateTime as String {format: "h"},  

// "K" outputs 12 o'clock as 0. Other hours match "h" output.  

"hourOfDay0to11-K" : myDateTime as String {format: "K"},  

"clockHourOfAmPm-k" : myDateTime as String {format: "k"},  

"hourOfDay0to23-H" : myDateTime as String {format: "H"},  

"minuteOfHour-m" : myDateTime as String {format: "m"},  

"secondOfMinute-s" : myDateTime as String {format: "s"},  

"fractionOfSecond-S" : myDateTime as String {format: "S"},  

"millisecondOfDay-A" : myDateTime as String {format: "A"},  

"nanosecondCountOfSecond-n" : myDateTime as String {format: "n"},  

"nanosecondCountOfDay-N" : myDateTime as String {format: "N"},  

"timeZoneID-VV" : myDateTime as String {format: "VV"},  

"timeZoneName-zz" : myDateTime as String {format: "zz"},  

"localizedZoneOffset-zzz" : myDateTime as String {format: "zzz"},  

"localizedZoneOffset-0" : myDateTime as String {format: "0"},  

"timeZoneOffsetZforZero-XXX" : myDateTime as String {format: "XXX"},  

"timeZoneOffsetZforZero-XX" : myDateTime as String {format: "XX"},  

"timeZoneOffsetZforZero-X" : myDateTime as String {format: "X"},  

"timeZoneOffset-xxx" : myDateTime as String {format: "xxx"},  

"timeZoneOffset-xx" : myDateTime as String {format: "xx"},  

"timeZoneOffset-x" : myDateTime as String {format: "x"},  

"timeZoneOffset-Z" : myDateTime as String {format: "Z"}  

}

```

	Africa/Porto-Novo	America/Antigua	Asia/Seoul	Pacific/Tahiti
Notice the use of the syntax to format the marker time.				
Africa/ Output	America -na/Mar Africa/Engo	America -la Engo	nd oul T+2	Etc/GM /Stockh Europe
				Pacific/ Tarawa

```
{
  "dateTime": "2020-11-10T13:44:12.283-08:00",
  "era-G": "AD",
  "year-u": "2020",
  "year-uu": "20",
  "year-y": "2020",
  "year-yy": "20",
  "dayOfYear-D": "315",
  "monthOfYear-MMMM": "November",
  "monthOfYear-MMM": "Nov",
  "monthOfYear-MM": "11",
  "monthOfYear-M": "11",
  "monthOfYear-LL": "11",
  "monthOfYear-L": "11",
  "dayOfMonth-d": "10",
  "quarterOfYear-qqq": "4",
  "quarterOfYear-qq": "04",
  "quarterOfYear-q": "4",
  "quarterOfYear-QQQ": "4th quarter",
  "quarterOfYear-QQQ": "Q4",
  "quarterOfYear-QQ": "04",
  "quarterOfYear-Q": "4",
  "weekBasedYear-YY": "20",
  "weekBasedYear-Y": "2020",
  "weekInYear-w": "46",
  "weekInMonth-W": "2",
  "dayOfWeekAbbreviatedName-E": "Tue",
  "dayOfWeekFullName-EEEE": "Tuesday",
  "localizedDayOfWeek-eeee": "Tuesday",
  "localizedDayOfWeek-eee": "Tue",
  "localizedDayOfWeek-ee": "03",
  "localizedDayOfWeek-e": "3",
  "localizedDayOfWeek-cccc": "Tuesday",
  "localizedDayOfWeek-ccc": "Tue",
  "localizedDayOfWeek-c": "3",
  "weekOfMonth-F": "3",
  "amORpm-a": "PM",
  "hourOfDay1to12-h": "1",
  "hourOfDay0to11-K": "1",
  "clockHourOfAmPm-k": "13",
  "hourOfDay0to23-H": "13",
  "minuteOfHour-m": "44",
  "secondOfMinute-s": "12",
  "fractionOfSecond-S": "2",
  "millisecondOfDay-A": "49452283",
  "nanosecondCountOfSecond-n": "283000000",
  "nanosecondCountOfDay-N": "49452283000000",
  "timeZoneID-VV": "-08:00",
  "timeZoneName-zz": "-08:00",
  "localizedZoneOffset-zzz": "-08:00",
  "localizedZoneOffset-O": "GMT-8",
}
```

America/Arge	America	Antarctica	Asia/Almaty	Europe/Zaporozhye	SystemV/EST5
a/Arge	Americ	ica/Ma wson	aanbaa tar	Etc/GM T-2	443

```

    "timeZoneOffsetZforZero-XXX": "-08:00",
    "timeZoneOffsetZforZero-XX": "-0800",
    "timeZoneOffsetZforZero-X": "-08",
    "timeZoneOffset-xxx": "-08:00",
    "timeZoneOffset-xx": "-0800",
    "timeZoneOffset-x": "-08",
    "timeZoneOffset-Z": "-0800"
}

```

## Related Examples

- [Parse Dates](#)
- [Add and Subtracting Dates](#)
- [Change a Time Zone](#)

## See Also

- [Date and Time \(dw::Core Types\)](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Work with Multipart Data

The following example iterates over a multipart payload and extracts data from each part.

The example uses the following functions:

- `read` (in the script's header) reads the multipart boundary to `34b21`.
- `mapObject` iterates over the parts in the multipart data returned by `read` and `name` key of each part within a JSON object.

```
%dw 2.0
output application/json
var multi = '--34b21
Content-Disposition: form-data; name="text"
Content-Type: text/plain

Book
--34b21
Content-Disposition: form-data; name="file1"; filename="a.json"
Content-Type: application/json

{
  "title": "Java 8 in Action",
  "author": "Mario Fusco",
  "year": 2014
}
--34b21
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html

<!DOCTYPE html>
<title>
  Available for download!
</title>
--34b21--
var parsed = read(multi, "multipart/form-data", {boundary:'34b21'})
---
parsed.parts mapObject ((value, key, index) -> {(index): key})
```

•  
*Output:* Ame

Americ

10

a

```
{  
  "0": "text",  
  "1": "file1",  
  "2": "file2"  
}
```

For additional examples, see the [Multipart Format \(Form Data\)](#) examples and functions referenced in the [dw::module::Multipart](#) API documentation.

**See Also**

- a/Bahia Americ
  - **Selectors** Americ
  - \_Bande Metla
  - **DataWeave Tas** Asia/Br
  - **Cookbook** unei
  - Americ Asia/Ca
  - a/Barb Americ lcutta
  - ados a/Mexi
  - co City

# Conditionally Reduce a List Via a Function

This DataWeave example flattens the input to something simpler. For each element in the input, it conditionally includes a `contentTypes` field only in case it has any values. It then rearranges this content into a more readable shape. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The examples use these functions:

- `map` to go through every element in the input array.
- `reduce` to aggregate the multiple elements in the `tags` array into one field.
- `splitBy` to parse the values in the "tags" array, so as to only select the first part of each string (through `[0]`) and discard the second.
- `++` to concatenate these elements into one single string, with commas as separators.
- `if` include the last field only if the `contentTypes` array is bigger than 0.

*DataWeave Script:*

```
%dw 2.0
output application/json
fun reduceMapFor(data) = data reduce((## splitBy ":")[0] ++ "," ++ ($ splitBy ":")[0])
---
payload.results map
{
    email: $.profile.email,
    name: $.profile.firstName,
    tags: reduceMapFor($.data.interests.tags[0]),
    (contenttypes: reduceMapFor($.data.interests.contenttypes[0])) if
(sizeOf($.data.interests.contenttypes[0]) > 0)
}
```

*Input JSON Payload:*

```
{
  "results": [
    {
      "profile": {
        "firstName": "john",
        "lastName": "doe",
        "email": "johndoe@demo.com"
      },
      "data": {
        "interests": [
          {
            "language": "English",
            "tags": [
              "digital-strategy:Digital Strategy",
              "cloud-computing:Cloud Computing"
            ]
          }
        ]
      }
    }
  ]
}
```

```

        "innovation:Innovation"
    ],
    "contenttypes": []
}
]
}
},
{
    "profile": {
        "firstName": "jane",
        "lastName": "doe",
        "email": "janedoe@demo.com"
    },
    "data": {
        "interests": [
            {
                "language": "English",
                "tags": [
                    "tax-reform:Tax Reform",
                    "retail-health:Retail Health"
                ],
                "contenttypes": [
                    "News",
                    "Analysis",
                    "Case studies",
                    "Press releases"
                ]
            }
        ]
    }
},
{
    "objectsCount": 2,
    "totalCount": 2,
    "statusCode": 200,
    "errorCode": 0,
    "statusReason": "OK"
}

```

- Americ  
a/Nipig  
on

- Americ  
a/Nom  
e

- Americ  
a/Noro  
nha

*Output JSON:*

```
[  
  {  
    "email": "johndoe@demo.com",  
    "name": "john",  
    "tags": "digital-strategy,innovation"  
  },  
  {  
    "email": "janedoe@demo.com",  
    "name": "jane",  
    "tags": "tax-reform,retail-health",  
    "contenttypes": "News,Analysis,Case studies,Press releases"  
  }  
]
```

## Related Examples

- [Exclude Fields from the Output](#)
- [Output a Field When Present](#)
- [Use Constant Directives](#)
- [Define a Function that Flattens Data in a List](#)
- [Pass Functions as Arguments](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Pass Functions as Arguments

This DataWeave example defines a base function in the header that receives two arguments: a function and an element to apply it on. The base function applies the received function onto the keys of the received element and also to every one of its child elements recursively. In this case, the function sent sets all keys to lower case. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The example uses the following:

- A custom function that is applied recursively, ie: it calls itself for each child in the element it receives originally.
- **mapObject** to sort through each child of the element.
- **lower** from the Strings library to make every Key in the input lower case.
- **is** to check if the passed element is of type **Object** (and therefore has children).

- **if** and **else** to only have the function call itself when a condition is met, and to otherwise end the recursive loop for that branch.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Strings
output application/xml
fun applyToKeys(element, func) =
    if (element is Object)
        element mapObject (value, key) -> {
            (func(key)) : applyToKeys( value, func)
        }
    else element
---
applyToKeys(payload, (key) -> lower(key))
```

*Input XML Payload:*

```
<CATALOG>
<CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
</CD>
<CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
</CD>
</CATALOG>
```

*Output XML:*

```
<?xml version='1.0' encoding='US-ASCII'?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
  <cd>
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <country>UK</country>
    <company>CBS Records</company>
    <price>9.90</price>
    <year>1988</year>
  </cd>
</catalog>
```

## Related Examples

- [Define a Custom Addition Function](#)
- [Dynamically Map Based On a Definition](#)
- [Define a Function that Flattens Data in a List](#)
- [Conditionally Reduce a List Via a Function](#)

## See Also

- [Selectors](#)
- [DataWeave Cookbook](#)

## Change a Script's MIME Type Output (Mule)

The DataWeave examples show how to differentiate the output MIME type from the MIME type in which the output data is formatted. Such differentiation enables a Mule application to output a MIME type that clients can use to determine how to render the application's output data. An example is HTTP, which uses the MIME type to generate its [Content-Type](#) header.

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The first example uses the "problem details" standard for APIs. The standard requires the [application/problem+json](#) content type in the response header and a JSON-formatted response body

that contains specific keys.

The example is a Mule flow that handles HTTP requests but always fails. The failure triggers the error handler, which sets the status code to 400 and uses a DataWeave script to produce a response that conforms to the standard.

The DataWeave output directive `output application/problem+json with json` instructs the script to produce a response with the `application/problem+json` MIME type and a JSON-formatted body.

A client that receives the 400 error uses the `Content-Type: application/problem+json` in the response to determine that it can handle the response data in the standard way, for example, using the `title` key it expects. By contrast, the MIME type `Content-Type: application/json`, without `problem+`, fails to indicate whether the client can handle the error in that way.

*Mule Flow:*

```
<flow name="problemDetailFlow">
    <http:listener config-ref="HTTP_Listener_config" path="/test">
        <http:error-response statusCode="#[vars.statusCode]" >
            <http:body ><![CDATA[#[payload]]]></http:body>
        </http:error-response>
    </http:listener>
    <raise-error doc:name="Raise error" type="CUSTOM:FORCED" description="This is
a manual error to force the execution of the error handler."/>
    <error-handler >
        <on-error-propagate>
            <set-variable value="#[400]" variableName="statusCode"/>
            <ee:transform>
                <ee:message>
                    <ee:set-payload><![CDATA[%dw 2.0
output application/problem+json with json
--->
{
    "type": "https://example.org/out-of-stock",
    title: "Out of Stock",
    status: vars.statusCode,
    detail: "Item B00027Y5QG is no longer available",
    product: "B00027Y5QG"
}]]></ee:set-payload>
                </ee:message>
            </ee:transform>
        </on-error-propagate>
    </error-handler>
</flow>
```

*Response:*

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json; charset=UTF-8
Content-Length: 173
Date: Thu, 02 Apr 2020 19:48:23 GMT
Connection: close

{
  "type": "https://example.org/out-of-stock",
  "title": "Out of Stock",
  "status": 400,
  "detail": "Item B00027Y5QG is no longer available",
  "product": "B00027Y5QG"
}
```

In the next example, the Mule flow returns [text/markdown](#) content generated by DataWeave as simple text. A browser that supports Markdown might render the input as bulleted content according to the rules of Markdown instead of rendering it in plain text with asterisks.

*Mule Flow:*

```
<flow name="listFlow">
  <http:listener config-ref="HTTP_Listener_config" path="/some.md"/>
  <ee:transform>
    <ee:message>
      <ee:set-payload ><![CDATA[%dw 2.0
output text/markdown with text
---
/* idea 1
* some other idea"]]></ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
```

*Response:*

```
HTTP/1.1 200
Content-Type: text/markdown; charset=UTF-8
Content-Length: 26
Date: Thu, 02 Apr 2020 21:28:36 GMT

* idea 1
* some other idea
```

## See Also

- [DataWeave Formats](#)

## Look Up Data in an Excel (XLSX) File (Mule)

This DataWeave example uses the `filter` function to return only the rows in an Excel (XLSX) input file that contain a specified value. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.



By default, files over 1.5 MB are stored on disk. Smaller files are stored in memory.

The following DataWeave script reads an XLSX file and returns filtered data from it. It assumes that a spreadsheet named `ourBugs.xlsx` contains data on bugs assigned to all the members of a team, including an assignee named `Fred M`.

*DataWeave Script:*

```
%dw 2.0
var myInput = readUrl("classpath://ourBugs.xlsx", "application/xlsx")
output application/json
---
myInput."Data" filter ((entry, index) -> entry."Assignee" == "Fred M")
```

- The script passes `classpath:ourBugs.xlsx` to the `readUrl` function to read the file from a Studio project directory (`src/main/resources`). It stores the results in the variable `myInput`.
- The script selects a sheet named `Data` from the XLSX file, then filters out all records except the ones where the `Assignee` column contains the value `Fred M`. It returns the results in an array of JSON objects, for example:

```
[
  {
    "Issue Key": "BUG-11708",
    "Issue Type": "Bug",
    "Summary": "Some Description of the Bug",
    "Assignee": "Fred M",
    "Reporter": "Natalie C",
    "Priority": "To be reviewed",
    "Status": "Closed",
    "Resolution": "Done",
    "Created": "2019-04-29T03:57:00",
    "Updated": "2019-05-06T10:40:00",
    "Due Date": ""
  },
  {
    "Issue Key": "BUG-4903",
    "Issue Type": "Story",
    "Summary": "Some Description of the Bug",
    "Assignee": "Fred M",
    "Reporter": "Fred M",
    "Priority": "To be reviewed",
    "Status": "In Progress",
    "Resolution": "",
    "Created": "2019-05-07T11:22:00",
    "Updated": "2019-05-08T10:16:00",
    "Due Date": ""
  },
  {
    "Issue Key": "BUG-4840",
    "Issue Type": "Story",
    "Summary": "Some Description of the Bug",
    "Assignee": "Fred M",
    "Reporter": "Pablo C",
    "Priority": "To be reviewed",
    "Status": "In Validation",
    "Resolution": "",
    "Created": "2019-04-30T07:11:00",
    "Updated": "2019-05-08T10:16:00",
    "Due Date": ""
  }
]
```

Using the same DataWeave script as in the previous example, the next example writes the results of the `filter` expression to a file, `fredBugs.json`. The example is a configuration XML from a Mule project in Studio.

## XML Configuration in Studio:

```
<file:config name="File_Read_Config" doc:name="File Read Config" />
<file:config name="File_Write_Config" doc:name="File Write Config" />
<flow name="xlsx-lookup" >
    <scheduler doc:name="Scheduler" >
        <scheduling-strategy >
            <fixed-frequency frequency="1" timeUnit="MINUTES"/>
        </scheduling-strategy>
    </scheduler>
    <ee:transform doc:name="Transform Message" >
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
var myInput = readUrl("classpath://ourBugs.xlsx", "application/xlsx")
output application/json
---
myInput."Data" filter ((entry, index) -> entry."Assignee" == "Fred M")]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <file:write doc:name="Write JSON"
        path="/path/to/fredBugs.json"
        config-ref="File_Write_Config">
    </file:write>
</flow>
```

- The Scheduler (**scheduler**) triggers the flow to execute the next component, Transform Message.
- The Transform Message component (**ee:transform**) provides a DataWeave script to return all records from the **"Data"** sheet for which the **Assignee** column contains the value **Fred M**, and it transforms the binary XLSX input to JSON output.
- The Write operation (**file:write**) from the File connector receives the JSON payload from **ee:transform** and writes it to a file called **fredBugs.json**.

## See Also

- [filter Function](#)
- [Supported Data Formats](#)

## Look Up Data in CSV File (Mule)

This DataWeave example uses **filter** with **map** to look up data in a CSV file and return a country code for each calling code found within an input array of phone numbers. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The following JSON payload provides an array of phone numbers to the DataWeave script.

```
[  
  "54-112724555",  
  "1-6298765432"  
]
```

The value of the Mule variable `country_codes` is a CSV file that maps calling codes to country codes.

```
CALLING_CODE,COUNTRY_CODE  
1,US  
7,RU  
54,AR  
20,EG  
32,BE  
33,FR  
505,NI  
506,CR  
1876,JM  
1905,CA  
1939,PR  
262262,RE  
262269,YT  
..,  
..,
```

The following DataWeave script reads each phone number in the payload and uses the `filter` function to search for the calling code within the Mule variable `country_code` that matches the calling code in the phone number. Then the `map` function gets the country code that corresponds to the calling code and returns the results to an array of JSON objects.

*DataWeave Script:*

```
%dw 2.0  
output application/json  
  
fun lookupCountryCode(phoneNumber: String): String = do {  
    var callingCode = (phoneNumber splitBy("-"))[0]  
    ---  
    (vars.country_code filter ((item, index) -> item.CALLING_CODE == callingCode) map  
    ((item, index) -> item.COUNTRY_CODE))[0]  
}  
---  
payload map ((item, index) -> { phone: item, countryCode: lookupCountryCode(item)})
```

*Sample Output (JSON):*

```
[  
  {  
    "phone": "54-112724555",  
    "countryCode": "AR"  
  },  
  {  
    "phone": "1-6298765432",  
    "countryCode": "US"  
  }  
]
```

## See Also

- [Supported Data Formats](#)

## Decode and Encode Base64 (Mule)

The following DataWeave examples show how to convert a file stream into Base64 and to convert a Base64 string into a file stream. They use a PDF file as the input and output. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

### Encode a PDF File to Base64

This example performs Base64 encoding on a PDF file. It simulates the transformation in the Read operation from the [XML configuration](#) by converting a Base64 file stream ([application/octet-stream](#)) to Binary and outputting the Binary in the [text/plain](#) format.

The example uses the following functions:

- `toBase64` from the `dw::core::Binaries` module to transform a binary value to a Base64 string.
- `readUrl` from the `Core (dw::Core)` module to input a PDF file.

The example assumes that the PDF file is in the `src/main/resources` directory of a Mule project in Anypoint Studio. It provides the name of the PDF file and specifies the MIME type as `"application/octet-stream"`. Using `"binary"` as the second `readUrl` argument is also valid for this example.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Binaries
var myPDF = readUrl("classpath://pdf-test.pdf", "application/octet-stream")
output text/plain
---
toBase64(myPDF as Binary)
```

*Output (partial results):*

```
JVBERi0xLjYNJeLjz9MNCjM3IDA...gMS9MIDIwNTk3L08gNDAvRSAx
...
```

## Decode Base64

This example simulates the transformation in the Write operation from the [XML configuration](#) by converting a Base64 file stream ([application/octet-stream](#)) to Binary and outputting the Binary in the [application/PDF with binary](#) format. The example does not generate a PDF file; a Write operation is required to generate such a file.

The examples use the following functions:

- [fromBase64](#) in the [dw::core::Binaries](#) module to transform a Base64 string to a binary value.
- [readUrl](#) to input a PDF file.

The example assumes that the PDF file is in the [src/main/resources](#) directory of a Mule project in Anypoint Studio. The function reads the input PDF as an octet-stream MIME type. Using the MIME type "binary" as the second [readUrl](#) argument is also valid for this example.

*DataWeave Script:*

```
%dw 2.0
import * from dw::core::Binaries
var myPDF = readUrl("classpath://pdf-test.pdf", "application/octet-stream")
var myPDFasBinary = toBase64(myPDF as Binary)
output application/PDF with binary
---
fromBase64(myPDFasBinary as String) as Binary
```

The output is the text representation of the PDF.

*Output (partial results):*

```
%PDF-1.6
%>>>
37 0 obj <</Linearized 1/L 20597/0 40/E 14115/N 1/T 19795/H [ 1005 215]>>
endobj

xref
37 34
0000000016 00000 n
...
```

## XML Configuration Example

In Anypoint Studio, you can create and run a Mule flow that encodes and decodes a PDF file. After you complete the procedure, the example reads the file from the specified local directory using the File Connector Read operation. It writes the PDF to the specified directory by using the File Connector Write operation, and it uses Transform Message components to encode the file to Base64 and to decode that output from Base64.

1. Create a Mule project in Anypoint Studio:
  - a. Select **File > New > Mule Project**.
  - b. Enter a name for your Mule project.
  - c. Click **Finish**.
2. Use **Add Module** in the Mule Palette to add the File Connector:
  - a. In the **Mule Palette** view, click **(X) Search in Exchange**.
  - b. In the **Add Dependencies to Project** window, type **file** in the search field.
  - c. Click **File Connector** in Available modules.
  - d. Click **Add**.
  - e. Click **Finish**.
3. In the Studio canvas, navigate to the **Configuration XML** tab.
4. Copy and paste the following example between the `<mule/>` elements of a Mule flow in Anypoint Studio:

```

<http:listener-config name="HTTP_Listener_config"
                      doc:name="HTTP Listener config" >
    <http:listener-connection
        host="0.0.0.0"
        port="8081" />
</http:listener-config>
<file:config name="File_Config"
              doc:name="File Config" >
    <file:connection workingDir="/Users/me/Downloads" />
</file:config>
<flow name="encode-decode-base64">
    <http:listener doc:name="Listener"
                  path="/transform"
                  config-ref="HTTP_Listener_config"/>
    <file:read doc:name="Read"
               config-ref="File_Config"
               path="pdf-test.pdf" />
    <ee:transform doc:name="Transform Message">
        <ee:message>
            <ee:set-payload><![CDATA[%dw 2.0
import * from dw::core::Binaries
output text/plain
---
toBase64(payload as Binary)]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <ee:transform doc:name="Transform Message" >
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
import * from dw::core::Binaries
output application/pdf with binary
---
fromBase64(payload as String) as Binary)]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <file:write doc:name="Write"
               path="/Users/me/pdf-test-result.pdf"/>
    <set-payload
        value='#[PDF file created in the directory specified in the Write
operation.]'
        doc:name="Set Payload" />
</flow>

```

Note that it is valid to replace the directive `output application/pdf with binary` in the second Transform Message component with `output application/octet-stream`.

5. Place a PDF file of your choice into a local directory on your machine.
6. Correct the input and output directories of the **Read** and **Write** operations:
  - In the **Read** operation, change the **File Path** value `/Users/me/Downloads` to the location of

your PDF file.

- In the **Write** operation, change the **Path** value `/Users/me/pdf-test.pdf` to the location and file name that you want to output the file.

7. Run the project in Studio.

8. Load the `0.0.0.0:8081/transform` into a browser.

This action creates a PDF file in the specified location and generates a separate `transform` file with the message in the Set Payload component, [PDF file created in the directory specified in the Write operation](#).

## See Also

- [dw::core::Binaries](#)
- [Selectors](#)
- [DataWeave Cookbook](#)

# Call Java Methods (Mule)

From a DataWeave statement, you can call Java methods from any Java class that's in your Mule project. Note that you can only call Static methods via DataWeave (methods that belong to a Java class, not methods that belong to a specific instance of a class). Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

## Call a Java Method

Before you can call a method, you must first import the class it belongs to into your DataWeave code. You can import Java classes just as you import DataWeave modules by including `java!` into the statement.

For example, below is a simple Java class with a single method that appends a random number at the end of a string. Assume that you created this class as part of a Java package named `org.mycompany.utils` in your Mule project's `src/main/java` folder.

```
package org.mycompany.utils;

import java.util.Random;

public class MyUtils {

    public static String appendRandom(String base) {
        return base + new Random().nextInt();
    }
}
```

You can call the method `appendRandom()` from DataWeave code, in any of the following ways.

- Import the class and then refer to the method:

```
%dw 2.0
import java!org::mycompany::utils::MyUtils
output application/json
---
{
    a: MyUtils::appendRandom("myString")
}
```

- Import one or more methods instead of the whole class:

```
%dw 2.0
import java!org::mycompany::utils::MyUtils::appendRandom
output application/json
---
{
    a: appendRandom("myString")
}
```

- If the method is a static method, import and call it in a single line:

```
%dw 2.0
output application/json
---
{
    a: java!org::mycompany::utils::MyUtils::appendRandom(vars.myString)
}
```

All three methods return the variable value with a random string appended:

```
{
    "a":"myString969858409"
}
```

## Instantiate a Java Class

Through DataWeave code, you can instantiate a new object of any class. Note that after creating an instance, you can't call its instance methods through DataWeave. However, you can reference its variables.

This simple Java class has a method and a variable.

```

package org.mycompany.utils;
public class MyClass {

    private String foo;
    public MyClass(String foo) {
        this.foo = foo;
    }

    public String getFoo() {
        return foo;
    }

}

```

To create the **MyClass** example in a Studio project:

1. Create a project for the class in Studio by clicking **New → Mule Project** and providing a name, such as **my-app**, and clicking **Finish**.
2. From **my-app**, create the **org.mycompany.utils** package by right-clicking **src/main/java** from Studio's **Package Explorer**, selecting **New → Package**, and naming the package **org.mycompany.utils**.
3. Create the **MyClass** class in the your new **org.mycompany.utils** package by right-clicking **org.mycompany.utils**, selecting **New → Class**, naming that class **MyClass**, and clicking **Finish**.
4. In the **MyClass.java** tab that opens in Studio, copy, paste, and save the contents **MyClass**.

The following DataWeave example imports **MyClass**, creates a new instance of the class, and calls its instance variable **foo**. (Note that it is not possible for DataWeave to call the object's private **getFoo()** method.)

```

%dw 2.0
import java!org:::mycompany::utils::MyClass
output application/json
---
{
    a: MyClass::new("myString").foo
}

```

To add this DataWeave example to your Studio project:

1. Returning to **my-app.xml** in **src/main/mule**, drag a Transform Message component into the project's canvas, and click to open it.
2. In the component's **Transform Message** tab, replace the entire script in the source code area (on the right) with the DataWeave script (above).
3. Save your changes and click **Preview** (located farthest right on the tab) to view the following output:

```
{  
    "a":"myString"  
}
```

Note that the XML for the Transform Message configuration in the Mule flow looks something like this:

```
<flow name="my-appFlow" >  
    <ee:transform doc:name="Transform Message" >  
        <ee:message >  
            <ee:set-payload ><![CDATA[%dw 2.0  
import java!org::mycompany::utils::MyClass  
output application/json  
---  
{  
a: MyClass::new("myString").foo  
}]]></ee:set-payload>  
            </ee:message>  
        </ee:transform>  
    </flow>
```

## Use a Java Bridge

DataWeave enables you to call any Java static function or constructor by using the Java bridge. This feature is useful when you need to reuse business logic written in Java or to instantiate any Java object that does not have an empty public constructor.

To use a Java bridge, transform a fully qualified Java name to a DataWeave name:

1. Add the prefix **!java**.
2. Replace the dots in the fully qualified name with **::**.

When invoking the Java function, DataWeave transforms each argument to the expected type in the function parameter.

The following example shows how to create a new instance of a **java.lang.NullPointerException**:

```
%dw 2.0  
output application/json  
---  
java!java::lang::NullPointerException::new("foo")
```

The following example invokes the function **java.lang.String.valueOf**:

```
%dw 2.0
output text/plain
import valueOf from java!java::lang::String
---
valueOf(true)
```

## Java Bridge Example

The following example shows an API that returns a greeting message based on the input `uriParam`. The Java bridge calls a static method within the `my.class.org.Greeter` class that builds the response message.

```
%dw 2.0
output application/json
import java!my::class::org::Greeter
---
{
  response: Greeter::greet(attributes.uriParams."name"
}
```

## XML for Java Bridge Example

Paste this code into your Studio XML editor to quickly load the flow for this example into your Mule app:

```

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

<mule xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core"
      xmlns:http="http://www.mulesoft.org/schema/mule/http"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.mulesoft.org/schema/mule/core
                           http://www.mulesoft.org/schema/mule/core/current/mule.xsd
                           http://www.mulesoft.org/schema/mule/http
                           http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
                           http://www.mulesoft.org/schema/mule/ee/core
                           http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
    <http:listener-config name="HTTP_Listener_config" doc:name="HTTP Listener config"
    doc:id="addcdb50-cdfa-400d-992f-495ab6dd373d" >
        <http:listener-connection host="0.0.0.0" port="8081" />
    </http:listener-config>
    <flow name="data-weave-java-bridge-example-flow" >
        <http:listener doc:name="Listener" config-ref="HTTP_Listener_config"
        path="/greet/{name}"/>
        <ee:transform doc:name="Transform Message" >
            <ee:message >
                <ee:set-payload ><![CDATA[%dw 2.0
output application/json
import java!my::class::org::Greeter
---
{
    response: Greeter::greet(attributes.uriParams."name")
}]]></ee:set-payload>
            </ee:message>
            </ee:transform>
        </flow>
    </mule>

```

## Test the App

To test your app, follow these steps:

1. Save and run your Mule app.
2. Run the following curl command: <http://localhost:8081/greet/Username>.

The Mule app returns the following response message:

```
{
  "response": "Greetings, Username!"
}
```

## See Also

- [Java Format](#)
- [Supported Data Formats](#)

## Read and Write a Flat File (Mule)

The following examples show you how to read the contents of a flat file, and also how to write a flat file following the specified schema. Each example consists of a Mule application that uses DataWeave to read and write flat files.

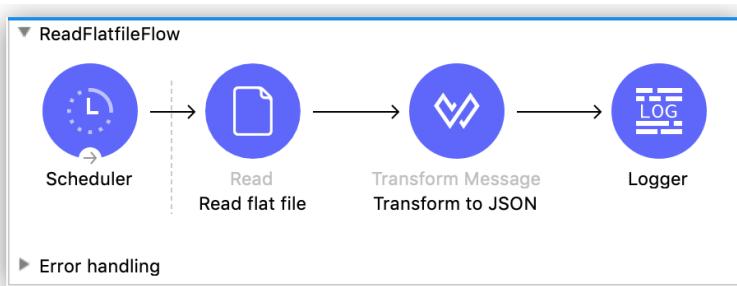
When you work with flat files, configure the flat file schema to use for reading and writing flat file content. The flat file schema defines the structure of a flat file, and this enables Mule to write and read in this format.

These example applications are configured to work with the sample flat file data and the example flat file schema provided in [Full Example Schema](#).

### Example of Reading a Flat File

The following example application consists of:

1. A Scheduler component that triggers the flow
2. A File Read operation that reads a flat file from the specified path
3. A Transform Message component that transforms the content read from the flat file to JSON format
4. A Logger component that outputs the current payload



## Application XML File

```
<mule xmlns:file="http://www.mulesoft.org/schema/mule/file"
      xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/current/mule.xsd
http://www.mulesoft.org/schema/mule/ee/core
http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd
http://www.mulesoft.org/schema/mule/file
http://www.mulesoft.org/schema/mule/file/current/mule-file.xsd">
    <file:config name="File_Config" doc:name="File Config" >
        <file:connection workingDir="${fileWorkingDir}" />
    </file:config>
    <configuration-properties doc:name="Configuration properties"
file="flatfileconfig.properties" />
    <flow name="ReadFlatfileFlow" >
        <scheduler doc:name="Scheduler">
            <scheduling-strategy>
                <fixed-frequency frequency="10000" />
            </scheduling-strategy>
        </scheduler>
        <file:read doc:name="Read flat file" config-ref="File_Config"
path="${fileReadPath}"
outputMimeType='application/flatfile; schemapath=${schemaPath}'/> ①
        <ee:transform doc:name="Transform to JSON" >
            <ee:message >
                <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
payload]]>
                </ee:set-payload>
            </ee:message>
        </ee:transform>
        <logger level="INFO" doc:name="Logger" message='#[["\n"]Reading flat file from:
${mule.home}/apps/${app.name}/${fileReadPath}#[["\n"]#[payload]' />
    </flow>
</mule>
```

① Note that the File Read operation configures the following attributes:

- **path**: The path of the flat file to read
- **outputMimeType**: The output MIME type of the content, in this case, **application/flatfile**
- **schemapath**: The reader property that specifies the path to the flat file schema (**.ffd** file)

After its execution, the example application generates the following output:

```
{
  "Batch": [
    {
      "TDR": [
        {
          "Record Type": "BAT",
          "Sequence Number": 2,
          "Amount": 32876,
          "Account Number": "0123456789",
          "Batch Function": "D",
          "Type": "CR"
        },
        {
          "Record Type": "BAT",
          "Sequence Number": 3,
          "Amount": 87326,
          "Account Number": "0123456788",
          "Batch Function": "D",
          "Type": "CR"
        }
      ],
      "BCF": {
        "Record Type": "BAT",
        "Sequence Number": 4,
        "Batch Transaction Count": 2,
        "Batch Transaction Amount": 120202,
        "Unique Batch Identifier": "A000000001",
        "Batch Function": "T",
        "Type": "CR"
      },
      "BCH": {
        "Record Type": "BAT",
        "Company Name": "ACME RESEARCH",
        "Sequence Number": 1,
        "Unique Batch Identifier": "A000000001",
        "Batch Function": "H"
      }
    },
    {
      "TDR": [
        {
          "Record Type": "BAT",
          "Sequence Number": 6,
          "Amount": 3582,
          "Account Number": "1234567890",
          "Batch Function": "D",
          "Type": "DB"
        },
        {
          "Record Type": "BAT",
          "Sequence Number": 7,
        }
      ]
    }
  ]
}
```

```

        "Amount": 256,
        "Account Number": "1234567891",
        "Batch Function": "D",
        "Type": "CR"
    }
],
"BCF": {
    "Record Type": "BAT",
    "Sequence Number": 8,
    "Batch Transaction Count": 2,
    "Batch Transaction Amount": 3326,
    "Unique Batch Identifier": "A000000002",
    "Batch Function": "T",
    "Type": "DB"
},
"BCH": {
    "Record Type": "BAT",
    "Company Name": "AJAX EXPLOSIVES",
    "Sequence Number": 5,
    "Unique Batch Identifier": "A000000002",
    "Batch Function": "H"
}
}
],
"RQF": {
    "Record Type": "RQF",
    "File Batch Count": 2,
    "File Transaction Count": 8,
    "File Transaction Amount": 116876,
    "Unique File Identifier": "A000000001",
    "Type": "CR"
},
"RQH": {
    "Record Type": "RQH",
    "File Creation Time": "10:10:00",
    "File Creation Date": "2018-09-01",
    "Currency": "USD",
    "Unique File Identifier": "A000000001"
}
}

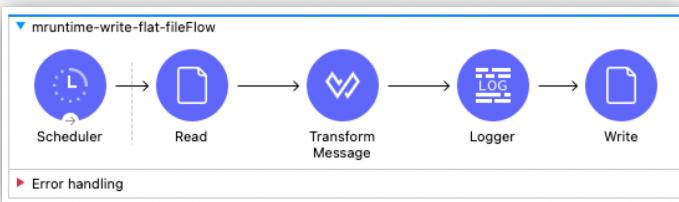
```

## Example of Writing a Flat File

The following example application contains:

1. A Scheduler component that triggers the flow
2. A File Read operation that reads a file containing the JSON output from the previous example
3. A Transform Message component that transforms the JSON payload to the specified flat file format

4. A Logger component that outputs the current payload
5. A File Write operation that outputs the payload to a flat file in the specified target path



#### *Application XML File*

```

<mule xmlns:file="http://www.mulesoft.org/schema/mule/file"
      xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/current/mule.xsd
http://www.mulesoft.org/schema/mule/ee/core
http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd
http://www.mulesoft.org/schema/mule/file
http://www.mulesoft.org/schema/mule/file/current/mule-file.xsd">
    <file:config name="File_Config" doc:name="File Config" >
        <file:connection workingDir="${fileWorkingDir}" />
    </file:config>
    <configuration-properties doc:name="Configuration properties"
file="flatfileconfig.properties" />
    <flow name="WriteFlatFile" >
        <scheduler doc:name="Scheduler" >
            <scheduling-strategy >
                <fixed-frequency frequency="10000" />
            </scheduling-strategy>
        </scheduler>
        <file:read doc:name="Read JSON file" config-ref="File_Config"
path="${jsonFilePath}"/>
        <ee:transform doc:name="Transform to flat file"> ①
            <ee:message>
                <ee:set-payload><![CDATA[%dw 2.0
output application/json schemaPath = "flatfileschema.ffd"
%dw 2.0
--->
{
    RQH: {
        "Record Type": payload.RQH."Record Type",
        "File Creation Date": payload.RQH."File Creation Date" as Date {format: "yyyy-MM-dd"},
        "File Creation Time": payload.RQH."File Creation Time" as Time,
        "Unique File Identifier": payload.RQH."Unique File Identifier",
        "Currency": payload.RQH.Currency
    },
    
```

```

Batch: payload.Batch map ( batch , indexOfBatch ) -> {
    BCH: {
        "Record Type": batch.BCH."Record Type",
        "Sequence Number": batch.BCH."Sequence Number",
        "Batch Function": batch.BCH."Batch Function",
        "Company Name": batch.BCH."Company Name",
        "Unique Batch Identifier": batch.BCH."Unique Batch Identifier"},
    TDR: batch.TDR map ( tdr, indexOfTdr ) ->
    {
        "Record Type": tdr.TDR."Record Type",
        "Sequence Number": tdr.TDR."Sequence Number",
        "Batch Function": tdr.TDR."Batch Function",
        "Account Number": tdr.TDR."Account Number",
        "Amount": tdr.TDR.Amount,
        "Type": tdr.TDR.'Type'
    },
    BCF: {
        "Record Type": batch.BCF."Record Type",
        "Sequence Number": batch.BCF."Sequence Number",
        "Batch Function": batch.BCF."Batch Function",
        "Batch Transaction Amount": batch.BCF."Batch Transaction Amount",
        "Type": batch.BCF.'Type',
        "Batch Transaction Count": batch.BCF."Batch Transaction Count",
        "Unique Batch Identifier": batch.BCF."Unique Batch Identifier"}
    },
    RQF: {
        "Record Type": payload.RQF."Record Type",
        "File Batch Count": payload.RQF."File Batch Count",
        "File Transaction Count": payload.RQF."File Transaction Count",
        "File Transaction Amount": payload.RQF."File Transaction Amount",
        Type: payload.RQF.Type,
        "Unique File Identifier": payload.RQF."Unique File Identifier"
    }
}
]]>
    </ee:set-payload>
    </ee:message>
    </ee:transform>
    <logger level="INFO" doc:name="Logger" message='#[["\n"]Writing flat file to:
${mule.home}/apps/${app.name}/${fileWritePath}#[["\n"]#[payload]'/>
    <file:write doc:name="Write flat file" config-ref="File_Config"
path="${fileWritePath}" /> ②
    </flow>
</mule>

```

① Note that the Transform component specifies the following writer properties in the script header:

- **output**: The output MIME type of the transformation, in this case, `application/flatfile`
- **schemapath**: The writer property that specifies the path to the flat file schema (`.ffd` file)

- ② Note that the File Write operation specifies the `path` property to define where to write the flat file

After its execution, the example application generates the following output:

```
RQH201809011010A000000001USD
BAT00001HACME RESEARCH           A00000001
BAT00000D             0000000000
BAT00000D             0000000000
BAT00004T0000120202CR00002A00000001
BAT00005HAJAX EXPLOSIVES         A00000002
BAT00000D             0000000000
BAT00000D             0000000000
BAT00008T000003326DB00002A00000002
RQF0020000800000116876CRA00000001
```

## Test the Examples Locally

If you want to test the examples locally, follow these steps to create and configure your example Mule application:

1. Create a new **Mule Project** in Anypoint Studio.
2. Copy the XML code of the application you want to test into the **Configuration XML** view of your project.
3. Use [Add Modules](#) to add the Anypoint Connector for File (File Connector) to your project.
4. Create a file named `flatfileconfig.properties` in the `src/main/resources` folder of your Mule application, and set its content with the following properties:

```
fileReadPath=flatfile
schemaPath=flatfileschema.ffd
fileWritePath=flatfilenew
jsonFileReadPath=flatFileJson.json
fileWorkingDir=#["${mule.home}/apps/${app.name}/"]
```

5. Create a file named `flatfile` in the `src/main/resources` folder of your Mule application, and set its content with the [flat file sample data](#) from the DataWeave flat file schema example.
6. Create a file named `flatfileschema.ffd` in the `src/main/resources` folder of your Mule application, and set its content with the [full schema example](#) from the DataWeave flat file schema example.
7. Create a file named `flatFileJson.json` in the `src/main/resources` folder of your Mule application and set its content with the [JSON transformation output](#) from the example that reads a flat file.

## See Also

- [Flat File Format](#)

- [Flatfile Schemas](#)
- [Set Reader and Writer Properties](#)
- [Configure Properties](#)

## Use a Reader Property through a Connector (Mule)

Connector operations that read input data provide an `outputMimeType` field that you can use to identify the *input* MIME type. This field also accepts one or more reader properties, which is useful for helping DataWeave read the input correctly.

By default, the DataWeave reader treats a comma (,) as a separator. However, assume that a CSV input is separated by the pipe (|) but that the content also contains commas. You can set a CSV reader property to recognize the pipe as the separator.

### Pipe-separated CSV Input

```
id | name | role
1234 | Shokida,Mr. | dev
2345 | Achaval,Mr. | arch
3456 | Felisatti,Ms. | mngr
4567 | Chibana,Mr. | dev
```

Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 examples](#). For other DataWeave versions, you can use the version selector in the DataWeave table of contents.

The following example uses the Read operation in the File connector to read the pipe-separated (|) CSV input, and it uses a DataWeave script in the Transform Message component to output a row of the input in comma-separated format.

## Mule Flow:

```
<flow name="ex-use-csv-reader-props" >
    <scheduler doc:name="Scheduler" >
        <scheduling-strategy >
            <fixed-frequency frequency="90" timeUnit="SECONDS"/>
        </scheduling-strategy>
    </scheduler>
    <file:read doc:name="Read" config-ref="File_Config" path="staff.csv"
        outputMimeType='application/csv; separator=|; header=true' />
    <ee:transform doc:name="Transform Message" >
        <ee:message>
            <ee:set-payload><![CDATA[%dw 2.0
output application/csv header = false
---
payload[1]]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <logger level="INFO" doc:name="Logger" message="#[payload]" />
</flow>
```

The Scheduler component (`<scheduler/>`) in the example generates a Mule event each time it triggers the flow.

The Read operation (`<file:read/>`) uses `outputMimeType='application/csv; separator=|; header=true'` to identify the input MIME type (application/csv), the CSV separator (|), and the header setting (true).

The script in the Transform Message component (`<ee:transform>`) returns the second row from the input CSV (2345 , Achaval\,Mr. , arch). The script uses `payload[1]` to select that row because DataWeave treats each row in the CSV as an index of an array. The output is comma-separated because the default CSV separator is a comma. The script omits the CSV header from the output by using the `output` directive (`output application/csv header = false`).

The Logger component (`<logger/>`), which is set to `payload`, writes the comma-separated output as INFO in its `LoggerMessageProcessor` message: 2345 , Achaval\,Mr. , arch. To avoid treating the comma before "Mr." as a separator, the CSV output escapes the comma with a backslash (Achaval\,Mr.).

The following examples pass reader properties through the `outputMimeType` field:

- [CSV example](#) that uses `streaming=true; header=true`
- [JSON example](#) that uses `streaming=true`

## See Also

- [DataWeave Formats](#)
- [Core Components](#)
- [DataWeave Cookbook](#)

# Use Dynamic Writer Properties (Mule)

The following example shows a Mule flow that transforms comma-separated CSV data into pipe-separated (|) CSV data. Instead of setting a literal value in the script, the example selects the pipe value from a [Mule variable](#), which is part of a Mule event that travels through data-processing components in the flow. The example uses the DataWeave writer function in the body of the script to change the comma-separated input to pipe-separated output.

The Scheduler component (`<scheduler/>`) in the example generates a Mule event each time it triggers the flow.

The Set Variable component (`<set-variable/>`) creates a Mule variable named `delimiter` with the value '|'. In Studio, the value is created as an `fx` expression (`value="#['|']"`), rather than a simple string (`value="'|'"`).

The Transform Message component (`<ee:transform-message/>`) contains a DataWeave script that reads comma-separated CSV input and uses the Mule variable to write pipe-separated CSV output.

The Logger component (`<logger/>`), which is set to `payload`, writes the pipe-separated output as INFO in its `LoggerMessageProcessor` message: `macaroni | rigatoni | ravioli | spaghetti`.

*Mule Flow:*

```
<flow name="ex-use-mule-var-as-prop-config-val" >
  <scheduler doc:name="Scheduler" >
    <scheduling-strategy >
      <fixed-frequency frequency="45" timeUnit="SECONDS"/>
    </scheduling-strategy>
  </scheduler>
  <set-variable value="#['|']" doc:name="Set Variable" variableName="delimiter" />
  <ee:transform doc:name="Transform Message" >
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
var myVar = read("macaroni , rigatoni , ravioli , spaghetti", "application/csv" ,
{"header":false, separator:','})
output application/csv with binary
---
write(myVar, "application/csv", {"header":false,
"separator":vars.delimiter})]]></ee:set-payload>
    </ee:message>
  </ee:transform>
  <logger level="INFO" doc:name="Logger" message="#[payload]"/>
</flow>
```

To provide comma-separated input for the example, the script in the Transform Message component uses a `read` function as a value to a DataWeave variable, `myVar`:

```
var myVar = read("macaroni , rigatoni , ravioli , spaghetti", "application/csv" ,
{"header":false, separator:','})
```

To write pipe-separated output, the DataWeave `write` function in the body of the script changes the comma-separated input to pipe-separated output. To select the pipe value from the Mule variable (`delimiter`), the function passes the writer property configuration `"separator":vars.delimiter` as an argument:

```
write(myVar, "application/csv", {"header":false, "separator":vars.delimiter})
```

The script uses the following DataWeave `output` directive in the script's header: `output application/csv with binary`. It is necessary to append the `with binary` setting (introduced in Mule 4.3.0) to invoke the binary writer because the `write` function returned CSV output. Without the setting, DataWeave would attempt to write CSV again and produce the following issue: `CSV Structure should be an Array<Object> or Object but got String, while writing CSV.`

Note that if you want to see pipe-separated content from the Transform Message component's **Preview** screen in the Studio UI, you must create sample data for it:

1. Double-click the Transform Message component from the **Message Flow** area of the Studio app.
2. Right-click **delimiter: String** in the **Context** tab, which is located on the bottom left of the configuration UI for the component.
3. Select **Edit Sample Data** to open a **vars-delimiter** tab.
4. Replace any content in the **vars-delimiter** with the following sample data for the **delimiter** value: `"|"`. You must put the delimiter in quotation marks.

For further guidance with sample data, see [Previewing the Output of a Transformation](#).

## See Also

- [DataWeave Formats](#)
- [Core Components](#)
- [DataWeave Cookbook](#)

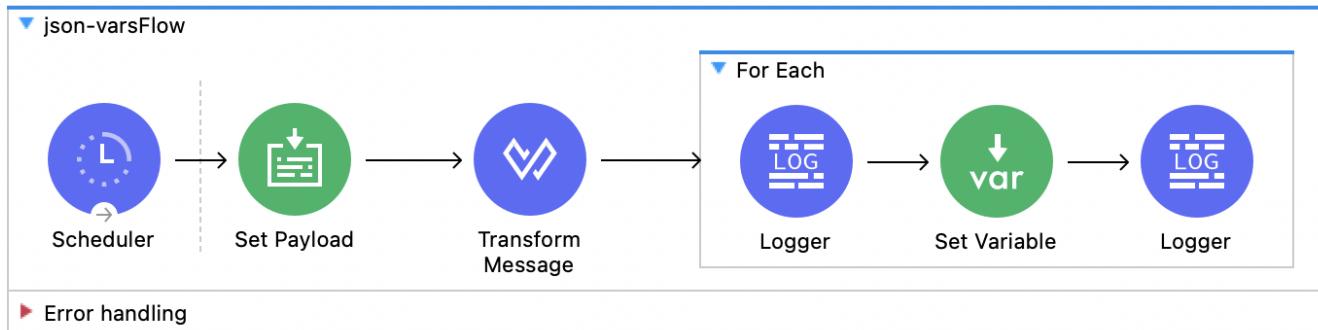
## Extract Key/Value Pairs with Pluck Function (Mule)

In addition to using the DataWeave functions such as `entriesOf`, `keysOf`, or `valuesOf` to work with key-value pairs, you can also use `pluck`. The following Mule app example shows how to use the DataWeave `pluck` function to extract key-value pairs from a JSON payload and store them into Mule event variables with variable names as the keys and their corresponding values.

The Mule app consists of:

- A Scheduler component that triggers the flow
- A Set Payload component that sets the JSON payload
- A Transform Message component that transforms the content read from the Set Payload component by extracting the key-value pairs with the `pluck` function and putting them into an array
- A For Each scope component that iterates over the array

- A Set Variable component that sets the variable name as the Key and variable value as the Value in the JSON array
- A Logger component that prints the variable results per each iteration



Application XML File (reformatted for readability):

```

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

<mule xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/current/mule.xsd
http://www.mulesoft.org/schema/mule/ee/core
http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
    <flow name="json-varsFlow" >
        <scheduler doc:name="Scheduler" >
            <scheduling-strategy >
                <fixed-frequency frequency="10000"/>
            </scheduling-strategy>
        </scheduler>
        <set-payload value='{"firstName": "jason", "job": "engineer",
"dept":"support"}' 
            doc:name="Set Payload"
            mimeType="application/json" />
        <ee:transform doc:name="Transform Message" >
            <ee:message >
                <ee:set-payload ><![CDATA[%dw 2.0
output application/json
--->
payload pluck (value,key) -> {
    Test:{
        Key: key,
        Value: value
    }
}]]></ee:set-payload>
            </ee:message>
        </ee:transform>
        <foreach doc:name="For Each"
            collection="#[payload.Test]">
            <logger level="INFO" doc:name="Logger" message="#[payload]" />
            <set-variable value="#[payload.Value]"
                doc:name="Set Variable"
                variableName="#[payload.Key]"/>
            <logger level="INFO"
                doc:name="Logger"
                message="#[vars[payload.Key]]"/>
        </foreach>
    </flow>
</mule>

```

Note that the Transform Message configures the following DataWeave script:

```
%dw 2.0
output application/json
---
payload pluck (value,key) -> {
    Test: {
        Key: key,
        Value: value
    }
}
```

When the Mule app executes with the following JSON payload:

```
{
  "firstName": "jason", "job": "engineer", "dept": "support"
}
```

The DataWeave transformation result is:

```
[
  {
    "Test": {
      "Key": "firstName",
      "Value": "jason"
    }
  },
  {
    "Test": {
      "Key": "job",
      "Value": "engineer"
    }
  },
  {
    "Test": {
      "Key": "dept",
      "Value": "support"
    }
  }
]
```

The For Each scope component iterates over the transformed values, and the app logs the values of the variable results (reformatted for readability):

```

INFO 2022-06-22 10:11:17,864 [[MuleRuntime].uber.02:
[jsonsample].json-varsFlow.CPU_INTENSIVE @5c652aa]
[processor: json-varsFlow/processors/2/processors/0; event: 545f5ef0-f24e-11ec-
8c92-147ddaaaf4f97]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
{
    "Key": "firstName",
    "Value": "jason"
}
INFO 2022-06-22 10:11:17,870 [[MuleRuntime].uber.02: [jsonsample].json-
varsFlow.CPU_INTENSIVE @5c652aa]
[processor: json-varsFlow/processors/2/processors/2; event: 545f5ef0-f24e-11ec-
8c92-147ddaaaf4f97]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
"jason"
INFO 2022-06-22 10:11:17,871 [[MuleRuntime].uber.02: [jsonsample].json-
varsFlow.CPU_INTENSIVE @5c652aa]
[processor: json-varsFlow/processors/2/processors/0; event: 545f5ef0-f24e-11ec-
8c92-147ddaaaf4f97]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
{
    "Key": "job",
    "Value": "engineer"
}
INFO 2022-06-22 10:11:17,875 [[MuleRuntime].uber.02:
[jsonsample].json-varsFlow.CPU_INTENSIVE @5c652aa]
[processor: json-varsFlow/processors/2/processors/2; event: 545f5ef0-f24e-11ec-
8c92-147ddaaaf4f97]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
"engineer"
INFO 2022-06-22 10:11:17,877 [[MuleRuntime].uber.02:
[jsonsample].json-varsFlow.CPU_INTENSIVE @5c652aa]
[processor: json-varsFlow/processors/2/processors/0; event: 545f5ef0-f24e-11ec-
8c92-147ddaaaf4f97]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
{
    "Key": "dept",
    "Value": "support"
}
INFO 2022-06-22 10:11:17,879 [[MuleRuntime].uber.02: [jsonsample].json-
varsFlow.CPU_INTENSIVE @5c652aa]
[processor: json-varsFlow/processors/2/processors/2; event: 545f5ef0-f24e-11ec-
8c92-147ddaaaf4f97]
org.mule.runtime.core.internal.processor.LoggerMessageProcessor:
"support"

```

## See Also

- [DataWeave Formats](#)

- DataWeave Cookbook

# DataWeave Reference

DataWeave functions are packaged in modules. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 operators](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

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

- `import dw::core::Strings`
- `import camelize, capitalize from dw::core::Strings`
- `import * from dw::core::Strings`

The way you import a module impacts the way you need to call its functions from a DataWeave script. If the directive does not list specific functions to import or use `* from` to import all functions from a function module, you need to specify the module when you call the function from your script. For example, this import directive does not identify any functions to import from the String module, so it calls the `pluralize` function like this: `Strings::pluralize("box")`.

## Transform

```
%dw 2.0
import dw::core::Strings
output application/json
---
{ 'plural': Strings::pluralize("box") }
```

The next example identifies a specific function to import from the String module, so it can call the method like this: `pluralize("box")`.

## Transform

```
%dw 2.0
import pluralize from dw::core::Strings
output application/json
---
{ 'plural': pluralize("box") }
```

The next example imports all functions from the String module, so it can call the method like this: `pluralize("box")`.

## *Transform*

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{ 'plural': pluralize("box") }
```

# DataWeave Modules

- [dw::core::Arrays](#)
- [dw::core::Binaries](#)
- [dw::util::Coercions](#)
- [dw::Core](#)
- [dw::extension::DataFormat](#)
- [dw::core::Dates](#)
- [dw::util::Diff](#)
- [dw::xml::Dtd](#)
- [Encryption \(dw::Crypto\)](#)
- [dw::util::Math](#)
- [dw::Mule](#)
- [dw::module::Multipart](#)
- [dw::core::Numbers](#)
- [dw::core::Objects](#)
- [dw::core::Periods](#)
- [dw::Runtime](#)
- [dw::core::Strings](#)
- [dw::System](#)
- [dw::util::Timer](#)
- [dw::util::Tree](#)
- [dw::core::Types](#)
- [dw::core::URL](#)
- [dw::util::Values](#)

# Function Signatures

Each DataWeave function in the DataWeave Reference is identified by its function signature. A function specifies a function name, zero or more parameters, and a return type.

Basic syntax of a two-parameter function signature:

```
function(parameterType,parameterType): returnType
```

For example, the signature `contains(String, String): Boolean` for the `contains` function has two parameters, each of which accepts a value of the `String` type. The function returns a `Boolean` value.

Parameters in the function signature correspond, *in order*, to their named parameters, which are described in Parameters tables for each signature. For example, in `contains(String, String): Boolean`, the Parameters table indicates that `text` is the first parameter and `toSearch` is the second.

Many DataWeave functions are overloaded to handle different data types. There is a unique function signature for each variant of the function. For example, `isEmpty` is overloaded to support an input value of an `Array`, `String`, `Object`, or `Null` type.

For more information on the data types, see [Type System](#).

## Type Parameters in Function Signatures

Function signatures can contain type parameters, which are similar to generics in some programming languages. For example, the `Array<T>` in `contains(Array<T>, Any): Boolean` indicates that the array can contain elements of any type (`T`) that DataWeave supports. By contrast, an array of a specific type or types specifies the type, for example, `Array<String>`, `Array<Number>`, or for both types `Array<String|Number>`.

## Function Types in Function Signatures

Functions can be passed as arguments. Some parameters within function signatures are function types that take one or more parameters of the form `parameterName:Type` and have a return type ( `ReturnType`). For example, the function type `(item: T, index: Number) → R` is a parameter type of the `map` function `(map(Array<T>, (item: T, index: Number) → R): Array<R>)`. The function type accepts a value of any type `T` and a value of type `Number`, which serve as parameters to return a value of type `R`.

## See Also

[Functions](#)

[About DataWeave](#)

[DataWeave Quickstart](#)

## DataWeave Operators

DataWeave supports several operators, including mathematical operators, equality operators, and operators such as `prepend`, `append` and `update`. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to [DataWeave version 1.2 operators](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

## Mathematical Operators

DataWeave supports the most common mathematical operators:

Operator	Description
<code>+</code>	For addition.
<code>-</code>	For subtraction.
<code>*</code>	For multiplication.
<code>/</code>	For division.

In addition to operating with numbers, the `(-)` and `(+)` operators can also operate with complex data structures like arrays, objects, and dates.

The following example uses mathematical operators with different data types:

*Source*

```
%dw 2.0
output application/json
---
{ "mathOperators" : [
    { "2 + 2" : (2 + 2) },
    { "2 - 2" : (2 - 2) },
    { "2 * 2" : (2 * 2) },
    { "2 / 2" : (2 / 2) },
    { "[1,2,3] - 1 + 4" : [1,2,3] - 1 + 4},
    { "{a:1, b:2, c:3} - 'a' " : {a:1, b:2, c:3} - "a"},
    { "|2021-03-02T10:39:59| - |P1D| + |PT3H|" : |2021-03-02T10:39:59| - |P1D| +
    |PT3H|}
]
}
```

*Output*

```
{
  "mathOperators": [
    { "2 + 2": 4 },
    { "2 - 2": 0 },
    { "2 * 2": 4 },
    { "2 / 2": 1 },
    { "[1,2,3] - 1 + 4": [2,3,4] },
    { "{a:1, b:2, c:3} - 'a' " : {"b": 2, "c": 3} },
    { "|2021-03-02T10:39:59| - |P1D| + |PT3H|" : "2021-03-01T13:39:59" }
  ]
}
```

Several DataWeave functions operate on numbers, for example: `sum`, `mod` (for modulo), and `avg` (for average).

## Equality and Relational Operators

DataWeave supports the following equality and relational operators:

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

Note that you can negate these operators by using the [logical operator](#), `not`.

The following example uses relational operators:

*Source*

```
%dw 2.0
output application/json
---
{ "relational" : [
    { "1 < 1" : (1 < 1) },
    { "1 > 2" : (1 > 2) },
    { "1 <= 1" : (1 <= 1) },
    { "1 >= 1" : (1 >= 1) }
]
```

*Output*

```
{ "relational": [
    { "(1 < 1)": false },
    { "(1 > 2)": false },
    { "(1 <= 1)": true },
    { "(1 >= 1)": true }
]
```

Note that if the operands of the relational operator belong to different types, DataWeave coerces the right-side operand to the type of the left-side operand. For example, in the expression `"123" > 12` DataWeave coerces `12` (a Number type) to `"12"` (a String type) and compares each String value lexicographically. In the expression `123 > "12"`, DataWeave coerces the String value `"12"` to the Number value `12` and compares the numbers.

These examples use equality operators:

## Source

```
%dw 2.0
output application/dw
---
{ "equality" :
  [
    (1 == 1),
    (1 == 2),
    ("true" == true),
    ("true" ~= true),
    (['true'] ~= [true]),
    ('1' ~= 1)
  ]
}
```

## Output

```
{
  equality: [ true, false, false, true, true, true ] }
```

## Logical Operators

DataWeave supports the following logical operators:

Operator	Description
not	Negates the result of the input. See also, !.
!	Negates the result of the input. See also, not. <i>Introduced in DataWeave 2.2.0. Supported by Mule 4.2 and later.</i>
and	Returns true if the result of all inputs is true, false if not.
or	Returns true if the result of any input is true, false if not.



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

The following examples use logical operators:

## Source

```
%dw 2.0
var myArray = [1,2,3,4,5]
var myMap = myArray map not (($ mod 2) == 0)
output application/json
---
{
    "not" : [
        "notTrue" : not true,
        "notFalse" : not false,
        "myMapWithNot" : myMap
    ],
    "and" : [
        "andTrueFalse" : true and false,
        "andIsTrue" : (1 + 1 == 2) and (2 + 2 == 4),
        "andIsFalse" : (1 + 1 == 2) and (2 + 2 == 2)
    ],
    "or" : [
        "orTrueFalse" : true or false,
        "orIsTrue" : (1 + 1 == 2) or (2 + 2 == 2),
        "orIsFalse" : (1 + 1 == 1) or (2 + 2 == 2)
    ],
    "!-vs-not" : [
        "example-!" : (! true or true),
        "example-not" : (not true or true)
    ]
}
```

Note that `myMap` iterates through the items in a list (`myArray`) and determines whether the modulo (`mod`) expression *does not* evaluate to `0` when applied to each given item.

## Output

```
{  
  "not": [  
    { "notTrue": false },  
    { "notFalse": true },  
    { "myMapWithNot": [ true, false, true, false, true ] }  
],  
  "and": [  
    { "andTrueFalse": false },  
    { "andIsTrue": true },  
    { "andIsFalse": false }  

```

Note that `not` works in expressions such as `not (true)`, but `not(true)` (without the space) does not work.

You can use logical operators together. The following example uses:

- `or not` as defined in the `orNot` expression.
- `and not` in `andNot`.
- `not and not` in `notWithAndNot`.

```
%dw 2.0
var orNot = if (1 + 1 == 4 or not 1 == 2) {"answer": "orNot - Condition met"}
            else {"answer": "nope"}
var andNot = if (1 + 1 == 2 and not 1 == 2) {"answer": "andNot - Condition met"}
            else {"answer": "nope"}
var notWithAndNot = if (not (1 + 1 == 2 and not 1 == 1)) {"answer": "notWithAndNot -
Condition met"}
            else {"answer": "nope"}
output application/json
---
{
  "answers" :
  [
    orNot,
    andNot,
    notWithAndNot
  ]
}
```

### Output

```
{
  "answers": [
    { "answer": "orNot - Condition met" },
    { "answer": "andNot - Condition met" },
    { "answer": "notWithAndNot - Condition met" }
  ]
}
```

DataWeave executes the code inside each `if` block because all conditional expressions in the example evaluate to `true`.

## Prepend, Append, and Remove Operators for Arrays

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

Operator	Description
<code>&gt;&gt;</code>	Prepends data on the left-hand side of the operator to items in the array on the right-hand side. For example, <code>1 &gt;&gt; [2]</code> results in <code>[ 1, 2 ]</code> , prepending <code>1</code> to <code>2</code> in the array.
<code>&lt;&lt;</code>	Appends data on the right-hand side of the operator to items in the array on the left-hand side. For example, <code>[1] &lt;&lt; 2</code> results in <code>[ 1, 2 ]</code> , appending <code>2</code> to <code>1</code> in the array.
<code>+</code>	Appends data on the right-hand side of the operator to items in the array on the left-hand side. For example, <code>[1] + 2</code> results in <code>[ 1, 2 ]</code> , appending <code>2</code> to <code>1</code> in the array. The array is always on the left-hand side of the operator.

Operator	Description
-	Removes a specified element of any supported type from an array.

The following examples show uses of prepend, append, and remove operators on arrays:

```
%dw 2.0
output application/json
---
{
  "prepend-append" : [
    // Array on right side when prepending.
    { "prepend" : 1 >> [2] },
    { "prepend-number" : 1 >> [1] },
    { "prepend-string" : "a" >> [1] },
    { "prepend-object" : { "a" : "b"} >> [1] },
    { "prepend-array" : [1] >> [2, 3] },
    { "prepend-binary" : (1 as Binary) >> [1] },
    { "prepend-date-time" : |23:57:59Z| >> [ |2017-10-01| ] },
    // Array is on left side when appending.
    { "append-number" : [1] << 2 },
    { "append-string" : [1] << "a" },
    { "append-object" : [1] << { "a" : "b"} },
    { "append-array" : [1,2] << [1, 2, 3] },
    { "append-binary" : [1] << (1 as Binary) },
    { "append-date-time" : [ |2017-10-01| ] << |23:57:59Z| },
    { "append-object-to-array" : [1,2] << {"a" : "b"} },
    { "append-array-to-array1" : ["a","b"] << ["c","d"] },
    { "append-array-to-array2" : [[{"a","b"}, {"c","d"}]] << [{"e","f"}] },
    // + always appends within the array
    { "append-with+" : [1] + 2 },
    { "append-with+" : [2] + 1 },
    { "removeNumberFromArray" : ( [1,2,3] - 2 ) },
    { "removeObjectFromArray" : ( [ {a : "b"}, {c : "d"} , { e : "f"} ] - { c : "d"} )
  }
]
}
```

## Output

```
{  
  "prepend-append": [  
    { "prepend": [ 1, 2 ] },  
    { "prepend-number": [ 1, 1 ] },  
    { "prepend-string": [ "a", 1 ] },  
    { "prepend-array": [ [ 1 ], 2, 3 ] },  
    { "prepend-object": [ { "a": "b" }, 1 ] },  
    { "prepend-binary": [ "\u0001", 1 ] },  
    { "prepend-date-time": [ "23:57:59Z", "2017-10-01" ] },  
    { "append-number": [ 1, 2 ] },  
    { "append-string": [ 1, "a" ] },  
    { "append-object": [ 1, { "a": "b" } ] },  
    { "append-array": [ 1, 2, [ 1, 2, 3 ] ] },  
    { "append-binary": [ 1, "\u0001" ] },  
    { "append-date-time": [ "2017-10-01", "23:57:59Z" ] },  
    { "append-object-to-array": [ 1, 2, { "a": "b" } ] },  
    { "append-array-to-array1": [ "a", "b", ["c","d"] ] },  
    { "append-array-to-array2": [ ["a","b"], ["c","d"], ["e","f"] ] },  
    { "append-with-+": [ 1, 2 ] },  
    { "append-with-+": [ 2, 1 ] },  
    { "removeNumberFromArray": [ 1, 3 ] },  
    { "removeObjectFromArray": [ { "a": "b" }, { "e": "f" } ] }  
  ]  
}
```

## Scope and Flow Control Operators

DataWeave supports operators that control the flow and scope of expressions:

- **do** and **using**
- **if** **else** and **else if**

*Table 4. Scope Operators:*

Operator	Description
<b>do</b>	Creates a scope in which new variables, functions, annotations, or namespaces can be declared and used. The syntax is similar to a mapping in that it is composed of a header and body separated by <b>---</b> . Its header is where all the declarations are defined, and its body is the result of the expression. See <a href="#">do</a> and <a href="#">Examples: Local DataWeave Variables</a> for examples.
<b>using</b>	Replaced by <b>do</b> . Supported for backwards compatibility only.

*Table 5. Flow Control Operators:*

Operator	Description
<code>if else</code>	An <code>if</code> operator evaluates a conditional expression and returns the value under the <code>if</code> only if the conditional expression is true. Otherwise, it returns the expression under <code>else</code> . Every <code>if</code> expression must have a matching <code>else</code> expression. See <a href="#">if else</a> for an example.
<code>else if</code>	An <code>else</code> operator chains expressions together within an if-else construct by incorporating <code>else if</code> . See <a href="#">else if</a> for an example.

## Update Operator

DataWeave supports the `update` operator, which enables you to update specified fields of a data structure with new values.

*Introduced in DataWeave 2.3.0. Supported by Mule 4.3 and later.*

Versions prior to Mule 4.3 require the following syntax to increase the `age` value in `myInput` by one without recreating the entire object:

```
%dw 2.0
var myInput = {
    "name": "Ken",
    "lastName": "Shokida",
    "age": 30
}
output application/json
---
myInput mapObject ((value,key) ->
    if(key as String == "age")
        {(key): value as Number + 1}
    else
        {(key): value} )
```

In Mule 4.3, the `update` operator simplifies the syntax:

```
%dw 2.0
var myInput = {
    "name": "Ken",
    "lastName": "Shokida",
    "age": 30
}
output application/json
---
myInput update {
    case age at .age -> age + 1
}
```

Both DataWeave scripts return the same result:

```
{  
    "name": "Ken",  
    "lastName": "Shokida",  
    "age": 31  
}
```

With **update**, casting or iterating through all the key-value pairs is not necessary.

The **update** syntax is as follows:

```
<value_to_update> update {  
    case <variable_name> at <update_expression>[!]? [if(<conditional_expression>)]? ->  
<new_value>  
    ...  
}
```

- **value\_to\_update** represents the original value to update.
- **update\_expression** is the selector expression that matches a value in **value\_to\_update**.
- **variable\_name** is the name of the variable to bind to the matched **update\_expression** value.
- **new\_value** is the expression with the new value.
- **[if(<conditional\_expression>)]?** If **conditional\_expression** returns **true**, the script updates the value. Use of a conditional expression is optional.
- **[!]** marks the selector as an upsert. If the expression doesn't find a match, the **!** makes the **update** operator create all the required elements and insert the new value.
- **update\_expression** supports multiple **case** expressions.



The **update** operator doesn't mutate the existing value. Instead, the operator creates a new value with the updated expressions.

## Selector Expressions

Use selector expressions with the **update** operator to specify the fields to update.

### Value Selector

This matching selector can check for matches to any value within an object.

The following example updates a field at the root level and a field in a nested level using the value selector:

## Source

```
%dw 2.0
var myInput = {
    "name": "Ken",
    "lastName": "Shokida",
    "age": 30,
    "address": {
        "street": "Amenabar",
        "zipCode": "AB1234"
    }
}
output application/json
---
myInput update {
    case age at .age -> age + 1
    case s at .address.street -> "First Street"
}
```

## Output

```
{
    "name": "Ken",
    "lastName": "Shokida",
    "age": 31,
    "address": {
        "street": "First Street",
        "zipCode": "AB1234"
    }
}
```

## Index Selector

This matching selector enables you to match any array by its index.

The following example updates an element that is at the first location of the array:

## Input

```
{
    "name": "Ken",
    "lastName": "Shokida",
    "age": 30,
    "addresses": [
        {
            "street": "Amenabar",
            "zipCode": "AB1234"
        }
    ]
}
```

## Source

```
%dw 2.0
output application/json
---
payload update {
    case s at .addresses[0] -> {
        "street": "Second Street",
        "zipCode": "ZZ123"
    }
}
```

## Output

```
{
  "name": "Ken",
  "lastName": "Shokida",
  "age": 30,
  "addresses": [
    {
      "street": "Second Street",
      "zipCode": "ZZ123"
    }
  ]
}
```

## Attribute Selector

This matching selector enables you to match any attribute value.

The following example updates an attribute value and changes it to upper case:

## Input

```
<user name="leandro"/>
```

## Source

```
%dw 2.0
output application/json
---
payload update {
    case name at .user.@name -> upper(name)
}
```

## Output

```
<?xml version='1.0' encoding='UTF-8'?>
<user name="LEANDRO"/>
```

## Dynamic Selector

Any selector can incorporate an expression to make the selection dynamic.

The following example dynamically selects and updates the value of a field that is not hardcoded. Such a field can change at runtime. The example uses the variable `theFieldName` to obtain the field with key `name`, and it changes the value of that field to `Shoki`.

## Input

```
{
  "name": "Ken",
  "lastName": "Shokida"
}
```

## Source

```
%dw 2.0
var theFieldName = "name"
output application/json
---
payload update {
  case s at ."$(" + theFieldName + ")" -> "Shoki"
}
```

## Output

```
{
  "name": "Shoki",
  "lastName": "Shokida"
}
```

## Conditional Update

The `update` operator has syntax for writing a conditional update if you want to modify a field under a given condition.

The following example updates the category based on the name of the user:

## Input

```
[{"name": "Ken", "age": 30}, {"name": "Tomo", "age": 70}, {"name": "Kajika", "age": 10}]
```

## Source

```
%dw 2.0
output application/json
---
payload map ((user) ->
    user update {
        case name at .name if(name == "Ken") -> name ++ " (Leandro)"
        case name at .name if(name == "Tomo") -> name ++ " (Christian)"
    }
)
```

## Output

```
[
{
    "name": "Ken (Leandro)",
    "age": 30
},
{
    "name": "Tomo (Christian)",
    "age": 70
},
{
    "name": "Kajika",
    "age": 10
}]
```

## Upserting

The `update` operator enables you to insert a field that is not present already by using the `!` symbol at the end of the selector expression. When the field is not present, the operator binds a `null` value to the variable.

The following example updates each object in the array `myInput`. If the field `name` is present in the object, the example applies the `upper` function to the value of `name`. If the field `name` is not present in the object, the example appends the key-value pair `"name": "JOHN"` to the object.

## Source

```
%dw 2.0
var myInput = [{"lastName": "Doe"}, {"lastName": "Parker", "name": "Peter"}]
output application/json
---
myInput map ((value) ->
    value update {
        case name at .name! -> if(name == null) "JOHN" else upper(name)
    }
)
```

## Output

```
[{
  "lastName": "Doe",
  "name": "JOHN"
},
{
  "lastName": "Parker",
  "name": "PETER"
}]
```

## Sugar Syntax

The `update` operator accepts a default variable named `$` in place of the longer `<variable_name>` at syntax.

The following example binds the value of the expression `.age` to the default variable name `$`, which is used to express the new value. The longer equivalent is shown in comments.

## Input

```
{
  "name": "Ken",
  "lastName": "Shokida",
  "age": 30,
  "address": {
    "street": "Amenabar",
    "zipCode": "AB1234"
  }
}
```

## Source

```
%dw 2.0
output application/json
---
payload update {
    //equivalent expression:
    //case age at .age -> age + 1
    case .age -> $ + 1
    case .address.street -> "First Street"
}
```

## Output

```
{
  "name": "Ken",
  "lastName": "Shokida",
  "age": 31,
  "address": {
    "street": "First Street",
    "zipCode": "AB1234"
  }
}
```

## Metadata Assignment Operator

DataWeave supports the metadata assignment operator `<~>`, which enables you to set the metadata of any value without using the `as` operator. This operator is useful when you want to set the metadata and retain the current value type. If you are coercing the value to a new type, use `as`.

*Introduced in DataWeave 2.5.0. Supported by Mule 4.5 and later.*

The operator syntax is as follows:

```
<valueB> <~> <objectA>
```

The operator assigns `objectA` as metadata to `valueB`. The arrow points to the value to which the metadata is assigned.

Versions prior to Mule 4.5 require the following syntax, for example, to add the `class` metadata field to an object:

```
%dw 2.0
---
{name: "Ken"} as Object {class: "Person"}
```

In Mule 4.5, the metadata assignment operator `<~>` simplifies the syntax:

```
%dw 2.0
---
{name: "Ken"} <~ {class: "Person"}
```

## Metadata

Metadata is any additional information attached to a value that doesn't affect its identity. Given any two values `A` and `B` that are equal, if you attach metadata to `A` and call it `A'`, then `A == A' == B`. So you can define a value that is formed by the value and the metadata attached to it.

## System Properties

DataWeave supports several system properties. To use these properties in application development and deployments, see [System Properties](#).

Property	Description
<code>com.mulesoft.dw.buffered_char_sequence.enabled</code>	<p>Controls whether an internal <code>BufferedCharSequence</code> improvement is enabled or disabled. When this property is set to <code>true</code>, DataWeave does not load large fields (greater than 1.5M) into memory. DataWeave reads these fields by chunk. This feature reduces performance to prevent an out-of-memory error.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>true</code></li> </ul>
<code>com.mulesoft.dw.buffersize</code>	<p>Sets the size (in bytes) of in-memory input and output buffers that DataWeave uses to retain processed inputs and outputs. DataWeave stores payloads that exceed this size in temporary files <code>dw-buffer-index-\${count}.tmp</code> and <code>dw-buffer-output-\${count}.tmp</code>. See <a href="#">Memory Management</a>.</p> <ul style="list-style-type: none"> <li>• Type: <code>Number</code> (in bytes)</li> <li>• Default: <code>8192</code> bytes (8 KB)</li> </ul>
<code>com.mulesoft.dw.coercionexception.verbose</code>	<p>By default, adds to coercion exceptions more information about data that fails to coerce. When this property is set to <code>false</code>, DataWeave does not display the additional metadata.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>true</code></li> </ul>

Property	Description
<code>com.mulesoft.dw.date_minus_back_compatibility</code>	<p>Restores the <a href="#">Add and Subtracting Dates</a> behaviors that changed in 2.3.0 when this property is set to <code>true</code>.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p>Available for language levels: 2.3, 2.4</p> <p>Example:</p> <pre>%dw 2.0 output application/dw ---  2019-10-01  -  2018-09-23 </pre> <ul style="list-style-type: none"> <li>• In DataWeave 2.3.0, the expression returns <a href="#">PT8952H</a>.</li> <li>• In DataWeave 2.2.0, the expression returns <a href="#">P1Y8D</a>.</li> </ul>
<code>com.mulesoft.dw.directbuffer.disable</code>	<p>Controls whether DataWeave uses off-heap memory (the default) or heap memory. DataWeave uses off-heap memory for internal buffering, which can cause issues on systems that have only a small amount of memory. See <a href="#">Memory Management</a>.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul>
<code>com.mulesoft.dw.dump_files</code>	<p><i>Experimental:</i> Dumps the input context and the failing script into a folder when this property is set to <code>true</code>. This behavior enables you to track the failing script and the data that makes the script fail, which is particularly useful for verifying that the received input data is valid. Incorrect scripts often fail when an upstream component generates invalid data.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p><code>com.mulesoft.dw.dump_files</code> is an <i>experimental feature</i> that is subject to change or removal from future versions of DataWeave. See <a href="#">Troubleshooting</a>.</p>

Property	Description
<code>com.mulesoft.dw.dump_folder</code>	<p><i>Experimental:</i> Specifies the path in which to dump files when <code>com.mulesoft.dw.dump_files</code> is set to <code>true</code>.</p> <ul style="list-style-type: none"> <li>• Type: <code>String</code></li> <li>• Default: <code>java.io.tmpdir</code> (system property value)</li> </ul> <p><code>com.mulesoft.dw.dump_folder</code> is an <i>experimental feature</i> that is subject to change or removal from future versions of DataWeave. See <a href="#">Troubleshooting</a>.</p>
<code>com.mulesoft.dw.dumper_fill_stacktrace</code>	<p><i>Experimental:</i> Adds the StackTrace for dumper exceptions when <code>com.mulesoft.dw.dump_files</code> is set to <code>true</code>.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p><code>com.mulesoft.dw.dumper_fill_stacktrace</code> is an <i>experimental feature</i> that is subject to change or removal from future versions of DataWeave. See <a href="#">Troubleshooting</a>.</p>
<code>com.mulesoft.dw.error_value_length</code>	<p>Sets the maximum length of exception messages to display to the user. The message is truncated to the maximum length. This setting is useful for avoiding long exception messages.</p> <ul style="list-style-type: none"> <li>• Type: <code>Number</code></li> <li>• Default: <code>80</code></li> </ul>
<code>com.mulesoft.dw.indexsize</code>	<p>Sets the maximum size (in bytes) of the page in memory that indexed readers use.</p> <ul style="list-style-type: none"> <li>• Type: <code>Number</code></li> <li>• Default: <code>1572864</code></li> </ul>
<code>com.mulesoft.dw.javaSqlDateToDate</code>	<p>When set, <code>java.sql.Date</code> maps to the DataWeave <code>Date</code> type. If <code>false</code>, it maps to <code>DateTime</code>.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p>Available for language levels: 2.4, 2.5</p>

Property	Description
<code>com.mulesoft.dw.max_memory_allocation</code>	<p>Sets the size (in bytes) of each slot in the off-heap memory pool. DataWeave stores payloads that exceed this size in temporary files <code>dw-buffer-input-\${count}.tmp</code> and <code>dw-buffer-output-\${count}.tmp</code>. See <a href="#">Memory Management</a>.</p> <ul style="list-style-type: none"> <li>• Type: <a href="#">Number</a></li> <li>• Default: <a href="#">1572864</a> (1.5 MB)</li> </ul>
<code>com.mulesoft.dw.memory_pool_size</code>	<p>Sets the number of slots in the memory pool. DataWeave buffers use off-heap memory from a pool, up to a defined size (<code>com.mulesoft.dw.memory_pool_size</code> * <code>com.mulesoft.dw.max_memory_allocation</code>). DataWeave allocates the remainder using heap memory. See <a href="#">Memory Management</a>.</p> <ul style="list-style-type: none"> <li>• Type: <a href="#">Number</a></li> <li>• Default: <a href="#">60</a></li> </ul>
<code>com.mulesoft.dw.multipart.defaultContentType</code>	<p>Sets the default Content-Type to use on parts of the <code>multipart/*</code> format when a <a href="#">Content-Type</a> is not specified. See also, the multipart reader property <a href="#">defaultContentType</a>. <i>Introduced in DataWeave 2.3 (2.3.0-20210720) for the August 2021 release of Mule 4.3.0-20210719.</i></p> <ul style="list-style-type: none"> <li>• Type: <a href="#">String</a></li> <li>• Default: <a href="#">application/octet-stream</a></li> </ul>
<code>com.mulesoft.dw.stacksize</code>	<p>Sets the maximum size of the stack. When a function recurses too deeply, DataWeave throws an error, such as Stack Overflow. The maximum size limit is 256.</p> <ul style="list-style-type: none"> <li>• Type: <a href="#">Number</a></li> <li>• Default: <a href="#">256</a></li> </ul>
<code>com.mulesoft.dw.track.cursor.close</code>	<p>When set to <code>true</code>, tracks the stack trace from which the <code>CursorProvider#close()</code> method is called. Use this property for troubleshooting, for example, if <code>CursorProvider#openCursor()</code> is called on a cursor that is already closed.</p> <ul style="list-style-type: none"> <li>• Type: <a href="#">Boolean</a></li> <li>• Default: <a href="#">false</a></li> </ul>

Property	Description
<pre>com.mulesoft.dw.valueSelector.selectsAlwaysFirst</pre>	<p>When set to set to <code>true</code>, returns the first occurrence of an element (even if the element appears more than once). Enabling this behavior degrades performance.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p>The following example illustrates the behavior that is controlled by this property. (Assume that the DataWeave script acts on the XML input.)</p> <p><i>XML input:</i></p> <pre>&lt;root&gt;   &lt;users&gt;     &lt;user&gt;       &lt;lname&gt;chibana&lt;/lname&gt;       &lt;name&gt;Shoki&lt;/name&gt;     &lt;/user&gt;     &lt;user&gt;       &lt;name&gt;Shoki&lt;/name&gt;       &lt;name&gt;Tomo&lt;/name&gt;     &lt;/user&gt;   &lt;/users&gt; &lt;/root&gt;</pre> <p><i>DataWeave script:</i></p> <pre>%dw 2.0 output application/json --- {   shokis: payload.root.users.*user map \$.name }</pre> <ul style="list-style-type: none"> <li>• If <code>com.mulesoft.dw.valueSelector.selectsAlwaysFirst</code> is set to <code>true</code>, the script returns the following output:</li> </ul> <pre>{   "shokis": [     "Shoki",     "Shoki"   ] }</pre> <ul style="list-style-type: none"> <li>• If <code>com.mulesoft.dw.valueSelector.selectsAlwaysFirst</code> is set to <code>false</code>, the script returns the following output:</li> </ul>

Property	Description
<code>com.mulesoft.dw.xml_reader.honourMixedContentStructure</code>	<p>When this property is set to <code>true</code>, DataWeave retains a mixed-content structure instead of grouping text with mixed content into a single text field.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p>Available for language levels: 2.4</p>
<code>com.mulesoft.dw.xml_reader.parseDtD</code>	<p>DataWeave parses a Doctype declaration when this property is set to <code>true</code>.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul> <p>Available for language levels: 2.5</p>
<code>com.mulesoft.dw.xml.supportDTD</code>	<p>Controls whether DTD handling is enabled or disabled. When this property is set to <code>false</code>, DataWeave skips processing of both internal and external subsets. Note that the default for this property changed from <code>true</code> to <code>false</code> in Mule version 4.3.0-20210427, which includes the May, 2021 patch of DataWeave version 2.3.0.</p> <ul style="list-style-type: none"> <li>• Type: <code>Boolean</code></li> <li>• Default: <code>false</code></li> </ul>

## Precedence Rules

DataWeave expressions are compiled in a specific order. The result of a compilation of something at one level can serve as input for expressions in higher levels, but not at lower levels. Before you begin, note that 2.x versions of DataWeave are used by Mule 4 apps. For DataWeave in Mule 3 apps, refer to the [DataWeave version 1.2 documentation](#). For other Mule versions, you can use the version selector in the DataWeave table of contents.

### Order of Compilation

The following table orders operators and functions from first compiled (1) to last compiled (10):

Level	Operator or Function
1	<ul style="list-style-type: none"> <li>• <b>using</b>: For initializing local variables in a DataWeave script.</li> <li>• Unary DataWeave operators at this level: <ul style="list-style-type: none"> <li>◦ <b>.^</b>: Schema selector.</li> <li>◦ <b>.#</b>: Namespace selector.</li> <li>◦ <b>..</b>: Descendants selector.</li> <li>◦ <b>not</b>: Logical operator for negation.</li> <li>◦ <b>.@</b>: All attribute selector, for example, in a case that uses the expression <code>payload.root.a.@</code> to return the attributes and values of the input payload <code>&lt;root&gt;&lt;a attr1="foo" attr2="bar" /&gt;&lt;/root&gt;</code> within the array of objects <code>{ "attr1": "foo", "attr2": "bar" }</code>.</li> </ul> </li> </ul>
2	<ul style="list-style-type: none"> <li>• <b>as</b>: Used for type coercion.</li> </ul>
3	<ul style="list-style-type: none"> <li>• <b>*</b>: Multiply.</li> <li>• <b>/</b>: Divide.</li> </ul>
4	<ul style="list-style-type: none"> <li>• <b>+</b>: Add.</li> <li>• <b>-</b>: Subtract.</li> <li>• <b>&gt;&gt;</b>: Prepend data.</li> <li>• <b>&lt;&lt;</b>: Append data.</li> </ul>
5	<ul style="list-style-type: none"> <li>• <b>&gt;</b>: Greater than or equal to.</li> <li>• <b>=</b>: Equal to.</li> <li>• <b>&lt;</b>: Less than or equal to.</li> <li>• <b>&lt;</b>: Less than.</li> <li>• <b>&gt;</b>: Greater than.</li> <li>• <b>is</b>: Compare type, for example, used in <a href="#">Remove Certain XML Attributes</a>.</li> </ul>
6	<ul style="list-style-type: none"> <li>• <b>!=</b>: Not equal to.</li> <li>• <b>~=</b>: Contains.</li> <li>• <b>==</b>: Equality operator that tries to coerce one value to the type of the other when the types are different.</li> </ul>
7	<ul style="list-style-type: none"> <li>• <b>and</b>: Logical <b>and</b> operator.</li> </ul>
8	<ul style="list-style-type: none"> <li>• <b>or</b>: Logical <b>or</b> operator.</li> </ul>

Level	Operator or Function
9	<ul style="list-style-type: none"> <li>• <b>default</b>: For setting a default value.</li> <li>• <b>case</b> and <b>else</b>: For pattern matching.</li> <li>• DataWeave functions at this level: <ul style="list-style-type: none"> <li>◦ <b>matches</b></li> <li>◦ <b>match</b></li> <li>◦ <b>map</b></li> <li>◦ <b>mapObject</b></li> <li>◦ <b>groupBy</b></li> <li>◦ <b>filter</b></li> </ul> </li> <li>• Binary DataWeave selectors at this level: <ul style="list-style-type: none"> <li>◦ <b>.@keyName</b> and <b>.*@</b>: Binary DataWeave attribute selectors.</li> <li>◦ <b>.*</b>: Multi-value selector.</li> <li>◦ <b>.^</b>: Metadata selector.</li> <li>◦ <b>.&amp;</b>: Key-value pair selector.</li> <li>◦ <b>[?()]</b>: Filter selector.</li> </ul> </li> <li>• <b>.keyName</b>: Value selector.</li> <li>• <b>[&lt;index&gt; to &lt;index&gt;]</b>: Range selector</li> </ul>
10	<ul style="list-style-type: none"> <li>• <b>if else</b>: Conditional expression used for flow control in DataWeave.</li> </ul>

## Order of Chained Function Calls

When functions are chained together in a series, the functions are processed in order from the first to the last function specified in the chain.

For example, the following script acts on the input array of objects defined by the variable **flights**. The script first calls **filter** to return an array that accepts all **price** values that are less than 500. Then it calls **orderBy** to return an array that reorders the objects found in the input from the lowest to the highest **price** value. Finally, it calls **groupBy** to group the input array of objects alphabetically by **toAirport** value.

*DataWeave Script:*

```
%dw 2.0
output application/json
var flights = [
    { "toAirport": "SFO", "price": 550, "airline": "American" },
    { "toAirport": "MUA", "price": 200, "airline": "American" },
    { "toAirport": "SFO", "price": 300, "airline": "American" },
    { "toAirport": "CLE", "price": 600, "airline": "American" },
    { "toAirport": "CLE", "price": 190, "airline": "American" },
    { "toAirport": "SFO", "price": 400, "airline": "American" }
]
---
flights filter $.price < 500 orderBy $.price groupBy $.toAirport
```

The result is an object that contains a collection of key-value pairs:

*Output:*

```
{
    "CLE": [
        {
            "toAirport": "CLE",
            "price": 190,
            "airline": "American"
        }
    ],
    "SFO": [
        {
            "toAirport": "SFO",
            "price": 300,
            "airline": "United"
        },
        {
            "toAirport": "SFO",
            "price": 400,
            "airline": "American"
        }
    ],
    "MUA": [
        {
            "toAirport": "MUA",
            "price": 200,
            "airline": "American"
        }
    ]
}
```

You can force the evaluation order by inserting evaluation parentheses ( and ) into the DataWeave expression. This example returns the same output as the previous example:

*DataWeave Script:*

```
%dw 2.0
output application/json
var flights = [
    { "toAirport": "SFO", "price": 550, "airline": "American" },
    { "toAirport": "MUA", "price": 200, "airline": "American" },
    { "toAirport": "SFO", "price": 300, "airline": "American" },
    { "toAirport": "CLE", "price": 600, "airline": "American" },
    { "toAirport": "CLE", "price": 190, "airline": "American" },
    { "toAirport": "SFO", "price": 400, "airline": "American" }
]
---
(( ( flights filter $.price < 500 ) orderBy $.price ) groupBy $.toAirport )
```

The result is an object that contains a collection of key-value pairs:

*Output:*

```
{
    "CLE": [
        {
            "toAirport": "CLE",
            "price": 190,
            "airline": "American"
        }
    ],
    "SFO": [
        {
            "toAirport": "SFO",
            "price": 300,
            "airline": "United"
        },
        {
            "toAirport": "SFO",
            "price": 400,
            "airline": "American"
        }
    ],
    "MUA": [
        {
            "toAirport": "MUA",
            "price": 200,
            "airline": "American"
        }
    ]
}
```

The order of the function calls is important. For example, the rearranged expression `flights groupBy $.toAirport filter $.price < 500 orderBy $.price` returns an error because `groupBy`

returns an object, and `filter` expects an array. The resulting error is `Expecting Type: Array>T<, but got: {String: Array<{|toA… airline: String|}>}.`

## See Also

- [Core \(dw::Core\) functions](#)
- [DataWeave Operators](#)
- [Selectors](#)
- [Variables](#)
- [Flow Control in DataWeave](#)
- [Pattern Matching in DataWeave](#)
- [Type Coercion with DataWeave](#)

## dw::Core

This module contains core DataWeave functions for data transformations. It is automatically imported into any DataWeave script. For documentation on DataWeave 1.0 functions, see [DataWeave Operators](#).

## Functions

Name	Description
<code>++</code>	Concatenates two values.
<code>&lt;&lt;dw-core-functions-minusminus::,-&gt;</code>	Removes specified values from an input value.
<code>abs</code>	Returns the absolute value of a number.
<code>avg</code>	Returns the average of numbers listed in an array.
<code>ceil</code>	Rounds a number up to the nearest whole number.
<code>contains</code>	Returns <code>true</code> if an input contains a given value, <code>false</code> if not.
<code>daysBetween</code>	Returns the number of days between two dates.
<code>distinctBy</code>	Iterates over the input and returns the unique elements in it.
<code>endsWith</code>	Returns <code>true</code> if a string ends with a provided substring, <code>false</code> if not.
<code>entriesOf</code>	Returns an array of key-value pairs that describe the key, value, and any attributes in the input object.
<code>filter</code>	Iterates over an array and applies an expression that returns matching values.
<code>filterObject</code>	Iterates a list of key-value pairs in an object and applies an expression that returns only matching objects, filtering out the rest from the output.
<code>find</code>	Returns indices of an input that match a specified value.
<code>flatMap</code>	Iterates over each item in an array and flattens the results.

Name	Description
<code>flatten</code>	Turns a set of subarrays (such as <code>[ [1,2,3], [4,5,[6]], [], [null] ]</code> ) into a single, flattened array (such as <code>[ 1, 2, 3, 4, 5, [6], null ]</code> ).
<code>floor</code>	Rounds a number down to the nearest whole number.
<code>groupBy</code>	Returns an object that groups items from an array based on specified criteria, such as an expression or matching selector.
<code>indexOf</code>	Returns the index of the <i>first</i> occurrence of the specified element in this array, or <code>-1</code> if this list does not contain the element.
<code>isBlank</code>	Returns <code>true</code> if the given string is empty (""), completely composed of whitespaces, or <code>null</code> . Otherwise, the function returns <code>false</code> .
<code>isDecimal</code>	Returns <code>true</code> if the given number contains a decimal, <code>false</code> if not.
<code>isEmpty</code>	Returns <code>true</code> if the given input value is empty, <code>false</code> if not.
<code>isEven</code>	Returns <code>true</code> if the number or numeric result of a mathematical operation is even, <code>false</code> if not.
<code>isInteger</code>	Returns <code>true</code> if the given number is an integer (which lacks decimals), <code>false</code> if not.
<code>isLeapYear</code>	Returns <code>true</code> if it receives a date for a leap year, <code>false</code> if not.
<code>isOdd</code>	Returns <code>true</code> if the number or numeric result of a mathematical operation is odd, <code>false</code> if not.
<code>joinBy</code>	Merges an array into a single string value and uses the provided string as a separator between each item in the list.
<code>keysOf</code>	Returns an array of keys from key-value pairs within the input object.
<code>lastIndexOf</code>	Returns the index of the <i>last</i> occurrence of the specified element in a given array or <code>-1</code> if the array does not contain the element.
<code>log</code>	Without changing the value of the input, <code>log</code> returns the input as a system log. So this makes it very simple to debug your code, because any expression or subexpression can be wrapped with <code>log</code> and the result will be printed out without modifying the result of the expression. The output is going to be printed in application/dw format.
<code>lower</code>	Returns the provided string in lowercase characters.
<code>map</code>	Iterates over items in an array and outputs the results into a new array.
<code>mapObject</code>	Iterates over an object using a mapper that acts on keys, values, or indices of that object.
<code>match</code>	Uses a Java regular expression (regex) to match a string and then separates it into capture groups. Returns the results in an array.
<code>matches</code>	Checks if an expression matches the entire input string.
<code>max</code>	Returns the highest <code>Comparable</code> value in an array.
<code>maxBy</code>	Iterates over an array and returns the highest value of <code>Comparable</code> elements from it.

Name	Description
min	Returns the lowest Comparable value in an array.
minBy	Iterates over an array to return the lowest value of comparable elements from it.
mod	Returns the modulo (the remainder after dividing the dividend by the divisor).
namesOf	Returns an array of strings with the names of all the keys within the given object.
now	Returns a DateTime value for the current date and time.
onNull	Executes a callback function if the preceding expression returns a null value and then replaces the null value with the result of the callback.
orderBy	Reorders the elements of an input using criteria that acts on selected elements of that input.
pluck	Useful for mapping an object into an array, pluck iterates over an object and returns an array of keys, values, or indices from the object.
pow	Raises the value of a base number to the specified power.
random	Returns a pseudo-random number greater than or equal to 0.0 and less than 1.0.
randomInt	Returns a pseudo-random whole number from 0 to the specified number (exclusive).
read	Reads a string or binary and returns parsed content.
readUrl	Reads a URL, including a classpath-based URL, and returns parsed content. This function works similar to the read function.
reduce	Applies a reduction expression to the elements in an array.
replace	Performs string replacement.
round	Rounds a number up or down to the nearest whole number.
scan	Returns an array with all of the matches found in an input string.
sizeOf	Returns the number of elements in an array. It returns 0 if the array is empty.
splitBy	Splits a string into a string array based on a value that matches part of that string. It filters out the matching part from the returned array.
sqrt	Returns the square root of a number.
startsWith	Returns true or false depending on whether the input string starts with a matching prefix.
sum	Returns the sum of numeric values in an array.
then	This function works as a pipe that passes the value returned from the preceding expression to the next (a callback) only if the value returned by the preceding expression is not null.

Name	Description
to	Returns a range with the specified boundaries.
trim	Removes any blank spaces from the beginning and end of a string.
typeOf	Returns the basic data type of a value.
unzip	Performs the opposite of <code>zip</code> . It takes an array of arrays as input.
upper	Returns the provided string in uppercase characters.
uuid	Returns a v4 UUID using random numbers as the source.
valuesOf	Returns an array of the values from key-value pairs in an object.
with	Helper function that specifies a replacement element. This function is used with <code>replace</code> , <code>update</code> or <code>mask</code> to perform data substitutions.
write	Writes a value as a string or binary in a supported format.
xsiType	Creates a <code>xsi:type</code> type attribute. This method returns an object, so it must be used with dynamic attributes.
zip	Merges elements from two arrays into an array of arrays.

## Types

- [Core Types](#)

## Namespaces

- [Core Namespaces](#)

## Annotations

- [Core Annotations](#)

`++`

`++<S, T>(source: Array<S>, with: Array<T>): Array<S | T>`

Concatenates two values.

This version of `++` concatenates the elements of two arrays into a new array. Other versions act on strings, objects, and the various date and time formats that DataWeave supports.

If the two arrays contain different types of elements, the resulting array is all of `S` type elements of `Array<S>` followed by all the `T` type elements of `Array<T>`. Either of the arrays can also have mixed-type elements. Also note that the arrays can contain any supported data type.

## Parameters

Name	Description
<code>source</code>	The source array.

Name	Description
with	The array to concatenate with the source array.

### Example

The example concatenates an `Array<Number>` with an `Array<String>`. Notice that it outputs the result as the value of a JSON object.

### Source

```
%dw 2.0
output application/json
---
{ "result" : [0, 1, 2] ++ ["a", "b", "c"] }
```

### Output

```
{ "result": [0, 1, 2, "a", "b", "c"] }
```

### Example

### Source

```
%dw 2.0
output application/json
---
{ "a" : [0, 1, true, "my string"] ++ [2, [3,4,5], {"a": 6}] }
```

### Output

```
{ "a": [0, 1, true, "my string", 2, [3, 4, 5], { "a": 6}] }
```

## `++(source: String, with: String): String`

Concatenates the characters of two strings.

Strings are treated as arrays of characters, so the `++` operator concatenates the characters of each string as if they were arrays of single-character string.

### Parameters

Name	Description
source	The source string.
with	The string to concatenate with the source string.

## Example

This example concatenates two strings. Here, `Mule` is treated as `Array<String> ["M", "u", "l", "e"]`. Notice that the example outputs the result `MuleSoft` as the value of a JSON object.

## Source

```
%dw 2.0
output application/json
---
{ "name" : "Mule" ++ "Soft" }
```

## Output

```
{ "name": "MuleSoft" }
```

`++<T <: Object, Q <: Object>(source: T, with: Q): T & Q`

Concatenates two objects and returns one flattened object.

The `++` operator extracts all the key-values pairs from each object, then combines them together into one result object.

## Parameters

Name	Description
<code>source</code>	The source object.
<code>with</code>	The object to concatenate with the source object.

## Example

This example concatenates two objects and transforms them to XML. Notice that it flattens the array of objects `{aa: "a", bb: "b"}` into separate XML elements and that the output uses the keys of the specified JSON objects as XML elements and the values of those objects as XML values.

## Source

```
%dw 2.0
output application/xml
---
{ concat : {aa: "a", bb: "b"} ++ {cc: "c"} }
```

## Output

```

<?xml version="1.0" encoding="UTF-8"?>
<concat>
  <aa>a</aa>
  <bb>b</bb>
  <cc>c</cc>
</concat>

```

## **++(date: Date, time: LocalTime): LocalDateTime**

Appends a [LocalTime](#) with a [Date](#) to return a [LocalDateTime](#) value.

[Date](#) and [LocalTime](#) instances are written in standard Java notation, surrounded by pipe (|) symbols. The result is a [LocalDateTime](#) object in the standard Java format. Note that the order in which the two objects are concatenated is irrelevant, so logically, `Date LocalTime` produces the same result as `LocalTime Date.`

### Parameters

Name	Description
<code>date</code>	A <a href="#">Date</a> .
<code>time</code>	A <a href="#">LocalTime</a> , a time format without a time zone.

### Example

This example concatenates a [Date](#) and [LocalTime](#) object to return a [LocalDateTime](#).

### Source

```

%dw 2.0
output application/json
---
{ "LocalDateTime" : (|2017-10-01| ++ |23:57:59|) }

```

### Output

```
{ "LocalDateTime": "2017-10-01T23:57:59" }
```

## **++(time: LocalTime, date: Date): LocalDateTime**

Appends a [LocalTime](#) with a [Date](#) to return a [LocalDateTime](#).

Note that the order in which the two objects are concatenated is irrelevant, so logically, `LocalTime Date` produces the same result as `Date LocalTime.`

### Parameters

Name	Description
time	A <a href="#">LocalTime</a> , a time format without a time zone.
date	A <a href="#">Date</a> .

## Example

This example concatenates [LocalTime](#) and [Date](#) objects to return a [LocalDateTime](#).

## Source

```
%dw 2.0
output application/json
---
{ "LocalDateTime" : (|23:57:59| ++ |2003-10-01|) }
```

## Output

```
{ "LocalDateTime": "2017-10-01T23:57:59" }
```

## ++(date: Date, time: Time): DateTime

Appends a [Date](#) to a [Time](#) in order to return a [DateTime](#).

Note that the order in which the two objects are concatenated is irrelevant, so logically, [Date + Time](#) produces the same result as [Time + Date](#).

## Parameters

Name	Description
date	A <a href="#">Date</a> .
time	A <a href="#">Time</a> , a time format that can include a time zone ( <a href="#">Z</a> or <a href="#">HH:mm</a> ).

## Example

This example concatenates [Date](#) and [Time](#) objects to return a [DateTime](#).

## Source

```
%dw 2.0
output application/json
---
[ |2017-10-01| ++ |23:57:59-03:00|, |2017-10-01| ++ |23:57:59Z| ]
```

## Output

```
[ "2017-10-01T23:57:59-03:00", "2017-10-01T23:57:59Z" ]
```

## **++(time: Time, date: Date): DateTime**

Appends a **Date** to a **Time** object to return a **DateTime**.

Note that the order in which the two objects are concatenated is irrelevant, so logically, **Date + Time** produces the same result as a **Time + Date**.

### **Parameters**

Name	Description
time	A <b>Time</b> , a time format that can include a time zone (Z or HH:mm).
date	A <b>Date</b> .

### **Example**

This example concatenates a **Date** with a **Time** to output a **DateTime**. Notice that the inputs are surrounded by pipes (|).

### **Source**

```
%dw 2.0
output application/json
---
|2018-11-30| ++ |23:57:59+01:00|
```

### **Output**

```
"2018-11-30T23:57:59+01:00"
```

### **Example**

This example concatenates **Time** and **Date** objects to return **DateTime** objects. Note that the first **LocalTime object is coerced to a 'Time**. Notice that the order of the date and time inputs does not change the order of the output **DateTime**.

### **Source**

```
%dw 2.0
output application/json
---
{
    "DateTime1" : (|23:57:59| as Time) ++ |2017-10-01|,
    "DateTime2" : |23:57:59Z| ++ |2017-10-01|,
    "DateTime3" : |2017-10-01| ++ |23:57:59+02:00|
}
```

## Output

```
{
    "DateTime1": "2017-10-01T23:57:59Z",
    "DateTime2": "2017-10-01T23:57:59Z",
    "DateTime3": "2017-10-01T23:57:59+02:00"
}
```

## **++(date: Date, timezone: TimeZone): DateTime**

Appends a **TimeZone** to a **Date** type value and returns a **DateTime** result.

### Parameters

Name	Description
<b>date</b>	A <b>Date</b> .
<b>timezone</b>	A <b>TimeZone</b> ( <b>Z</b> or <b>HH:mm</b> ).

### Example

This example concatenates **Date** and **TimeZone (-03:00)** to return a **DateTime**. Note the local time in the **DateTime** is **00:00:00** (midnight).

### Source

```
%dw 2.0
output application/json
---
{ "DateTime" : (|2017-10-01| ++ |-03:00|) }
```

## Output

```
{ "DateTime": "2017-10-01T00:00:00-03:00" }
```

## **++(timezone: TimeZone, date: Date): DateTime**

Appends a [Date](#) to a [TimeZone](#) in order to return a [DateTime](#).

#### Parameters

Name	Description
<code>date</code>	A <a href="#">Date</a> .
<code>timezone</code>	A <a href="#">TimeZone</a> (Z or HH:mm).

#### Example

This example concatenates [TimeZone \(-03:00\)](#) and [Date](#) to return a [DateTime](#). Note the local time in the [DateTime](#) is **00:00:00** (midnight).

#### Source

```
%dw 2.0
output application/json
---
{ "DateTime" : |-03:00| ++ |2017-10-01| }
```

#### Output

```
{ "DateTime": "2017-10-01T00:00:00-03:00" }
```

### **++(dateTime: LocalDateTime, timezone: TimeZone): DateTime**

Appends a [TimeZone](#) to a [LocalDateTime](#) in order to return a [DateTime](#).

#### Parameters

Name	Description
<code>dateTime</code>	A <a href="#">LocalDateTime</a> , a date and time without a time zone.
<code>timezone</code>	A <a href="#">TimeZone</a> (Z or HH:mm).

#### Example

This example concatenates [LocalDateTime](#) and [TimeZone \(-03:00\)](#) to return a [DateTime](#).

#### Source

```
%dw 2.0
output application/json
---
{ "DateTime" : (|2003-10-01T23:57:59| ++ |-03:00|) }
```

#### Output

```
{ "DateTime": "2003-10-01T23:57:59-03:00" }
```

## ++(timezone: TimeZone, datetime: LocalDateTime): DateTime

Appends a [LocalDateTime](#) to a [TimeZone](#) in order to return a [DateTime](#).

### Parameters

Name	Description
dateTime	A <a href="#">LocalDateTime</a> , a date and time without a time zone.
timezone	A <a href="#">TimeZone</a> (Z or HH:mm).

### Example

This example concatenates [TimeZone](#) (-03:00) and [LocalDateTime](#) to return a [DateTime](#).

### Source

```
%dw 2.0
output application/json
---
{ "TimeZone" : (|-03:00| ++ |2003-10-01T23:57:59|) }
```

### Output

```
{ "TimeZone": "2003-10-01T23:57:59-03:00" }
```

## ++(time: LocalTime, timezone: TimeZone): Time

Appends a [TimeZone](#) to a [LocalTime](#) in order to return a [Time](#).

### Parameters

Name	Description
time	A <a href="#">LocalTime</a> , time format without a time zone.
timezone	A <a href="#">TimeZone</a> (Z or HH:mm).

### Example

This example concatenates [LocalTime](#) and [TimeZone](#) (-03:00) to return a [Time](#). Note that the output returns `:00` for the unspecified seconds.

### Source

```
%dw 2.0
output application/json
---
{ "Time" : (|23:57| ++ |-03:00|) }
```

## Output

```
{ "Time": "23:57:00-03:00" }
```

### **++(timezone: TimeZone, time: LocalTime): Time**

Appends a [LocalTime](#) to a [TimeZone](#) in order to return a [Time](#).

#### Parameters

Name	Description
<a href="#">time</a>	A <a href="#">LocalTime</a> , a time format without a time zone.
<a href="#">timezone</a>	A <a href="#">TimeZone</a> (Z or HH:mm).

#### Example

This example concatenates [TimeZone \(-03:00\)](#) and [LocalTime](#) to return a [Time](#). Note that the output returns `:00` for the unspecified seconds.

#### Source

```
%dw 2.0
output application/json
---
{ "Time" : (|-03:00| ++ |23:57|) }
```

## Output

```
{
  "Time": "23:57:00-03:00"
}
```

### --<S>(source: Array<S>, toRemove: Array<Any>): Array<S>

Removes specified values from an input value.

This version of [--](#) removes all instances of the specified items from an array. Other versions act on objects, strings, and the various date and time formats that are supported by DataWeave.

Name	Description
source	The array containing items to remove.
toRemove	Items to remove from the source array.

### Example

This example removes specified items from an array. Specifically, it removes all instances of the items listed in the array on the right side of `--` from the array on the left side of the function, leaving `[0]` as the result.

### Source

```
%dw 2.0
output application/json
---
{ "a" : [0, 1, 1, 2] -- [1,2] }
```

### Output

```
{ "a": [0] }
```

`--<K, V>(source: { (K)?: V }, toRemove: Object): { (K)?: V }`

Removes specified key-value pairs from an object.

### Parameters

Name	Description
source	The source object (an <code>Object</code> type).
toRemove	Object that contains the key-value pairs to remove from the source object.

### Example

This example removes a key-value pair from the source object.

### Source

```
%dw 2.0
output application/json
---
{ "hello" : "world", "name" : "DW" } -- { "hello" : "world" }
```

### Output

```
{ "name": "DW" }
```

### --(source: Object, keys: Array<String>)

Removes all key-value pairs from the source object that match the specified search key.

#### Parameters

Name	Description
source	The source object (an <code>Object</code> type).
toRemove	An array of keys to specify the key-value pairs to remove from the source object.

#### Example

This example removes two key-value pairs from the source object.

#### Source

```
%dw 2.0
output application/json
---
{ "yes" : "no", "good" : "bad", "old" : "new" } -- ["yes", "old"]
```

#### Output

```
{ "good": "bad" }
```

### --(source: Object, keys: Array<Key>)

Removes specified key-value pairs from an object.

#### Parameters

Name	Description
source	The source object (an <code>Object</code> type).
keys	A keys for the key-value pairs to remove from the source object.

#### Example

This example specifies the key-value pair to remove from the source object.

#### Source

```
%dw 2.0
output application/json
---
{ "hello" : "world", "name" : "DW" } -- ["hello" as Key]
```

## Output

```
{ "name": "DW" }
```

### --(source: Null, keys: Any)

Helper function that enables `--` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## abs

### abs(number: Number): Number

Returns the absolute value of a number.

#### Parameters

Name	Description
number	The number to evaluate.

#### Example

This example returns the absolute value of the specified numbers.

#### Source

```
%dw 2.0
output application/json
---
[ abs(-2), abs(2.5), abs(-3.4), abs(3) ]
```

## Output

```
[ 2, 2.5, 3.4, 3 ]
```

## avg

### avg(values: Array<Number>): Number

Returns the average of numbers listed in an array.

An array that is empty or that contains a non-numeric value results in an error.

#### Parameters

Name	Description
values	The input array of numbers.

#### Example

This example returns the average of multiple arrays.

#### Source

```
%dw 2.0
output application/json
---
{ a: avg([1, 1000]), b: avg([1, 2, 3]) }
```

#### Output

```
{ "a": 500.5, "b": 2.0 }
```

## ceil

### ceil(number: Number): Number

Rounds a number up to the nearest whole number.

#### Parameters

Name	Description
number	The number to round.

#### Example

This example rounds numbers up to the nearest whole numbers. Notice that `2.1` rounds up to `3`.

#### Source

```
%dw 2.0
output application/json
---
[ ceil(1.5), ceil(2.1), ceil(3) ]
```

## Output

```
[ 2, 3, 3 ]
```

## contains

**contains<T>(@StreamCapable items: Array<T>, element: Any): Boolean**

Returns **true** if an input contains a given value, **false** if not.

This version of **contains** accepts an array as input. Other versions accept a string and can use another string or regular expression to determine whether there is a match.

### Parameters

Name	Description
items	The input array.
elements	Element to find in the array. Can be any supported data type.

### Example

This example finds that **2** is in the input array, so it returns **true**.

### Source

```
%dw 2.0
output application/json
---
[ 1, 2, 3, 4 ] contains(2)
```

## Output

```
true
```

### Example

This example indicates whether the input array contains "3".

### Source

```
%dw 2.0
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 }
```

### contains(text: String, toSearch: String): Boolean

Indicates whether a string contains a given substring. Returns **true** or **false**.

#### Parameters

Name	Description
text	An input string (a <b>String</b> ).
toSearch	The substring (a <b>String</b> ) to find in the input string.

#### Example

This example finds "mule" in the input string "mulesoft", so it returns **true**.

#### Source

```
%dw 2.0
output application/json
---
"mulesoft" contains("mule")
```

## Output

```
true
```

## Example

This example finds that the substring "me" is in "some string", so it returns **true**.

## Source

```
%dw 2.0
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 }
```

## contains(text: String, matcher: Regex): Boolean

Returns **true** if a string contains a match to a regular expression, **false** if not.

### Parameters

Name	Description
text	An input string.
matcher	A Java regular expression for matching characters in the input <b>text</b> .

## Example

This example checks for any of the letters **e** through **g** in the input **mulesoft**, so it returns **true**.

## Source

```
%dw 2.0
output application/json
---
contains("mulesoft", /[e-g]/)
```

## Output

```
true
```

## Example

This example finds a match to `/s[t|p]rin/` within "A very long string", so it returns `true`. The `[t|p]` in the regex means `t` or `p`.

## Source

```
%dw 2.0
output application/json
---
ContainsString: payload.root.mystring contains /s[t|p]rin/
```

## Input

```
<?xml version="1.0" encoding="UTF-8"?>
<root><mystring>A very long string</mystring></root>
```

## Output

```
{ "ContainsString": true }
```

## contains(text: Null, matcher: Any): false

Helper function that enables `contains` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## daysBetween

### daysBetween(from: Date, to: Date): Number

Returns the number of days between two dates.

#### Parameters

Name	Description
from	From date (a <code>Date</code> type).
to	To date (a <code>Date</code> type). Note that if the <code>to</code> date is <i>earlier</i> than the <code>from</code> date, the function returns a negative number equal to the number of days between the two dates.

## Example

This example returns the number of days between the specified dates.

## Source

```
%dw 2.0
output application/json
---
{ days : daysBetween('2016-10-01T23:57:59-03:00', '2017-10-01T23:57:59-03:00') }
```

## Output

```
{ "days" : 365 }
```

## distinctBy

`distinctBy<T>(@StreamCapable items: Array<T>, criteria: (item: T, index: Number) -> Any): Array<T>`

Iterates over the input and returns the unique elements in it.

DataWeave uses the result of the provided lambda as the uniqueness criteria.

This version of `distinctBy` finds unique values in an array. Other versions act on an object and handle a `null` value.

## Parameters

Name	Description
items	The array to evaluate.
criteria	The criteria used to select an <code>item</code> and/or <code>index</code> from the array.

## Example

This example inputs an array that contains duplicate numbers and returns an array with unique numbers from that input. Note that you can write the same expression using an anonymous parameter for the values: `[0, 1, 2, 3, 3, 2, 1, 4] distinctBy $`

## Source

```
%dw 2.0
output application/json
---
[0, 1, 2, 3, 3, 2, 1, 4] distinctBy (value) -> { "unique" : value }
```

## Output

```
[ 0, 1, 2, 3, 4]
```

## Example

This example removes duplicates of "[Kurt Cagle](#)" from an array.

## Source

```
%dw 2.0
output application/json
var record = {
    "title": "XQuery Kick Start",
    "author": [
        "James McGovern",
        "Per Bothner",
        "Kurt Cagle",
        "James Linn",
        "Kurt Cagle",
        "Kurt Cagle",
        "Kurt Cagle",
        "Vaidyanathan Nagarajan"
    ],
    "year": "2000"
}
---
{
    "book" : {
        "title" : record.title,
        "year" : record.year,
        "authors" : record.author distinctBy $ 
    }
}
```

## Output

```
{
  "book": {
    "title": "XQuery Kick Start",
    "year": "2000",
    "authors": [
      "James McGovern",
      "Per Bothner",
      "Kurt Cagle",
      "James Linn",
      "Vaidyanathan Nagarajan"
    ]
  }
}
```

## distinctBy<K, V>(object: { (K)?: V }, criteria: (value: V, key: K) -> Any): Object

Removes duplicate key-value pairs from an object.

### Parameters

Name	Description
object	The object from which to remove the key-value pairs.
criteria	The <code>key</code> and/or <code>value</code> used to identify the key-value pairs to remove.

### Example

This example inputs an object that contains duplicate key-value pairs and returns an object with key-value pairs from that input. Notice that the keys (`a` and `A`) are not treated with case sensitivity, but the values (`b` and `B`) are. Also note that you can write the same expression using an anonymous parameter for the values: `{a : "b", a : "b", A : "b", a : "B"} distinctBy $`

### Source

```
%dw 2.0
output application/json
---
{a : "b", a : "b", A : "b", a : "B"} distinctBy (value) -> { "unique" : value }
```

### Output

```
{ "a": "b", "a": "B" }
```

### Example

This example removes duplicates (`<author>James McGovern</author>`) from `<book/>`.

## Source

```
%dw 2.0
output application/xml
---
{
    book : {
        title : payload.book.title,
        authors: payload.book.&author distinctBy $}
}
```

## Input

```
<book>
    <title> "XQuery Kick Start"</title>
    <author>James Linn</author>
    <author>Per Bothner</author>
    <author>James McGovern</author>
    <author>James McGovern</author>
    <author>James McGovern</author>
</book>
```

## Output

```
<book>
    <title> "XQuery Kick Start"</title>
    <authors>
        <author>James Linn</author>
        <author>Per Bothner</author>
        <author>James McGovern</author>
    </authors>
</book>
```

**distinctBy(@StreamCapable items: Null, criteria: (item: Nothing, index: Nothing) -> Any): Null**

Helper function that enables **distinctBy** to work with a **null** value.

## endsWith

**endsWith(text: String, suffix: String): Boolean**

Returns **true** if a string ends with a provided substring, **false** if not.

## Parameters

Name	Description
text	The input string (a <a href="#">String</a> ).
suffix	The suffix string to find at the end of the input string.

## Example

This example finds "no" (but not "to") at the end of "Mariano".

## Source

```
%dw 2.0
output application/json
---
[ "Mariano" endsWith "no", "Mariano" endsWith "to" ]
```

## Output

```
[ true, false ]
```

## endsWith(text: Null, suffix: Any): false

Helper function that enables [endsWith](#) to work with a [null](#) value.

*Introduced in DataWeave version 2.4.0.*

## entriesOf

### entriesOf<T <: Object>(obj: T): Array<{| key: Key, value: Any, attributes: Object |}>

Returns an array of key-value pairs that describe the key, value, and any attributes in the input object.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
obj	The object to describe.

## Example

This example returns the key, value, and attributes from the object specified in the variable [myVar](#). The object is the XML input to the [read](#) function.

## Source

```
%dw 2.0
var myVar = read('<xml attr="x"><a>true</a><b>1</b></xml>', 'application/xml')
output application/json
---
{ "entriesOf" : entriesOf(myVar) }
```

## Output

```
{
  "entriesOf": [
    {
      "key": "xml",
      "value": {
        "a": "true",
        "b": "1"
      },
      "attributes": {
        "attr": "x"
      }
    }
  ]
}
```

## entriesOf(obj: Null): Null

Helper function that enables `entriesOf` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## filter

**filter<T>(@StreamCapable items: Array<T>, criteria: (item: T, index: Number) -> Boolean): Array<T>**

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

The expression must return `true` or `false`. If the expression returns `true` for a value or index in the array, the value gets captured in the output array. If it returns `false` for a value or index in the array, that item gets filtered out of the output. If there are no matches, the output array will be empty.

### Parameters

Name	Description
<code>items</code>	The array to filter.
<code>criteria</code>	Boolean expression that selects an <code>item</code> and/or <code>index</code> .

## Example

This example returns an array of values in the array that are greater than 2.

## Source

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

## Output

```
[9,3,4,5]
```

## Example

This example returns an array of all the users with age bigger or equal to 30. The script accesses data of each element from within the lambda expression.

## Source

```
%dw 2.0
---
[{"name": "Mariano", "age": 37}, {"name": "Shoki", "age": 30}, {"name": "Tomo", "age": 25}, {"name": "Ana", "age": 29}]
    filter ((value, index) -> value.age >= 30)
```

## Output

```
[
  {
    "name": "Mariano",
    "age": 37
  },
  {
    "name": "Shoki",
    "age": 30
  }
]
```

## Example

This example returns an array of all items found at an index (\$\$) greater than 1 where the value of the element is less than 5. Notice that it is using anonymous parameters as selectors instead of using named parameters in an anonymous function.

## Source

```
%dw 2.0
output application/json
---
[9, 2, 3, 4, 5] filter (($$ > 1) and ($ < 5))
```

## Output

```
[3,4]
```

## Example

This example reads a JSON array that contains objects with `user` and `error` keys, and uses the `filter` function to return only the objects in which the `error` key is `null`.

## Source

```
%dw 2.0
output application/json

var users = [
  {
    "user": {
      "name": "123",
      "lastName": "Smith"
    },
    "error": "That name doesn't exists"
  },
  {
    "user": {
      "name": "John",
      "lastName": "Johnson"
    },
    "error": null
  }
]
---
users filter ((item, index) -> item.error == null)
```

## Output

```
[  
 {  
   "user": {  
     "name": "John",  
     "lastName": "Johnson"  
   },  
   "error": null  
 }  
]
```

### Example

This example reads a JSON array and uses the **filter** function to extract the phone numbers that are active.

### Input

```
{  
  "Id": "118400110000000517",  
  "marketCode": "US",  
  "languageCode": "en-US",  
  "profile": {  
    "base": {  
      "username": "TheMule",  
      "activeInd": "R",  
      "phone": [  
        {  
          "activeInd": "Y",  
          "type": "mobile",  
          "primaryInd": "Y",  
          "number": "230678123"  
        },  
        {  
          "activeInd": "N",  
          "type": "mobile",  
          "primaryInd": "N",  
          "number": ""  
        },  
        {  
          "activeInd": "Y",  
          "type": "mobile",  
          "primaryInd": "Y",  
          "number": "154896523"  
        }  
      ]  
    }  
  }  
}
```

## Source

```
%dw 2.0
output application/json
---
{
    id: payload.Id,
    markCode: payload.marketCode,
    languageCode: payload.languageCode,
    username: payload.profile.base.username,
    phoneNumber: (payload.profile.base.phone filter ($.activeInd == "Y" and
$.primaryInd== "Y")).number default []
}
```

## Output

```
{
    "id": "118400110000000517",
    "markCode": "US",
    "languageCode": "en-US",
    "username": "TheMule",
    "phoneNumber": [
        "230678123"
        "154896523"
    ]
}
```

**filter(@StreamCapable text: String, criteria: (character: String, index: Number) -> Boolean): String**

Iterates over a string and applies an expression that returns matching values.

The expression must return **true** or **false**. If the expression returns **true** for a character or index in the array, the character gets captured in the output string. If it returns **false** for a character or index in the array, that character gets filtered out of the output. If there are no matches, the output string will be empty.

## Parameters

Name	Description
text	The text to filter.
criteria	The criteria to use.

## Example

This example shows how **filter** can be used to remove all characters in odd positions.

## Source

```
%dw 2.0
output application/json
---
"hello world" filter ($$ mod 2) == 0
```

## Output

```
"hlowrd"
```

## filter(@StreamCapable value: Null, criteria: (item: Nothing, index: Nothing) -> Any): Null

Helper function that enables `filter` to work with a `null` value.

## filterObject

### filterObject<K, V>(@StreamCapable value: { (K)?: V }, criteria: (value: V, key: K, index: Number) -> Boolean): { (K)?: V }

Iterates a list of key-value pairs in an object and applies an expression that returns only matching objects, filtering out the rest from the output.

The expression must return `true` or `false`. If the expression returns `true` for a key, value, or index of an object, the object gets captured in the output. If it returns `false` for any of them, the object gets filtered out of the output. If there are no matches, the output array will be empty.

### Parameters

Name	Description
<code>value</code>	The source object to evaluate.
<code>criteria</code>	Boolean expression that selects a <code>value</code> , <code>key</code> , or <code>index</code> of the object.

### Example

This example outputs an object if its value equals "apple".

### Source

```
%dw 2.0
output application/json
---
{"a" : "apple", "b" : "banana"} filterObject ((value) -> value == "apple")
```

## Output

```
{ "a": "apple" }
```

## Example

This example only outputs an object if the key starts with "letter". The DataWeave `startsWith` function returns `true` or `false`. Note that you can use the anonymous parameter for the key to write the expression `((value, key) → key startsWith "letter")`:`( $$ startsWith "letter")``

## Source

```
%dw 2.0
output application/json
---
{"letter1": "a", "letter2": "b", "id": 1} filterObject ((value, key) -> key startsWith
"letter")
```

## Output

```
{ "letter1": "a", "letter2": "b" }
```

## Example

This example only outputs an object if the index of the object in the array is less than 1, which is always true of the first object. Note that you can use the anonymous parameter for the index to write the expression `((value, key, index) → index < 1)`:`( $$$ < 1)`

## Source

```
%dw 2.0
output application/json
---
{ "1": "a", "2": "b", "3": "c"} filterObject ((value, key, index) -> index < 1)
```

## Output

```
{ "1": "a" }
```

## Example

This example outputs an object that contains only the values that are not `null` in the input JSON object.

## Source

```
%dw 2.0
output application/json
var myObject = {
    str1 : "String 1",
    str2 : "String 2",
    str3 : null,
    str4 : "String 4",
}
---
myObject filterObject $ != null
```

## Output

```
{
  "str1": "String 1",
  "str2": "String 2",
  "str4": "String 4"
}
```

**filterObject(value: Null, criteria: (value: Nothing, key: Nothing, index: Nothing) -> Any): Null**

Helper function that enables **filterObject** to work with a **null** value.

## find

**find<T>(@StreamCapable() elements: Array<T>, elementToFind: Any): Array<Number>**

Returns indices of an input that match a specified value.

This version of the function returns indices of an array. Others return indices of a string.

### Parameters

Name	Description
<b>elements</b>	An array with elements of any type.
<b>elementToFind</b>	Value to find in the input array.

### Example

This example finds the index of an element in a string array.

### Source

```
%dw 2.0
output application/json
---
["Bond", "James", "Bond"] find "Bond"
```

## Output

```
[0,2]
```

### **find(@StreamCapable() text: String, matcher: Regex): Array<Array<Number>>**

Returns the indices in the text that match the specified regular expression (regex), followed by the capture groups.

The first element in each resulting sub-array is the index in the text that matches the regex, and the next ones are the capture groups in the regex (if present).

Note: To retrieve parts of the text that match a regex. use the [scan](#) function.

## Parameters

Name	Description
text	A string.
matcher	A Java regular expression for matching characters in the <a href="#">text</a> .

## Example

This example finds the beginning and ending indices of words that contain ea

## Source

```
%dw 2.0
output application/json
---
"I heart DataWeave" find /\w*ea\w*(\b)/
```

## Output

```
[ [2,7], [8,17] ]
```

### **find(@StreamCapable() text: String, textToFind: String): Array<Number>**

Lists indices where the specified characters of a string are present.

## Parameters

Name	Description
text	A source string.
textToFind	The string to find in the source string.

## Example

This example lists the indices of "a" found in "aabccdbce".

## Source

```
%dw 2.0
output application/json
---
"aabccdbce" find "a"
```

## Output

```
[0,1]
```

## find(@StreamCapable() text: Null, textToFind: Any): Array<Nothing>

Helper function that enables `find` to work with a `null` value.

## flatMap

### flatMap<T, R>(@StreamCapable items: Array<T>, mapper: (item: T, index: Number) -> Array<R>): Array<R>

Iterates over each item in an array and flattens the results.

Instead of returning an array of arrays (as `map` does when you iterate over the values within an input like `[ [1,2], [3,4] ]`), `flatMap` returns a flattened array that looks like this: `[1, 2, 3, 4]`. `flatMap` is similar to `flatten`, but `flatten` only acts on the values of the arrays, while `flatMap` can act on values and indices of items in the array.

## Parameters

Name	Description
items	The array to map.
mapper	Expression or selector for an <code>item</code> and/or <code>index</code> in the array to flatten.

## Example

This example returns an array containing each value in order. Though it names the optional `index` parameter in its anonymous function `(value, index) -> value`, it does not use `index` as a selector for the output, so it is possible to write the anonymous function using `(value) -> value`. You can also use

an anonymous parameter for the value to write the example like this: [ [3,5], [0.9,5.5] ] flatMap \$. Note that this example produces the same result as `flatten([ [3,5], [0.9,5.5] ])`, which uses `flatten`.

## Source

```
%dw 2.0
output application/json
---
[ [3,5], [0.9,5.5] ] flatMap (value, index) -> value
```

## Output

```
[ 3, 5, 0.9, 5.5]
```

**flatMap<T, R>(@StreamCapable value: Null, mapper: (item: Nothing, index: Nothing) -> Any): Null**

Helper function that enables `flatMap` to work with a `null` value.

## flatten

**flatten<T, Q>(@StreamCapable items: Array<Array<T> | Q>): Array<T | Q>**

Turns a set of subarrays (such as [ [1,2,3], [4,5,[6]], [], [null] ]) into a single, flattened array (such as [ 1, 2, 3, 4, 5, [6], null ]).

Note that it flattens only the first level of subarrays and omits empty subarrays.

## Parameters

Name	Description
<code>items</code>	The input array of arrays made up of any supported types.

## Example

This example defines three arrays of numbers, creates another array containing those three arrays, and then uses the flatten function to convert the array of arrays into a single array with all values.

## Source

```
%dw 2.0
output application/json
var array1 = [1,2,3]
var array2 = [4,5,6]
var array3 = [7,8,9]
var arrayOfArrays = [array1, array2, array3]
---
flatten(arrayOfArrays)
```

## Output

```
[ 1,2,3,4,5,6,7,8,9 ]
```

## Example

This example returns a single array from nested arrays of objects.

## Source

```
%dw 2.0
var myData =
{ user : [
  {
    group : "dev",
    myarray : [
      { name : "Shoki", id : 5678 },
      { name : "Mariano", id : 9123 }
    ]
  },
  {
    group : "test",
    myarray : [
      { name : "Sai", id : 2001 },
      { name : "Peter", id : 2002 }
    ]
  }
]
output application/json
---
flatten(myData.user.myarray)
```

## Output

```
[  
  {  
    "name": "Shoki",  
    "id": 5678  
  },  
  {  
    "name": "Mariano",  
    "id": 9123  
  },  
  {  
    "name": "Sai",  
    "id": 2001  
  },  
  {  
    "name": "Peter",  
    "id": 2002  
  }  
]
```

Note that if you use `myData.user.myarray` to select the array of objects in `myarray`, instead of using `flatten(myData.user.myarray)`, the output is a nested array of objects:

```
[  
  [  
    {  
      "name": "Shoki",  
      "id": 5678  
    },  
    {  
      "name": "Mariano",  
      "id": 9123  
    }  
  ]  
]
```

## **flatten(@StreamCapable value: Null): Null**

Helper function that enables `flatten` to work with a `null` value.

## **floor**

### **floor(number: Number): Number**

Rounds a number down to the nearest whole number.

#### **Parameters**

Name	Description
number	The number to evaluate.

## Example

This example rounds numbers down to the nearest whole numbers. Notice that `1.5` rounds down to `1`.

## Source

```
%dw 2.0
output application/json
---
[ floor(1.5), floor(2.2), floor(3) ]
```

## Output

```
[ 1, 2, 3]
```

## groupBy

`groupBy<T, R>(items: Array<T>, criteria: (item: T, index: Number) -> R): {} | (R): Array<T> | {}`

Returns an object that groups items from an array based on specified criteria, such as an expression or matching selector.

This version of `groupBy` groups the elements of an array using the `criteria` function. Other versions act on objects and handle null values.

## Parameters

Name	Description
items	The array to group.
criteria	Expression providing the criteria by which to group the items in the array.

## Example

This example groups items from the input array `["a", "b", "c"]` by their indices. Notice that it returns the numeric indices as strings and that items (or values) of the array are returned as arrays, in this case, with a single item each. The items in the array are grouped based on an anonymous function `(item, index) → index` that uses named parameters `item` and `index`. Note that you can produce the same result using the anonymous parameter `$$` to identify the indices of the array like this: `["a", "b", "c"] groupBy $$`

## Source

```
%dw 2.0
output application/json
---
["a","b","c"] groupBy (item, index) -> index
```

## Output

```
{ "2": [ "c" ], "1": [ "b" ], "0": [ "a" ] }
```

## Example

This example groups the elements of an array based on the language field. Notice that it uses the `item.language` selector to specify the grouping criteria. So the resulting object uses the "language" values ("Scala" and "Java") from the input to group the output. Also notice that the output places the each input object in an array.

## Source

```
%dw 2.0
var myArray = [
  { "name": "Foo", "language": "Java" },
  { "name": "Bar", "language": "Scala" },
  { "name": "FooBar", "language": "Java" }
]
output application/json
---
myArray groupBy (item) -> item.language
```

## Output

```
{
  "Scala": [
    { "name": "Bar", "language": "Scala" }
  ],
  "Java": [
    { "name": "Foo", "language": "Java" },
    { "name": "FooBar", "language": "Java" }
  ]
}
```

## Example

This example uses `groupBy "myLabels"` to return an object where `"`myLabels`" is the key, and an array of selected values (`["Open New", "Zoom In", "Zoom Out", "Original View"]`) is the value. It uses the selectors (`myVar.menu.items.*label`) to create that array. Notice that the selectors retain all values where "`label`" is the key but filter out values where "`id`" is the key.

## Source

```
%dw 2.0
var myVar = { menu: {
    header: "Move Items",
    items: [
        {"id": "internal"},
        {"id": "left", "label": "Move Left"},
        {"id": "right", "label": "Move Right"},
        {"id": "up", "label": "Move Up"},
        {"id": "down", "label": "Move Down"}
    ]
}}
output application/json
---
(myVar.menu.items.*label groupBy "myLabels")
```

## Output

```
{ "myLabels": [ "Move Left", "Move Right", "Move Up", "Move Down" ] }
```

### groupBy<R>(text: String, criteria: (character: String, index: Number) -> R): { (R): String }

Returns an object that groups characters from a string based on specified criteria, such as an expression or matching selector.

This version of `groupBy` groups the elements of an array using the `criteria` function. Other versions act on objects and handle `null` values.

#### Parameters

Name	Description
<code>text</code>	The string to group by.
<code>criteria</code>	The criteria to use.

## Example

This example shows how you can use `groupBy` to split a string into vowels and non-vowels.

## Source

```
%dw 2.0
output application/json
---
"hello world!" groupBy (not isEmpty($ find /[aeiou]/))
```

## Output

```
{
  "false": "hll wrld!",
  "true": "eo"
}
```

## groupBy<K, V, R>(object: { (K)?: V }, criteria: (value: V, key: K) -> R): { (R): { (K)?: V } }

Groups elements of an object based on criteria that the `groupBy` uses to iterate over elements in the input.

### Parameters

Name	Description
<code>object</code>	The object containing objects to group.
<code>criteria</code>	The grouping criteria to apply to elements in the input object, such as a <code>key</code> and/or <code>value</code> of the object to use for grouping.

### Example

This example groups objects within an array of objects using the anonymous parameter `$` for the value of each key in the input objects. It applies the DataWeave `upper` function to those values. In the output, these values become upper-case keys. Note that you can also write the same example using a named parameter for the within an anonymous function like this: `{ "a" : "b", "c" : "d"} groupBy (value) -> upper(value)`

### Source

```
%dw 2.0
output application/json
---
{ "a" : "b", "c" : "d"} groupBy upper($)
```

### Output

```
{ "D": { "c": "d" }, "B": { "a": "b" } }
```

### Example

This example uses `groupBy "costs"` to produce a JSON object from an XML object where `"costs"` is the key, and the selected values of the XML element `prices` becomes the JSON value (`{ "price": "9.99", "price": "10.99" }`).

### Source

```
%dw 2.0
var myRead =
read("<prices><price>9.99</price><price>10.99</price></prices>","application/xml")
output application/json
---
myRead.prices groupBy "costs"
```

## Output

```
{ "costs" : { "price": "9.99", "price": "10.99" } }
```

## groupBy(value: Null, criteria: (Nothing, Nothing) -> Any): Null

Helper function that enables `groupBy` to work with a `null` value.

## indexOf

### indexOf(array: Array, value: Any): Number

Returns the index of the *first* occurrence of the specified element in this array, or `-1` if this list does not contain the element.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>array</code>	The array of elements to search.
<code>value</code>	The value to search.

#### Example

This example shows how `indexOf` behaves given different inputs.

#### Source

```
%dw 2.0
output application/json
---
{
  present: ["a","b","c","d"] indexOf "c",
  notPresent: ["x","w","x"] indexOf "c",
  presentMoreThanOnce: ["a","b","c","c"] indexOf "c",
}
```

## Output

```
{  
    "present": 2,  
    "notPresent": -1,  
    "presentMoreThanOnce": 2  
}
```

## indexOf(theString: String, search: String): Number

Returns the index of the **first** occurrence of the specified String in this String.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
theString	The string to search.
search	The string to find within <code>theString</code> .

### Example

This example shows how the `indexOf` behaves under different inputs.

### Source

```
%dw 2.0  
output application/json  
---  
{  
    present: "abcd" indexOf "c",  
    notPresent: "xyz" indexOf "c",  
    presentMoreThanOnce: "abcdc" indexOf "c",  
}
```

### Output

```
{  
    "present": 2,  
    "notPresent": -1,  
    "presentMoreThanOnce": 2  
}
```

## indexOf(array: Null, value: Any): Number

Helper method to make `indexOf` null friendly

*Introduced in DataWeave version 2.4.0.*

## isBlank

**isBlank(text: String | Null): Boolean**

Returns `true` if the given string is empty (""), completely composed of whitespaces, or `null`. Otherwise, the function returns `false`.

### Parameters

Name	Description
<code>text</code>	An input string to evaluate.

### Example

This example indicates whether the given values are blank. It also uses the `not` and `!` operators to check that a value is not blank. The `!` operator is supported starting in Dataweave 2.2.0. Use `!` only in Mule 4.2 and later versions.

### Source

```
%dw 2.0
output application/json
var someString = "something"
var nullString = null
---
{
    // checking if the string is blank
    "emptyString" : isBlank(""),
    "stringWithSpaces" : isBlank("      "),
    "textString" : isBlank(someString),
    "somePayloadValue" : isBlank(payload.nonExistingValue),
    "nullString" : isBlank(nullString),

    // checking if the string is not blank
    "notEmptyTextString" : not isBlank(" 1234"),
    "notEmptyTextStringTwo" : ! isBlank("")
}
```

### Output

```
{  
    "emptyString": true,  
    "stringWithSpaces": true,  
    "textString": false,  
    "somePayloadValue": true,  
    "nullString": true,  
    "notEmptyTextString": true,  
    "notEmptyTextStringTwo": false  
}
```

## isDecimal

### isDecimal(number: Number): Boolean

Returns `true` if the given number contains a decimal, `false` if not.

#### Parameters

Name	Description
<code>number</code>	The number to evaluate.

#### Example

This example indicates whether a number has a decimal. Note that numbers within strings get coerced to numbers.

#### Source

```
%dw 2.0  
output application/json  
---  
[ isDecimal(1.1), isDecimal(1), isDecimal("1.1") ]
```

#### Output

```
[ true, false, true ]
```

## isEmpty

### isEmpty(elements: Array<Any>): Boolean

Returns `true` if the given input value is empty, `false` if not.

This version of `isEmpty` acts on an array. Other versions act on a string or object, and handle null values.

## Parameters

Name	Description
elements	The input array to evaluate.

## Example

This example indicates whether the input array is empty.

## Source

```
%dw 2.0
output application/json
---
[ isEmpty([]), isEmpty([1]) ]
```

## Output

```
[ true, false ]
```

## isEmpty(value: String): Boolean

Returns `true` if the input string is empty, `false` if not.

## Parameters

Name	Description
value	A string to evaluate.

## Example

This example indicates whether the input strings are empty.

## Source

```
%dw 2.0
output application/json
---
[ isEmpty(""), isEmpty("DataWeave") ]
```

## Output

```
[ true, false ]
```

## isEmpty(value: Object): Boolean

Returns **true** if the given object is empty, **false** if not.

#### Parameters

Name	Description
<b>value</b>	The object to evaluate.

#### Example

This example indicates whether the input objects are empty.

#### Source

```
%dw 2.0
output application/json
---
[ isEmpty({}), isEmpty({name: "DataWeave"}) ]
```

#### Output

```
[ true, false ]
```

### isEmpty(value: Null): true

Returns **true** if the input is **null**.

#### Parameters

Name	Description
<b>value</b>	<b>null</b> is the value in this case.

#### Example

This example indicates whether the input is **null**.

#### Source

```
%dw 2.0
output application/json
---
{ "nullValue" : isEmpty(null) }
```

#### Output

```
{ "nullValue": true }
```

## isEven

### isEven(number: Number): Boolean

Returns `true` if the number or numeric result of a mathematical operation is even, `false` if not.

#### Parameters

Name	Description
<code>number</code>	The number to evaluate.

#### Example

This example indicates whether the numbers and result of an operation are even.

#### Source

```
%dw 2.0
output application/json
---
{ "isEven" : [ isEven(0), isEven(1), isEven(1+1) ] }
```

#### Output

```
{ "isEven" : [ true, false, true ] }
```

## isInteger

### isInteger(number: Number): Boolean

Returns `true` if the given number is an integer (which lacks decimals), `false` if not.

#### Parameters

Name	Description
<code>number</code>	The number to evaluate.

#### Example

This example indicates whether the input is an integer for different values. Note numbers within strings get coerced to numbers.

#### Source

```
%dw 2.0
output application/json
---
[isInteger(1), isInteger(2.0), isInteger(2.2), isInteger("1")]
```

## Output

```
[ true, true, false, true ]
```

## isLeapYear

### isLeapYear(dateTime: DateTime): Boolean

Returns `true` if it receives a date for a leap year, `false` if not.

This version of `leapYear` acts on a `DateTime` type. Other versions act on the other date and time formats that DataWeave supports.

#### Parameters

Name	Description
<code>dateTime</code>	The <code>DateTime</code> value to evaluate.

#### Example

This example indicates whether the input is a leap year.

#### Source

```
%dw 2.0
output application/json
---
[ isLeapYear(|2016-10-01T23:57:59|), isLeapYear(|2017-10-01T23:57:59|) ]
```

## Output

```
[ true, false ]
```

### isLeapYear(date: Date): Boolean

Returns `true` if the input `Date` is a leap year, 'false' if not.

#### Parameters

Name	Description
date	The <a href="#">Date</a> value to evaluate.

## Example

This example indicates whether the input is a leap year.

## Source

```
%dw 2.0
output application/json
---
[ isLeapYear(|2016-10-01|), isLeapYear(|2017-10-01|) ]
```

## Output

```
[ true, false ]
```

## isLeapYear(datetime: LocalDateTime): Boolean

Returns [true](#) if the input local date-time is a leap year, 'false' if not.

## Parameters

Name	Description
datetime	A <a href="#">LocalDateTime</a> value to evaluate.

## Example

This example indicates whether the input is a leap year. It uses a [map](#) function to iterate through the array of its [LocalDateTime](#) values, applies the [isLeapYear](#) to those values, returning the results in an array.

## Source

```
%dw 2.0
output application/json
---
[ |2016-10-01T23:57:59-03:00|, |2016-10-01T23:57:59Z| ] map isLeapYear($)
```

## Output

```
[ true, true ]
```

## isOdd

### isOdd(number: Number): Boolean

Returns `true` if the number or numeric result of a mathematical operation is odd, `false` if not.

#### Parameters

Name	Description
<code>number</code>	A number to evaluate.

#### Example

This example indicates whether the numbers are odd.

#### Source

```
%dw 2.0
output application/json
---
{ "isOdd" : [ isOdd(0), isOdd(1), isOdd(2+2) ] }
```

#### Output

```
{ "isOdd": [ false, true, false ] }
```

## joinBy

### joinBy(elements: Array<StringCoerceable>, separator: String): String

Merges an array into a single string value and uses the provided string as a separator between each item in the list.

Note that `joinBy` performs the opposite task of `splitBy`.

#### Parameters

Name	Description
<code>elements</code>	The input array.
<code>separator</code>	A <code>String</code> used to join elements in the list.

#### Example

This example joins the elements with a hyphen (-).

#### Source

```
%dw 2.0
output application/json
---
{ "hyphenate" : ["a","b","c"] joinBy "-" }
```

## Output

```
{ "hyphenate": "a-b-c" }
```

## joinBy(n: Null, separator: Any): Null

Helper function that enables `joinBy` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## keysOf

### keysOf<K, V>(obj: { (K)?: V }): Array<K>

Returns an array of keys from key-value pairs within the input object.

The returned keys belong to the Key type. To return each key as a string, you can use `namesOf`, instead.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
<code>object</code>	The object to evaluate.

### Example

This example returns the keys from the key-value pairs within the input object.

### Source

```
%dw 2.0
output application/json
---
{ "keysOf" : keysOf({ "a" : true, "b" : 1}) }
```

## Output

```
{ "keysOf" : ["a","b"] }
```

## Example

This example illustrates a difference between `keysOf` and `namesOf`. Notice that `keysOf` retains the attributes (`name` and `lastName`) and namespaces (`xmlns`) from the XML input, while `namesOf` returns `null` for them because it does not retain them.

## Source

```
%dw 2.0
var myVar = read('<users xmlns="http://test.com">
    <user name="Mariano" lastName="Achaval"/>
    <user name="Stacey" lastName="Duke"/>
</users>', 'application/xml')
output application/json
---
{ keysOfExample: flatten([keysOf(myVar.users) map $.#,
                           keysOf(myVar.users) map $.@])
}
++
{ namesOfExample: flatten([namesOf(myVar.users) map $.#,
                           namesOf(myVar.users) map $.@])
}
```

## Output

```
{
  "keysOfExample": [
    "http://test.com",
    "http://test.com",
    {
      "name": "Mariano",
      "lastName": "Achaval"
    },
    {
      "name": "Stacey",
      "lastName": "Duke"
    }
  ],
  "namesOfExample": [
    null,
    null,
    null,
    null
  ]
}
```

## keysOf(obj: Null): Null

Helper function that enables `keysOf` to work with a `null` value.

Introduced in DataWeave version 2.4.0.

## lastIndexOf

### lastIndexOf(array: Array, value: Any): Number

Returns the index of the *last* occurrence of the specified element in a given array or **-1** if the array does not contain the element.

Introduced in DataWeave version 2.4.0.

#### Parameters

Name	Description
array	The array of elements to search.
value	The value to search.

#### Example

This example shows how **indexOf** behaves given different inputs.

#### Source

```
%dw 2.0
output application/json
---
{
  present: ["a","b","c","d"] lastIndexOf "c",
  notPresent: ["x","w","x"] lastIndexOf "c",
  presentMoreThanOnce: ["a","b","c","c"] lastIndexOf "c",
}
```

#### Output

```
{
  "present": 2,
  "notPresent": -1,
  "presentMoreThanOnce": 3
}
```

### lastIndexOf(array: String, value: String): Number

Takes a string as input and returns the index of the *last* occurrence of a given search string within the input. The function returns **-1** if the search string is not present in the input.

Introduced in DataWeave version 2.4.0.

#### Parameters

Name	Description
string	The string to search.
value	A string value to search for within the input string.

## Example

This example shows how the `indexOf` behaves given different inputs.

## Source

```
%dw 2.0
output application/json
---
{
  present: "abcd" lastIndexOf "c",
  notPresent: "xyz" lastIndexOf "c",
  presentMoreThanOnce: "abcdc" lastIndexOf "c",
}
```

## Output

```
{
  "present": 2,
  "notPresent": -1,
  "presentMoreThanOnce": 4
}
```

## lastIndexOf(array: Null, value: Any): Number

Helper function that enables `lastIndexOf` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## log

### log<T>(prefix: String = "", value: T): T

Without changing the value of the input, `log` returns the input as a system log. So this makes it very simple to debug your code, because any expression or subexpression can be wrapped with `log` and the result will be printed out without modifying the result of the expression. The output is going to be printed in application/dw format.

The prefix parameter is optional and allows to easily find the log output.

Use this function to help with debugging DataWeave scripts. A Mule app outputs the results through the `DefaultLoggingService`, which you can see in the Studio console.

## Parameters

Name	Description
prefix	An optional string that typically describes the log.
value	The value to log.

## Example

This example logs the specified message. Note that the `DefaultLoggingService` in a Mule app that is running in Studio returns the message `WARNING - "Houston, we have a problem,"` adding the dash - between the prefix and value. The Logger component's `LoggerMessageProcessor` returns the input string `"Houston, we have a problem."`, without the `WARNING` prefix.

## Source

```
%dw 2.0
output application/json
---
log("WARNING", "Houston, we have a problem")
```

## Output

### Console Output

```
"WARNING - Houston, we have a problem"
```

### Expression Output

```
"Houston, we have a problem"
```

## Example

This example shows how to log the result of expression `myUser.user` without modifying the original expression `myUser.user.friend.name`.

## Source

```
%dw 2.0
output application/json

var myUser = {user: {friend: {name: "Shoki"}, id: 1, name: "Tomo"}, accountId: "leansh" }
---
log("User", myUser.user).friend.name
```

## Output

## Console output

```
User - {  
    friend: {  
        name: "Shoki"  
    },  
    id: 1,  
    name: "Tomo"  
}
```

## Expression Output

```
"Shoki"
```

## lower

### lower(text: String): String

Returns the provided string in lowercase characters.

#### Parameters

Name	Description
text	The input string.

#### Example

This example converts uppercase characters to lower-case.

#### Source

```
%dw 2.0  
output application/json  
---  
{ "name" : lower("MULESOFT") }
```

#### Output

```
{ "name": "mulesoft" }
```

### lower(value: Null): Null

Helper function that enables `lower` to work with a `null` value.

## map

**map<T, R>(@StreamCapable items: Array<T>, mapper: (item: T, index: Number) -> R): Array<R>**

Iterates over items in an array and outputs the results into a new array.

### Parameters

Name	Description
items	The array to map.
mapper	Expression or selector used to act on each item and optionally, each index of that item.

### Example

This example iterates over an input array (`["jose", "pedro", "mateo"]`) to produce an array of DataWeave objects. The anonymous function (`value, index) -> {index: value}`) maps each item in the input to an object. As `{index: value}` shows, each index from the input array becomes a key for an output object, and each value of the input array becomes the value of that object.

### Source

```
%dw 2.0
output application/json
---
["jose", "pedro", "mateo"] map (value, index) -> { (index) : value}
```

### Output

```
[ { "0": "jose" }, { "1": "pedro" }, { "2": "mateo" } ]
```

### Example

This example iterates over the input array (`['a', 'b', 'c']`) using an anonymous function that acts on the items and indices of the input. For each item in the input array, it concatenates the `index + 1` (`index plus 1`) with an underscore (`_`), and the corresponding `value` to return the array, `[ "1_a", "2_b", "3_c" ]`.

### Source

```
%dw 2.0
output application/json
---
['a', 'b', 'c'] map ((value, index) -> (index + 1) ++ '_' ++ value)
```

## Output

```
[ "1_a", "2_b", "3_c" ]
```

## Example

If the parameters of the `mapper` function are not named, the index can be referenced with `$$`, and the value with `$`. This example iterates over each item in the input array `['joe', 'pete', 'matt']` and returns an array of objects where the index is selected as the key. The value of each item in the array is selected as the value of the returned object. Note that the quotes around `$$` are necessary to convert the numeric keys to strings.

## Source

```
%dw 2.0
output application/json
---
['joe', 'pete', 'matt'] map ( $$ : $)
```

## Output

```
[
  { "0": "joe" },
  { "1": "pete" },
  { "2": "matt" }
]
```

## Example

This example iterates over a list of objects and transform the values into CSV. Each of these objects represent a CSV row. The `map` operation generates an object with `age` and `address` for each entry in the list. `$` represents the implicit variable under iteration.

## Source

```
%dw 2.0
output application/csv
---
[{
  "age": 14 ,
  "name": "Claire"
}, {
  "age": 56,
  "name": "Max"
}, {
  "age": 89,
  "name": "John"
}] map {
  age: $.age,
  name: $.name
}
```

## Output

```
age,name
14,Claire
56,Max
89,John
```

## map(@StreamCapable value: Null, mapper: (item: Nothing, index: Nothing) -> Any): Null

Helper function that enables `map` to work with a `null` value.

## mapObject

### mapObject<K, V>(@StreamCapable object: { (K)?: V }, mapper: (value: V, key: K, index: Number) -> Object): Object

Iterates over an object using a mapper that acts on keys, values, or indices of that object.

#### Parameters

Name	Description
<code>object</code>	The object to map.
<code>mapper</code>	Expression or selector that provides the <code>key</code> , <code>value</code> , or <code>index</code> used for mapping the specified object into an output object.

#### Example

This example iterates over the input `{ "a":"b", "c":"d"}` and uses the anonymous mapper function `((value,key,index) → { (index) : { (value):key} })` to invert the keys and values in each specified object and to return the indices of the objects as keys. The mapper uses named parameters to

identify the keys, values, and indices of the input object. Note that you can write the same expression using anonymous parameters, like this: `{"a":"b","c":"d"} mapObject { ($$$) : { ($):$$$} }`

## Source

```
%dw 2.0
output application/json
---
{"a":"b","c":"d"} mapObject (value,key,index) -> { (index) : { (value):key} }
```

## Output

```
{ "0": { "b": "a" }, "1": { "d": "c" } }
```

## Example

This example increases each price by 5 and formats the numbers to always include 2 decimals.

## Source

```
%dw 2.0
output application/xml
---
{
  prices: payload.prices mapObject (value, key) -> {
    (key): (value + 5) as Number {format: "##.00"}
  }
}
```

## Input

```
<?xml version='1.0' encoding='UTF-8'?>
<prices>
  <basic>9.99</basic>
  <premium>53</premium>
  <vip>398.99</vip>
</prices>
```

## Output

```
<?xml version='1.0' encoding='UTF-8'?>
<prices>
  <basic>14.99</basic>
  <premium>58.00</premium>
  <vip>403.99</vip>
</prices>
```

## mapObject(value: Null, mapper: (value: Nothing, key: Nothing, index: Nothing) -> Any): Null

Helper function that enables `mapObject` to work with a `null` value.

### Example

Using the previous example, you can test that if the input of the `mapObject` is `null`, the output result is `null` as well. In XML `null` values are written as empty tags. You can change these values by using the writer property `writeNilOnNull=true`.

### Input

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

### Output

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

## match

### match(text: String, matcher: Regex): Array<String>

Uses a Java regular expression (regex) to match a string and then separates it into capture groups. Returns the results in an array.

Note that you can use `match` for pattern matching expressions that include [case statements](#).

### Parameters

Name	Description
<code>text</code>	A string.
<code>matcher</code>	A Java regex for matching characters in the <code>text</code> .

### Example

In this example, the regex matches the input email address and contains two capture groups within parentheses (located before and after the @). The result is an array of elements: The first matching the entire regex, the second matching the initial capture group () in the the regex, the third matching the last capture group ([a-z]).

### Source

```
%dw 2.0
output application/json
---
"me@mulesoft.com" match(/([a-z]*)@([a-z]*).com/)
```

### Output

```
[
  "me@mulesoft.com",
  "me",
  "mulesoft"
]
```

### Example

This example outputs matches to values in an array that end in 4. It uses `flatMap` to iterate over and flatten the list.

### Source

```
%dw 2.0
var a = '192.88.99.0/24'
var b = '192.168.0.0/16'
var c = '192.175.48.0/24'
output application/json
---
[ a, b, c ] flatMap ( $ match(/.*[$4]/) )
```

### Output

```
[ "192.88.99.0/24", "192.175.48.0/24" ]
```

## match(text: Null, matcher: Any): Null

Helper function that enables `match` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## matches

**matches(text: String, matcher: Regex): Boolean**

Checks if an expression matches the entire input string.

For use cases where you need to output or conditionally process the matched value, see [Pattern Matching in DataWeave](#).

### Parameters

Name	Description
text	The input string.
matcher	A Java regular expression for matching characters in the string.

### Example

This example indicates whether the regular expression matches the input text. Note that you can also use the `matches(text,matcher)` notation (for example, `matches("admin123", /a.*\d+/)`).

### Source

```
%dw 2.0
output application/json
---
[ ("admin123" matches /a.*\d+/, ("admin123" matches /^b.+/) ]
```

### Output

```
[ true, false ]
```

**matches(text: Null, matcher: Any): false**

Helper function that enables `matches` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## max

**max<T <: Comparable>(@StreamCapable values: Array<T>): T | Null**

Returns the highest `Comparable` value in an array.

The items must be of the same type, or the function throws an error. The function returns `null` if the array is empty.

### Parameters

Name	Description
values	The input array. The elements in the array can be any supported type.

## Example

This example returns the maximum value of each input array.

## Source

```
%dw 2.0
output application/json
---
{ a: max([1, 1000]), b: max([1, 2, 3]), c: max([1.5, 2.5, 3.5]) }
```

## Output

```
{ "a": 1000, "b": 3, "c": 3.5 }
```

## maxBy

**maxBy<T>(@StreamCapable array: Array<T>, criteria: (item: T) -> Comparable): T | Null**

Iterates over an array and returns the highest value of **Comparable** elements from it.

The items must be of the same type. **maxBy** throws an error if they are not, and the function returns **null** if the array is empty.

## Parameters

Name	Description
array	The input array.
criteria	Expression for selecting an item from the array, where the item is a <b>Number</b> , <b>Boolean</b> , <b>DateTime</b> , <b>LocalDateTime</b> , <b>Date</b> , <b>LocalTime</b> , <b>Time</b> , or <b>TimeZone</b> data type. Can be referenced with <b>\$</b> .

## Example

This example returns the greatest numeric value within objects (key-value pairs) in an array. Notice that it uses **item.a** to select the value of the object. You can also write the same expression like this, using an anonymous parameter: `[ { "a" : 1 }, { "a" : 3 }, { "a" : 2 } ] maxBy $.a`

## Source

```
%dw 2.0
output application/json
---
[ { "a" : 1 }, { "a" : 3 }, { "a" : 2 } ] maxBy ((item) -> item.a)
```

## Output

```
{ "a" : 3 }
```

## Example

This example gets the latest `DateTime`, `Date`, and `Time` from inputs defined in the variables `myDateTime1` and `myDateTime2`. It also shows that the function returns null on an empty array.

## Source

```
%dw 2.0
var myDateTime1 = "2017-10-01T22:57:59-03:00"
var myDateTime2 = "2018-10-01T23:57:59-03:00"
output application/json
---
{
    myMaxBy: {
        byDateTime: [ myDateTime1, myDateTime2 ] maxBy ((item) -> item),
        byDate: [ myDateTime1 as Date, myDateTime2 as Date ] maxBy ((item) -> item),
        byTime: [ myDateTime1 as Time, myDateTime2 as Time ] maxBy ((item) -> item),
        emptyArray: [] maxBy ((item) -> item)
    }
}
```

## Output

```
{
    "myMaxBy": {
        "byDateTime": "2018-10-01T23:57:59-03:00",
        "byDate": "2018-10-01",
        "byTime": "23:57:59-03:00",
        "emptyArray": null
    }
}
```

## min

`min<T <: Comparable>(@StreamCapable values: Array<T>): T | Null`

Returns the lowest `Comparable` value in an array.

The items must be of the same type or `min` throws an error. The function returns `null` if the array is empty.

#### Parameters

Name	Description
<code>values</code>	The input array. The elements in the array can be any supported type.

#### Example

This example returns the lowest numeric value of each input array.

#### Source

```
%dw 2.0
output application/json
---
{ a: min([1, 1000]), b: min([1, 2, 3]), c: min([1.5, 2.5, 3.5]) }
```

#### Output

```
{ "a": 1, "b": 1, "c": 1.5 }
```

## minBy

`minBy<T>(@StreamCapable array: Array<T>, criteria: (item: T) -> Comparable): T | Null`

Iterates over an array to return the lowest value of comparable elements from it.

The items need to be of the same type. `minBy` returns an error if they are not, and it returns null when the array is empty.

#### Parameters

Name	Description
<code>item</code>	Element in the input array (of type <code>Number</code> , <code>Boolean</code> , <code>DateTime</code> , <code>LocalDateTime</code> , <code>Date</code> , <code>LocalTime</code> , <code>Time</code> , or <code>TimeZone</code> ). Can be referenced with <code>\$</code> .

#### Example

This example returns the lowest numeric value within objects (key-value pairs) in an array. Notice that it uses `item.a` to select the value of the object. You can also write the same expression like this, using an anonymous parameter: `[ { "a" : 1 }, { "a" : 3 }, { "a" : 2 } ] minBy $.a`

#### Source

```
%dw 2.0
output application/json
---
[ { "a" : 1 }, { "a" : 2 }, { "a" : 3 } ] minBy (item) -> item.a
```

## Output

```
{ "a" : 1 }
```

## Example

This example gets the latest `DateTime`, `Date`, and `Time` from inputs defined in the variables `myDateTime1` and `myDateTime2`. It also shows that the function returns null on an empty array.

## Source

```
%dw 2.0
var myDateTime1 = "2017-10-01T22:57:59-03:00"
var myDateTime2 = "2018-10-01T23:57:59-03:00"
output application/json
---
{
    myMinBy: {
        byDateTime: [ myDateTime1, myDateTime2 ] minBy ((item) -> item),
        byDate: [ myDateTime1 as Date, myDateTime2 as Date ] minBy ((item) -> item),
        byTime: [ myDateTime1 as Time, myDateTime2 as Time ] minBy ((item) -> item),
        aBoolean: [ true, false, (0 > 1), (1 > 0) ] minBy $,
        emptyArray: [] minBy ((item) -> item)
    }
}
```

## Output

```
{
    "myMinBy": {
        "byDateTime": "2017-10-01T22:57:59-03:00",
        "byDate": "2017-10-01",
        "byTime": "22:57:59-03:00",
        "aBoolean": false,
        "emptyArray": null
    }
}
```

## mod

**mod(dividend: Number, divisor: Number): Number**

Returns the modulo (the remainder after dividing the **dividend** by the **divisor**).

#### Parameters

Name	Description
<b>dividend</b>	The number that serves as the dividend for the operation.
<b>divisor</b>	The number that serves as the divisor for the operation.

#### Example

This example returns the modulo of the input values. Note that you can also use the `mod(dividend, divisor)` notation (for example, `mod(3, 2)` to return `1`).

#### Source

```
%dw 2.0
output application/json
---
[ (3 mod 2), (4 mod 2), (2.2 mod 2) ]
```

#### Output

```
[ 1, 0, 0.2]
```

## namesOf

**namesOf(obj: Object): Array<String>**

Returns an array of strings with the names of all the keys within the given object.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
<b>obj</b>	The object to evaluate.

#### Example

This example returns the keys from the key-value pairs within the input object.

#### Source

```
%dw 2.0
output application/json
---
{ "namesOf" : namesOf({ "a" : true, "b" : 1}) }
```

## Output

```
{ "namesOf" : ["a","b"] }
```

### namesOf(obj: Null): Null

Helper function that enables `namesOf` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## now

### now(): DateTime

Returns a `DateTime` value for the current date and time.

#### Example

This example uses `now()` to return the current date and time as a `DateTime` value. It also shows how to return a date and time in a specific time zone. Java 8 time zones are supported.

#### Source

```
%dw 2.0
output application/json
---
{
  nowCalled: now(),
  nowCalledSpecificTimeZone: now() >> "America/New_York"
}
```

## Output

```
{
  "nowCalled": "2019-08-26T13:32:10.64-07:00",
  "nowCalledSpecificTimeZone": "2019-08-26T16:32:10.643-04:00"
}
```

#### Example

This example shows uses of the `now()` function with valid selectors. It also shows how to get the

epoch time with `now()` as `Number`. For additional examples, see [Date and Time \(dw::Core Types\)](#).

## Source

```
%dw 2.0
output application/json
---
{
  now: now(),
  epochTime : now() as Number,
  nanoseconds: now().nanoseconds,
  milliseconds: now().milliseconds,
  seconds: now().seconds,
  minutes: now().minutes,
  hour: now().hour,
  day: now().day,
  month: now().month,
  year: now().year,
  quarter: now().quarter,
  dayOfWeek: now().dayOfWeek,
  dayOfYear: now().dayOfYear,
  offsetSeconds: now().offsetSeconds,
  formattedDate: now() as String {format: "y-MM-dd"},
  formattedTime: now() as String {format: "hh:mm:ss"}
}
```

## Output

```
{
  "now": "2019-06-18T16:55:46.678-07:00",
  "epochTime": 1560902146,
  "nanoseconds": 678000000,
  "milliseconds": 678,
  "seconds": 46,
  "minutes": 55,
  "hour": 16,
  "day": 18,
  "month": 6,
  "year": 2019,
  "quarter": 2,
  "dayOfWeek": 2,
  "dayOfYear": 169,
  "offsetSeconds": -25200,
  "formattedDate": "2019-06-18",
  "formattedTime": "04:55:46"
}
```

## onNull

**onNull<R>(previous: Null, callback: () -> R): R**

Executes a callback function if the preceding expression returns a `null` value and then replaces the `null` value with the result of the callback.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>previous</code>	The value of the preceding expression.
<code>callback</code>	Callback that generates a new value if <code>previous</code> returns <code>null</code> .

### Example

This example shows how `onNull` behaves when it receives a `null` value.

### Source

```
%dw 2.0
output application/json
---
{
    "onNull": []
        reduce ((item, accumulator) -> item ++ accumulator)
        then ((result) -> sizeOf(result))
        onNull "Empty Text"
}
```

### Output

```
{
    "onNull": "Empty Text"
}
```

**onNull<T>(previous: T, callback: () -> Any): T**

Helper function that enables `onNull` to work with a *non-null* value.

*Introduced in DataWeave version 2.4.0.*

## orderBy

**orderBy<K, V, R, O <: { (K)?: V }>(object: O, criteria: (value: V, key: K) -> R): O**

Reorders the elements of an input using criteria that acts on selected elements of that input.

This version of `orderBy` takes an object as input. Other versions act on an input array or handle a `null` value.

Note that you can reference the index with the anonymous parameter `$$` and the value with `$`.

## Parameters

Name	Description
<code>object</code>	The object to reorder.
<code>criteria</code>	The result of the function is used as the criteria to reorder the object.

## Example

This example alphabetically orders the values of each object in the input array. Note that `orderBy($.letter)` produces the same result as `orderBy($[0])`.

## Source

```
%dw 2.0
output application/json
---
{ letter: "b", letter: "c", letter: "a" } orderBy ((value, key) -> value)
```

## Output

```
{
  "letter": "a",
  "letter": "b",
  "letter": "c"
}
```

## orderBy<T, R>(array: Array<T>, criteria: (item: T, index: Number) -> R): Array<T>

Sorts an array using the specified criteria.

## Parameters

Name	Description
<code>array</code>	The array to sort.
<code>criteria</code>	The result of the function serves as criteria for sorting the array. It should return a simple value ( <code>String</code> , <code>Number</code> , and so on).

## Example

This example sorts an array of numbers based on the numeric values.

## Source

```
%dw 2.0
output application/json
---
[3,2,3] orderBy $
```

## Output

```
[ 2, 3, 3 ]
```

## Example

The `orderBy` function does not have an option to order in descending order instead of ascending. In these cases, you can simply invert the order of the resulting array using `-`, for example:

## Source

```
%dw 2.0
output application/json
---
orderDescending: ([3,8,1] orderBy -$)
```

## Output

```
{ "orderDescending": [8,3,1] }
```

## Example

This example sorts an array of people based on their age.

## Source

```
%dw 2.0
output application/json
---
[{name: "Santiago", age: 42},{name: "Leandro", age: 29}, {name: "Mariano", age: 35}]
orderBy (person) -> person.age
```

## Output

```
[  
  {  
    name: "Leandro",  
    age: 29  
  },  
  {  
    name: "Mariano",  
    age: 35  
  },  
  {  
    name: "Santiago",  
    age: 42  
  }  
]
```

### Example

This example sorts an array of DateTime in descending order.

### Source

```
%dw 2.0  
output application/json  
---  
[|2020-10-01T23:57:59.017Z|, |2022-12-22T12:12:12.011Z|, |2020-10-01T12:40:10.012Z|,  
|2020-10-01T23:57:59.021Z|]  
orderBy -(# as Number {unit: "milliseconds"})
```

### Output

```
[  
  "2022-12-22T12:12:12.011Z",  
  "2020-10-01T23:57:59.021Z",  
  "2020-10-01T23:57:59.017Z",  
  "2020-10-01T12:40:10.012Z"  
]
```

### Example

This example changes the order of the objects in a JSON array. The expression first orders them alphabetically by the value of the **Type** key, then reverses the order based on the [\[-1 to 0\]](#).

### Source

```
%dw 2.0
var myInput = [
    {
        "AccountNumber": "987999321",
        "NameOnAccount": "QA",
        "Type": "AAAA",
        "CDetail": {
            "Status": "Open"
        }
    },
    {
        "AccountNumber": "12399978",
        "NameOnAccount": "QA",
        "Type": "BBBB",
        "CDetail": {}
    },
    {
        "AccountNumber": "32199974",
        "NameOnAccount": "QA",
        "Type": "CCCC",
        "CDetail": {}
    }
]
output application/json
---
(myInput orderBy $.Type)[-1 to 0]
```

## Output

```
[
  {
    "AccountNumber": "32199974",
    "NameOnAccount": "QA",
    "Type": "CCCC",
    "CDetail": {

    }
  },
  {
    "AccountNumber": "12399978",
    "NameOnAccount": "QA",
    "Type": "BBBB",
    "CDetail": {

    }
  },
  {
    "AccountNumber": "987999321",
    "NameOnAccount": "QA",
    "Type": "AAAA",
    "CDetail": {
      "Status": "Open"
    }
  }
]
```

## orderBy(value: Null, criteria: (item: Nothing, index: Nothing) -> Null): Null

Helper function that enables `orderBy` to work with a `null` value.

## pluck

`pluck<K, V, R>(@StreamCapable object: { (K)?: V }, mapper: (value: V, key: K, index: Number) -> R): Array<R>`

Useful for mapping an object into an array, `pluck` iterates over an object and returns an array of keys, values, or indices from the object.

It is an alternative to `mapObject`, which is similar but returns an object, instead of an array.

### Parameters

Name	Description
<code>object</code>	The object to map.
<code>mapper</code>	Expression or selector that provides the <code>key</code> , <code>value</code> , and/or <code>index</code> (optional) used for mapping the specified object into an array.

## Example

This example iterates over `{ "a": "b", "c": "d"}` using the anonymous mapper function `((value, key, index) → { (index) : { (value):key} })` to invert each key-value pair in the specified object and to return their indices as keys. The mapper uses named parameters to identify the keys, values, and indices of the object. Note that you can write the same expression using anonymous parameters, like this: `{"a": "b", "c": "d"} pluck { ($$$) : { ($): $$} }` Unlike the almost identical example that uses `mapObject`, `pluck` returns the output as an array.

## Source

```
%dw 2.0
output application/json
---
>{"a": "b", "c": "d"} pluck (value,key,index) -> { (index) : { (value):key} }
```

## Output

```
[ { "0": { "b": "a" } }, { "1": { "d": "c" } } ]
```

## Example

This example uses `pluck` to iterate over each element within `<prices>` and returns arrays of their keys, values, and indices. It uses anonymous parameters to capture them. Note that it uses `as Number` to convert the values to numbers. Otherwise, they would return as strings.

## Source

```
%dw 2.0
output application/json
var readXml = read("<prices>
    <basic>9.99</basic>
    <premium>53.00</premium>
    <vip>398.99</vip>
</prices>", "application/xml")
---
"result" : {
    "keys" : readXml.prices pluck($$),
    "values" : readXml.prices pluck($) as Number,
    "indices" : readXml.prices pluck($$$)
}
```

## Output

```
{
  "result": {
    "keys": [ "basic", "premium", "vip" ],
    "values": [ 9.99, 53, 398.99 ],
    "indices": [ 0, 1, 2 ]
  }
}
```

## pluck(value: Null, mapper: (value: Nothing, key: Nothing, index: Nothing) -> Any): Null

Helper function that enables `pluck` to work with a `null` value.

## pow

### pow(base: Number, power: Number): Number

Raises the value of a `base` number to the specified `power`.

#### Parameters

Name	Description
<code>base</code>	A number ( <code>Number</code> type) that serves as the base.
<code>power</code>	A number ( <code>Number</code> type) that serves as the power.

#### Example

This example raises the value a `base` number to the specified `power`. Note that you can also use the `pow(base,power)` notation (for example, `pow(2,3)` to return 8).

#### Source

```
%dw 2.0
output application/json
---
[ (2 pow 3), (3 pow 2), (7 pow 3) ]
```

#### Output

```
[ 8, 9, 343 ]
```

## random

### random(): Number

Returns a pseudo-random number greater than or equal to `0.0` and less than `1.0`.

## Example

This example generates a pseudo-random number and multiplies it by 1000.

## Source

```
%dw 2.0
output application/json
---
{ price: random() * 1000 }
```

## Output

```
{ "price": 65.02770292248383 }
```

## randomInt

**randomInt(upperBound: Number): Number**

Returns a pseudo-random whole number from `0` to the specified number (exclusive).

## Parameters

Name	Description
upperBound	A number that sets the upper bound of the random number.

## Example

This example returns an integer from 0 to 1000 (exclusive).

## Source

```
%dw 2.0
output application/json
---
{ price: randomInt(1000) }
```

## Output

```
{ "price": 442.0 }
```

## read

**read(stringToParse: String | Binary, contentType: String = "application/dw", readerProperties: Object = {}): Any**

Reads a string or binary and returns parsed content.

This function can be useful if the reader cannot determine the content type by default.

## Parameters

Name	Description
<code>stringToParse</code>	The string or binary to read.
<code>contentType</code>	A supported format (or content type). Default: <code>application/dw</code> .
<code>readerProperties</code>	Optional: Sets reader configuration properties. For other formats and reader configuration properties, see <a href="#">Supported Data Formats</a> .

## Example

This example reads a JSON object `{ "hello" : "world" }`, and it uses the `"application/json"` argument to indicate *input* content type. By contrast, the `output application/xml` directive in the header of the script tells the script to transform the JSON content into XML output. Notice that the XML output uses `hello` as the root XML element and `world` as the value of that element. The `hello` in the XML corresponds to the key `"hello"` in the JSON object, and `world` corresponds to the JSON value `"world"`.

## Source

```
%dw 2.0
output application/xml
---
read('{ "hello" : "world" }','application/json')
```

## Output

```
<?xml version='1.0' encoding='UTF-8'?><hello>world</hello>
```

## Example

This example reads a string as a CSV format without a header and transforms it to JSON. Notice that it adds column names as keys to the output object. Also, it appends `[0]` to the function call here to select the first index of the resulting array, which avoids producing the results within an array (with square brackets surrounding the entire output object).

## Source

```
%dw 2.0
var myVar = "Some, Body"
output application/json
---
read(myVar,"application/csv",{header:false})[0]
```

## Output

```
{ "column_0": "Some", "column_1": " Body" }
```

## Example

This example reads the specified XML and shows the syntax for a reader property, in this case, `{ indexedReader: "false" }`.

## Source

```
%dw 2.0
output application/xml
---
{
    "XML" : read("<prices><basic>9.99</basic></prices>",
                  "application/xml",
                  { indexedReader: "false" })."prices"
}
```

## Output

```
<?xml version='1.0' encoding='UTF-8'?>
<XML>
    <basic>9.99</basic>
</XML>
```

## readUrl

**readUrl(url: String, contentType: String = "application/dw", readerProperties: Object = {}): Any**

Reads a URL, including a classpath-based URL, and returns parsed content. This function works similar to the `read` function.

The classpath-based URL uses the `classpath:` protocol prefix, for example: `classpath://myfolder/myFile.txt` where `myFolder` is located under `src/main/resources` in a Mule project. Other than the URL, `readURL` accepts the same arguments as `read`.

## Parameters

Name	Description
<code>url</code>	The URL string to read. It also accepts a classpath-based URL.
<code>contentType</code>	A supported format (or MIME type). Default: <code>application/dw</code> .
<code>readerProperties</code>	Optional: Sets reader configuration properties. For other formats and reader configuration properties, see <a href="#">Supported Data Formats</a> .

## Example

This example reads a JSON object from a URL. (For readability, the output values shown below are shortened with `...`.)

### Source

```
%dw 2.0
output application/json
---
readUrl("https://jsonplaceholder.typicode.com/posts/1", "application/json")
```

### Output

```
{ "userId": 1, "id": 1, "title": "sunt aut ...", "body": "quia et ..." }
```

## Example

This example reads a JSON object from a `myJsonSnippet.json` file located in the `src/main/resources` directory in Studio. (Sample JSON content for that file is shown in the Input section below.) After reading the file contents, the script transforms selected fields from JSON to CSV. Reading files in this way can be useful when trying out a DataWeave script on sample data, especially when the source data is large and your script is complex.

### Source

```
%dw 2.0
var myJsonSnippet = readUrl("classpath://myJsonSnippet.json", "application/json")
output application/csv
---
(myJsonSnippet.results map(item) -> item.profile)
```

### Input

```
{
  "results": [
    {
      "profile": {
        "firstName": "john",
        "lastName": "doe",
        "email": "johndoe@demo.com"
      },
      "data": {
        "interests": [
          {
            "language": "English",
            "tags": [
              "digital-strategy:Digital Strategy",
              "innovation:Innovation"
            ],
            "contenttypes": []
          }
        ]
      }
    },
    {
      "profile": {
        "firstName": "jane",
        "lastName": "doe",
        "email": "janedoe@demo.com"
      },
      "data": {
        "interests": [
          {
            "language": "English",
            "tags": [
              "tax-reform:Tax Reform",
              "retail-health:Retail Health"
            ],
            "contenttypes": [
              "News",
              "Analysis",
              "Case studies",
              "Press releases"
            ]
          }
        ]
      }
    }
  ]
}
```

## Output

```
firstName,lastName,email  
john,doe,johndoe@demo.com  
jane,doe,janedoe@demo.com
```

### Example

This example reads a CSV file from a URL, sets reader properties to indicate that there's no header, and then transforms the data to JSON.

### Source

```
%dw 2.0  
output application/json  
---  
readUrl("https://mywebsite.com/data.csv", "application/csv", {"header" : false})
```

### Input

```
Max,the Mule,MuleSoft
```

### Output

```
[  
 {  
   "column_0": "Max",  
   "column_1": "the Mule",  
   "column_2": "MuleSoft"  
 }
```

### Example

This example reads a simple `dw1` file from the `src/main/resources` directory in Studio, then dynamically reads the value of the key `name` from it. (Sample content for the input file is shown in the Input section below.)

### Source

```
%dw 2.0  
output application/json  
---  
(readUrl("classpath://name.dw1", "application/dw")).firstName
```

### Input

```
{
  "firstName" : "Somebody",
  "lastName" : "Special"
}
```

## Output

```
"Somebody"
```

## reduce

**reduce<T>(@StreamCapable items: Array<T>, callback: (item: T, accumulator: T) -> T): T | Null**

Applies a reduction expression to the elements in an array.

For each element of the input array, in order, **reduce** applies the reduction lambda expression (function), then replaces the accumulator with the new result. The lambda expression can use both the current input array element and the current accumulator value.

Note that if the array is empty and no default value is set on the accumulator parameter, a null value is returned.

### Parameters

Name	Description
<b>item</b>	Item in the input array. It provides the value to reduce. Can also be referenced as <b>\$</b> .
<b>acc</b>	The accumulator. Can also be referenced as <b>\$\$</b> . Used to store the result of the lambda expression after each iteration of the <b>reduce</b> operation.  The accumulator parameter can be set to an initial value using the syntax <b>acc = initialValue</b> . In this case, the lambda expression is called with the first element of the input array. Then the result is set as the new accumulator value.  If an initial value for the accumulator is not set, the accumulator is set to the first element of the input array. Then the lambda expression is called with the second element of the input array.
	The initial value of the accumulator and the lambda expression dictate the type of result produced by the <b>reduce</b> function. If the accumulator is set to <b>acc = {}</b> , the result is usually of type <b>Object</b> . If the accumulator is set to <b>acc = []</b> , the result is usually of type <b>Array</b> . If the accumulator is set to <b>acc = ""</b> , the result is usually a <b>String</b> .

### Example

This example returns the sum of the numeric values in the first input array.

### Source

```
%dw 2.0
output application/json
---
[2, 3] reduce ($ + $$)
```

### Output

```
5
```

### Example

This example adds the numbers in the `sum` example, concatenates the same numbers in `concat`, and shows that an empty array `[]` (defined in `myEmptyList`) returns `null` in `emptyList`.

### Source

```
%dw 2.0
var myNums = [1,2,3,4]
var myEmptyList = []
output application/json
---
{
  "sum" : myNums reduce ($$ + $),
  "concat" : myNums reduce ($$ ++ $),
  "emptyList" : myEmptyList reduce ($$ ++ $)
}
```

### Output

```
{ "sum": 10, "concat": "1234", "emptyList": null }
```

### Example

This example sets the first element from the first input array to `"z"`, and it adds `3` to the sum of the second input array. In `multiply`, it shows how to multiply each value in an array by the next (`[2,3,3] reduce ((item, acc) → acc * item)`) to produce a final result of `18` ( $= 2 * 3 * 3$ ). The final example, `multiplyAcc`, sets the accumulator to `3` to multiply the result of `acc * item` ( $= 12$ ) by `3` (that is,  $3 * 2 * 3 = 36$ ), as shown in the output.

### Source

```
%dw 2.0
output application/json
---
{
    "concat" : ["a", "b", "c", "d"] reduce ((item, acc = "z") -> acc ++ item),
    "sum": [0, 1, 2, 3, 4, 5] reduce ((item, acc = 3) -> acc + item),
    "multiply" : [2,3,3] reduce ((item, acc) -> acc * item),
    "multiplyAcc" : [2,2,3] reduce ((item, acc = 3) -> acc * item)
}
```

## Output

```
{ "concat": "zabcd", "sum": 18, "multiply": 18, "multiplyAcc": 36 }
```

## Example

This example shows a variety of uses of **reduce**, including its application to arrays of boolean values and objects.

## Source

```

%dw 2.0
output application/json
var myVar =
{
    "a": [0, 1, 2, 3, 4, 5],
    "b": ["a", "b", "c", "d", "e"],
    "c": [{"letter": "a"}, {"letter": "b"}, {"letter": "c"}],
    "d": [true, false, false, true, true]
}
---
{
    "a" : [0, 1, 2, 3, 4, 5] reduce $$,
    "b": ["a", "b", "c", "d", "e"] reduce $$,
    "c": [{"letter": "a"}, {"letter": "b"}, {"letter": "c"}] reduce ((item, acc = "z") -> acc ++ item.letter),
    "d": [{letter: "a"}, {letter: "b"}, {letter: "c"}] reduce $$,
    "e": [true, false, false, true, true] reduce ($$ and $),
    "f": [true, false, false, true, true] reduce ((item, acc) -> acc and item),
    "g": [true, false, false, true, true] reduce ((item, acc = false) -> acc and item),
    "h": [true, false, false, true, true] reduce $$,
    "i": myVar.a reduce ($$ + $),
    "j": myVar.a reduce ((item, acc) -> acc + item),
    "k": myVar.a reduce ((item, acc = 3) -> acc + item),
    "l": myVar.a reduce $$,
    "m": myVar.b reduce ($$ ++ $),
    "n": myVar.b reduce ((item, acc) -> acc ++ item),
    "o": myVar.b reduce ((item, acc = "z") -> acc ++ item),
    "p": myVar.b reduce $$,
    "q": myVar.c reduce ((item, acc = "z") -> acc ++ item.letter),
    "r": myVar.c reduce $$,
    "s": myVar.d reduce ($$ and $),
    "t": myVar.d reduce ((item, acc) -> acc and item),
    "u": myVar.d reduce ((item, acc = false) -> acc and item),
    "v": myVar.d reduce $$,
    "w": ([0, 1, 2, 3, 4] reduce ((item, acc = {}) -> acc ++ { a: item })) pluck $,
    "x": [] reduce $$,
    "y": [] reduce ((item, acc = 0) -> acc + item)
}

```

## Output

```

    "a": 0,
    "b": "a",
    "c": "zabc",
    "d": { "letter": "a" },
    "e": false,
    "f": false,
    "g": false,
    "h": true,
    "i": 15,
    "j": 15,
    "k": 18,
    "l": 0,
    "m": "abcde",
    "n": "abcde",
    "o": "zabcde",
    "p": "a",
    "q": "zabc",
    "r": { "letter": "a" },
    "s": false,
    "t": false,
    "u": false,
    "v": true,
    "w": [ 0,1,2,3,4 ],
    "x": null,
    "y": 0
}

```

**reduce<T, A>(@StreamCapable items: Array<T>, callback: (item: T, accumulator: A) -> A): A**

**reduce(@StreamCapable text: String, callback: (item: String, accumulator: String) -> String): String**

Applies a reduction expression to the characters in a string.

For each character of the input string, in order, **reduce** applies the reduction lambda expression (function), then replaces the accumulator with the new result. The lambda expression can use both the current character and the current accumulator value.

Note that if the string is empty and no default value is set on the accumulator parameter, an empty string is returned.

#### Parameters

Name	Description
<b>text</b>	The string to reduce.
<b>callback</b>	The function to apply.

#### Example

This example shows how `reduce` can be used to reverse a string.

## Source

```
%dw 2.0
output application/json
---
"hello world" reduce (item, acc = "") -> item ++ acc
```

## Output

```
"dlrow olleh"
```

`reduce<A>(@StreamCapable text: String, callback: (item: String, accumulator: A) -> A): A`

`reduce<T, A>(@StreamCapable items: Null, callback: (item: T, accumulator: A) -> A): Null`

Helper function that enables `reduce` to work with a `null` value.

## replace

`replace(text: String, matcher: Regex): ((Array<String>, Number) -> String) -> String`

Performs string replacement.

This version of `replace` accepts a Java regular expression for matching part of a string. It requires the use of the `with` helper function to specify a replacement string for the matching part of the input string.

## Parameters

Name	Description
<code>text</code>	A string to match.
<code>matcher</code>	A Java regular expression for matching characters in the input <code>text</code> string.

## Example

The first example in the source replaces all characters up to and including the second hyphen (`123-456-`) with an empty value, so it returns the last four digits. The second replaces the characters `b13e` in the input string with a hyphen (`-`).

## Source

```
%dw 2.0
output application/json
---
["123-456-7890" replace /.*-/ with(""), "abc123def" replace /[b13e]/ with("-")]
```

## Output

```
[ 7890, "a-c-2-d-f" ]
```

## Example

This example replaces the numbers **123** in the input strings with **ID**. It uses the regular expression **(\d+)**, where the **\d** metacharacter means any digit from 0-9, and **+** means that the digit can occur one or more times. Without the **+**, the output would contain one **ID** per digit. The example also shows how to write the expression using infix notation, then using prefix notation.

## Source

```
%dw 2.0
output application/json
---
[ "my123" replace /(\d+)/ with("ID"), replace("myOther123", /(\d+)/) with("ID") ]
```

## Output

```
[ "myID", "myOtherID" ]
```

## replace(text: String, matcher: String): ((Array<String>, Number) -> String) -> String

Performs string replacement.

This version of **replace** accepts a string that matches part of a specified string. It requires the use of the **with** helper function to pass in a replacement string for the matching part of the input string.

## Parameters

Name	Description
<b>text</b>	The string to match.
<b>matcher</b>	The string for matching characters in the input <b>text</b> string.

## Example

This example replaces the numbers **123** from the input string with the characters **ID**, which are passed through the **with** function.

## Source

```
%dw 2.0
output application/json
---
{ "replace": "admin123" replace "123" with("ID") }
```

## Output

```
{ "replace": "adminID" }
```

**replace(text: Null, matcher: Any): ((Nothing, Nothing) -> Any) -> Null**

Helper function that enables `replace` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## round

**round(number: Number): Number**

Rounds a number up or down to the nearest whole number.

### Parameters

Name	Description
number	The number to evaluate.

### Example

This example rounds decimal numbers to the nearest whole numbers.

## Source

```
%dw 2.0
output application/json
---
[ round(1.2), round(4.6), round(3.5) ]
```

## Output

```
[ 1, 5, 4 ]
```

## scan

## scan(text: String, matcher: Regex): Array<Array<String>>

Returns an array with all of the matches found in an input string.

Each match is returned as an array that contains the complete match followed by any capture groups in your regular expression (if present).

### Parameters

Name	Description
text	The input string to scan.
regex	A Java regular expression that describes the pattern match in the <code>text</code> .

### Example

In this example, the `regex` describes a URL. It contains three capture groups within the parentheses, the characters before and after the period (`.`). It produces an array of matches to the input URL and the capture groups. It uses `flatten` to change the output from an array of arrays into a simple array. Note that a `regex` is specified within forward slashes (`//`).

### Source

```
%dw 2.0
output application/json
---
flatten("www.mulesoft.com" scan(/([w]*).([a-z]*).([a-z]*)/))
```

### Output

```
[ "www.mulesoft.com", "www", "mulesoft", "com" ]
```

### Example

In the example, the `regex` describes an email address. It contains two capture groups, the characters before and after the `@`. It produces an array matches to the email addresses and capture groups in the input string.

### Source

```
%dw 2.0
output application/json
---
"anypt@mulesoft.com,max@mulesoft.com" scan(/([a-z]*)@([a-z]*).com/)
```

### Output

```
[  
  [ "anypt@mulesoft.com", "anypt", "mulesoft" ],  
  [ "max@mulesoft.com", "max", "mulesoft" ]  
]
```

## scan(text: Null, matcher: Any): Null

Helper function that enables `scan` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## sizeOf

### sizeOf(array: Array<Any>): Number

Returns the number of elements in an array. It returns `0` if the array is empty.

This version of `sizeOf` takes an array or an array of arrays as input. Other versions act on arrays of objects, strings, or binary values.

#### Parameters

Name	Description
<code>array</code>	The input array. The elements in the array can be any supported type.

#### Example

This example counts the number of elements in the input array. It returns `3`.

#### Source

```
%dw 2.0  
output application/json  
---  
sizeOf([ "a", "b", "c" ])
```

#### Output

```
3
```

#### Example

This example returns a count of elements in the input array.

#### Source

```
%dw 2.0
output application/json
---
{
  "arraySizes": {
    "size3": sizeOf([1,2,3]),
    "size2": sizeOf([[1,2,3],[4]]),
    "size0": sizeOf([])
  }
}
```

## Output

```
{
  "arraySizes": {
    "size3": 3,
    "size2": 2,
    "size0": 0
  }
}
```

## sizeOf(object: Object): Number

Returns the number of key-value pairs in an object.

This function accepts an array of objects. Returns **0** if the input object is empty.

### Parameters

Name	Description
object	The input object that contains one or more key-value pairs.

### Example

This example counts the key-value pairs in the input object, so it returns **2**.

### Source

```
%dw 2.0
output application/json
---
sizeOf({a: 1, b: 2})
```

## Output

```
2
```

## Example

This example counts the key-value pairs in an object.

## Source

```
%dw 2.0
output application/json
---
{
  objectSizes : {
    sizeIs2: sizeOf({a:1,b:2}),
    sizeIs0: sizeOf({})
  }
}
```

## Output

```
{
  "objectSize": {
    "sizeIs2": 2,
    "sizeIs0": 0
  }
}
```

## sizeOf(binary: Binary): Number

Returns the number of elements in an array of binary values.

## Parameters

Name	Description
binary	The input array of binary values.

## Example

This example returns the size of an array of binary values.

## Source

```
%dw 2.0
output application/json
---
sizeOf(["\u0000" as Binary, "\u0001" as Binary, "\u0002" as Binary])
```

## Output

**sizeOf(text: String): Number**

Returns the number of characters (including white space) in an string.

Returns **0** if the string is empty.

**Parameters**

Name	Description
text	The input text.

**Example**

This example returns the number of characters in the input string **"abc"**.

**Source**

```
%dw 2.0
output application/json
---
sizeOf("abc")
```

**Output**

3

**Example**

This example returns the number of characters in the input strings. Notice it counts blank spaces in the string **"my string"** and that **sizeOf("123" as Number)** returns **1** because **123** is coerced into a number, so it is not a string.

**Source**

```
%dw 2.0
output application/json
---
{
  sizeOfString2 : sizeOf("my string"),
  sizeOfEmptyString: sizeOf(""),
  sizeOfNumber : sizeOf("123" as Number)
}
```

**Output**

```
{  
  "sizeOfString2": 9,  
  "sizeOfEmptyString": 0,  
  "sizeOfNumber": 1  
}
```

## sizeOf(n: Null): Null

Helper function that enables `sizeOf` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## splitBy

### splitBy(text: String, regex: Regex): Array<String>

Splits a string into a string array based on a value that matches part of that string. It filters out the matching part from the returned array.

This version of `splitBy` accepts a Java regular expression (regex) to match the input string. The regex can match any character in the input string. Note that `splitBy` performs the opposite operation of `joinBy`.

#### Parameters

Name	Description
<code>text</code>	The input string to split.
<code>regex</code>	A Java regular expression used to split the string. If it does not match some part of the string, the function will return the original, unsplit string in the array.

#### Example

This example uses a Java regular expression to split an address block by the periods and forward slash in it. Notice that the regular expression goes between forward slashes.

#### Source

```
%dw 2.0  
output application/json  
---  
"192.88.99.0/24" splitBy(/[.\\/]/)
```

#### Output

```
["192", "88", "99", "0", "24"]
```

## Example

This example uses several regular expressions to split input strings. The first uses `\/*^.b./` to split the string by `-b-`. The second uses `/\s/` to split by a space. The third example returns the original input string in an array (`[ "no match" ]`) because the regex `/\s/` (for matching the first character if it is `s`) does not match the first character in the input string ("no match"). The fourth, which uses `/^n.../`, matches the first characters in "no match", so it returns `[ "", "match" ]`. The last removes all numbers and capital letters from a string, leaving each of the lower case letters in the array. Notice that the separator is omitted from the output.

## Source

```
%dw 2.0
output application/json
---
{ "splitters" : {
    "split1" : "a-b-c" splitBy(/\/*^.b./),
    "split2" : "hello world" splitBy(/\s/),
    "split3" : "no match" splitBy(/\s/),
    "split4" : "no match" splitBy(/^n.../),
    "split5" : "a1b2c3d4A1B2C3D" splitBy(/\/*[0-9A-Z]/)
  }
}
```

## Output

```
{
  splitters: {
    split1: [ "a", "c" ],
    split2: [ "hello", "world" ],
    split3: [ "no match" ],
    split4: [ "", "match" ],
    split5: [ "a", "b", "c", "d" ]
  }
}
```

## Example

This example splits the number by `.` and applies the index selector `[0]` to the result of the `splitBy` function. The `splitBy` returns `[ "192", "88", "99", "0" ]` so the index `*` selector `[0]` just returns the first element in the array ("192").

## Source

```
%dw 2.0
output application/json
---
("192.88.99.0" splitBy("."))[0]
```

## Output

```
"192"
```

## Example

This example uses a Java regular expression to split a string by `.` at every point the input string matches the regex. Note that the regular expression does not consider the periods between the backticks `\``.

## Source

```
%dw 2.0
output application/json
---
'root.sources.data.\`test.branch.BranchSource\`.source.traits'
splitBy(/[.](?=(:[^']*[^']*')*[^\']*$/))
```

## Output

```
[
  "root",
  "sources",
  "data",
  "\`test.branch.BranchSource\`",
  "source",
  "traits"
]
```

## splitBy(text: String, separator: String): Array<String>

Splits a string into a string array based on a separating string that matches part of the input string. It also filters out the matching string from the returned array.

The separator can match any character in the input. Note that `splitBy` performs the opposite operation of `joinBy`.

## Parameters

Name	Description
text	The string to split.
separator	A string used to separate the input string. If it does not match some part of the string, the function will return the original, unsplit string in the array.

## Example

This example splits a string containing an IP address by its periods.

## Source

```
%dw 2.0
output application/json
---
"192.88.99.0" splitBy(".")
```

## Output

```
["192", "88", "99", "0"]
```

## Example

The first example (`splitter1`) uses a hyphen (-) in "a-b-c" to split the string. The second uses an empty string ("") to split each character (including the blank space) in the string. The third example splits based on a comma (,) in the input string. The last example does not split the input because the function is case sensitive, so the upper case `NO` does not match the lower case `no` in the input string. Notice that the separator is omitted from the output.

## Source

```
%dw 2.0
output application/json
---
{ "splitters" : {
    "split1" : "a-b-c" splitBy("-"),
    "split2" : "hello world" splitBy(""),
    "split3" : "first,middle,last" splitBy(","),
    "split4" : "no split" splitBy("NO")
  }
}
```

## Output

```
{
  splitters: {
    split1: [ "a", "b", "c" ],
    split2: [ "h", "e", "l", "l", "o", "", "w", "o", "r", "l", "d" ],
    split3: [ "first", "middle", "last" ],
    split4: [ "no split" ]
  }
}
```

## splitBy(text: Null, separator: Any)

Helper function that enables `splitBy` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## sqrt

### sqrt(number: Number): Number

Returns the square root of a number.

#### Parameters

Name	Description
<code>number</code>	The number to evaluate.

#### Example

This example returns the square root of a number.

#### Source

```
%dw 2.0
output application/json
---
[ sqrt(4), sqrt(25), sqrt(100) ]
```

#### Output

```
[ 2.0, 5.0, 10.0 ]
```

## startsWith

### startsWith(text: String, prefix: String): Boolean

Returns `true` or `false` depending on whether the input string starts with a matching prefix.

## Parameters

Name	Description
text	The input string.
prefix	A string that identifies the prefix.

## Example

This example indicates whether the strings start with a given prefix. Note that you can use the `startsWith(text,prefix)` or `text startsWith(prefix)` notation (for example, `startsWith("Mari","Mar")` or `"Mari" startsWith("Mar")`).

## Source

```
%dw 2.0
output application/json
---
[ "Mari" startsWith("Mar"), "Mari" startsWith("Em") ]
```

## Output

```
[ true, false ]
```

## startsWith(text: Null, prefix: Any): false

Helper function that enables `startsWith` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## sum

### sum(values: Array<Number>): Number

Returns the sum of numeric values in an array.

Returns `0` if the array is empty and produces an error when non-numeric values are in the array.

## Parameters

Name	Description
values	The input array of numbers.

## Example

This example returns the sum of the values in the input array.

## Source

```
%dw 2.0
output application/json
---
sum([1, 2, 3])
```

## Output

```
6
```

## then

**then(value: Null, callback: (previousResult: Nothing) -> Any): Null**

Helper function that enables `then` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

**then<T, R>(previous: T, callback: (result: T) -> R): R**

This function works as a pipe that passes the value returned from the preceding expression to the next (a callback) only if the value returned by the preceding expression is not `null`.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>previous</code>	The value of the preceding expression.
<code>callback</code>	Callback that processes the result of <code>previous</code> if the result is not <code>null</code> .

### Example

This example shows how to use `then` to chain and continue processing the result of the previous expression.

### Source

```
%dw 2.0
output application/json
---
{
    "chainResult": ["mariano", "de Achaval"]
        reduce ((item, accumulator) -> item ++ accumulator)
        then ((result) -> sizeOf(result)),
    "referenceResult" : ["mariano", "de Achaval"]
        map ((item, index) -> upper(item))
        then {
            name: #[0],
            lastName: #[1],
            length: sizeOf($)
        },
    "onNullReturnNull": []
        reduce ((item, accumulator) -> item ++ accumulator)
        then ((result) -> sizeOf(result))
}
}
```

## Output

```
{
    "chainResult": 17,
    "referenceResult": {
        "name": "MARIANO",
        "lastName": "DE ACHAVAL",
        "length": 2
    },
    "onNullReturnNull": null
}
```

## to

### **to(from: Number, to: Number): Range**

Returns a range with the specified boundaries.

The upper boundary is inclusive.

#### Parameters

Name	Description
from	Number value that starts the range. The output includes the <b>from</b> value.
to	Number value that ends the range. The output includes the <b>from</b> value.

#### Example

This example lists a range of numbers from 1 to 10.

### Source

```
%dw 2.0
output application/json
---
{ "myRange": 1 to 10 }
```

### Output

```
{ "myRange": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] }
```

### Example

DataWeave treats a string as an array of characters. This example applies **to** to a string.

### Source

```
%dw 2.0
var myVar = "Hello World!"
output application/json
---
{
  indices2to6 : myVar[2 to 6],
  indicesFromEnd : myVar[6 to -1],
  reversal : myVar[11 to -0]
}
```

### Output

```
{
  "indices2to6": "llo W",
  "indicesFromEnd": "World!",
  "reversal": "!dlroW olleH"
}
```

## trim

### trim(text: String): String

Removes any blank spaces from the beginning and end of a string.

### Parameters

Name	Description
text	The string from which to remove any blank spaces.

## Example

This example trims a string. Notice that it does not remove any spaces from the middle of the string, only the beginning and end.

## Source

```
%dw 2.0
output application/json
---
{ "trim": trim("    my really long text      ") }
```

## Output

```
{ "trim": "my really long text" }
```

## Example

This example shows how `trim` handles a variety strings and how it handles a null value.

## Source

```
%dw 2.0
output application/json
---
{
  "null": trim(null),
  "empty": trim(""),
  "blank": trim("    "),
  "noBlankSpaces": trim("abc"),
  "withSpaces": trim("    abc    ")
}
```

## Output

```
{
  "null": null,
  "empty": "",
  "blank": "",
  "noBlankSpaces": "abc",
  "withSpaces": "abc"
}
```

## trim(value: Null): Null

Helper function that enables `trim` to work with a `null` value.

## typeOf

### typeOf<T>(value: T): Type<T>

Returns the basic data type of a value.

A value's type is taken from its runtime representation and is never one of the arithmetic types (intersection, union, `Any`, or `Nothing`) nor a type alias. If present, metadata of a value is included in the result of `typeOf` (see [metadataOf](#)).

#### Parameters

Name	Description
<code>value</code>	Input value to evaluate.

#### Example

This example identifies the type of several input values.

#### Source

```
%dw 2.0
output application/json
---
[ typeOf("A b"), typeOf([1,2]), typeOf(34), typeOf(true), typeOf({ a : 5 }) ]
```

#### Output

```
[ "String", "Array", "Number", "Boolean", "Object" ]
```

#### Example

This example shows that the type of a value is independent of the type with which it is declared.

#### Source

```
%dw 2.0
output application/json

var x: String | Number = "clearly a string"
var y: "because" = "because"
---
[typeOf(x), typeOf(y)]
```

## Output

```
["String", "String"]
```

## unzip

**unzip<T>(items: Array<Array<T>>): Array<Array<T>>**

Performs the opposite of [zip](#). It takes an array of arrays as input.

The function groups the values of the input sub-arrays by matching indices, and it outputs new sub-arrays with the values of those matching indices. No sub-arrays are produced for unmatching indices. For example, if one input sub-array contains four elements (indices 0-3) and another only contains three (indices 0-2), the function will not produce a sub-array for the value at index 3.

### Parameters

Name	Description
items	The input array of arrays.

### Example

This example unzips an array of arrays. It outputs the first index of each sub-array into one array [ 0, 1, 2, 3 ], and the second index of each into another [ "a", "b", "c", "d" ].

### Source

```
%dw 2.0
output application/json
---
unzip([ [0,"a"], [1,"b"], [2,"c"], [3,"d"] ])
```

## Output

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

### Example

This example unzips an array of arrays. Notice that the number of elements in the input arrays is not all the same. The function creates only as many full sub-arrays as it can, in this case, just one.

### Source

```
%dw 2.0
output application/json
---
unzip([ [0,"a"], [1,"a","foo"], [2], [3,"a"] ])
```

## Output

```
[0,1,2,3]
```

## upper

### upper(text: String): String

Returns the provided string in uppercase characters.

#### Parameters

Name	Description
text	The string to convert to uppercase.

#### Example

This example converts lowercase characters to uppercase.

#### Source

```
%dw 2.0
output application/json
---
{ "name" : upper("mulesoft") }
```

## Output

```
{ "name": "MULESOFT" }
```

### upper(value: Null): Null

Helper function that enables `upper` to work with a `null` value.

## uuid

### uuid(): String

Returns a v4 UUID using random numbers as the source.

## Example

This example generates a random v4 UUID.

## Source

```
%dw 2.0
output application/json
---
uuid()
```

## Output

```
"7cc64d24-f2ad-4d43-8893-fa24a0789a99"
```

## valuesOf

**valuesOf<K, V>(obj: { (K)?: V }): Array<V>**

Returns an array of the values from key-value pairs in an object.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
obj	The object to evaluate.

## Example

This example returns the values of key-value pairs within the input object.

## Source

```
%dw 2.0
output application/json
---
{ "valuesOf" : valuesOf({a: true, b: 1}) }
```

## Output

```
{ "valuesOf" : [true,1] }
```

## valuesOf(obj: Null): Null

Helper function that enables **valuesOf** to work with a **null** value.

Introduced in DataWeave version 2.4.0.

## with

**with<V, U, R, X>(toBeReplaced: ((V, U) -> R) -> X, replacer: (V, U) -> R): X**

Helper function that specifies a replacement element. This function is used with [replace](#), [update](#) or [mask](#) to perform data substitutions.

### Parameters

Name	Description
toBeReplaced	The value to be replaced.
replacer	The replacement value for the input value.

### Example

This example replaces all numbers in a string with "x" characters. The [replace](#) function specifies the base string and a regex to select the characters to replace, and [with](#) provides the replacement string to use.

### Source

```
%dw 2.0
output application/json
---
{ "ssn" : "987-65-4321" replace /[0-9]/ with("x") }
```

### Output

```
{ "ssn": "xxx-xx-xxxx" }
```

## write

**write(value: Any, contentType: String = "application/dw", writerProperties: Object = {}): String | Binary**

Writes a value as a string or binary in a supported format.

Returns a String or Binary with the serialized representation of the value in the specified format (MIME type). This function can write to a different format than the input. Note that the data must validate in that new format, or an error will occur. For example, [application/xml](#) content is not valid within an [application/json](#) format, but [text/plain](#) can be valid. It returns a [String](#) value for all text-based data formats (such as XML, JSON , CSV) and a [Binary](#) value for all the binary formats (such as Excel, MultiPart, OctetStream).

### Parameters

Name	Description
<code>value</code>	The value to write. The value can be of any supported data type.
<code>contentType</code>	A supported format (or MIME type) to write. Default: <a href="#">application/dw</a> .
<code>writerProperties</code>	Optional: Sets writer configuration properties. For writer configuration properties (and other supported MIME types), see <a href="#">Supported Data Formats</a> .

## Example

This example writes the string `world` in plain text (`text/plain`). It outputs that string as the value of a JSON object with the key `hello`.

## Source

```
%dw 2.0
output application/json
---
{ hello : write("world", "text/plain") }
```

## Output

```
{ "hello": "world" }
```

## Example

This example takes JSON input and writes the payload to a CSV format that uses a pipe (|) separator and includes the header (matching keys in the JSON objects). Note that if you instead use `"header":false` in your script, the output will lack the `Name|Email|Id|Title` header in the output.

## Source

```
%dw 2.0
output application/xml
---
{ "output" : write(payload, "application/csv", {"header":true, "separator" : "|"}) }
```

## 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 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>
```

## xsiType

### xsiType(name: String, namespace: Namespace)

Creates a `xsi:type` type attribute. This method returns an object, so it must be used with dynamic attributes.

*Introduced in DataWeave version 2.2.2.*

#### Parameters

Name	Description
<code>name</code>	The name of the schema <code>type</code> that is referenced without the prefix.
<code>namespace</code>	The namespace of that type.

#### Example

This example shows how the `xsiType` behaves under different inputs.

#### Source

```
%dw 2.0
output application/xml
ns acme http://acme.com
---
{
    user @((xsiType("user", acme))): {
        name: "Peter",
        lastName: "Parker"
    }
}
```

## Output

```
<?xml version='1.0' encoding='UTF-8'?>
<user xsi:type="acme:user" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:acme="http://acme.com">
    <name>Peter</name>
    <lastName>Parker</lastName>
</user>
```

## zip

**zip<T, R>(left: Array<T>, right: Array<R>): Array<Array<T | R>>**

Merges elements from two arrays into an array of arrays.

The first sub-array in the output array contains the first indices of the input sub-arrays. The second index contains the second indices of the inputs, the third contains the third indices, and so on for every case where there are the same number of indices in the arrays.

### Parameters

Name	Description
left	The array on the left-hand side of the function.
right	The array on the right-hand side of the function.

### Example

This example zips the arrays located to the left and right of **zip**. Notice that it returns an array of arrays where the first index, (**[0,1]**) contains the first indices of the specified arrays. The second index of the output array (**[1,"b"]**) contains the second indices of the specified arrays.

### Source

```
%dw 2.0
output application/json
---
[0,1] zip ["a","b"]
```

## Output

```
[ [0,"a"], [1,"b"] ]
```

## Example

This example zips elements of the left-hand and right-hand arrays. Notice that only elements with counterparts at the same index are returned in the array.

## Source

```
%dw 2.0
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"],
  "d" : [0, 1, 2] zip ["a", "b", "c", "d"]
}
```

## Output

```
{  
  "a": [  
    [0,"a"],  
    [1,"b"],  
    [2,"c"],  
    [3,"d"]  
  ],  
  "b": [  
    [0,"a"]  
  ],  
  "c": [  
    [0,"a"],  
    [1,"b"]  
  ],  
  "d": [  
    [0,"a"],  
    [1,"b"],  
    [2,"c"]  
  ]  
}
```

### Example

This example zips more than two arrays. Notice that items from `["aA", "bB"]` in `list4` are not in the output because the other input arrays only have two indices.

### Source

```
%dw 2.0  
output application/json  
var myvar = {  
  "list1": ["a", "b"],  
  "list2": [1, 2, 3],  
  "list3": ["aa", "bb"],  
  "list4": [{"A": "B", "C": "D"}, [11, 12], ["aA", "bB"]]  
}  
---  
((myvar.list1 zip myvar.list2) zip myvar.list3) zip myvar.list4
```

### Output

```
[  
  [  
    [ [ "a", 1 ], "aa" ], [ "A", "B", "C" ]  
  ],  
  [  
    [ [ "b", 2 ], "bb" ], [ 11, 12 ]  
  ]  
]
```

## Core Types

Type	Definition	Description
Any	<code>type Any = Any</code>	The top-level type. <code>Any</code> extends all of the system types, which means that anything can be assigned to a <code>Any</code> typed variable.
Array	<code>type Array = Array</code>	Array type that requires a <code>Type(T)</code> to represent the elements of the list. Example: <code>Array&lt;Number&gt;</code> represents an array of numbers, and <code>Array&lt;Any&gt;</code> represents an array of any type.  Example: <code>[1, 2, "a", "b", true, false, { a : "b"}, [1, 2, 3] ]</code>
Binary	<code>type Binary = Binary</code>	A blob.
Boolean	<code>type Boolean = Boolean</code>	A <code>Boolean</code> type of <code>true</code> or <code>false</code> .
CData	<code>type CData = String {cdata: true}</code>	XML defines a <code>CData</code> custom type that extends from <code>String</code> 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 string arrives inside a CDATA block. <code>CData</code> inherits from the type <code>String</code> .  <b>Source:</b>  <code>output application/xml --- { "user" : "Shoki" as CData }</code>  <b>Output:</b>  <code>&lt;?xml version="1.0" encoding="UTF-8"?&gt;&lt;user&gt;&lt;![CDATA[Shoki]]&gt;&lt;/user&gt;</code>

Type	Definition	Description
Comparable	<code>type Comparable = String Number   Boolean   DateTime LocalDateTime   Date LocalTime   Time   TimeZone</code>	A union type that represents all the types that can be compared to each other.
Date	<code>type Date = Date</code>	A date represented by a year, month, and day. For example: <code> 2018-09-17 </code>
DateTime	<code>type DateTime = DateTime</code>	A <code>Date</code> and <code>Time</code> within a <code>TimeZone</code> . For example: <code> 2018-09-17T22:13:00Z </code>
Dictionary	<code>type Dictionary = { _: T }</code>	Generic dictionary interface.
Enum	<code>type Enum = String {enumeration: true}</code>	<p>This type is based on the <a href="#">Enum Java class</a>.</p> <p>It must always be used with the <code>class</code> property, specifying the full Java class name of the class, as shown in the example below.</p> <p><b>Source:</b></p> <pre>"Max" as Enum {class: "com.acme.MuleyEnum"}</pre>
Iterator	<code>type Iterator = Array {iterator: true}</code>	<p>This type is based on the <a href="#">iterator Java class</a>. The iterator contains a collection and includes methods to iterate through and filter it.</p> <p>Just like the Java class, <code>Iterator</code> is designed to be consumed only once. For example, if you pass it to a <a href="#">Logger component</a>, the Logger consumes it, so it becomes unreadable by further elements in the flow.</p>
Key	<code>type Key = Key</code>	<p>A key of an <a href="#">Object</a>.</p> <p>Examples: <code>{ myKey : "a value" }, { myKey : { a : 1, b : 2} }, { myKey : [1,2,3,4] }</code></p>
LocalDateTime	<code>type LocalDateTime = LocalDateTime</code>	A <code>DateTime</code> in the current <code>TimeZone</code> . For example: <code> 2018-09-17T22:13:00 </code>
LocalTime	<code>type LocalTime = LocalTime</code>	A <code>Time</code> in the current <code>TimeZone</code> . For example: <code> 22:10:18 </code>

Type	Definition	Description
NaN	<code>type NaN = Null {NaN: true}</code>	<code>java.lang.Float</code> and <code>java.lang.Double</code> have special cases for <code>NaN</code> and <code>Infinit</code> . DataWeave does not have these concepts for its number multi-precision nature. So when it is mapped to DataWeave values, it is wrapped in a Null with a Schema marker.
Namespace	<code>type Namespace = Namespace</code>	A <code>Namespace</code> type represented by a <code>URI</code> and a prefix.
Nothing	<code>type Nothing = Nothing</code>	Bottom type. This type can be assigned to all the types.
Null	<code>type Null = Null</code>	A Null type, which represents the <code>null</code> value.
Number	<code>type Number = Number</code>	A number type: Any number, decimal, or integer is represented by the Number` type.
Object	<code>type Object = Object</code>	Type that represents any object, which is a collection of <code>Key</code> and value pairs.  Examples: <code>{ myKey : "a value" }, { myKey : { a : 1, b : 2} }, { myKey : [1,2,3,4] }</code>
Pair	<code>type Pair = { l: LEFT, r: RIGHT }</code>	A type used to represent a pair of values.  <i>Introduced in DataWeave version 2.2.0.</i>
Period	<code>type Period = Period</code>	A period.
Range	<code>type Range = Range</code>	A <code>Range</code> type represents a sequence of numbers.
Regex	<code>type Regex = Regex</code>	A Java regular expression (regex) type.
SimpleType	<code>type SimpleType = String   Boolean   Number   DateTime   LocalDateTime   Date   LocalTime   Time   TimeZone   Period</code>	A union type that represents all the simple types.
String	<code>type String = String</code>	<code>String</code> type

Type	Definition	Description
StringCoerceable	<code>type StringCoerceable = String   Boolean   Number   DateTime   LocalDateTime   Date   LocalTime   Time   TimeZone   Period   Key   Binary   Uri   Type&lt;Any&gt;   Regex   Namespace</code>	A union type of all the types that can be coerced to String type.  <i>Introduced in DataWeave version 2.3.0.</i>
Time	<code>type Time = Time</code>	A time in a specific <code>TimeZone</code> . For example: <code> 22:10:18Z </code>
TimeZone	<code>type TimeZone = TimeZone</code>	A time zone.
Type	<code>type Type = Type</code>	A type in the DataWeave type system.
Uri	<code>type Uri = Uri</code>	A URI.

## Core Namespaces

Namespace	Definition	Description
xsi	<code>xsi</code> <code>= http://www.w3.org/2001/XMLSchema-instance</code>	Namespace declaration of XMLSchema.

## Core Annotations

Annotation	Definition	Description
AnnotationTarget	<code>@AnnotationTarget(targets: Array&lt;"Function"   "Parameter"   "Variable"   "Import"&gt;)</code>	<p>Annotation that limits the application of an annotation. An example is <code>@AnnotationTarget(targets = ["Function", "Variable"])</code>, which limits the scope of the annotation <code>TailRec()</code> to functions and variables. If no <code>AnnotationTarget</code> is specified, an annotation can apply to any valid target.</p> <p>Annotation Targets:</p> <ul style="list-style-type: none"> <li>• <code>Parameter</code>: For function parameters.</li> <li>• <code>Function</code>: For function definitions.</li> <li>• <code>Variable</code>: For variable definitions.</li> <li>• <code>Import</code>: For import definitions.</li> </ul>

Annotation	Definition	Description
Deprecated	<code>@Deprecated(since: String, replacement: String)</code>	Annotation that marks a function as deprecated. <i>Introduced in DataWeave version 2.4.0.</i>
DesignOnlyType	<code>@DesignOnlyType()</code>	Annotation that marks a parameter type as <i>design only</i> to indicate that the field type is validated only at design time. At runtime, only minimal type validation takes place. This annotation is useful for performance, especially with complex Object types.
Experimental	<code>@Experimental()</code>	Annotation that identifies a feature as experimental and subject to change or removal in the future. <i>Introduced in DataWeave version 2.4.0.</i>
GlobalDescription	<code>@GlobalDescription()</code>	Annotation used to identify the function description to use for the function's documentation. This annotation is useful for selecting the correct function description when the function is overloaded. <i>Introduced in DataWeave version 2.4.0.</i>
Interceptor	<code>@Interceptor(interceptorFunction: String   (annotationArgs: Object, targetFunctionName: String, args: Array&lt;Any&gt;, callback: (args: Array&lt;Any&gt;) -&gt; Any) -&gt; Any)</code>	Annotation that marks another annotation as an Interceptor so that the marked annotation will wrap an annotated function with an <code>interceptorFunction</code> . An example is the <code>RuntimePrivilege</code> annotation, which is annotated by <code>@Interceptor(interceptorFunction = "@native system::SecurityManagerCheckFunctionValue")</code> . The <code>readUrl</code> function definition is annotated by <code>@RuntimePrivilege(requirements = "Resource")</code> . <i>Experimental:</i> This experimental feature is subject to change or removal from future versions of DataWeave.

Annotation	Definition	Description
Internal	@Internal(permits: Array<String>)	<p>Annotation that marks a function as <i>internal</i> and not to be used.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p style="margin: 0;">_Introduced in DataWeave 2.4.0.</p> <p style="margin: 0;">Supported by Mule 4.4.0 and later._</p> </div> <p><i>Experimental:</i> This experimental feature is subject to change or removal from future versions of DataWeave.</p>
Labels	@Labels(labels: Array<String>)	<p>Annotation for labeling a function or variable definition so that it becomes more easy to discover. An example is <code>@Labels(labels = ["append", "concat"])</code>.</p> <p><i>Introduced in DataWeave version 2.4.0.</i></p>
Lazy	@Lazy()	<p>Annotation that marks a variable declaration for lazy initialization.</p> <p><i>Introduced in DataWeave version 2.3.0.</i></p>
RuntimePrivilege	@RuntimePrivilege(requires: String)	<p>Annotation used to indicate that a function requires runtime privileges to execute. An example is <code>@RuntimePrivilege(requires = "Resource")</code>, which annotates the <code>readUrl</code> function definition.</p>
Since	@Since(version: String)	<p>Annotation that identifies the DataWeave version in which the annotated functionality was introduced. An example is <code>@Since(version = "2.4.0")</code>.</p> <p><i>Introduced in DataWeave 2.3.0. Supported by Mule 4.3 and later.</i></p>
StreamCapable	@StreamCapable()	<p>Annotation that marks a parameter as stream capable, which means that this field will consume an array of objects in a forward-only manner. Examples of functions with <code>@StreamCapable</code> fields are <code>map</code>, <code>mapObject</code>, and <code>pluck</code>.</p>

Annotation	Definition	Description
TailRec	<code>@TailRec()</code>	Annotation that marks a function as tail recursive. If a function with this annotation is not tail recursive, the function will fail.
UntrustedCode	<code>@UntrustedCode(privileges: Array&lt;String&gt;)</code>	Annotation that marks a script as untrusted, which means that the script has no privileges. For example, such a script cannot gain access to environment variables or read a resource from a URL.  <i>Experimental:</i> This experimental feature is subject to change or removal from future versions of DataWeave.

## dw::core::Arrays

This module contains helper functions for working with arrays.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Arrays` to the header of your DataWeave script.

### Functions

Name	Description
<code>countBy</code>	Counts the elements in an array that return <code>true</code> when the matching function is applied to the value of each element.
<code>divideBy</code>	Breaks up an array into sub-arrays that contain the specified number of elements.
<code>drop</code>	Drops the first <code>n</code> elements. It returns the original array when <code>n &lt;= 0</code> and an empty array when <code>n &gt; sizeOf(array)</code> .
<code>dropWhile</code>	Drops elements from the array while the condition is met but stops the selection process when it reaches an element that fails to satisfy the condition.
<code>every</code>	Returns <code>true</code> if every element in the array matches the condition.
<code>firstWith</code>	Returns the first element that satisfies the condition, or returns <code>null</code> if no element meets the condition.
<code>indexOf</code>	Returns the index of the first occurrence of an element within the array. If the value is not found, the function returns <code>-1</code> .
<code>indexWhere</code>	Returns the index of the first occurrence of an element that matches a condition within the array. If no element matches the condition, the function returns <code>-1</code> .

Name	Description
join	Joins two arrays of objects by a given ID criteria.
leftJoin	Joins two arrays of objects by a given ID criteria.
outerJoin	Joins two array of objects by a given ID criteria.
partition	Separates the array into the elements that satisfy the condition from those that do not.
slice	Selects the interval of elements that satisfy the condition: <code>from &lt;= indexOf(array) &lt; until</code>
some	Returns <code>true</code> if at least one element in the array matches the specified condition.
splitAt	Splits an array into two at a given position.
splitWhere	Splits an array into two at the first position where the condition is met.
sumBy	Returns the sum of the values of the elements in an array.
take	Selects the first <code>n</code> elements. It returns an empty array when <code>n &lt;= 0</code> and the original array when <code>n &gt; sizeOf(array)</code> .
takeWhile	Selects elements from the array while the condition is met but stops the selection process when it reaches an element that fails to satisfy the condition.

## countBy

**countBy<T>(array: Array<T>, matchingFunction: (T) -> Boolean): Number**

Counts the elements in an array that return `true` when the matching function is applied to the value of each element.

### Parameters

Name	Description
array	The input array that contains elements to match.
matchingFunction	A function to apply to elements in the input array.

### Example

This example counts the number of elements in the input array ([1, 2, 3, 4]) that return `true` when the function `((\$ mod 2) == 0)` is applied their values. In this case, the values of *two* of the elements, both 2 and 4, match because `2 mod 2 == 0` and `4 mod 2 == 0`. As a consequence, the `countBy` function returns 2. Note that `mod` returns the modulus of the operands.

### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
---
{ "countBy" : [1, 2, 3, 4] countBy (($ mod 2) == 0) }
```

## Output

```
{ "countBy": 2 }
```

### countBy(array: Null, matchingFunction: (Nothing) -> Any): Null

Helper function that enables `countBy` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## divideBy

### divideBy<T>(items: Array<T>, amount: Number): Array<Array<T>>

Breaks up an array into sub-arrays that contain the specified number of elements.

When there are fewer elements in the input array than the specified number, the function fills the sub-array with those elements. When there are more elements, the function fills as many sub-arrays needed with the extra elements.

#### Parameters

Name	Description
items	Items in the input array.
amount	The number of elements allowed per sub-array.

#### Example

This example breaks up arrays into sub-arrays based on the specified `amount`.

#### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
---
{
  "divideBy" : [
    { "divideBy2" : [1, 2, 3, 4, 5] divideBy 2 },
    { "divideBy2" : [1, 2, 3, 4, 5, 6] divideBy 2 },
    { "divideBy3" : [1, 2, 3, 4, 5] divideBy 3 }
  ]
}
```

## Output

```
{
  "divideBy": [
    {
      "divideBy2": [
        [ 1, 2 ],
        [ 3, 4 ],
        [ 5 ]
      ]
    },
    {
      "divideBy2": [
        [ 1, 2 ],
        [ 3, 4 ],
        [ 5, 6 ]
      ]
    },
    {
      "divideBy3": [
        [ 1, 2, 3 ],
        [ 4, 5 ]
      ]
    }
  ]
}
```

## divideBy(items: Null, amount: Any): Null

Helper function that enables `divideBy` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## drop

### drop<T>(array: Array<T>, n: Number): Array<T>

Drops the first `n` elements. It returns the original array when `n <= 0` and an empty array when `n > sizeOf(array)`.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>array</code>	The left array of elements.
<code>n</code>	The number of elements to take.

#### Example

This example returns an array that only contains the third element of the input array. It drops the first two elements from the output.

#### Source

```
%dw 2.0
import * from dw::core::Arrays
var users = ["Mariano", "Leandro", "Julian"]
output application/json
---
drop(users, 2)
```

#### Output

```
[ "Julian" ]
```

### drop(array: Null, n: Any): Null

Helper function that enables `drop` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

### dropWhile

#### dropWhile<T>(array: Array<T>, condition: (item: T) -> Boolean): Array<T>

Drops elements from the array while the condition is met but stops the selection process when it reaches an element that fails to satisfy the condition.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
array	The array of elements.
condition	The condition (or expression) used to match an element in the array.

## Example

This example returns an array that omits elements that are less than or equal to `2`. The last two elements (`2` and `1`) are included in the output array because the function stops dropping elements when it reaches the `3`, which is greater than `2`.

## Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var arr = [0,1,3,2,1]
---
arr dropWhile $ < 3
```

## Output

```
[
  3,
  2,
  1
]
```

## dropWhile(array: Null, condition: (item: Nothing) -> Any): Null

Helper function that enables `dropWhile` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## every

### every<T>(list: Array<T>, condition: (T) -> Boolean): Boolean

Returns `true` if every element in the array matches the condition.

The function stops iterating after the first negative evaluation of an element in the array.

## Parameters

Name	Description
list	The input array.
condition	A condition (or expression) to apply to elements in the input array.

## Example

This example applies a variety of expressions to the input arrays. The `$` references values of the elements.

## Source

```
%dw 2.0
import * from dw::core::Arrays
var arr0 = [] as Array<Number>
output application/json
---
{ "results" : [
    "ok" : [
        [1,1,1] every ($ == 1),
        [1] every ($ == 1)
    ],
    "err" : [
        [1,2,3] every ((log('should stop at 2 ==', $) mod 2) == 1),
        [1,1,0] every ($ == 1),
        [0,1,1,0] every (log('should stop at 0 ==', $) == 1),
        [1,2,3] every ($ == 1),
        arr0 every true,
    ]
]
}
```

## Output

```
{
  "results": [
    {
      "ok": [ true, true ]
    },
    {
      "err": [ false, false, false, false, false ]
    }
  ]
}
```

## **every(value: Null, condition: (Nothing) -> Any): Boolean**

Helper function that enables `every` to work with a `null` value.

*Introduced in DataWeave version 2.3.0.*

## **firstWith**

### **firstWith<T>(array: Array<T>, condition: (item: T, index: Number) -> Boolean): T | Null**

Returns the first element that satisfies the condition, or returns `null` if no element meets the condition.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
<code>array</code>	The array of elements to search.
<code>condition</code>	The condition to satisfy.

## Example

This example shows how `firstWith` behaves when an element matches and when an element does not match.

## Source

```
%dw 2.0
output application/json
import firstWith from dw::core::Arrays
var users = [{name: "Mariano", lastName: "Achaval"}, {name: "Ana", lastName: "Felisatti"}, {name: "Mariano", lastName: "de Sousa"}]
---
{
  a: users firstWith ((user, index) -> user.name == "Mariano"),
  b: users firstWith ((user, index) -> user.name == "Peter")
}
```

## Output

```
{
  "a": {
    "name": "Mariano",
    "lastName": "Achaval"
  },
  "b": null
}
```

## `firstWith(array: Null, condition: (item: Nothing, index: Nothing) -> Any): Null`

Helper function that enables `firstWith` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## indexOf

### indexOf<T>(array: Array<T>, toFind: T): Number

Returns the index of the first occurrence of an element within the array. If the value is not found, the function returns **-1**.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
array	The array of elements.
toFind	The element to find.

#### Example

This example returns the index of the matching value from the input array. The index of "Julian" is 2.

#### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var users = ["Mariano", "Leandro", "Julian"]
---
indexOf(users, "Julian")
```

#### Output

```
2
```

## indexWhere

### indexWhere<T>(array: Array<T>, condition: (item: T) -> Boolean): Number

Returns the index of the first occurrence of an element that matches a condition within the array. If no element matches the condition, the function returns **-1**.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
array	The array of elements.

Name	Description
condition	The condition (or expression) used to match an element in the array.

## Example

This example returns the index of the value from the input array that matches the condition in the lambda expression, `(item) -> item startsWith "Jul"`.

## Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var users = ["Mariano", "Leandro", "Julian"]
---
users indexWhere (item) -> item startsWith "Jul"
```

## Output

```
2
```

## indexWhere(array: Null, condition: (item: Nothing) -> Any): Null

Helper function that enables `indexWhere` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## join

`join<L <: Object, R <: Object>(left: Array<L>, right: Array<R>, leftCriteria: (leftValue: L) -> String, rightCriteria: (rightValue: R) -> String): Array<Pair<L, R>>`

Joins two arrays of objects by a given ID criteria.

`join` returns an array all the `left` items, merged by ID with any right items that exist.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
<code>left</code>	The left-side array of objects.
<code>right</code>	The right-side array of objects.
<code>leftCriteria</code>	The criteria used to extract the ID for the left collection.
<code>rightCriteria</code>	The criteria used to extract the ID for the right collection.

## Example

This example shows how join behaves. Notice that the output only includes objects where the values of the input `user.id` and `product.ownerId` match. The function includes the "`l`" and "`r`" keys in the output.

## Source

```
%dw 2.0
import * from dw::core::Arrays
var users = [{"id": "1", "name": "Mariano"}, {"id": "2", "name": "Leandro"}, {"id": "3", "name": "Julian"}, {"id": "5", "name": "Julian"}]
var products = [{"ownerId": "1", "name": "DataWeave"}, {"ownerId": "1", "name": "BAT"}, {"ownerId": "3", "name": "DataSense"}, {"ownerId": "4", "name": "SmartConnectors"}]
output application/json
---
join(users, products, (user) -> user.id, (product) -> product.ownerId)
```

## Output

```
[
  {
    "l": {
      "id": "1",
      "name": "Mariano"
    },
    "r": {
      "ownerId": "1",
      "name": "DataWeave"
    }
  },
  {
    "l": {
      "id": "1",
      "name": "Mariano"
    },
    "r": {
      "ownerId": "1",
      "name": "BAT"
    }
  },
  {
    "l": {
      "id": "3",
      "name": "Julian"
    },
    "r": {
      "ownerId": "3",
      "name": "DataSense"
    }
  }
]
```

## leftJoin

**leftJoin<L <: Object, R <: Object>(left: Array<L>, right: Array<R>, leftCriteria: (leftValue: L) -> String, rightCriteria: (rightValue: R) -> String): Array<{ l: L, r?: R }>**

Joins two arrays of objects by a given ID criteria.

**leftJoin** returns an array all the **left** items, merged by ID with any right items that meet the joining criteria.

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
<b>left</b>	The left-side array of objects.

Name	Description
right	The right-side array of objects.
leftCriteria	The criteria used to extract the ID for the left collection.
rightCriteria	The criteria used to extract the ID for the right collection.

## Example

This example shows how join behaves. Notice that it returns all objects from the left-side array (`left`) but only joins items from the right-side array (`right`) if the values of the left-side `user.id` and right-side `product.ownerId` match.

## Source

```
%dw 2.0
import * from dw::core::Arrays
var users = [{"id": "1", name:"Mariano"}, {"id": "2", name:"Leandro"}, {"id": "3", name:"Julian"}, {"id": "5", name:"Julian"}]
var products = [{"ownerId: "1", name:"DataWeave"}, {"ownerId: "1", name:"BAT"}, {"ownerId: "3", name:"DataSense"}, {"ownerId: "4", name:"SmartConnectors"}]
output application/json
---
leftJoin(users, products, (user) -> user.id, (product) -> product.ownerId)
```

## Output

```
[
  {
    "l": {
      "id": "1",
      "name": "Mariano"
    },
    "r": {
      "ownerId": "1",
      "name": "DataWeave"
    }
  },
  {
    "l": {
      "id": "1",
      "name": "Mariano"
    },
    "r": {
      "ownerId": "1",
      "name": "BAT"
    }
  },
  {
    "l": {
      "id": "2",
      "name": "Leandro"
    }
  },
  {
    "l": {
      "id": "3",
      "name": "Julian"
    },
    "r": {
      "ownerId": "3",
      "name": "DataSense"
    }
  },
  {
    "l": {
      "id": "5",
      "name": "Julian"
    }
  }
]
```

## outerJoin

**outerJoin<L <: Object, R <: Object>(left: Array<L>, right: Array<R>, leftCriteria: (leftValue: L) -> String, rightCriteria: (rightValue: R) -> String): Array<{ l?: L, r?: R }>**

Joins two array of objects by a given **ID** criteria.

**outerJoin** returns an array with all the **left** items, merged by ID with the **right** items in cases where any exist, and it returns **right** items that are not present in the **left**.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
<b>left</b>	The left-side array of objects.
<b>right</b>	The right-side array of objects.
<b>leftCriteria</b>	The criteria used to extract the ID for the left collection.
<b>rightCriteria</b>	The criteria used to extract the ID for the right collection.

## Example

This example shows how join behaves. Notice that the output includes objects where the values of the input `user.id` and `product.ownerId` match, and it includes objects where there is no match for the value of the `user.id` or `product.ownerId`.

## Source

```
%dw 2.0
import * from dw::core::Arrays
var users = [{"id": "1", "name": "Mariano"}, {"id": "2", "name": "Leandro"}, {"id": "3", "name": "Julian"}, {"id": "5", "name": "Julian"}]
var products = [{"ownerId": "1", "name": "DataWeave"}, {"ownerId": "1", "name": "BAT"}, {"ownerId": "3", "name": "DataSense"}, {"ownerId": "4", "name": "SmartConnectors"}]
output application/json
---
outerJoin(users, products, (user) -> user.id, (product) -> product.ownerId)
```

## Output

```
[  
  {  
    "l": {  
      "id": "1",  
      "name": "Mariano"  
    },  
    "r": {  
      "ownerId": "1",  
      "name": "DataWeave"  
    }  
  },  
  {  
    "l": {  
      "id": "1",  
      "name": "Mariano"  
    },  
    "r": {  
      "ownerId": "1",  
      "name": "BAT"  
    }  
  },  
  {  
    "l": {  
      "id": "2",  
      "name": "Leandro"  
    }  
  },  
  {  
    "l": {  
      "id": "3",  
      "name": "Julian"  
    },  
    "r": {  
      "ownerId": "3",  
      "name": "DataSense"  
    }  
  },  
  {  
    "l": {  
      "id": "5",  
      "name": "Julian"  
    }  
  },  
  {  
    "r": {  
      "ownerId": "4",  
      "name": "SmartConnectors"  
    }  
  }  
]
```

## partition

**partition<T>(array: Array<T>, condition: (item: T) -> Boolean): { success: Array<T>, failure: Array<T> }**

Separates the array into the elements that satisfy the condition from those that do not.

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
array	The array of elements to split.
condition	The condition (or expression) used to match an element in the array.

### Example

This example partitions numbers found within an input array. The even numbers match the criteria set by the lambda expression `(item) -> isEven(item)`. The odd do not. The function generates the "success" and "failure" keys within the output object.

### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var arr = [0,1,2,3,4,5]
---
arr partition (item) -> isEven(item)
```

### Output

```
{
  "success": [
    0,
    2,
    4
  ],
  "failure": [
    1,
    3,
    5
  ]
}
```

## partition(array: Null, condition: (item: Nothing) -> Any): Null

Helper function that enables `partition` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## **slice**

**slice<T>(array: Array<T>, from: Number, until: Number): Array<T>**

Selects the interval of elements that satisfy the condition: `from <= indexOf(array) < until`

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
<code>array</code>	The array of elements.
<code>from</code>	The starting index of the interval of elements to include from the array. If this value is negative, the function starts including from the first element of the array. If this value is higher than the last index of the array, the function returns an empty array ( <code>[]</code> ).
<code>until</code>	The ending index of the interval of elements to include from the array. If this value is higher than the last index of the array, the function includes up to the last element of the array. If this value is lower than the first index of the array, the function returns an empty array ( <code>[]</code> ).

### Example

This example returns an array that contains the values of indices 1, 2, and 3 from the input array. It excludes the values of indices 0, 4, and 5.

### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var arr = [0,1,2,3,4,5]
---
slice(arr, 1, 4)
```

### Output

```
[1,
 2,
 3]
```

**slice(array: Null, from: Any, until: Any): Null**

Helper function that enables `slice` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## some

### `some<T>(list: Array<T>, condition: (T) -> Boolean): Boolean`

Returns `true` if at least one element in the array matches the specified condition.

The function stops iterating after the first element that matches the condition is found.

#### Parameters

Name	Description
<code>list</code>	The input array.
<code>condition</code>	A condition (or expression) used to match elements in the array.

#### Example

This example applies a variety of expressions to elements of several input arrays. The `$` in the condition is the default parameter for the current element of the array that the condition evaluates. Note that you can replace the default `$` parameter with a lambda expression that contains a named parameter for the current array element.

#### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
---
{ "results" : [
    "ok" : [
        [1,2,3] some (($ mod 2) == 0),
        [1,2,3] some ((nextNum) -> (nextNum mod 2) == 0),
        [1,2,3] some (($ mod 2) == 1),
        [1,2,3,4,5,6,7,8] some (log('should stop at 2 ==', $) == 2),
        [1,2,3] some ($ == 1),
        [1,1,1] some ($ == 1),
        [1] some ($ == 1)
    ],
    "err" : [
        [1,2,3] some ($ == 100),
        [1] some ($ == 2)
    ]
]
}
```

#### Output

```
{  
  "results": [  
    {  
      "ok": [ true, true, true, true, true, true, true ]  
    },  
    {  
      "err": [ false, false ]  
    }  
  ]  
}
```

## some(list: Null, condition: (Nothing) -> Any): Boolean

Helper function that enables `some` to work with a `null` value.

*Introduced in DataWeave version 2.3.0.*

## splitAt

### splitAt<T>(array: Array<T>, n: Number): Pair<Array<T>, Array<T>>

Splits an array into two at a given position.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
array	The array of elements.
n	The index at which to split the array.

#### Example

#### Source

```
%dw 2.0  
import * from dw::core::Arrays  
output application/json  
var users = ["Mariano", "Leandro", "Julian"]  
---  
users splitAt 1
```

#### Output

```
{
  "l": [
    "Mariano"
  ],
  "r": [
    "Leandro",
    "Julian"
  ]
}
```

## splitAt(array: Null, n: Any): Null

Helper function that enables `splitAt` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## splitWhere

### splitWhere<T>(array: Array<T>, condition: (item: T) -> Boolean): Pair<Array<T>, Array<T>>

Splits an array into two at the first position where the condition is met.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>array</code>	The array of elements to split.
<code>condition</code>	The condition (or expression) used to match an element in the array.

#### Example

#### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var users = ["Mariano", "Leandro", "Julian", "Tomo"]
---
users splitWhere (item) -> item startsWith "Jul"
```

#### Output

```
{
  "l": [
    "Mariano",
    "Leandro"
  ],
  "r": [
    "Julian",
    "Tomo"
  ]
}
```

## splitWhere(array: Null, condition: (item: Nothing) -> Any): Null

Helper function that enables `splitWhere` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## sumBy

### sumBy<T>(array: Array<T>, numberSelector: (T) -> Number): Number

Returns the sum of the values of the elements in an array.

#### Parameters

Name	Description
<code>array</code>	The input array.
<code>numberSelector</code>	A DataWeave selector that selects the values of the numbers in the input array.

#### Example

This example calculates the sum of the values of elements some arrays. Notice that both of the `sumBy` function calls produce the same result.

#### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
---
{
  "sumBy" : [
    [ { a: 1 }, { a: 2 }, { a: 3 } ] sumBy $.a,
    sumBy([ { a: 1 }, { a: 2 }, { a: 3 } ], (item) -> item.a)
  ]
}
```

## Output

```
{ "sumBy" : [ 6, 6 ] }
```

## sumBy(array: Null, numberSelector: (Nothing) -> Any): Null

Helper function that enables `sumBy` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## take

### take<T>(array: Array<T>, n: Number): Array<T>

Selects the first `n` elements. It returns an empty array when `n <= 0` and the original array when `n > sizeOf(array)`.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>array</code>	The array of elements.
<code>n</code>	The number of elements to select.

#### Example

This example outputs an array that contains the values of first two elements of the input array.

#### Source

```
%dw 2.0
import * from dw::core::Arrays
var users = ["Mariano", "Leandro", "Julian"]
output application/json
---
take(users, 2)
```

## Output

```
[ "Mariano", "Leandro" ]
```

## take(array: Null, n: Any): Null

Helper function that enables `take` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## takeWhile

**takeWhile<T>(array: Array<T>, condition: (item: T) -> Boolean): Array<T>**

Selects elements from the array while the condition is met but stops the selection process when it reaches an element that fails to satisfy the condition.

To select all elements that meet the condition, use the `filter` function.

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
<code>array</code>	The array of elements.
<code>condition</code>	The condition (or expression) used to match an element in the array.

### Example

This example iterates over the elements in the array and selects only those with an index that is `<= 1` and stops selecting elements when it reaches one that is greater than `2`. Notice that it does not select the second `1` because of the `2` that precedes it in the array. The function outputs the result into an array.

### Source

```
%dw 2.0
import * from dw::core::Arrays
output application/json
var arr = [0,1,2,1]
---
arr takeWhile $ <= 1
```

### Output

```
[0,1]
```

**takeWhile(array: Null, condition: (item: Nothing) -> Any): Null**

Helper function that enables `takeWhile` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## dw::core::Binaries

This module contains helper functions for working with binaries.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Binaries` to the header of your DataWeave script.

## Functions

Name	Description
<code>concatWith</code>	Concatenates the content of two binaries.
<code>fromBase64</code>	Transforms a Base64 string into a binary value.
<code>fromHex</code>	Transforms a hexadecimal string into a binary.
<code>readLinesWith</code>	Splits the specified binary content into lines and returns the results in an array.
<code>toBase64</code>	Transforms a binary value into a Base64 string.
<code>toHex</code>	Transforms a binary value into a hexadecimal string.
<code>writeLinesWith</code>	Writes the specified lines and returns the binary content.

### concatWith

`concatWith(source: Binary, with: Binary): Binary`

Concatenates the content of two binaries.

*Introduced in DataWeave version 2.5.0.*

#### Parameters

Name	Type	Description
<code>source</code>	Binary	The source binary content.
<code>with</code>	Binary	The binary to append.

#### Example

This example concats two binaries into one binary.

#### Source

```
%dw 2.0
import * from dw::core::Binaries
output application/dw
---
"CAFE" as Binary {base: "16"} concatWith "ABCD" as Binary {base: "16"}
```

## Output

```
"yv6rzQ==" as Binary {base: "64"}
```

### **concatWith(source: Binary, with: Null): Binary**

Helper function that enables `concatWith` to work with a `null` value.

### **concatWith(source: Null, with: Binary): Binary**

Helper function that enables `concatWith` to work with a `null` value.

## fromBase64

### **fromBase64(base64String: String): Binary**

Transforms a Base64 string into a binary value.

#### Parameters

Name	Description
<code>base64String</code>	The Base64 string to transform.

#### Example

This example takes a Base64 encoded string and transforms it into a binary value. This example assumes that the `payload` contains the Base64 string generated from an image in example [Example](#). The output of this function is a binary value that represents the image generated in example [toBase64](#).

#### Source

```
%dw 2.0
import * from dw::core::Binaries
output application/octet-stream
---
fromBase64(payload)
```

## fromHex

### fromHex(hexString: String): Binary

Transforms a hexadecimal string into a binary.

#### Parameters

Name	Description
hexString	A hexadecimal string to transform.

#### Example

This example transforms a hexadecimal string to "Mule". To make the resulting type clear, it outputs data in the `application/dw` format.

#### Source

```
%dw 2.0
import * from dw::core::Binaries
output application/dw
---
{ "hexToBinary": fromHex("4D756C65") }
```

#### Output

```
{
  hexToBinary: "TXVsZQ==" as Binary {base: "64"}
}
```

## readLinesWith

### readLinesWith(content: Binary, charset: String): Array<String>

Splits the specified binary content into lines and returns the results in an array.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
content	Binary data to read and split.
charset	String representing the encoding to read.

#### Example

This example transforms binary content, which is separated into new lines (`\n`), in a comma-

separated array.

## Source

```
%dw 2.0
import * from dw::core::Binaries
var content = read("Line 1\nLine 2\nLine 3\nLine 4\nLine 5\n", "application/octet-
stream")
output application/json
---
{
  lines : (content readLinesWith "UTF-8"),
  showType: typeOf(content)
}
```

## Output

```
{
  "lines": [ "Line 1", "Line 2", "Line 3", "Line 4", "Line 5" ],
  "showType": "Binary"
}
```

## toBase64

### toBase64(content: Binary): String

Transforms a binary value into a Base64 string.

#### Parameters

Name	Description
content	The binary value to transform.

#### Example

This example transforms a binary value into a Base64 encoded string. In this case, the binary value represents an image.

## Source

```
%dw 2.0

import dw::Crypto
import toBase64 from dw::core::Binaries

var emailChecksum = Crypto::MD5("achaval@gmail.com" as Binary)
var image = readUrl(log("https://www.gravatar.com/avatar/${emailChecksum}"),
"application/octet-stream")

output application/json
---
toBase64(image)
```

## Output

This example outputs a Base64 encoded string. The resulting string was shortened for readability purposes:

```
"/9j/4AAQSkZJRgABAQEAYABgAAD//..."
```

## toHex

### **toHex(content: Binary): String**

Transforms a binary value into a hexadecimal string.

#### Parameters

Name	Description
content	The <b>Binary</b> value to transform.

#### Example

This example transforms a binary version of "Mule" (defined in the variable, **myBinary**) to hexadecimal.

#### Source

```
%dw 2.0
import * from dw::core::Binaries
output application/json
var myBinary = "Mule" as Binary
var testType = typeOf(myBinary)
---
{
    "binaryToHex" : toHex(myBinary)
}
```

## Output

```
{ "binaryToHex": "4D756C65" }
```

## writeLinesWith

**writeLinesWith(content: Array<String>, charset: String): Binary**

Writes the specified lines and returns the binary content.

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
content	Array of items to write.
charset	String representing the encoding to use when writing.

### Example

This example inserts a new line (`\n`) after each iteration. Specifically, it uses `map` to iterate over the result of `to(1, 10), [1,2,3,4,5]`, then writes the specified content ("Line \$"), which includes the unnamed variable `$` for each number in the array.

Note that without `writeLinesWith "UTF-8"`, the expression `{ lines: to(1, 10) map "Line $" }` simply returns an array of line numbers as the value of an object: `{ "lines": [ "line 1", "line 2", "line 3", "line 4", "line 5" ] }`.

### Source

```
%dw 2.0
import * from dw::core::Binaries
output application/json
---
{ lines: to(1, 10) map "Line $" writeLinesWith "UTF-8" }
```

## Output

```
{  
  "lines": "Line 1\nLine 2\nLine 3\nLine 4\nLine 5\n"
```

# dw::core::Dates

This module contains functions for creating and manipulating dates.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Dates` to the header of your DataWeave script.

*Introduced in DataWeave version 2.4.0.*

## Functions

Name	Description
<code>atBeginningOfDay</code>	Returns a new <code>DateTime</code> value that changes the <code>Time</code> value in the input to the beginning of the specified <i>day</i> .
<code>atBeginningOfHour</code>	Returns a new <code>DateTime</code> value that changes the <code>Time</code> value in the input to the beginning of the specified <i>hour</i> .
<code>atBeginningOfMonth</code>	Returns a new <code>DateTime</code> value that changes the <code>Day</code> value from the input to the first day of the specified <i>month</i> . It also sets the <code>Time</code> value to <code>00:00:00</code> .
<code>atBeginningOfWeek</code>	Returns a new <code>DateTime</code> value that changes the <code>Day</code> and <code>Time</code> values from the input to the beginning of the first day of the specified <i>week</i> .
<code>atBeginningOfYear</code>	Takes a <code>DateTime</code> value as input and returns a <code>DateTime</code> value for the first day of the <i>year</i> specified in the input. It also sets the <code>Time</code> value to <code>00:00:00</code> .
<code>date</code>	Creates a <code>Date</code> value from values specified for <code>year</code> , <code>month</code> , and <code>day</code> fields.
<code>dateTime</code>	Creates a <code>DateTime</code> value from values specified for <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minutes</code> , <code>seconds</code> , and <code>timezone</code> fields.
<code>localDateTime</code>	Creates a <code>LocalDateTime</code> value from values specified for <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minutes</code> , and <code>seconds</code> fields.
<code>localTime</code>	Creates a <code>LocalTime</code> value from values specified for <code>hour</code> , <code>minutes</code> , and <code>seconds</code> fields.
<code>time</code>	Creates a <code>Time</code> value from values specified for <code>hour</code> , <code>minutes</code> , <code>seconds</code> , and <code>timezone</code> fields.
<code>today</code>	Returns the date for today as a <code>Date</code> type.
<code>tomorrow</code>	Returns the date for tomorrow as a <code>Date</code> type.
<code>yesterday</code>	Returns the date for yesterday as a <code>Date</code> type.

## Types

- [Dates Types](#)

## atBeginningOfDay

### atBeginningOfDay(dateTime: DateTime): DateTime

Returns a new **DateTime** value that changes the **Time** value in the input to the beginning of the specified *day*.

The hours, minutes, and seconds in the input change to **00:00:00**.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
dateTime	The <b>DateTime</b> value to reference.

#### Example

This example changes the **Time** value within the **DateTime** input to the beginning of the specified day.

#### Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  "atBeginningOfDayDateTime": atBeginningOfDay(|2020-10-06T18:23:20.351-03:00|)
}
```

#### Output

```
{
  "atBeginningOfDayDateTime": "2020-10-06T00:00:00-03:00"
}
```

### atBeginningOfDay(localDateTime: LocalDateTime): LocalDateTime

Returns a new **LocalDateTime** value that changes the **Time** value within the input to the start of the specified *day*.

The hours, minutes, and seconds in the input change to **00:00:00**.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
localDateTime	The <code>LocalDateTime</code> value to reference.

## Example

This example changes the `Time` value within the `LocalDateTime` input to the beginning of the specified day.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  "atBeginningOfDayLocalDateTime": atBeginningOfDay(|2020-10-06T18:23:20.351|)
}
```

## Output

```
{
  "atBeginningOfDayLocalDateTime": "2020-10-06T00:00:00"
}
```

## atBeginningOfHour

### atBeginningOfHour(dateTime: DateTime): DateTime

Returns a new `DateTime` value that changes the `Time` value in the input to the beginning of the specified *hour*.

The minutes and seconds in the input change to `00:00`.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
dateTime	The <code>DateTime</code> value to reference.

## Example

This example changes the `Time` value within the `DateTime` input to the beginning of the specified *hour*.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    "atBeginningOfHourDateTime": atBeginningOfHour(|2020-10-06T18:23:20.351-03:00|)
}
```

## Output

```
{
    "atBeginningOfHourDateTime": "2020-10-06T18:00:00-03:00"
}
```

### atBeginningOfHour(localDateTime: LocalDateTime): LocalDateTime

Returns a new `LocalDateTime` value that changes the `Time` value in the input to the beginning of the specified *hour*.

The minutes and seconds in the input change to `00:00`.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>localDateTime</code>	The <code>LocalDateTime</code> value to reference.

#### Example

This example changes the `Time` value within the `LocalDateTime` input to the beginning of the specified hour.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    "atBeginningOfHourLocalDateTime": atBeginningOfHour(|2020-10-06T18:23:20.351|)
```

## Output

```
{  
    "atBeginningOfHourLocalDateTime": "2020-10-06T18:00:00"  
}
```

## atBeginningOfHour(localTime: LocalTime): LocalTime

Returns a new [LocalTime](#) value that changes its value in the input to the beginning of the specified *hour*.

The minutes and seconds in the input change to [00:00](#).

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
localTime	The <a href="#">LocalTime</a> value to reference.

### Example

This example changes the [LocalTime](#) value to the beginning of the specified hour.

### Source

```
%dw 2.0  
import * from dw::core::Dates  
output application/json  
---  
{  
    "atBeginningOfHourLocalTime": atBeginningOfHour(|18:23:20.351|)  
}
```

### Output

```
{  
    "atBeginningOfHourLocalTime": "18:00:00"  
}
```

## atBeginningOfHour(time: Time): Time

Returns a new [Time](#) value that changes the input value to the beginning of the specified *hour*.

The minutes and seconds in the input change to [00:00](#).

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
time	The <b>Time</b> value to reference.

## Example

This example changes the **Time** value to the beginning of the specified hour.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    "atBeginningOfHourTime": atBeginningOfHour(|18:23:20.351-03:00|)
}
```

## Output

```
{
    "atBeginningOfHourTime": "18:00:00-03:00"
}
```

# atBeginningOfMonth

## atBeginningOfMonth(dateTime: DateTime): DateTime

Returns a new **DateTime** value that changes the **Day** value from the input to the first day of the specified *month*. It also sets the **Time** value to **00:00:00**.

The day and time in the input changes to **01T00:00:00**.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
dateTime	The <b>DateTime</b> value to reference.

## Example

This example changes the **Day** value within the **DateTime** input to the first day of the specified month and sets the **Time** value to **00:00:00**.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  "atBeginningOfMonthDateTime": atBeginningOfMonth(|2020-10-06T18:23:20.351-03:00|)
}
```

## Output

```
{
  "atBeginningOfMonthDateTime": "2020-10-01T00:00:00-03:00"
}
```

### atBeginningOfMonth(localDateTime: LocalDateTime): LocalDateTime

Returns a new **LocalDateTime** value that changes the **Day** and **LocalTime** values from the input to the beginning of the specified *month*.

The day and time in the input changes to **01T00:00:00**.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
<b>localDateTime</b>	The <b>LocalDateTime</b> value to reference.

## Example

This example changes the **Day** and **LocalTime** values within the **LocalDateTime** input to the beginning of the specified month.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  "atBeginningOfMonthLocalDateTime": atBeginningOfMonth(|2020-10-06T18:23:20.351|)
```

## Output

```
{  
    "atBeginningOfMonthLocalDateTime": "2020-10-01T00:00:00"  
}
```

## atBeginningOfMonth(date: Date): Date

Returns a new **Date** value that changes the **Day** value from the input to the first day of the specified *month*.

The day in the input changes to **01**.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<b>date</b>	The <b>Date</b> value to reference.

### Example

This example changes the **Day** value within the **Date** input to the first day of the specified month.

### Source

```
%dw 2.0  
import * from dw::core::Dates  
output application/json  
---  
{  
    atBeginningOfMonthDate: atBeginningOfMonth(|2020-10-06|)  
}
```

### Output

```
{  
    "atBeginningOfMonthDate": "2020-10-01"  
}
```

## atBeginningOfWeek

### atBeginningOfWeek(dateTime: DateTime): DateTime

Returns a new **DateTime** value that changes the **Day** and **Time** values from the input to the beginning of the first day of the specified *week*.

The function treats Sunday as the first day of the week.

Introduced in DataWeave version 2.4.0.

## Parameters

Name	Description
dateTime	The <code>DateTime</code> value to reference.

## Example

This example changes the `Day` and `Time` values (`06T18:23:20.351`) within the `DateTime` input to the beginning of the first day of the specified `week` (`04T00:00:00`).

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  atBeginningOfWeekDateTime: atBeginningOfWeek(|2020-10-06T18:23:20.351-03:00|)
}
```

## Output

```
{
  "atBeginningOfWeekDateTime": "2020-10-04T00:00:00-03:00"
}
```

## atBeginningOfWeek(localDateTime: LocalDateTime): LocalDateTime

Returns a new `LocalDateTime` value that changes the `Day` and `Time` values from the input to the beginning of the first day of the specified `week`.

The function treats Sunday as the first day of the week.

Introduced in DataWeave version 2.4.0.

## Parameters

Name	Description
localDateTime	The <code>LocalDateTime</code> value to reference.

## Example

This example changes the `Day` and `Time` values (`06T18:23:20.351`) within the `LocalDateTime` input to the beginning of the first day of the specified `week` (`04T00:00:00`).

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  atBeginningOfWeekLocalDateTime: atBeginningOfWeek(|2020-10-06T18:23:20.351|)
}
```

## Output

```
{
  "atBeginningOfWeekLocalDateTime": "2020-10-04T00:00:00"
}
```

### atBeginningOfWeek(date: Date): Date

Returns a new **Date** value that changes the **Date** input input to the first day of the specified *week*.

The function treats Sunday as the first day of the week.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
date	The <b>Date</b> value to reference.

#### Example

This example changes the **Day** value (**06**) within the **Date** input to the first day of the week that contains **2020-10-06** (a Tuesday), which is **2020-10-04** (a Sunday). The **Day** value changes from **06** to **04**.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  atBeginningOfWeekDate: atBeginningOfWeek(|2020-10-06|)
}
```

## Output

```
{  
    "atBeginningOfWeekDate": "2020-10-04"  
}
```

## atBeginningOfYear

### atBeginningOfYear(dateTime: DateTime): DateTime

Takes a **DateTime** value as input and returns a **DateTime** value for the first day of the *year* specified in the input. It also sets the **Time** value to **00:00:00**.

The month, day, and time in the input changes to **01-01T00:00:00**.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
dateTime	The <b>DateTime</b> value to reference.

#### Example

This example transforms the **DateTime** input (**|2020-10-06T18:23:20.351-03:00|**) to the date of the first day of the **Year** value (**2020**) in the input.

#### Source

```
%dw 2.0  
import * from dw::core::Dates  
output application/json  
---  
{  
    atBeginningOfYearDateTime: atBeginningOfYear(|2020-10-06T18:23:20.351-03:00|)  
}
```

#### Output

```
{  
    "atBeginningOfYearDateTime": "2020-01-01T00:00:00.000-03:00"  
}
```

### atBeginningOfYear(localDateTime: LocalDateTime): LocalDateTime

Takes a **LocalDateTime** value as input and returns a **LocalDateTime** value for the first day of the *year* specified in the input. It also sets the **Time** value to **00:00:00**.

The month, day, and time in the input changes to **01-01T00:00:00**.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<b>localDateTime</b>	The <b>LocalDateTime</b> value to reference.

#### Example

This example transforms the **LocalDateTime** input (**|2020-10-06T18:23:20.351|**) to the date of the first day of the **Year** value (**2020**) in the input.

#### Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    atBeginningOfYearLocalDateTime: atBeginningOfYear(|2020-10-06T18:23:20.351|)
}
```

#### Output

```
{
    "atBeginningOfYearLocalDateTime": "2020-01-01T00:00:00"
}
```

### atBeginningOfYear(date: Date): Date

Takes a **Date** value as input and returns a **Date** value for the first day of the *year* specified in the input.

The month and day in the input changes to **01-01**.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<b>date</b>	The <b>Date</b> value to reference.

#### Example

This example transforms **Date** input (|2020-10-06|) to the date of the first day of the **Year** value (2020) in the input.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    atBeginningOfYearDate: atBeginningOfYear(|2020-10-06|)
}
```

## Output

```
{
    "atBeginningOfYearDate": "2020-01-01"
}
```

# date

## date(parts: DateFactory): Date

Creates a **Date** value from values specified for **year**, **month**, and **day** fields.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
parts	Number values for <b>year</b> , <b>month</b> , and <b>day</b> fields. The <b>month</b> must be a value between 1 and 12, and the <b>day</b> value must be between 1 and 31. You can specify the name-value pairs in any order, but the output is ordered by default as a Date value, such as 2012-10-11. The input fields are parts of a <b>DateFactory</b> type.

## Example

This example shows how to create a value of type **Date**.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    newDate: date({year: 2012, month: 10, day: 11})
}
```

## Output

```
{
    "newDate": "2012-10-11"
}
```

## dateTime

### dateTime(parts: DateTimeFactory): DateTime

Creates a `DateTime` value from values specified for `year`, `month`, `day`, `hour`, `minutes`, `seconds`, and `timezone` fields.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
parts	Number values for <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minutes</code> , and <code>seconds</code> fields followed by a <code>TimeZone</code> value for the <code>timezone</code> field. Valid values are numbers between 1 and 12 for the <code>month</code> , 1 through 31 for the <code>day</code> , 0 through 23 for the <code>hour</code> , 0 through 59 for <code>minutes</code> , and 0 through 59 (including decimals, such as 59.99) for seconds. You can specify the name-value pairs in any order, but the output is ordered by default as a <code>DateTime</code> value, such as <code>2012-10-11T10:10:10-03:00</code> . The input fields are parts of a <code>DateTimeFactory</code> type.

#### Example

This example shows how to create a value of type `DateTime`.

#### Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    newDateTime: dateTime({year: 2012, month: 10, day: 11, hour: 12, minutes: 30,
seconds: 40 , timeZone: |-03:00|})
}
```

## Output

```
{
    "newDateTime": "2012-10-11T12:30:40-03:00"
}
```

## localDateTime

### localDateTime(parts: LocalDateTimeFactory): LocalDateTime

Creates a `LocalDateTime` value from values specified for `year`, `month`, `day`, `hour`, `minutes`, and `seconds` fields.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>parts</code>	Number values for <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minutes</code> , and <code>seconds</code> fields. Valid values are numbers between 1 and 12 for the <code>month</code> , 1 through 31 for the <code>day</code> , 0 through 23 for the <code>hour</code> , 0 through 59 for <code>minutes</code> , and 0 through 59 (including decimals, such as 59.99) for <code>seconds</code> fields. You can specify the name-value pairs in any order, but the output is ordered as a default <code>LocalDateTime</code> value, such as <code>2012-10-11T10:10:10</code> . The input fields are parts of a <code>LocalDateTimeFactory</code> type.

#### Example

This example shows how to create a value of type `LocalDateTime`.

#### Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    newLocalDateTime: localDateTime({year: 2012, month: 10, day: 11, hour: 12,
minutes: 30, seconds: 40})
}
```

## Output

```
{
    "newLocalDateTime": "2012-10-11T12:30:40"
}
```

## localTime

### localTime(parts: LocalTimeFactory): LocalTime

Creates a [LocalTime](#) value from values specified for [hour](#), [minutes](#), and [seconds](#) fields.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
parts	Number values for <a href="#">hour</a> , <a href="#">minutes</a> , and <a href="#">seconds</a> fields. Valid values are 0 through 23 for the <a href="#">hour</a> , 0 through 59 for <a href="#">minutes</a> , and 0 through 59 (including decimals, such as 59.99) for <a href="#">seconds</a> fields. You can specify the name-value pairs in any order, but the output is ordered as a default <a href="#">LocalTime</a> value, such as <a href="#">10:10:10</a> . The input fields are parts of a <a href="#">LocalDateTimeFactory</a> type.

#### Example

This example shows how to create a value of type [LocalTime](#).

#### Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
    newLocalTime: localTime({ hour: 12, minutes: 30, seconds: 40})
}
```

## Output

```
{
    "newLocalTime": "12:30:40"
}
```

## time

### time(parts: TimeFactory): Time

Creates a **Time** value from values specified for **hour**, **minutes**, **seconds**, and **timezone** fields.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
parts	Number values for <b>hour</b> , <b>minutes</b> , and <b>seconds</b> fields, and a <b>TimeZone</b> value for the <b>timezone</b> field. Valid values are 0 through 23 for the <b>hour</b> , 0 through 59 for <b>minutes</b> , and 0 through 59 (including decimals, such as 59.99) for <b>seconds</b> fields. The <b>timezone</b> must be a valid <b>TimeZone</b> value, such as <b>-03:00</b> . You can specify the name-value pairs in any order, but the output is ordered as a default <b>Time</b> value, such as <b>10:10:10-03:00</b> . The input fields are parts of a <b>TimeFactory</b> type.

#### Example

This example shows how to create a value of type **Time**.

#### Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
{
  newTime: time({ hour: 12, minutes: 30, seconds: 40 , timeZone: |-03:00| })
}
```

## Output

```
{
  "newTime": "12:30:40-03:00"
}
```

# today

## today(): Date

Returns the date for today as a [Date](#) type.

*Introduced in DataWeave version 2.4.0.*

## Example

This example shows the output of `today` function.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
today()
```

## Output

```
"2021-05-15"
```

# tomorrow

## tomorrow(): Date

Returns the date for tomorrow as a [Date](#) type.

*Introduced in DataWeave version 2.4.0.*

## Example

This example shows the output of `tomorrow` function.

## Source

```
%dw 2.0
import tomorrow from dw::core::Dates
output application/json
---
tomorrow()
```

## Output

```
"2021-05-16"
```

# yesterday

## yesterday(): Date

Returns the date for yesterday as a `Date` type.

*Introduced in DataWeave version 2.4.0.*

## Example

This example shows the output of `yesterday` function.

## Source

```
%dw 2.0
import * from dw::core::Dates
output application/json
---
yesterday()
```

## Output

```
"2021-05-14"
```

# Dates Types

Type	Definition	Description
DateFactory	<pre>type DateFactory = { day: Number, month: Number, year: Number }</pre>	Type containing selectable <code>day</code> , <code>month</code> , and <code>year</code> keys and corresponding <code>Number</code> values, such as <code>{day: 21, month: 1, year: 2021}</code> . The fields accept a <code>Number</code> value. Numbers preceded by <code>0</code> , such as <code>01</code> , are not valid.
DateTimeFactory	<pre>type DateTimeFactory = DateFactory &amp; LocalTimeFactory &amp; Zoned</pre>	Type that combines <code>DateFactory</code> , <code>LocalTimeFactory</code> , and <code>Zoned</code> types. For example, <code>{day: 21, month: 1, year: 2021, hour: 8, minutes: 31, seconds: 55, timeZone :  -03:00 }</code> as <code>DateTimeFactory</code> is a valid <code>DateTimeFactory</code> value.
LocalDateTimeFactory	<pre>type LocalDateTimeFactory = DateFactory &amp; LocalTimeFactory</pre>	Type that combines <code>DateFactory</code> and <code>LocalTimeFactory</code> types. For example, <code>{day: 21, month: 1, year: 2021, hour: 8, minutes: 31, seconds: 55, timeZone :  -03:00 }</code> as <code>LocalDateTimeFactory</code> is a valid <code>LocalDateTimeFactory</code> value. The <code>timeZone</code> field is optional.
LocalTimeFactory	<pre>type LocalTimeFactory = { hour: Number, minutes: Number, seconds: Number }</pre>	Type containing selectable <code>hour</code> , <code>minutes</code> , and <code>seconds</code> keys and corresponding <code>Number</code> values, such as <code>{hour: 8, minutes: 31, seconds: 55}</code> . The fields accept any <code>Number</code> value.
TimeFactory	<pre>type TimeFactory = LocalTimeFactory &amp; Zoned</pre>	Type that combines <code>LocalTimeFactory</code> and <code>Zoned</code> types. For example, <code>{hour: 8, minutes: 31, seconds: 55, timeZone :  -03:00 }</code> as <code>TimeFactory</code> is a valid <code>TimeFactory</code> value.
Zoned	<pre>type Zoned = { timeZone: TimeZone }</pre>	Type containing a selectable <code>timeZone</code> key and <code>TimeZone</code> value, such as <code>{ timezone :  -03:00 }</code> .

## dw::core::Numbers

This module contains helper functions for working with numbers.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Numbers` to the header of your DataWeave script.

*Introduced in DataWeave version 2.2.0.*

## Functions

Name	Description
<a href="#">fromBinary</a>	Transforms from a binary number into a decimal number.
<a href="#">fromHex</a>	Transforms a hexadecimal number into decimal number.
<a href="#">fromRadixNumber</a>	Transforms a number in the specified radix into decimal number
<a href="#">toBinary</a>	Transforms a decimal number into a binary number.
<a href="#">toHex</a>	Transforms a decimal number into a hexadecimal number.
<a href="#">toRadixNumber</a>	Transforms a decimal number into a number string in other radix.

## fromBinary

**fromBinary(binaryText: String): Number**

Transforms from a binary number into a decimal number.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
binaryText	The binary number represented in a <i>String</i> .

## Example

This example shows how the `toBinary` behaves with different inputs.

## Source

## Output

**fromBinary(binaryText: Null): Null**

Helper function that enables `fromBinary` to work with null value.

*Introduced in DataWeave version 2.2.0.*

## fromHex

**fromHex(hexText: String): Number**

Transforms a hexadecimal number into decimal number.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
hexText	The hexadecimal number represented in a <a href="#">String</a> .

## Example

This example shows how the `toBinary` behaves with different inputs.

## Source

```
%dw 2.0
import fromHex from dw::core::Numbers
output application/json
---
{
  a: fromHex("-1"),
  b: fromHex("3e3aeb4ae1383562f4b82261d969f7ac94ca40000000000000000"),
  c: fromHex(0),
  d: fromHex(null),
  e: fromHex("f"),
}
```

## Output

**fromHex(hexText: Null): Null**

Helper function that enables `fromHex` to work with null value.

*Introduced in DataWeave version 2.2.0.*

## **fromRadixNumber**

**fromRadixNumber(numberStr: String, radix: Number): Number**

Transforms a number in the specified radix into decimal number

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
numberText	The number text.
radix	The radix number.

## Example

This example shows how the `fromRadixNumber` behaves under different inputs.

## Source

```
%dw 2.0
import fromRadixNumber from dw::core::Numbers
output application/json
---
{
    a: fromRadixNumber("10", 2),
    b: fromRadixNumber("FF", 16)
}
```

## Output

```
{  
  "a": 2,  
  "b": 255  
}
```

## toBinary

**toBinary(number: Number): String**

Transforms a decimal number into a binary number.

Introduced in DataWeave version 2.2.0.

## Parameters

Name	Description
number	The input number.

## Example

This example shows how the `toBinary` behaves with different inputs.

## Source

## Output

**toBinary(number: Null): Null**

Helper function that enables `toBinary` to work with null value.

*Introduced in DataWeave version 2.2.0.*

## toHex

**toHex(number: Number): String**

Transforms a decimal number into a hexadecimal number.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
number	The input number.

## Example

This example shows how `toHex` behaves with different inputs.

## Source

## Output

```
{  
  "a": "-1",  
  "b": "3e3aeb4ae1383562f4b82261d969f7ac94ca40000000000000000",  
  "c": "0",  
  "d": null,  
  "e": "f"  
}
```

## toHex(number: Null): Null

Helper function that enables `toHex` to work with null value.

*Introduced in DataWeave version 2.2.0.*

## toRadixNumber

### toRadixNumber(number: Number, radix: Number): String

Transforms a decimal number into a number string in other radix.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
number	The decimal number.
radix	The radix of the result number.

#### Example

This example shows how the `toRadixNumber` behaves under different inputs.

#### Source

```
%dw 2.0  
import toRadixNumber from dw::core::Numbers  
output application/json  
---  
{  
  a: toRadixNumber(2, 2),  
  b: toRadixNumber(255, 16)  
}
```

## Output

```
{
  "a": "10",
  "b": "ff"
}
```

## dw::core::Objects

This module contains helper functions for working with objects.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Objects` to the header of your DataWeave script.

## Functions

Name	Description
<code>divideBy</code>	Breaks up an object into sub-objects that contain the specified number of key-value pairs.
<code>entrySet</code>	Returns an array of key-value pairs that describe the key, value, and any attributes in the input object.
<code>everyEntry</code>	Returns <code>true</code> if every entry in the object matches the condition.
<code>keySet</code>	Returns an array of key names from an object.
<code>mergeWith</code>	Appends any key-value pairs from a source object to a target object.
<code>nameSet</code>	Returns an array of keys from an object.
<code>someEntry</code>	Returns <code>true</code> if at least one entry in the object matches the specified condition.
<code>takeWhile</code>	Selects key-value pairs from the object while the condition is met.
<code>valueSet</code>	Returns an array of the values from key-value pairs in an object.

### divideBy

**divideBy(items: Object, amount: Number): Array<Object>**

Breaks up an object into sub-objects that contain the specified number of key-value pairs.

If there are fewer key-value pairs in an object than the specified number, the function will fill the object with those pairs. If there are more pairs, the function will fill another object with the extra pairs.

#### Parameters

Name	Description
<code>items</code>	Key-value pairs in the source object.

Name	Description
amount	The number of key-value pairs allowed in an object.

## Example

This example breaks up objects into sub-objects based on the specified `amount`.

## Source

```
%dw 2.0
import divideBy from dw::core::Objects
output application/json
---
{ "divideBy" : { "a": 1, "b" : true, "a" : 2, "b" : false, "c" : 3} divideBy 2 }
```

## Output

```
{
  "divideBy": [
    {
      "a": 1,
      "b": true
    },
    {
      "a": 2,
      "b": false
    },
    {
      "c": 3
    }
  ]
}
```

## entrySet

`entrySet<T <: Object>(obj: T): Array<{| key: Key, value: Any, attributes: Object |}>`

Returns an array of key-value pairs that describe the key, value, and any attributes in the input object.

*This function is **Deprecated**. Use [dw::Core::entriesOf](#), instead.*

## Parameters

Name	Description
obj	The <code>Object</code> to describe.

## Example

This example returns the key, value, and attributes in the object specified in the variable `myVar`.

## Source

```
%dw 2.0
import * from dw::core::Objects
var myVar = read('<xml attr="x"><a>true</a><b>1</b></xml>', 'application/xml')
output application/json
---
{ "entrySet" : entrySet(myVar) }
```

## Output

```
{
  "entrySet": [
    {
      "key": "xml",
      "value": {
        "a": "true",
        "b": "1"
      },
      "attributes": {
        "attr": "x"
      }
    }
  ]
}
```

## everyEntry

**everyEntry(object: Object, condition: (value: Any, key: Key) -> Boolean): Boolean**

Returns `true` if every entry in the object matches the condition.

The function stops iterating after the first negative evaluation of an element in the object.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
<code>object</code>	The object to evaluate.
<code>condition</code>	The condition to apply to each element.

## Example

This example shows how `everyEntry` behaves with different inputs.

## Source

```
%dw 2.0
import everyEntry from dw::core::Objects
output application/json
---
{
    a: {} everyEntry (value, key) -> value is String,
    b: {a: "", b: "123"} everyEntry (value, key) -> value is String,
    c: {a: "", b: 123} everyEntry (value, key) -> value is String,
    d: {a: "", b: 123} everyEntry (value, key) -> key as String == "a",
    e: {a: ""} everyEntry (value, key) -> key as String == "a",
    f: null everyEntry ((value, key) -> key as String == "a")
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false,
    "d": false,
    "e": true,
    "f": true
}
```

## everyEntry(list: Null, condition: (Nothing, Nothing) -> Boolean): Boolean

Helper function that enables `everyEntry` to work with a `null` value.

*Introduced in DataWeave version 2.3.0.*

## keySet

### keySet<K, V>(obj: { (K)?: V }): Array<K>

Returns an array of key names from an object.

*This function is **Deprecated**. Use `dw::Core::keysOf` instead.*

#### Parameters

Name	Description
object	The object to evaluate.

#### Example

This example returns the keys from the input object.

## Source

```
%dw 2.0
import * from dw::core::Objects
output application/json
---
{ "keySet" : keySet({ "a" : true, "b" : 1}) }
```

## Output

```
{ "keySet" : ["a", "b"] }
```

## Example

This example illustrates a difference between `keySet` and `nameSet`. Notice that `keySet` retains the attributes (`name` and `lastName`) and namespaces (`xmlns`) from the XML input, while `nameSet` returns `null` for them because it does not retain them.

## Source

```
%dw 2.0
import * from dw::core::Objects
var myVar = read('<users xmlns="http://test.com">
    <user name="Mariano" lastName="Achaval"/>
    <user name="Stacey" lastName="Duke"/>
</users>', 'application/xml')
output application/json
---
{ keySetExample: flatten([keySet(myVar.users) map $.#, keySet(myVar.users) map $.@])
}
++
{ nameSet: flatten([nameSet(myVar.users) map $.#, nameSet(myVar.users) map $.@])
}
```

## Output

```
{
  "keySet": [
    "http://test.com",
    "http://test.com",
    {
      "name": "Mariano",
      "lastName": "Achaval"
    },
    {
      "name": "Stacey",
      "lastName": "Duke"
    }
  ],
  "nameSet": [
    null,
    null,
    null,
    null
  ]
}
```

## mergeWith

**mergeWith<T <: Object, V <: Object>(source: T, target: V): ?**

Appends any key-value pairs from a source object to a target object.

If source and target objects have the same key, the function appends that source object to the target and removes that target object from the output.

### Parameters

Name	Description
source	The object to append to the <b>target</b> .
target	The object to which the <b>source</b> object is appended.

### Example

This example appends the source objects to the target. Notice that **"a" : true**, is removed from the output, and **"a" : false** is appended to the target.

### Source

```
%dw 2.0
import mergeWith from dw::core::Objects
output application/json
---
{ "mergeWith" : { "a" : true, "b" : 1} mergeWith { "a" : false, "c" : "Test"} }
```

## Output

```
"mergeWith": {
    "b": 1,
    "a": false,
    "c": "Test"
}
```

### **mergeWith<T <: Object>(a: Null, b: T): T**

Helper function that enables `mergeWith` to work with a `null` value.

### **mergeWith<T <: Object>(a: T, b: Null): T**

Helper function that enables `mergeWith` to work with a `null` value.

## nameSet

### **nameSet(obj: Object): Array<String>**

Returns an array of keys from an object.

*This function is **Deprecated**. Use [dw::Core::namesOf](#), instead.*

#### Parameters

Name	Description
<code>obj</code>	The object to evaluate.

#### Example

This example returns the keys from the input object.

#### Source

```
%dw 2.0
import * from dw::core::Objects
output application/json
---
{ "nameSet" : nameSet({ "a" : true, "b" : 1}) }
```

## Output

```
{ "nameSet" : [ "a", "b" ] }
```

## someEntry

**someEntry(obj: Object, condition: (value: Any, key: Key) -> Boolean): Boolean**

Returns **true** if at least one entry in the object matches the specified condition.

The function stops iterating after the first element that matches the condition is found.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
<b>obj</b>	The object to evaluate.
<b>condition</b>	The condition to use when evaluating elements in the object.

### Example

This example shows how the **someEntry** behaves with different inputs.

### Source

```
%dw 2.0
import someEntry from dw::core::Objects
output application/json
---
{
    a: {} someEntry (value, key) -> value is String,
    b: {a: "", b: "123"} someEntry (value, key) -> value is String,
    c: {a: "", b: 123} someEntry (value, key) -> value is String,
    d: {a: "", b: 123} someEntry (value, key) -> key as String == "a",
    e: {a: ""} someEntry (value, key) -> key as String == "b",
    f: null someEntry (value, key) -> key as String == "a"
}
```

## Output

```
{  
  "a": false,  
  "b": true,  
  "c": true,  
  "d": true,  
  "e": false,  
  "f": false  
}
```

## someEntry(obj: Null, condition: (value: Nothing, key: Nothing) -> Boolean): Boolean

Helper function that enables `someEntry` to work with a `null` value.

*Introduced in DataWeave version 2.3.0.*

## takeWhile

### takeWhile<T>(obj: Object, condition: (value: Any, key: Key) -> Boolean): Object

Selects key-value pairs from the object while the condition is met.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
<code>obj</code>	The object to filter.
<code>condition</code>	The condition (or expression) used to match a key-value pairs in the object.

#### Example

This example iterates over the key-value pairs in the object and selects the elements while the condition is met. It outputs the result into an object.

#### Source

```
%dw 2.0
import * from dw::core::Objects
output application/json
var obj = {
    "a": 1,
    "b": 2,
    "c": 5,
    "d": 1
}
---
obj takeWhile ((value, key) -> value < 3)
```

## Output

```
{
    "a": 1,
    "b": 2
}
```

## valueSet

**valueSet<K, V>(obj: { (K)?: V }): Array<V>**

Returns an array of the values from key-value pairs in an object.

*This function is **Deprecated**. Use [dw::Core::valuesOf](#), instead.*

### Parameters

Name	Description
obj	The object to evaluate.

### Example

This example returns the values from the input object.

### Source

```
%dw 2.0
import * from dw::core::Objects
output application/json
---
{ "valueSet" : valueSet({a: true, b: 1}) }
```

## Output

```
{ "valueSet" : [true,1] }
```

## dw::core::Periods

This module contains functions for working with and creating Period values.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Periods` to the header of your DataWeave script.

### Functions

Name	Description
<code>between</code>	Returns a Period (P) value consisting of the number of years, months, and days between two Date values.
<code>days</code>	Creates a Period value from the provided number of days.
<code>duration</code>	Creates a Period value that represents a number of days, hours, minutes, or seconds.
<code>hours</code>	Creates a Period value from the provided number of hours.
<code>minutes</code>	Creates a Period value from the provided number of minutes.
<code>months</code>	Creates a Period value from the provided number of months.
<code>period</code>	Creates a Period value as a date-based number of years, months, and days in the ISO-8601 calendar system.
<code>seconds</code>	Creates a Period value from the provided number of seconds.
<code>years</code>	Creates a Period value from the provided number of years.

### `between`

#### `between(endDateExclusive: Date, startDateInclusive: Date): Period`

Returns a Period (P) value consisting of the number of years, months, and days between two Date values.

The start date is included, but the end date is not. The result of this method can be a negative period if the end date (`endDateExclusive`) is before the start date (`startDateInclusive`).

Note that the first parameter of the function is the `endDateExclusive` and the second one is the `startDateInclusive`.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
endDateExclusive	The end date, exclusive.
startDateInclusive	The start date, inclusive.

## Example

This example shows how `between` behaves with different inputs.

## Source

```
import * from dw::core::Periods
output application/json
---
{
    a: between(|2010-12-12|, |2010-12-10|),
    b: between(|2011-12-11|, |2010-11-10|),
    c: between(|2020-02-29|, |2020-03-30|)
}
```

## Output

```
{
    "a": "P2D",
    "b": "P1Y1M1D",
    "c": "P-1M-1D"
}
```

## days

### days(nDays: Number): Period

Creates a Period value from the provided number of days.

The function applies the `period` function to input that is a whole number and the `duration` function to decimal input.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
nDays	The number of hours as a whole or decimal number. A positive or negative number is valid.

## Example

This example shows how `days` behaves with different inputs. It adds and subtracts hours from

DateTime values. It also converts the decimal value **4.555** into a number of hours, minutes, and second in the Period format (**PT109H19M12S**) and the whole number **4** into a number of days in the Period format (**P4D**).

## Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    tomorrow: |2020-10-05T20:22:34.385Z| + days(1),
    yesterday: |2020-10-05T20:22:34.385Z| - days(1),
    decimalDaysPlusQuarter: |2020-10-05T00:00:00.000Z| + days(0.25),
    decimalDaysPlusHalf: |2020-10-05T00:00:00.000Z| + days(0.5),
    decimalDaysPlusThreeQuarters: |2020-10-05T00:00:00.000Z| + days(0.75),
    decimalInputAsPeriod : days(4.555),
    fourDayPeriod: days(4),
    negativeValue: days(-1)
}
```

## Output

```
{
    "tomorrow": "2020-10-06T20:22:34.385Z",
    "yesterday": "2020-10-04T20:22:34.385Z",
    "decimalDaysPlusQuarter": "2020-10-05T06:00:00Z",
    "decimalDaysPlusHalf": "2020-10-05T12:00:00Z",
    "decimalDaysPlusThreeQuarters": "2020-10-05T18:00:00Z",
    "decimalInputAsPeriod": "PT109H19M12S",
    "fourDayPeriod": "P4D",
    "negativeValue": "P-1D"
}
```

## duration

**duration(period: { days?: Number, hours?: Number, minutes?: Number, seconds?: Number }): Period**

Creates a Period value that represents a number of days, hours, minutes, or seconds.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
period	An object such as <code>{days:4, hours:11, minutes:45, seconds: 55}</code> . The key-value pairs are optional and can be specified in any order. An empty object ( <code>{}</code> ) returns the Period value "PT0S" (zero seconds). The default value of each key is <code>0</code> . Valid values are whole or decimal numbers, which can be positive or negative. Key names are selectable.

## Example

This example shows how `duration` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    dayAfterDateTime: |2020-10-05T20:22:34.385Z| + duration({days: 1}),
    dayAndHourBeforeDateTime: |2020-10-05T20:22:34.385Z| - duration({days: 1, hours: 1}),
    pointInTimeBefore: |2020-10-05T20:22:34.385Z| - duration({days: 1, hours: 1, minutes: 20, seconds: 10}),
    emptyDuration: duration({}),
    constructDuration: duration({days:4, hours:11, minutes:28}),
    selectHoursFromDuration: duration({days:4, hours:11, minutes:28}).hours,
    decimalAsPeriod: duration({seconds: 30.5}),
    addNegativeValue: duration({ minutes : 1 }) + duration({ seconds : -1 })
}
```

## Output

```
{
    "dayAfterDateTime": "2020-10-06T20:22:34.385Z",
    "dayAndHourBeforeDateTime": "2020-10-04T19:22:34.385Z",
    "pointInTimeBefore": "2020-10-04T19:02:24.385Z",
    "emptyDuration": "PT0S",
    "constructDuration": "PT107H28M",
    "selectHoursFromDuration": 11,
    "decimalAsPeriod": "PT30.5S",
    "addNegativeValue": 59
}
```

## hours

**hours(nHours: Number): Period**

Creates a Period value from the provided number of hours.

The function applies the `duration` function to the input value.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
<code>nHours</code>	The number of hours as a whole or decimal number. A positive or negative number is valid.

## Example

This example shows how `hours` behaves with different inputs. It adds and subtracts hours from DateTime and LocalTime values. It also converts the decimal value `4.555` into the Period format (`PT4H33M18S`) and the whole number `4` into the Period format (`PT4H`). Notice that `hours(-1) + hours(2)` returns the number of seconds.

## Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    nextHour: |2020-10-05T20:22:34.385Z| + hours(1),
    previousHour: |2020-10-05T20:22:34.385Z| - hours(1),
    threeHoursLater: |20:22| + hours(3),
    addDecimalInput: |20:22| + hours(3.5),
    decimalInputAsPeriod : hours(4.555),
    fourHourPeriod : hours(4),
    addNegativeValue: hours(-1) + hours(2)
}
```

## Output

```
{
    "nextHour": "2020-10-05T21:22:34.385Z",
    "previousHour": "2020-10-05T19:22:34.385Z",
    "threeHoursLater": "23:22:00",
    "addDecimalInput": "23:52:00",
    "decimalInputAsPeriod": "PT4H33M18S",
    "fourHourPeriod": "PT4H",
    "addNegativeValue": 3600
}
```

## minutes

### minutes(nMinutes: Number): Period

Creates a Period value from the provided number of minutes.

The function applies the `duration` function to the input value.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
nMinutes	The number of minutes as a whole or decimal number. A positive or negative number is valid.

#### Example

This example shows how `minutes` behaves with different inputs. It adds and subtracts hours from DateTime values. It also converts the decimal value `4.555` into the Period format (`PT4M33.3S`) and the whole number `4` into the Period format (`PT4M`). Notice that `minutes(-1) + minutes(2)` returns the number of seconds.

#### Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    nextMinute: |2020-10-05T20:22:34.385Z| + minutes(1),
    previousMinute: |2020-10-05T20:22:34.385Z| - minutes(1),
    decimalInputPeriod: minutes(4.555),
    wholeNumberInputPeriod: minutes(4),
    addNegativeValue: minutes(-1) + minutes(2)
}
```

#### Output

```
{
    "nextMinute": "2020-10-05T20:23:34.385Z",
    "previousMinute": "2020-10-05T20:21:34.385Z",
    "decimalInputPeriod": "PT4M33.3S",
    "wholeNumberInputPeriod": "PT4M",
    "addNegativeValue": 60
}
```

## months

### months(nMonths: Number): Period

Creates a Period value from the provided number of months.

The function applies the [period](#) function to the input value.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
nMonths	The number of months as a whole number. A positive or negative number is valid.

#### Example

This example shows how `months` behaves with different inputs. It adds a month to a DateTime value, and it converts the whole number `4` to a number of months in the Period format (`P4M`).

#### Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    nextMonth: |2020-10-05T20:22:34.385Z| + months(1),
    fourMonthPeriod : months(4),
    addNegativeValue: months(-1) + months(2)
}
```

#### Output

```
{
    "nextMonth": "2020-11-05T20:22:34.385Z",
    "fourMonthPeriod": "P4M",
    "addNegativeValue": 1
}
```

## period

### period(period: { years?: Number, months?: Number, days?: Number }): Period

Creates a Period value as a date-based number of years, months, and days in the ISO-8601 calendar system.

Introduced in DataWeave version 2.4.0.

## Parameters

Name	Description
<code>period</code>	An object such as <code>{years:4, months:11, days:28}</code> . The key-value pairs are optional and can be specified in any order. An empty object ( <code>{}</code> ) returns the Period value <code>"P0D"</code> (zero days). The default value of each key is <code>0</code> . Valid values are whole numbers, which can be positive or negative. Decimal values produce an error message. Key names are selectable.

## Example

This example shows how `period` behaves with different inputs. The example adds a subtracts and adds the result of a `period` function to `DateTime` and `Date` values. It also constructs a `Period` value from `period` objects and selects a `months` value from the object.

## Source

```
%dw 2.0
output application/json
import * from dw::core::Periods
---
{
    dayBeforeDateTime: |2020-10-05T20:22:34.385Z| - period({days:1}),
    dayAfterDate: |2020-10-05| + period({days:1}),
    yearMonthDayAfterDate: |2020-10-05| + period({years:1, months:1, days:1}),
    emptyPeriod: period({}),
    constructPeriod: period({years:4, months:11, days:28}),
    selectMonthsFromPeriod: period({years:4, months:11, days:28}).months
}
```

## Output

```
{
    "dayBeforeDateTime": "2020-10-04T20:22:34.385Z",
    "dayAfterDate": "2020-10-06",
    "yearMonthDayAfterDate": "2021-11-06",
    "emptyPeriod": "P0D",
    "constructPeriod": "P4Y11M28D",
    "selectMonthsFromPeriod": 11
}
```

## seconds

### seconds(nSecs: Number): Period

Creates a `Period` value from the provided number of seconds.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
nSecs	The number of seconds as a whole or decimal number. A positive or negative number is valid.

## Example

This example shows how `seconds` behaves with different inputs. It adds and subtracts seconds from DateTime values. It also converts the decimal value `4.555` into the Period format (`PT4.555S`) and the whole number `4` into the Period format (`PT4S`)

## Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    nextSecond: |2020-10-05T20:22:34.385Z| + seconds(1),
    previousSecond: |2020-10-05T20:22:34.385Z| - seconds(1),
    decimalInputPeriod: seconds(4.555),
    wholeNumberInputPeriod: seconds(4),
    addNegativeValue: seconds(-1) + seconds(2)
}
```

## Output

```
{
    "nextSecond": "2020-10-05T20:22:35.385Z",
    "previousSecond": "2020-10-05T20:22:33.385Z",
    "decimalInputPeriod": "PT4.555S",
    "wholeNumberInputPeriod": "PT4S",
    "addNegativeValue": 1
}
```

## years

### years(nYears: Number): Period

Creates a Period value from the provided number of years.

The function applies the `period` function to the input value.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
nYears	A whole number for the number of years. A positive or negative number is valid.

## Example

This example shows how `years` behaves with different inputs. It adds a year to a `DateTime` value, and it converts the whole number 4 into a number of years in the `Period` format (`P4Y`). Notice that `years(-1) + years(2)` returns the number of months.

## Source

```
%dw 2.0
import * from dw::core::Periods
output application/json
---
{
    nextYear: |2020-10-05T20:22:34.385Z| + years(1),
    fourYearPeriod: years(4),
    addNegativeValue: years(-1) + years(2)
}
```

## Output

```
{
    "nextYear": "2021-10-05T20:22:34.385Z",
    "fourYearPeriod": "P4Y",
    "addNegativeValue": 12
}
```

# dw::core::Strings

This module contains helper functions for working with strings.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Strings` to the header of your DataWeave script.

## Functions

Name	Description
appendIfMissing	Appends the <code>suffix</code> to the end of the <code>text</code> if the <code>text</code> does not already ends with the <code>suffix</code> .
camelize	Returns a string in camel case based on underscores in the string.

Name	Description
<a href="#">capitalize</a>	Capitalizes the first letter of each word in a string.
<a href="#">charCode</a>	Returns the Unicode for the first character in an input string.
<a href="#">charCodeAt</a>	Returns the Unicode for a character at the specified index.
<a href="#">collapse</a>	Collapses the string into substrings of equal characters.
<a href="#">countCharactersBy</a>	Counts the number of times an expression that iterates through each character in a string returns <code>true</code> .
<a href="#">countMatches</a>	Counts the number of matches in a string.
<a href="#">dasherize</a>	Replaces spaces, underscores, and camel-casing in a string with dashes (hyphens).
<a href="#">everyCharacter</a>	Checks whether a condition is valid for <i>every</i> character in a string.
<a href="#">first</a>	Returns characters from the beginning of a string to the specified number of characters in the string, for example, the first two characters of a string.
<a href="#">fromCharCode</a>	Returns a character that matches the specified Unicode.
<a href="#">hammingDistance</a>	Returns the Hamming distance between two strings.
<a href="#">isAlpha</a>	Checks if the <code>text</code> contains only Unicode digits. A decimal point is not a Unicode digit and returns <code>false</code> .
<a href="#">isAlphanumeric</a>	Checks if the <code>text</code> contains only Unicode letters or digits.
<a href="#">isLowerCase</a>	Checks if the <code>text</code> contains only lowercase characters.
<a href="#">isNumeric</a>	Checks if the <code>text</code> contains only Unicode digits.
<a href="#">isUpperCase</a>	Checks if the <code>text</code> contains only uppercase characters.
<a href="#">isWhitespace</a>	Checks if the <code>text</code> contains only whitespace.
<a href="#">last</a>	Returns characters from the end of string to a specified number of characters, for example, the last two characters of a string.
<a href="#">leftPad</a>	The specified <code>text</code> is <i>left</i> -padded to the <code>size</code> using the <code>padText</code> . By default <code>padText</code> is <code>" "</code> .
<a href="#">levenshteinDistance</a>	Returns the Levenshtein distance (or edit distance) between two strings.
<a href="#">lines</a>	Returns an array of lines from a string.
<a href="#">mapString</a>	Applies an expression to every character of a string.
<a href="#">ordinalize</a>	Returns a number as an ordinal, such as <code>1st</code> or <code>2nd</code> .
<a href="#">pluralize</a>	Pluralizes a singular string.
<a href="#">prependIfMissing</a>	Prepends the <code>prefix</code> to the beginning of the string if the <code>text</code> does not already start with that prefix.
<a href="#">remove</a>	Removes all occurrences of a specified pattern from a string.
<a href="#">repeat</a>	Repeats a <code>text</code> the number of specified <code>times</code> .

Name	Description
replaceAll	Replaces all substrings that match a literal search string with a specified replacement string.
reverse	Reverses sequence of characters in a string.
rightPad	The specified <code>text</code> is <i>right</i> -padded to the <code>size</code> using the <code>padText</code> . By default <code>padText</code> is " ".
singularize	Converts a plural string to its singular form.
someCharacter	Checks whether a condition is valid for at least one of the characters or blank spaces in a string.
substring	Returns a substring that spans from the character at the specified <code>from</code> index to the last character before the <code>until</code> index.
substringAfter	Gets the substring after the first occurrence of a separator. The separator is not returned.
substringAfterLast	Gets the substring after the last occurrence of a separator. The separator is not returned.
substringBefore	Gets the substring before the first occurrence of a separator. The separator is not returned.
substringBeforeLast	Gets the substring before the last occurrence of a separator. The separator is not returned.
substringBy	Splits a string at each character where the <code>predicate</code> expression returns <code>true</code> .
substringEvery	Splits a string into an array of substrings equal to a specified length.
underscore	Replaces hyphens, spaces, and camel-casing in a string with underscores.
unwrap	Unwraps a given <code>text</code> from a <code>wrapper</code> text.
withMaxSize	Checks that the string length is no larger than the specified <code>maxLength</code> . If the string's length is larger than the <code>maxLength</code> , the function cuts characters from left to right, until the string length meets the length limit.
words	Returns an array of words from a string.
wrapIfMissing	Wraps <code>text</code> with <code>wrapper</code> if that <code>wrapper</code> is missing from the start or end of the given string.
wrapWith	Wraps the specified <code>text</code> with the given <code>wrapper</code> .

## appendIfMissing

**appendIfMissing(text: String, suffix: String): String**

Appends the `suffix` to the end of the `text` if the `text` does not already ends with the `suffix`.

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
text	The input string.
suffix	The text used as the suffix.

## Example

This example shows how `appendIfMissing` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import appendIfMissing from dw::core::Strings
output application/json
---
{
  "a": appendIfMissing(null, ""),
  "b": appendIfMissing("abc", ""),
  "c": appendIfMissing("", "xyz"),
  "d": appendIfMissing("abc", "xyz"),
  "e": appendIfMissing("abcxyz", "xyz")
}
```

## Output

```
{
  "a": null,
  "b": "abc",
  "c": "xyz",
  "d": "abcxyz",
  "e": "abcxyz"
}
```

## appendIfMissing(text: Null, suffix: String): Null

Helper function that enables `appendIfMissing` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## camelize

### camelize(text: String): String

Returns a string in camel case based on underscores in the string.

All underscores are deleted, including any underscores at the beginning of the string.

## Parameters

Name	Description
text	The string to convert to camel case.

## Example

This example converts a string that contains underscores to camel case.

## Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "a" : camelize("customer_first_name"),
  "b" : camelize("_name_starts_with_underscore")
}
```

## Output

```
{
  "a": "customerFirstName",
  "b": "nameStartsWithUnderscore"
}
```

## camelize(text: Null): Null

Helper function that enables `camelize` to work with a `null` value.

## capitalize

### capitalize(text: String): String

Capitalizes the first letter of each word in a string.

It also removes underscores between words and puts a space before each capitalized word.

## Parameters

Name	Description
text	The string to capitalize.

## Example

This example capitalizes a set of strings.

## Source

```
%dw 2.0
import * from dw::core::Strings
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"
}
```

### **capitalize(text: Null): Null**

Helper function that enables `capitalize` to work with a `null` value.

## charCode

### **charCode(text: String): Number**

Returns the Unicode for the first character in an input string.

For an empty string, the function fails and returns `Unexpected empty string`.

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example returns Unicode for the "M" in "Mule".

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "charCode" : charCode("Mule")
}
```

## Output

```
{ "charCode" : 77 }
```

### charCode(text: Null): Null

Helper function that enables `charCode` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## charCodeAt

### charCodeAt(content: String, position: Number): Number

Returns the Unicode for a character at the specified index.

This function fails if the index is invalid.

#### Parameters

Name	Description
<code>content</code>	The input string.
<code>position</code>	The index (a <code>Number</code> type) of a character in the string (as a string array). Note that the index of the first character is <code>0</code> .

#### Example

This example returns Unicode for the "u" at index `1` in "MuleSoft".

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "charCodeAt" : charCodeAt("MuleSoft", 1)
}
```

## Output

```
{ "charCodeAt": 117 }
```

## charCodeAt(content: Null, position: Any): Null

Helper function that enables `charCodeAt` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## collapse

### collapse(text: String): Array<String>

Collapses the string into substrings of equal characters.

Each substring contains a single character or identical characters that are adjacent to one another in the input string. Empty spaces are treated as characters.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>text</code>	The string to collapse.

#### Example

This example shows how the function collapses characters. Notice that the empty space (" ") is treated as a character.

#### Source

```
%dw 2.0
import collapse from dw::core::Strings
output application/json
---
collapse("a  b babb a")
```

## Output

```
["a", " ", "b", " ", "b", "a", "bb", " ", "a"]
```

### collapse(text: Null): Null

Helper function that enables `collapse` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## countCharactersBy

**countCharactersBy(text: String, predicate: (character: String) -> Boolean): Number**

Counts the number of times an expression that iterates through each character in a string returns **true**.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
text	The string to which the <b>predicate</b> applies.
predicate	Expression to apply to each character in the <b>text</b> string. The expression must return a Boolean value.

### Example

This example counts the digits in a string.

### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
"42 = 11 * 2 + 20" countCharactersBy isNumeric($)
```

### Output

```
7
```

**countCharactersBy(text: Null, predicate: (character: Nothing) -> Any): Null**

Helper function to make **countCharactersBy** work with a **null** value.

*Introduced in DataWeave version 2.4.0.*

## countMatches

**countMatches(text: String, pattern: String): Number**

Counts the number of matches in a string.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
text	The string to search for matches.
pattern	A substring to find in the text.

## Example

This example counts matches in a string.

## Source

```
%dw 2.0
import countMatches from dw::core::Strings
output application/json
---
"hello world!" countMatches "lo"
```

## Output

```
2
```

## countMatches(text: String, pattern: Regex): Number

Counts the number of times a regular expression matches text in a string.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
text	The string to search for matches.
pattern	The regex pattern to use in the search.

## Example

This example counts the vowels in a string.

## Source

```
%dw 2.0
import countMatches from dw::core::Strings
output application/json
---
"hello, ciao!" countMatches /[aeiou]/

```

## Output

```
5
```

## countMatches(text: Null, pattern: Any): Null

Helper function that enables `countMatches` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## dasherize

### dasherize(text: String): String

Replaces spaces, underscores, and camel-casing in a string with dashes (hyphens).

If no spaces, underscores, and camel-casing are present, the output will match the input.

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example replaces the spaces, underscores, and camel-casing in the input. Notice that the input "customer" is not modified in the output.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "a" : dasherize("customer"),
  "b" : dasherize("customer_first_name"),
  "c" : dasherize("customer NAME"),
  "d" : dasherize("customerName")
}
```

## Output

```
{  
  "a": "customer",  
  "b": "customer-first-name",  
  "c": "customer-name",  
  "d": "customer-name"  
}
```

## dasherize(text: Null): Null

Helper function that enables `dasherize` to work with a `null` value.

## everyCharacter

### everyCharacter(text: String, condition: (character: String) -> Boolean): Boolean

Checks whether a condition is valid for *every* character in a string.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>text</code>	The string to check.
<code>condition</code>	Expression that iterates through the characters in the string that it checks and returns a Boolean value.

#### Example

This example determines whether a string is composed of only digits and spaces.

#### Source

```
%dw 2.0  
import * from dw::core::Strings  
output application/json  
---  
"12 34 56" everyCharacter $ == " " or isNumeric($)
```

#### Output

```
true
```

### everyCharacter(text: Null, condition: (character: Nothing) -> Any): true

Helper function that enables `everyCharacter` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## first

### first(text: String, amount: Number): String

Returns characters from the beginning of a string to the specified number of characters in the string, for example, the first two characters of a string.

If the number is equal to or greater than the number of characters in the string, the function returns the entire string.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
text	The string to process.
amount	The number of characters to return. Negative numbers and <code>0</code> return an empty string. Decimals are rounded down to the nearest whole number.

#### Example

This example returns the first five characters from a string.

#### Source

```
%dw 2.0
import first from dw::core::Strings
output application/json
---
"hello world!" first 5
```

#### Output

```
"hello"
```

### first(text: Null, amount: Any): Null

Helper function that enables `first` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## fromCharCode

### fromCharCode(charCode: Number): String

Returns a character that matches the specified Unicode.

## Parameters

Name	Description
charCode	The input Unicode (a <a href="#">Number</a> ).

## Example

This example inputs the Unicode number [117](#) to return the character "u".

## Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "fromCharCode" : fromCharCode(117)
}
```

## Output

```
{ "fromCharCode": "u" }
```

## fromCharCode(charCode: Null): Null

Helper function that enables [fromCharCode](#) to work with a [null](#) value.

*Introduced in DataWeave version 2.4.0.*

## hammingDistance

### hammingDistance(a: String, b: String): Number | Null

Returns the Hamming distance between two strings.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
a	The first string.
b	The second string.

## Example

This example shows how [hammingDistance](#) behaves with different strings.

## Source

```
%dw 2.0
import hammingDistance from dw::core::Strings
output application/json
---
"holu" hammingDistance "chau"
```

## Output

```
3
```

## isAlpha

### isAlpha(text: String): Boolean

Checks if the `text` contains only Unicode digits. A decimal point is not a Unicode digit and returns `false`.

Note that the method does not allow for a leading sign, either positive or negative.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example shows how `isAlpha` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import isAlpha from dw::core::Strings
output application/json
---
{
  "a": isAlpha(null),
  "b": isAlpha(""),
  "c": isAlpha(" "),
  "d": isAlpha("abc"),
  "e": isAlpha("ab2c"),
  "f": isAlpha("ab-c")
}
```

## Output

```
{  
  "a": false,  
  "b": false,  
  "c": false,  
  "d": true,  
  "e": false,  
  "f": false  
}
```

## isAlpha(text: Null): Boolean

Helper function that enables `isAlpha` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## isAlphanumeric

### isAlphanumeric(text: String): Boolean

Checks if the `text` contains only Unicode letters or digits.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example shows how `isAlphanumeric` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import isAlphanumeric from dw::core::Strings
output application/json
---
{
    "a": isAlphanumeric(null),
    "b": isAlphanumeric(""),
    "c": isAlphanumeric(" "),
    "d": isAlphanumeric("abc"),
    "e": isAlphanumeric("ab c"),
    "f": isAlphanumeric("ab2c"),
    "g": isAlphanumeric("ab-c")
}
```

## Output

```
{
    "a": false,
    "b": false,
    "c": false,
    "d": true,
    "e": false,
    "f": true,
    "g": false
}
```

## isAlphanumeric(text: Null): Boolean

Helper function that enables `isAlphanumeric` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## isLowerCase

### isLowerCase(text: String): Boolean

Checks if the `text` contains only lowercase characters.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example shows how `isLowerCase` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import isLowerCase from dw::core::Strings
output application/json
---
{
  "a": isLowerCase(null),
  "b": isLowerCase(""),
  "c": isLowerCase(" "),
  "d": isLowerCase("abc"),
  "e": isLowerCase("aBC"),
  "f": isLowerCase("a c"),
  "g": isLowerCase("a1c"),
  "h": isLowerCase("a/c")
}
```

## Output

```
{
  "a": false,
  "b": false,
  "c": false,
  "d": true,
  "e": false,
  "f": false,
  "g": false,
  "h": false
}
```

### **isLowerCase(text: Null): Boolean**

Helper function that enables `isLowerCase` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

### **isNumeric**

#### **isNumeric(text: String): Boolean**

Checks if the `text` contains only Unicode digits.

A decimal point is not a Unicode digit and returns false. Note that the method does not allow for a leading sign, either positive or negative.

*Introduced in DataWeave version 2.2.0.*

#### **Parameters**

Name	Description
text	The input string.

## Example

This example shows how `isNumeric` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import isNumeric from dw::core::Strings
output application/json
---
{
  "a": isNumeric(null),
  "b": isNumeric(""),
  "c": isNumeric(" "),
  "d": isNumeric("123"),
  "e": isNumeric("一二"),
  "f": isNumeric("12 3"),
  "g": isNumeric("ab2c"),
  "h": isNumeric("12-3"),
  "i": isNumeric("12.3"),
  "j": isNumeric("-123"),
  "k": isNumeric("+123")
}
```

## Output

```
{
  "a": false,
  "b": false,
  "c": false,
  "d": true,
  "e": true,
  "f": false,
  "g": false,
  "h": false,
  "i": false,
  "j": false,
  "k": false
}
```

## isNumeric(text: Null): Boolean

Helper function that enables `isNumeric` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## isUpperCase

### isUpperCase(text: String): Boolean

Checks if the `text` contains only uppercase characters.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example shows how `isUpperCase` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import isUpperCase from dw::core::Strings
output application/json
---
{
    "a": isUpperCase(null),
    "b": isUpperCase(""),
    "c": isUpperCase(" "),
    "d": isUpperCase("ABC"),
    "e": isUpperCase("aBC"),
    "f": isUpperCase("A C"),
    "g": isUpperCase("A1C"),
    "h": isUpperCase("A/C")
}
```

#### Output

```
{
    "a": false,
    "b": false,
    "c": false,
    "d": true,
    "e": false,
    "f": false,
    "g": false,
    "h": false
}
```

### isUpperCase(text: Null): Boolean

Helper function that enables `isUpperCase` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## isWhitespace

### isWhitespace(text: String): Boolean

Checks if the `text` contains only whitespace.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.

#### Example

This example shows how `isWhitespace` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import isWhitespace from dw::core::Strings
output application/json
---
{
  "a": isWhitespace(null),
  "b": isWhitespace(""),
  "c": isWhitespace(" "),
  "d": isWhitespace("abc"),
  "e": isWhitespace("ab2c"),
  "f": isWhitespace("ab-c")
}
```

#### Output

```
{
  "a": false,
  "b": true,
  "c": true,
  "d": false,
  "e": false,
  "f": false
}
```

### isWhitespace(text: Null): Boolean

Helper function that enables `isWhitespace` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## last

**last(text: String, amount: Number): String**

Returns characters from the end of string to a specified number of characters, for example, the last two characters of a string.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>text</code>	The string to process.
<code>amount</code>	The number of characters to return. Negative numbers and <code>0</code> return an empty string. Decimals are rounded up to the nearest whole number.

### Example

This example returns the last six characters from a string.

### Source

```
%dw 2.0
import last from dw::core::Strings
output application/json
---
"hello world!" last 6
```

### Output

```
"world!"
```

**last(text: Null, amount: Any): Null**

Helper function that enables `last` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## leftPad

**leftPad(text: String, size: Number, padText: String = " "): String**

The specified `text` is *left*-padded to the `size` using the `padText`. By default `padText` is `" "`.

Returns left-padded **String** or original **String** if no padding is necessary.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
<code>text</code>	The input string.
<code>size</code>	The size to pad to.
<code>padText</code>	The text to pad with. It defaults to one space if not specified.

## Example

This example shows how **leftPad** behaves with different inputs and sizes.

## Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "a": leftPad(null, 3),
    "b": leftPad("", 3),
    "c": leftPad("bat", 5),
    "d": leftPad("bat", 3),
    "e": leftPad("bat", -1)
}
```

## Output

```
{
    "a": null,
    "b": "  ",
    "c": " bat",
    "d": "bat",
    "e": "bat"
}
```

## **leftPad(text: Null, size: Any, padText: Any = " ")**: Null

Helper function that enables **leftPad** to work with a **null** value.

*Introduced in DataWeave version 2.2.0.*

## levenshteinDistance

**levenshteinDistance(a: String, b: String): Number**

Returns the Levenshtein distance (or edit distance) between two strings.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
a	The first string.
b	The second string.

### Example

This example shows how `levenshteinDistance` behaves with different strings.

### Source

```
%dw 2.0
import levenshteinDistance from dw::core::Strings
output application/json
---
"kitten" levenshteinDistance "sitting"
```

### Output

```
3
```

## lines

**lines(text: String): Array<String>**

Returns an array of lines from a string.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
text	The string to split into lines.

### Example

This example divides a string into lines. An \n represents a line break.

## Source

```
%dw 2.0
import lines from dw::core::Strings
output application/json
---
lines("hello world\n\nhere    data-weave")
```

## Output

```
["hello world", "", "here    data-weave"]
```

### lines(text: Null): Null

Helper function that enables `lines` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## mapString

`mapString(@StreamCapable text: String, mapper: (character: String, index: Number) -> String): String`

Applies an expression to every character of a string.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>text</code>	The string to map.
<code>mapper</code>	Expression that applies to each character (\$) or index (\$\$) of the <code>text</code> string and returns a string.

### Example

This example redacts sensitive data from a string.

## Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{ balance: ("\$234" mapString if (isNumeric($)) "~" else $) }
```

## Output

```
{  
  "balance": "$~~~"  
}
```

**mapString(@StreamCapable text: Null, mapper: (character: Nothing, index: Nothing) -> Any): Null**

Helper function that enables `mapString` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## ordinalize

**ordinalize(num: Number): String**

Returns a number as an ordinal, such as `1st` or `2nd`.

### Parameters

Name	Description
<code>num</code>	An input number to return as an ordinal.

### Example

This example returns a variety of input numbers as ordinals.

### Source

```
%dw 2.0  
import * from dw::core::Strings  
output application/json  
---  
{  
  "a" : ordinalize(1),  
  "b": ordinalize(2),  
  "c": ordinalize(5),  
  "d": ordinalize(103)  
}
```

### Output

```
{  
  "a": "1st",  
  "b": "2nd",  
  "c": "5th",  
  "d": "103rd"  
}
```

## **ordinalize(num: Null): Null**

Helper function that enables `ordinalize` to work with a `null` value.

## **pluralize**

### **pluralize(text: String): String**

Pluralizes a singular string.

If the input is already plural (for example, "boxes"), the output will match the input.

#### **Parameters**

Name	Description
<code>text</code>	The string to pluralize.

#### **Example**

This example pluralizes the input string "box" to return "boxes".

#### **Source**

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{ "pluralize" : pluralize("box") }
```

#### **Output**

```
{ "pluralize" : "boxes" }
```

## **pluralize(text: Null): Null**

Helper function that enables `pluralize` to work with a `null` value.

## **prependIfMissing**

### **prependIfMissing(text: String, prefix: String): String**

Prepends the `prefix` to the beginning of the string if the `text` does not already start with that prefix.

*Introduced in DataWeave version 2.2.0.*

#### **Parameters**

Name	Description
text	The input string.
prefix	The text to use as prefix.

## Example

This example shows how `prependIfMissing` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import prependIfMissing from dw::core::Strings
output application/json
---
{
  "a": prependIfMissing(null, ""),
  "b": prependIfMissing("abc", ""),
  "c": prependIfMissing("", "xyz"),
  "d": prependIfMissing("abc", "xyz"),
  "e": prependIfMissing("xyzabc", "xyz")
}
```

## Output

```
{
  "a": null,
  "b": "abc",
  "c": "xyz",
  "d": "xyzabc",
  "e": "xyzabc"
}
```

## prependIfMissing(text: Null, prefix: String): Null

Helper function that enables `prependIfMissing` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## remove

### remove(text: String, toRemove: String): String

Removes all occurrences of a specified pattern from a string.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
text	The text to remove from.
toRemove	The pattern to remove.

## Example

This example shows how the `remove` can be used to remove some unwanted properties.

## Source

```
%dw 2.0
import remove from dw::core::Strings
output application/json
---
"lazyness purity state higher-order stateful" remove "state"
```

## Output

```
"lazyness purity  higher-order ful"
```

## remove(text: Null, toRemove: Any): Null

Helper function that enables `remove` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## repeat

### repeat(text: String, times: Number): String

Repeats a `text` the number of specified `times`.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
text	The input string.
times	Number of times to repeat char. Negative is treated as zero.

## Example

This example shows how `repeat` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "a": repeat("e", 0),
  "b": repeat("e", 3),
  "c": repeat("e", -2)
}
```

## Output

```
{
  "a": "",
  "b": "eee",
  "c": ""
}
```

## repeat(text: Null, times: Any): Null

Helper function that enables `repeat` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## replaceAll

### replaceAll(text: String, target: String, replacement: String): String

Replaces all substrings that match a literal search string with a specified replacement string.

Replacement proceeds from the beginning of the string to the end. For example, the result of replacing `"aa"` with `"b"` in the string ``"aaa"` is `"ba"`, rather than `"ab"`.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>text</code>	The string to search.
<code>target</code>	The string to find and replace in <code>text</code> .
<code>replacement</code>	The replacement string.

#### Example

This example shows how `replaceAll` behaves with different inputs.

#### Source

```
import * from dw::core::Strings
output application/json
---
{
    a: replaceAll("Mariano", "a" , "A"),
    b: replaceAll("AAAA", "AAA" , "B"),
    c: replaceAll(null, "aria" , "A"),
    d: replaceAll("Mariano", "j" , "Test"),
}
```

## Output

```
{
    "a": "MAriAno",
    "b": "BA",
    "c": null,
    "d": "Mariano"
}
```

## replaceAll(text: Null, oldValue: String, newValue: String): Null

Helper function that enables `replaceAll` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## reverse

### reverse(text: String): String

Reverses sequence of characters in a string.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
text	The string to reverse.

#### Example

This example shows how `reverse` behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    a: reverse("Mariano"),
    b: reverse(null),
    c: reverse("")
}
```

## Output

```
{
    "a": "onairam",
    "b": null,
    "c": ""
}
```

## reverse(text: Null): Null

Helper function that enables `reverse` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## rightPad

### rightPad(text: String, size: Number, padChar: String = " "): String

The specified `text` is *right*-padded to the `size` using the `padText`. By default `padText` is `" "`.

Returns right padded `String` or original `String` if no padding is necessary.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.
<code>size</code>	The size to pad to.
<code>padText</code>	The text to pad with. It defaults to one space if not specified.

#### Example

This example shows how `rightPad` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "a": rightPad(null, 3),
  "b": rightPad("", 3),
  "c": rightPad("bat", 5),
  "d": rightPad("bat", 3),
  "e": rightPad("bat", -1)
}
```

## Output

```
{
  "a": null,
  "b": "  ",
  "c": "bat  ",
  "d": "bat",
  "e": "bat"
}
```

## rightPad(text: Null, size: Any, padText: Any = " "): Null

Helper function that enables `rightPad` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## singularize

### singularize(text: String): String

Converts a plural string to its singular form.

If the input is already singular (for example, "box"), the output will match the input.

#### Parameters

Name	Description
<code>text</code>	The string to convert to singular form.

#### Example

This example converts the input string "boxes" to return "box".

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{ "singularize" : singularize("boxes") }
```

## Output

```
{ "singularize" : "box" }
```

### singularize(text: Null): Null

Helper function that enables `singularize` to work with a `null` value.

## someCharacter

### someCharacter(text: String, condition: (character: String) -> Boolean): Boolean

Checks whether a condition is valid for at least one of the characters or blank spaces in a string.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>text</code>	The string to check.
<code>condition</code>	Expression that iterates through the characters and spaces in the string and returns a Boolean value.

#### Example

This example determines whether a string has any uppercase characters.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
"someCharacter" someCharacter isUpperCase($)
```

## Output

```
true
```

## **someCharacter(text: Null, condition: (character: Nothing) -> Any): false**

Helper function that enables `someCharacter` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## **substring**

### **substring(text: String, from: Number, until: Number): String**

Returns a substring that spans from the character at the specified `from` index to the last character before the `until` index.

The characters in the substring satisfy the condition `from <= indexOf(string) < until`.

*Introduced in DataWeave version 2.4.0.*

#### **Parameters**

Name	Description
<code>text</code>	The string, treated as an array of characters.
<code>from</code>	The lowest index to include from the character array.
<code>until</code>	The lowest index to exclude from the character array.

#### **Example**

This example returns the substring with characters at indices `1` through `4` in the input string.

#### **Source**

```
%dw 2.0
import * from dw::core::Strings
output application/json
var text = "hello world!"
---
substring(text, 1, 5)
```

#### **Output**

```
"ello"
```

### **substring(text: Null, from: Any, until: Any): Null**

Helper function that enables `substring` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## substringAfter

### substringAfter(text: String, separator: String): String

Gets the substring after the first occurrence of a separator. The separator is not returned.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
text	The input string.
separator	String to search for.

#### Example

This example shows how `substringAfter` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "a": substringAfter(null, ''),
    "b": substringAfter("", "-"),
    "c": substringAfter("abc", "b"),
    "d": substringAfter("abcba", "b"),
    "e": substringAfter("abc", "d"),
    "f": substringAfter("abc", "")
}
```

#### Output

```
{
    "a": null,
    "b": "",
    "c": "c",
    "d": "cba",
    "e": "",
    "f": "bc"
}
```

### substringAfter(text: Null, separator: String): Null

Helper function that enables `substringAfter` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## substringAfterLast

### substringAfterLast(text: String, separator: String): String

Gets the substring after the last occurrence of a separator. The separator is not returned.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
text	The input string.
separator	String to search for.

#### Example

This example shows how `substringAfterLast` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "a": substringAfterLast(null, ""),
    "b": substringAfterLast("", "-"),
    "c": substringAfterLast("abc", "b"),
    "d": substringAfterLast("abcba", "b"),
    "e": substringAfterLast("abc", "d"),
    "f": substringAfterLast("abc", "")
}
```

#### Output

```
{
    "a": null,
    "b": "",
    "c": "c",
    "d": "a",
    "e": "",
    "f": null
}
```

### substringAfterLast(text: Null, separator: String): Null

Helper function that enables `substringAfterLast` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## substringBefore

### substringBefore(text: String, separator: String): String

Gets the substring before the first occurrence of a separator. The separator is not returned.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.
<code>separator</code>	String to search for.

#### Example

This example shows how `substringBefore` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "a": substringBefore(null, ""),
    "b": substringBefore("", "-"),
    "c": substringBefore("abc", "b"),
    "d": substringBefore("abc", "c"),
    "e": substringBefore("abc", "d"),
    "f": substringBefore("abc", "")
}
```

#### Output

```
{
    "a": null,
    "b": "",
    "c": "a",
    "d": "ab",
    "e": "",
    "f": ""}
```

## substringBefore(text: Null, separator: String): Null

Helper function that enables `substringBefore` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## substringBeforeLast

### substringBeforeLast(text: String, separator: String): String

Gets the substring before the last occurrence of a separator. The separator is not returned.

*Introduced in DataWeave version 2.2.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.
<code>separator</code>	String to search for.

#### Example

This example shows how `substringBeforeLast` behaves with different inputs and sizes.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "a": substringBeforeLast(null, ""),
    "b": substringBeforeLast("", "-"),
    "c": substringBeforeLast("abc", "b"),
    "d": substringBeforeLast("abcba", "b"),
    "e": substringBeforeLast("abc", "d"),
    "f": substringBeforeLast("abc", "")}
```

#### Output

```
{
  "a": null,
  "b": "",
  "c": "a",
  "d": "abc",
  "e": "",
  "f": "ab"
}
```

## substringBeforeLast(text: Null, separator: String): Null

Helper function that enables `substringBeforeLast` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## substringBy

`substringBy(text: String, predicate: (character: String, index: Number) -> Boolean): Array<String>`

Splits a string at each character where the `predicate` expression returns `true`.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>text</code>	The string to split. The string is treated as an array of characters.
<code>predicate</code>	Expression that tests each character and returns a Boolean value. The expression can iterate over each character and index of the string.

### Example

This example splits a string where any of the specified characters ("~", "=", or "\_" ) are present.

### Source

```
%dw 2.0
import substringBy from dw::core::Strings
output application/json
---
"hello~world=here_data-weave" substringBy $ == "~" or $ == "=" or $ == "_"
```

### Output

```
["hello", "world", "here", "data-weave"]
```

## **substringBy(text: Null, predicate: (character: Nothing, index: Nothing) -> Any): Null**

Helper function that enables `substringBy` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## **substringEvery**

### **substringEvery(text: String, amount: Number): Array<String>**

Splits a string into an array of substrings equal to a specified length.

The last substring can be shorter than that length. If the length is greater than or equal to the length of the string to split, the function returns the entire string.

*Introduced in DataWeave version 2.4.0.*

#### **Parameters**

Name	Description
<code>text</code>	The string to split.
<code>amount</code>	The desired length of each substring.

#### **Example**

This example shows how `substringEvery` behaves when splitting an input string. The last returned substring is shorter than the others.

#### **Source**

```
%dw 2.0
import substringEvery from dw::core::Strings
output application/json
---
substringEvery("substringEvery", 3)
```

#### **Output**

```
["sub", "str", "ing", "Eve", "ry"]
```

### **substringEvery(text: Null, amount: Any): Null**

Helper function that enables `substringEvery` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## underscore

### underscore(text: String): String

Replaces hyphens, spaces, and camel-casing in a string with underscores.

If no hyphens, spaces, and camel-casing are present, the output will match the input.

#### Parameters

Name	Description
text	The input string.

#### Example

This example replaces the hyphens and spaces in the input. Notice that the input "customer" is not modified in the output.

#### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
    "a" : underscore("customer"),
    "b" : underscore("customer-first-name"),
    "c" : underscore("customer NAME"),
    "d" : underscore("customerName")
}
```

#### Output

```
{
    "a": "customer",
    "b": "customer_first_name",
    "c": "customer_name",
    "d": "customer_name"
}
```

### underscore(text: Null): Null

Helper function that enables `underscore` to work with a `null` value.

## unwrap

### unwrap(text: String, wrapper: String): String

Unwraps a given `text` from a `wrapper` text.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
<code>text</code>	The input string.
<code>wrapper</code>	The text used to unwrap.

## Example

This example shows how `unwrap` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import unwrap from dw::core::Strings
output application/json
---
{
    "a": unwrap(null, ""),
    "b": unwrap(null, '\0'),
    "c": unwrap("abc", ""),
    "d": unwrap("AABabcBAA", 'A'),
    "e": unwrap("A", '#'),
    "f": unwrap("#A", '#'),
    "g": unwrap("A#", '#')
}
```

## Output

```
{
    "a": null,
    "b": null,
    "c": "abc",
    "d": "ABabcBA",
    "e": "A",
    "f": "A#",
    "g": "#A"
}
```

## unwrap(`text`: Null, `wrapper`: String): Null

Helper function that enables `unwrap` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## withMaxSize

### withMaxSize(text: String, maxLength: Number): String

Checks that the string length is no larger than the specified `maxLength`. If the string's length is larger than the `maxLength`, the function cuts characters from left to right, until the string length meets the length limit.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
<code>text</code>	The input string.
<code>maxLength</code>	The maximum length of the string.

#### Example

This example shows how `withMaxSize` behaves with different `maxLength` values for the same string. Note that if `withMaxSize` is 0, the function returns an unmodified string. If the input is `null`, the output is always `null`.

#### Source

```
%dw 2.0
import withMaxSize from dw::core::Strings
output application/json
---
{
    a: "123" withMaxSize 10,
    b: "123" withMaxSize 3,
    c: "123" withMaxSize 2,
    d: "123" withMaxSize 0,
    e: null withMaxSize 23,
}
```

#### Output

```
{
    "a": "123",
    "b": "123",
    "c": "12",
    "d": "123",
    "e": null
}
```

### withMaxSize(text: Null, maxLength: Number): Null

Helper function that enables `withMaxSize` to work with a `null` value.

*Introduced in DataWeave version 2.3.0.*

## words

### `words(text: String): Array<String>`

Returns an array of words from a string.

Separators between words include blank spaces, new lines, and tabs.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>text</code>	The string to split into words.

#### Example

This example divides a string by the `words` it contains. An `\n` represents a line break, and `\t` represents a tab.

#### Source

```
%dw 2.0
import words from dw::core::Strings
output application/json
---
words("hello world\nhere\t\t\data-weave")
```

#### Output

```
["hello", "world", "here", "data-weave"]
```

### `words(text: Null): Null`

Helper function that enables `words` to work with a `null` value.

*Introduced in DataWeave version 2.4.0.*

## wrapIfMissing

### `wrapIfMissing(text: String, wrapper: String): String`

Wraps `text` with `wrapper` if that `wrapper` is missing from the start or end of the given string.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
text	The input string.
wrapper	The content used to wrap.

## Example

This example shows how `wrapIfMissing` behaves with different inputs and sizes.

## Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "a": wrapIfMissing(null, ""),
  "b": wrapIfMissing("", ""),
  "c": wrapIfMissing("ab", "x"),
  "d": wrapIfMissing("ab'", ""),
  "e": wrapIfMissing("/", '/'),
  "f": wrapIfMissing("a/b/c", '/'),
  "g": wrapIfMissing("/a/b/c", '/'),
  "h": wrapIfMissing("a/b/c/", '/')
}
```

## Output

```
{
  "a": null,
  "b": "",
  "c": "xabx",
  "d": "'ab'",
  "e": "/",
  "f": "/a/b/c/",
  "g": "/a/b/c/",
  "h": "/a/b/c/"}
```

## `wrapIfMissing(text: Null, wrapper: String): Null`

Helper function that enables `wrapIfMissing` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

## wrapWith

**wrapWith(text: String, wrapper: String): String**

Wraps the specified `text` with the given `wrapper`.

*Introduced in DataWeave version 2.2.0.*

### Parameters

Name	Description
<code>text</code>	The input string.
<code>wrapper</code>	The content used to wrap.

### Example

This example shows how `wrapWith` behaves with different inputs and sizes.

### Source

```
%dw 2.0
import * from dw::core::Strings
output application/json
---
{
  "a": wrapWith(null, ""),
  "b": wrapWith("", ""),
  "c": wrapWith("ab", "x"),
  "d": wrapWith("'ab'", ""),
  "e": wrapWith("ab", "")
}
```

### Output

```
{
  "a": null,
  "b": "",
  "c": "xabx",
  "d": "'ab'",
  "e": "'ab'"}
```

**wrapWith(text: Null, wrapper: Any): Null**

Helper function that enables `wrapWith` to work with a `null` value.

*Introduced in DataWeave version 2.2.0.*

# dw::core::Types

This module enables you to perform type introspection.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::Types` to the header of your DataWeave script.

*Introduced in DataWeave version 2.3.0.*

## Functions

Name	Description
<code>arrayItem</code>	Returns the type of the given array. This function fails if the input is not an Array type.
<code>baseTypeOf</code>	Returns the base type of the given type.
<code>functionParamTypes</code>	Returns the list of parameters from the given function type. This function fails if the provided type is not a Function type.
<code>functionReturnType</code>	Returns the type of a function's return type. This function fails if the input type is not a Function type.
<code>intersectionItems</code>	Returns an array of all the types that define a given Intersection type. This function fails if the input is not an Intersection type.
<code>isAnyType</code>	Returns <code>true</code> if the input is the Any type.
<code>isArrayType</code>	Returns <code>true</code> if the input type is the Array type.
<code>isBinaryType</code>	Returns <code>true</code> if the input is the Binary type.
<code>isBooleanType</code>	Returns <code>true</code> if the input is the Boolean type.
<code>isDateTimeType</code>	Returns <code>true</code> if the input is the DateTime type.
<code>isDateType</code>	Returns <code>true</code> if the input is the Date type.
<code>isFunctionType</code>	Returns <code>true</code> if the input is the Function type.
<code>isIntersectionType</code>	Returns <code>true</code> if the input type is the Intersection type.
<code>isKeyType</code>	Returns <code>true</code> if the input is the Key type.
<code>isLiteralType</code>	Returns <code>true</code> if the input is the Literal type.
<code>isLocalDateTimeType</code>	Returns <code>true</code> if the input is the LocalDateTime type.
<code>isLocalTimeType</code>	Returns <code>true</code> if the input is the LocalTime type.
<code>isNamespaceType</code>	Returns <code>true</code> if the input is the Namespace type.
<code>isNothingType</code>	Returns <code>true</code> if the input is the Nothing type.
<code>isNullType</code>	Returns <code>true</code> if the input is the Null type.
<code>isNumberType</code>	Returns <code>true</code> if the input is the Number type.
<code>isObjectType</code>	Returns <code>true</code> if the input is the Object type.
<code>isPeriodType</code>	Returns <code>true</code> if the input is the Period type.

Name	Description
<code>isRangeType</code>	Returns <code>true</code> if the input is the Range type.
<code>isReferenceType</code>	Returns <code>true</code> if the input type is a Reference type.
<code>isRegexType</code>	Returns <code>true</code> if the input is the Regex type.
<code>isStringType</code>	Returns <code>true</code> if the input is the String type.
<code>isTimeType</code>	Returns <code>true</code> if the input is the Time type.
<code>isTimeZoneType</code>	Returns <code>true</code> if the input is the TimeZone type.
<code>isTypeType</code>	Returns <code>true</code> if the input is the Type type.
<code>isUnionType</code>	Returns <code>true</code> if the input type is the Union type.
<code>isUriType</code>	Returns <code>true</code> if the input is the Uri type.
<code>literalValueOf</code>	Returns the value of an input belongs to the Literal type.
<code>metadataOf</code>	Returns metadata that is attached to the given type.
<code>nameOf</code>	Returns the name of the input type.
<code>objectFields</code>	Returns the array of fields from the given Object type. This function fails if the type is not an Object type.
<code>unionItems</code>	Returns an array of all the types that define a given Union type. This function fails if the input is not a Union type.

## Types

- [Types Types](#)

## arrayItem

### arrayItem(t: Type): Type

Returns the type of the given array. This function fails if the input is not an Array type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
<code>t</code>	The type to check.

#### Example

This example shows how `arrayItem` behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Types
type ArrayOfString = Array<String>
type ArrayOfNumber = Array<Number>
type ArrayOfAny = Array<Any>
type ArrayOfAnyDefault = Array<Any>
output application/json
---
{
    a: arrayItem(ArrayOfString),
    b: arrayItem(ArrayOfNumber),
    c: arrayItem(ArrayOfAny),
    d: arrayItem(ArrayOfAnyDefault)
}
```

## Output

```
{
    "a": "String",
    "b": "Number",
    "c": "Any",
    "d": "Any"
}
```

## baseTypeOf

### baseTypeOf(t: Type): Type

Returns the base type of the given type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how `baseTypeOf` behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Types
type AType = String {format: "YYYY-MM-dd"}
output application/json
---
{
    a: baseTypeOf(AType)
}
```

## Output

```
{
    "a": "String"
}
```

## functionParamTypes

**functionParamTypes(t: Type): Array<FunctionParam>**

Returns the list of parameters from the given function type. This function fails if the provided type is not a Function type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The function type.

### Example

This example shows how **functionParamTypes** behaves with different inputs.

### Source

```
%dw 2.0
output application/json
import * from dw::core::Types
type AFunction = (String, Number) -> Number
type AFunction2 = () -> Number
---
{
    a: functionParamTypes(AFunction),
    b: functionParamTypes(AFunction2)
}
```

## Output

```
{  
  "a": [  
    {  
      "paramType": "String",  
      "optional": false  
    },  
    {  
      "paramType": "Number",  
      "optional": false  
    }  
,  
  ],  
  "b": [  
  ]  
}
```

## functionReturnType

**functionReturnType(t: Type): Type | Null**

Returns the type of a function's return type. This function fails if the input type is not a Function type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The function type.

### Example

This example shows how `functionReturnType` behaves with different inputs.

### Source

```
%dw 2.0  
output application/json  
import * from dw::core::Types  
type AFunction = (String, Number) -> Number  
type AFunction2 = () -> Number  
---  
{  
  a: functionReturnType(AFunction),  
  b: functionReturnType(AFunction2)  
}
```

## Output

```
{  
  "a": "Number",  
  "b": "Number"  
}
```

## intersectionItems

### intersectionItems(t: Type): Array<Type>

Returns an array of all the types that define a given Intersection type. This function fails if the input is not an Intersection type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how `intersectionItems` behaves with different inputs. Note that the `AType` variable defines an Intersection type `{name: String} & {age: Number}` by using an `&` between the two objects.

#### Source

```
%dw 2.0  
import * from dw::core::Types  
type AType = {name: String} & {age: Number}  
output application/json  
---  
{  
  a: intersectionItems(AType)  
}
```

## Output

```
{  
  "a": ["Object", "Object"]  
}
```

## isAnyType

**isAnyType(t: Type): Boolean**

Returns **true** if the input is the Any type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isAnyType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type AAny = Any
output application/json
---
{
    a: isAnyType(AAny),
    b: isAnyType(Any),
    c: isAnyType(String),
}
```

### Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isArrayType

**isArrayType(t: Type): Boolean**

Returns **true** if the input type is the Array type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

## Example

This example shows how `isArrayType` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type AType = Array<String>
output application/json
---
{
    a: isArrayType(AType),
    b: isArrayType(Boolean),
}
```

## Output

```
{
    "a": true,
    "b": false
}
```

## isBinaryType

`isBinaryType(t: Type): Boolean`

Returns `true` if the input is the Binary type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to check.

## Example

This example shows how `isBinaryType` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ABinary = Binary
output application/json
---
{
    a: isBinaryType(ABinary),
    b: isBinaryType(Binary),
    c: isBinaryType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isBooleanType

**isBooleanType(t: Type): Boolean**

Returns **true** if the input is the Boolean type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isBooleanType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ABoolean = Boolean
output application/json
---
{
    a: isBooleanType(ABoolean),
    b: isBooleanType(Boolean),
    c: isBooleanType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

# isDateTimeType

**isDateTimeType(t: Type): Boolean**

Returns **true** if the input is the DateTime type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to check.

## Example

This example shows how **isDateTimeType** behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ADateTime = DateTime
output application/json
---
{
    a: isDateTimeType(ADateTime),
    b: isDateTimeType(DateTime),
    c: isDateTimeType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isDateTime

**isDateTime(t: Type): Boolean**

Returns **true** if the input is the Date type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isDateTime** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ADate = Date
output application/json
---
{
    a: isDateType(ADate),
    b: isDateType(Date),
    c: isDateType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isFunctionType

**isFunctionType(t: Type): Boolean**

Returns **true** if the input is the Function type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isFunctionType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type AFunction = (String) -> String
output application/json
---
{
    a: isFunctionType(AFunction),
    b: isFunctionType(Boolean)
}
```

## Output

```
{  
  "a": true,  
  "b": false  
}
```

## isIntersectionType

**isIntersectionType(t: Type): Boolean**

Returns **true** if the input type is the Intersection type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isIntersectionType** behaves with different inputs.

### Source

```
%dw 2.0  
import * from dw::core::Types  
type AType = {name: String} & {age: Number}  
output application/json  
---  
{  
  a: isIntersectionType(AType),  
  b: isIntersectionType(Boolean),  
}
```

## Output

```
{  
  "a": true,  
  "b": false  
}
```

## isKeyType

**isKeyType(t: Type): Boolean**

Returns **true** if the input is the Key type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how **isKeyType** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Types
type AKey = Key
output application/json
---
{
    a: isKeyType(AKey),
    b: isKeyType(Key),
    c: isKeyType(Boolean),
}
```

#### Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isLiteralType

**isLiteralType(t: Type): Boolean**

Returns **true** if the input is the Literal type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

## Example

This example shows how `isLiteralType` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ALiteralType = "Mariano"
output application/json
---
{
    a: isLiteralType(ALiteralType),
    b: isLiteralType(Boolean)
}
```

## Output

```
{
    "a": true,
    "b": false
}
```

## isLocalDateTimeType

**isLocalDateTimeType(t: Type): Boolean**

Returns `true` if the input is the `LocalDateTime` type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to check.

## Example

This example shows how `isLocalDateTimeType` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ALocalDateTime = LocalDateTime
output application/json
---
{
    a: isLocalDateTimeType(ALocalDateTime),
    b: isLocalDateTimeType(LocalDateTime),
    c: isLocalDateTimeType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

# isLocalTimeType

**isLocalTimeType(t: Type): Boolean**

Returns **true** if the input is the LocalTime type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to check.

## Example

This example shows how **isLocalTimeType** behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ALocalTime = LocalTime
output application/json
---
{
    a: isLocalTimeType(ALocalTime),
    b: isLocalTimeType(LocalTime),
    c: isLocalTimeType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isNamespaceType

**isNamespaceType(t: Type): Boolean**

Returns **true** if the input is the Namespace type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isNamespaceType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ANamespace = Namespace
output application/json
---
{
    a: isNamespaceType(ANamespace),
    b: isNamespaceType(Namespace),
    c: isNamespaceType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isNothingType

**isNothingType(t: Type): Boolean**

Returns **true** if the input is the Nothing type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isNothingType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ANothing = Nothing
output application/json
---
{
    a: isNothingType(ANothing),
    b: isNothingType(Nothing),
    c: isNothingType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isNullType

### isNullType(t: Type): Boolean

Returns **true** if the input is the Null type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how **isNullType** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Types
type ANull = Null
output application/json
---
{
    a: isNullType(ANull),
    b: isNullType(Null),
    c: isNullType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

# isNumberType

**isNumberType(t: Type): Boolean**

Returns **true** if the input is the Number type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to check.

## Example

This example shows how **isNumberType** behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ANumber = Number
output application/json
---
{
    a: isNumberType(ANumber),
    b: isNumberType(Number),
    c: isNumberType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isObjectType

**isObjectType(t: Type): Boolean**

Returns **true** if the input is the Object type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isObjectType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type AType = {name: String}
output application/json
---
{
    a: isObjectType(AType),
    b: isObjectType(Boolean),
}
```

## Output

```
{  
  "a": true,  
  "b": false  
}
```

## isPeriodType

**isPeriodType(t: Type): Boolean**

Returns **true** if the input is the Period type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isPeriodType** behaves with different inputs.

### Source

```
%dw 2.0  
import * from dw::core::Types  
type APeriod = Period  
output application/json  
---  
{  
  a: isPeriodType(APeriod),  
  b: isPeriodType(Period),  
  c: isPeriodType(String),  
}
```

## Output

```
{  
  "a": true,  
  "b": true,  
  "c": false  
}
```

## isRangeType

**isRangeType(t: Type): Boolean**

Returns **true** if the input is the Range type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isRangeType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ARange = Range
output application/json
---
{
    a: isRangeType(ARange),
    b: isRangeType(Range),
    c: isRangeType(String),
}
```

### Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isReferenceType

**isReferenceType(t: Type): Boolean**

Returns **true** if the input type is a Reference type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

## Example

This example shows how `isReferenceType` behaves with different inputs.

## Source

```
%dw 2.0
output application/json
import * from dw::core::Types
type AArray = Array<String> {n: 1}
type AArray2 = Array<AArray>
---
{
    a: isReferenceType( AArray ),
    b: isReferenceType(arrayItem(AArray2)),
    c: isReferenceType(String)
}
```

## Output

```
{
    "a": false,
    "b": true,
    "c": false
}
```

# isRegexType

## isRegexType(t: Type): Boolean

Returns `true` if the input is the Regex type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to check.

## Example

This example shows how `isRegexType` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type ARegex = Regex
output application/json
---
{
    a: isRegexType(ARegex),
    b: isRegexType(Regex),
    c: isRegexType(Boolean),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isStringType

**isStringType(t: Type): Boolean**

Returns **true** if the input is the String type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isStringType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type AString = String
output application/json
---
{
    a: isStringType(AString),
    b: isStringType(String),
    c: isStringType(Boolean),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

# isTimeType

## isTimeType(t: Type): Boolean

Returns **true** if the input is the Time type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isTimeType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ATime = Time
output application/json
---
{
    a: isTimeType(ATime),
    b: isTimeType(Time),
    c: isTimeType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isTimeZoneType

**isTimeZoneType(t: Type): Boolean**

Returns **true** if the input is the TimeZone type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isTimeZoneType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type ATimeZone = TimeZone
output application/json
---
{
    a: isTimeZoneType(ATimeZone),
    b: isTimeZoneType(TimeZone),
    c: isTimeZoneType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

# isTypeType

## isTypeType(t: Type): Boolean

Returns **true** if the input is the Type type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how **isTypeType** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type AType = Type
output application/json
---
{
    a: isTypeType(AType),
    b: isTypeType(Type),
    c: isTypeType(String),
}
```

## Output

```
{
    "a": true,
    "b": true,
    "c": false
}
```

## isUnionType

### isUnionType(t: Type): Boolean

Returns **true** if the input type is the Union type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how **isUnionType** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Types
type AType = String | Number
output application/json
---
{
    a: isUnionType(AType),
    b: isUnionType(Boolean),
}
```

## Output

```
{  
  "a": true,  
  "b": false  
}
```

## isUriType

### isUriType(t: Type): Boolean

Returns **true** if the input is the Uri type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how **isUriType** behaves with different inputs.

#### Source

```
%dw 2.0  
import * from dw::core::Types  
type AUri = Uri  
output application/json  
---  
{  
  a: isUriType(AUri),  
  b: isUriType(Uri),  
  c: isUriType(String),  
}
```

## Output

```
{  
  "a": true,  
  "b": true,  
  "c": false  
}
```

## literalValueOf

**literalValueOf(t: Type): String | Number | Boolean**

Returns the value of an input belongs to the Literal type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

### Example

This example shows how `literalValueOf` behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::core::Types
type AType = "Mariano"
output application/json
---
{
    a: literalValueOf(AType)
}
```

### Output

```
{
    "a": "Mariano"
}
```

## metadataOf

**metadataOf(t: Type): Object**

Returns metadata that is attached to the given type.

*Introduced in DataWeave version 2.3.0.*

### Parameters

Name	Description
t	The type to check.

## Example

This example shows how `metadataOf` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::core::Types
type AType = String {format: "YYYY-MM-dd"}
output application/json
---
{
    a: metadataOf(AType)
}
```

## Output

```
{
    "a": {"format": "YYYY-MM-dd"}
```

## nameOf

`nameOf(t: Type): String`

Returns the name of the input type.

*Introduced in DataWeave version 2.3.0.*

## Parameters

Name	Description
t	The type to query

## Example

This example shows how `nameOf` behaves with different inputs.

## Source

```
%dw 2.0
output application/json
import * from dw::core::Types
type AArray = Array<String> {n: 1}
type AArray2 = Array<String>
---
{
    a: nameOf(AArray),
    b: nameOf(AArray2),
    c: nameOf(String)
}
```

## Output

```
{
    "a": "AArray",
    "b": "AArray2",
    "c": "String"
}
```

## objectFields

### objectFields(t: Type): Array<Field>

Returns the array of fields from the given Object type. This function fails if the type is not an Object type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The function type.

#### Example

This example shows how `objectFields` behaves with different inputs.

#### Source

```

import * from dw::core::Types
ns ns0 http://acme.com
type ADictionary = {_ : String}
type ASchema = {ns0#name @(ns0#foo: String): {}}
type AUser = {name @(foo?: String,l: Number)?: String, lastName*: Number}
---
{
  a: objectFields(ADictionary),
  b: objectFields(ASchema),
  c: objectFields(Object),
  d: objectFields(AUser)
}

```

## Output

```
{
  "a": [
    {
      "key": {
        "name": {
          "localName": "_",
          "namespace": null
        },
        "attributes": [
          {}
        ],
        "required": true,
        "repeated": false,
        "value": "String"
      }
    },
    "b": [
      {
        "key": {
          "name": {
            "localName": "name",
            "namespace": "http://acme.com"
          },
          "attributes": [
            {
              "name": {
                "localName": "foo",
                "namespace": "http://acme.com"
              },
              "value": "String",
              "required": true
            }
          ]
        }
      }
    }
  ]
}
```

```
        },
        "required": true,
        "repeated": false,
        "value": "Object"
    }
],
"c": [
],
"d": [
{
    "key": {
        "name": {
            "localName": "name",
            "namespace": null
        },
        "attributes": [
            {
                "name": {
                    "localName": "foo",
                    "namespace": null
                },
                "value": "String",
                "required": false
            },
            {
                "name": {
                    "localName": "l",
                    "namespace": null
                },
                "value": "Number",
                "required": true
            }
        ]
    },
    "required": false,
    "repeated": false,
    "value": "String"
},
{
    "key": {
        "name": {
            "localName": "lastName",
            "namespace": null
        },
        "attributes": [
        ],
        "required": true,
        "repeated": true,
```

```
        "value": "Number"
    }
]
}
```

## unionItems

### unionItems(t: Type): Array<Type>

Returns an array of all the types that define a given Union type. This function fails if the input is not a Union type.

*Introduced in DataWeave version 2.3.0.*

#### Parameters

Name	Description
t	The type to check.

#### Example

This example shows how `unionItems` behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::core::Types
type AType = String | Number
output application/json
---
{
    a: unionItems(AType)
}
```

#### Output

```
{
    "a": ["String", "Number"]
}
```

## Types Types

Type	Definition	Description
Attribute	<pre>type Attribute = { name: QName, required: Boolean, value: Type }</pre>	Represents an Attribute definition that is part of an Object field Key.  <i>Introduced in DataWeave version 2.3.0.</i>
Field	<pre>type Field = { key: { name: QName, attributes: Array&lt;Attribute&gt; }, required: Boolean, repeated: Boolean, value: Type }</pre>	Represents a Field description that is part of an Object.  <i>Introduced in DataWeave version 2.3.0.</i>
FunctionParam	<pre>type FunctionParam = { paramType: Type, optional: Boolean }</pre>	Represents a Function parameter that is part of a Function type.  <i>Introduced in DataWeave version 2.3.0.</i>
QName	<pre>type QName = { localName: String, namespace: Namespace   Null }</pre>	Represents a Qualified Name definition with a <code>localName</code> (a string) and a <code>namespace</code> . If the QName does not have a Namespace, its value is <code>null</code> .  <i>Introduced in DataWeave version 2.3.0.</i>

## dw::core::URL

This module contains helper functions for working with URIs.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::core::URL` to the header of your DataWeave script.

### Functions

Name	Description
<code>compose</code>	Uses a custom string interpolator to replace URL components with a <code>encodeURIComponent</code> result. You can call this function using the standard call, or a simplified version.
<code>decodeURI</code>	Decodes the escape sequences (such as <code>%20</code> ) in a URI.
<code>decodeURIComponent</code>	Decodes a Uniform Resource Identifier (URI) component previously created by <code>encodeURIComponent</code> or a similar routine.
<code>encodeURI</code>	Encodes a URI with UTF-8 escape sequences.
<code>encodeURIComponent</code>	Escapes certain characters in a URI component using UTF-8 encoding.
<code>parseURI</code>	Parses a URL and returns a <code>URI</code> object.

## Types

- [URL Types](#)

## compose

### compose(parts: Array<String>, interpolation: Array<String>): String

Uses a custom string interpolator to replace URL components with a [encodeURIComponent](#) result. You can call this function using the standard call, or a simplified version.

#### Parameters

Name	Description
<code>baseStringArray</code>	A string array that contains the URL parts to interpolate using the strings in the <code>interpolationStringArray</code> parameter.
<code>interpolationStringArray</code>	A string array that contains the strings used to interpolate the <code>baseStringArray</code> .

#### Example

The following example uses the compose function to form an encoded URL, the first parameter is an array of two strings that are part of the URL and the second parameter is the `urlPath` variable that is used to interpolate the strings in the first parameter. Notice that the spaces in the input are encoded in the output URL as `%20`.

#### Source

```
%dw 2.0
output application/json
var urlPath = "content folder"
import * from dw::core::URL
---
{ "encodedURL" : compose(["http://examplewebsite.com/", "/page.html"],
["$(urlPath)"] ) }
```

#### Output

```
{ "encodedURL" : "http://examplewebsite.com/content%20folder/page.html" }
```

#### Example

You can also call this function using the simplified syntax, which uses backticks (`) to enclose the string that includes the variable to encode. This example shows how to use the simplified syntax to obtain the same result as in the previous example.

#### Source

```
%dw 2.0
output application/json
var urlPath = "content folder"
import * from dw::core::URL
---
{ "encodedURL" : compose `http://examplewebsite.com/${urlPath}/page.html`}
```

## Output

```
{ "encodedURL" : "http://examplewebsite.com/content%20folder/page.html" }
```

## decodeURI

### decodeURI(text: String): String

Decodes the escape sequences (such as `%20`) in a URI.

The function replaces each escape sequence in the encoded URI with the character that it represents, but does not decode escape sequences that could not have been introduced by `encodeURI`. The character `#` is not decoded from escape sequences.

#### Parameters

Name	Description
<code>text</code>	The URI to decode.

#### Example

This example decodes a URI that contains the URL percent encoding `%20`, which is used for spaces.

#### Source

```
%dw 2.0
import * from dw::core::URL
output application/json
---
{
  "decodeURI" : decodeURI('http://asd/%20text%20to%20decode%20/text')
}
```

## Output

```
{
  "decodeURI": "http://asd/ text to decode /text"
}
```

## decodeURIComponent

### decodeURIComponent(text: String): String

Decodes a Uniform Resource Identifier (URI) component previously created by [encodeURIComponent](#) or a similar routine.

For an example, see [encodeURIComponent](#).

#### Parameters

Name	Description
text	URI component string.

#### Example

This example decodes a variety of URI components.

#### Source

```
%dw 2.0
import * from dw::core::URL
output application/json
---
{
    "decodeURIComponent": {
        "decodeURIComponent": decodeURIComponent("%20PATH/%20TO%20/DECODE%20"),
        "decodeURIComponent": decodeURIComponent("%3B%2C%2F%3F%3A%40%26%3D"),
        "decodeURIComponent": decodeURIComponent("%2D%5F%2E%21%7E%2A%27%28%29%24"),
    }
}
```

#### Output

```
{
    decodeURIComponent: {
        decodeURIComponent: " PATH/ TO /DECODE ",
        decodeURIComponent: ";,/?:@&=",
        decodeURIComponent: "-_.!~*'()\\$"
    }
}
```

## encodeURIComponent

### encodeURIComponent(text: String): String

Encodes a URI with UTF-8 escape sequences.

Applies up to four escape sequences for characters composed of two "surrogate" characters. The function assumes that the URI is a complete URI, so it does not encode reserved characters that have special meaning.

The function *does not encode these characters* with UTF-8 escape sequences:

Type (not escaped)	Examples
Reserved characters	; , / ? : @ & = \$
Unescaped characters	alphabetic, decimal digits, - _ . ! ~ * ' ( )
Number sign	#

## Parameters

Name	Description
text	The URI to encode.

## Example

This example shows encodes spaces in one URL and lists some characters that do not get encoded in the `not_encoded` string.

## Source

```
%dw 2.0
import * from dw::core::URL
output application/json
---
{
    "encodeURI" : encodeURI("http://asd/ text to decode /text"),
    "not_encoded": encodeURI("http://:;/?:@&=\$\_._.!~*'()")
}
```

## Output

```
{
    "encodeURI": "http://asd/%20text%20to%20decode%20/%25/\\"/text",
    "not_encoded": "http://:;/?:@&=$_._.!~*'()"
}
```

## encodeURIComponent

### encodeURIComponent(text: String): String

Escapes certain characters in a URI component using UTF-8 encoding.

There can be only four escape sequences for characters composed of two "surrogate" \* characters.

`encodeURIComponent` escapes all characters *except the following*: alphabetic, decimal digits, - \_ . ! ~ \* ' ( ). Note that `encodeURIComponent` differs from `encodeURI` in that it encodes reserved characters and the Number sign # of `encodeURI`:

Type	Includes
Reserved characters	
Unescaped characters	alphabetic, decimal digits, - _ . ! ~ * ' ( )
Number sign	

## Parameters

Name	Description
<code>text</code>	URI component string.

## Example

This example encodes a variety of URI components.

## Source

```
%dw 2.0
import * from dw::core::URL
output application/json
---
{
    "comparing_encode_functions_output" : {
        "encodeURIComponent" : encodeURI(" PATH/ TO /ENCODE "),
        "encodeURI" : encodeURI(" PATH/ TO /ENCODE "),
        "encodeURIComponent_to_hex" : encodeURIComponent("; , /?:@&="),
        "encodeURI_not_to_hex" : encodeURI("; , /?:@&="),
        "encodeURIComponent_not_encoded" : encodeURIComponent("-_.!~*'()"),
        "encodeURI_not_encoded" : encodeURI("-_.!~*'()")
    }
}
```

## Output

```
{
  "comparing_encode_functions_output": {
    "encodeURIComponent": "%20PATH%20T0%20/ENCODE%20",
    "encodeURI": "%20PATH%20T0%20/ENCODE%20",
    "encodeURIComponent_to_hex": "%3B%2C%2F%3F%3A%40%26%3D",
    "encodeURI_not_to_hex": ";,/?:@&=",
    "encodeURIComponent_not_encoded": "-_.!~*'()",
    "encodeURI_not_encoded": "-_.!~*'()"
  }
}
```

## parseURI

### parseURI(uri: String): URI

Parses a URL and returns a [URI](#) object.

The `isValid: Boolean` property in the output [URI](#) object indicates whether the parsing process succeeded. Every field in this object is optional, and a field will appear in the output only if it was present in the URL input.

#### Parameters

Name	Description
<code>uri</code>	The URI input.

#### Example

This example parses a URL.

#### Source

```
%dw 2.0
import * from dw::core::URL
output application/json
---
{
  'composition':
  parseURI('https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#footer')
}
```

#### Output

```
{
  "composition": {
    "isValid": true,
    "raw": "https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#footer",
    "host": "en.wikipedia.org",
    "authority": "en.wikipedia.org",
    "fragment": "footer",
    "path": "/wiki/Uniform_Resource_Identifier",
    "scheme": "https",
    "isAbsolute": true,
    "isOpaque": false
  }
}
```

## URL Types

Type	Definition	Description
URI	<code>type URI = { isValid: Boolean, host?: String, authority?: String, fragment?: String, path?: String, port?: Number, query?: String, scheme?: String, user?: String, isAbsolute?: Boolean, isOpaque?: Boolean }</code>	Describes the URI type. For descriptions of the fields, see <a href="#">URL Types (dw::core::URL)</a> .

## dw::Crypto

This module provide functions that perform encryptions through common algorithms, such as MD5, SHA1, and so on.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::Crypto` to the header of your DataWeave script.

## Functions

Name	Description
<a href="#">HMACBinary</a>	Computes an HMAC hash (with a secret cryptographic key) on input content.
<a href="#">HMACWith</a>	Computes an HMAC hash (with a secret cryptographic key) on input content, then transforms the result into a lowercase, hexadecimal string.
<a href="#">MD5</a>	Computes the MD5 hash and transforms the binary result into a hexadecimal lower case string.
<a href="#">SHA1</a>	Computes the SHA1 hash and transforms the result into a hexadecimal, lowercase string.

Name	Description
<a href="#">hashWith</a>	Computes the hash value of binary content using a specified algorithm.

## HMACBinary

**HMACBinary(secret: Binary, content: Binary, algorithm: String = "HmacSHA1"): Binary**

Computes an HMAC hash (with a secret cryptographic key) on input content.

See also, [HMACWith](#).

### Parameters

Name	Description
<b>secret</b>	The secret cryptographic key (a binary value) used when encrypting the content.
<b>content</b>	The binary input value.
<b>algorithm</b>	The hashing algorithm. <code>HmacSHA1</code> is the default. Valid values depend on the JDK version you are using. For JDK 8 and JDK 11, <code>HmacMD5</code> , <code>HmacSHA1</code> , <code>HmacSHA224</code> , <code>HmacSHA256</code> , <code>HmacSHA384</code> , and <code>HmacSHA512</code> are valid algorithms. For JDK 11, <code>HmacSHA512/224</code> and <code>HmacSHA512/256</code> are also valid.

### Example

This example uses HMAC with a secret value to encrypt the input content.

### Source

```
%dw 2.0
import dw::Crypto
output application/json
---
{
  "HMACBinary" : Crypto::HMACBinary("confidential" as Binary, "xxxxx" as Binary,
  "HmacSHA512")
}
```

### Output

```
{
  "HMACBinary": 
    "\ufffd\ufffd\ufffd\ufffd^h\ufffd!3\u0005\ufffd\u00017\ufffd\ufffd`\ufffd8?\ufffdn7\ufffdbs;\t\ufffd\ufffd\ufffdx&g\ufffd~\ufffd\ufffd%\ufffd7>1\ufffdK\u000e@\ufffdC\u0011\ufffdT\ufffd}W"
}
```

## HMACWith

**HMACWith(secret: Binary, content: Binary, @Since(version = "2.2.0") algorithm: String = "HmacSHA1"): String**

Computes an HMAC hash (with a secret cryptographic key) on input content, then transforms the result into a lowercase, hexadecimal string.

See also, [HMACBinary](#).

### Parameters

Name	Description
<code>secret</code>	The secret cryptographic key (a binary value) used when encrypting the <code>content</code> .
<code>content</code>	The binary input value.
<code>algorithm</code>	(Introduced in DataWeave 2.2.0. Supported by Mule 4.2 and later.) The hashing algorithm. By default, <code>HmacSHA1</code> is used. Other valid values are <code>HmacSHA256</code> and <code>HmacSHA512</code> .

### Example

This example uses HMAC with a secret value to encrypt the input content using the `HmacSHA256` algorithm.

### Source

```
%dw 2.0
import dw::Crypto
output application/json
---
{ "HMACWith" : Crypto::HMACWith("secret_key" as Binary, "Some value to hash" as Binary, "HmacSHA256") }
```

### Output

```
{ "HMACWith": "b51b4fe8c4e37304605753272b5b4321f9644a9b09cb1179d7016c25041d1747" }
```

## MD5

### MD5(content: Binary): String

Computes the MD5 hash and transforms the binary result into a hexadecimal lower case string.

#### Parameters

Name	Description
content	A binary input value to encrypt.

#### Example

This example uses the MD5 algorithm to encrypt a binary value.

#### Source

```
%dw 2.0
import dw::Crypto
output application/json
---
{ "md5" : Crypto::MD5("asd" as Binary) }
```

#### Output

```
{ "md5": "7815696ecbf1c96e6894b779456d330e" }
```

## SHA1

### SHA1(content: Binary): String

Computes the SHA1 hash and transforms the result into a hexadecimal, lowercase string.

#### Parameters

Name	Description
content	A binary input value to encrypt.

#### Example

This example uses the SHA1 algorithm to encrypt a binary value.

#### Source

```
%dw 2.0
import dw::Crypto
output application/json
---
{ "sha1" : Crypto::SHA1("dsasd" as Binary) }
```

## Output

```
{ "sha1": "2fa183839c954e6366c206367c9be5864e4f4a65" }
```

## hashWith

### hashWith(content: Binary, algorithm: String = "SHA-1"): Binary

Computes the hash value of binary content using a specified algorithm.

The first argument specifies the binary content to use to calculate the hash value, and the second argument specifies the hashing algorithm to use. The second argument must be any of the accepted Algorithm names:

Algorithm names	Description
MD2	The MD2 message digest algorithm as defined in <a href="#">RFC 1319</a> .
MD5	The MD5 message digest algorithm as defined in <a href="#">RFC 1321</a> .
SHA-1, SHA-256, SHA-384, SHA-512	Hash algorithms defined in the <a href="#">FIPS PUB 180-2</a> . SHA-256 is a 256-bit hash function intended to provide 128 bits of security against collision attacks, while SHA-512 is a 512-bit hash function intended to provide 256 bits of security. A 384-bit hash may be obtained by truncating the SHA-512 output.

## Parameters

Name	Description
content	The binary input value to hash.
algorithm	The name of the algorithm to use for calculating the hash value of <b>content</b> . This value is a string. Defaults to <b>SHA-1</b> .

## Example

This example uses the MD2 algorithm to encrypt a binary value.

## Source

```
%dw 2.0
import dw::Crypto
output application/json
---
{ "md2" : Crypto::hashWith("hello" as Binary, "MD2") }
```

## Output

```
{ "md2": "\ufffd\u0041s\ufffd\u0031\ufffd\ufffd}8\u0041\ufffd\u006U" }
```

# dw::extension::DataFormat

This module provides resources for registering a new data format for the DataWeave language.

For an example, see [Custom Data Formats Example](#).

## Types

- [DataFormat Types](#)

## Annotations

- [DataFormat Annotations](#)

## DataFormat Types

Type	Definition	Description
DataFormat	<pre>type DataFormat = {     binaryFormat?: Boolean,     defaultCharset?: String,     fileExtensions?: Array&lt;String&gt;,     acceptedMimeTypes?: Array&lt;MimeType&gt;,     reader: (content: Binary, settings: ReaderSettings) -&gt; Any,     writer: (value: Any, settings: WriterSettings) -&gt; Binary }</pre>	<p>Represents the <code>DataFormat</code> definition and contains the following fields:</p> <ul style="list-style-type: none"> <li>• <code>binaryFormat</code>: True if this is data format is represented as binary representation instead of text. False if not present.</li> <li>• <code>defaultCharset</code>: Default character set of this format, if any.</li> <li>• <code>fileExtensions</code>: Returns the list of file extensions with the <code>.</code> (for example, <code>.json</code>, <code>.xml</code>) to assign to this data format.</li> <li>• <code>acceptedMimeTypes</code>: The list of MIME types to accept.</li> <li>• <code>reader</code>: Function that reads raw content and transforms it into the canonical DataWeave model.</li> <li>• <code>writer</code>: Function that writes the canonical DataWeave model into binary content.</li> </ul> <p><i>Introduced in DataWeave version 2.2.0.</i></p>
EmptySettings	<code>type EmptySettings = Object</code>	<p>Represents a configuration with no settings.</p> <p><i>Introduced in DataWeave version 2.2.0.</i></p>
EncodingSettings	<pre>type EncodingSettings = {     encoding?: String     {defaultValue: "UTF-8"} }</pre>	<p>Represents encoding settings and contains the following field:</p> <ul style="list-style-type: none"> <li>• <code>encoding</code>: Encoding that the writer uses for output. Defaults to "UTF-8".</li> </ul> <p><i>Introduced in DataWeave version 2.2.0.</i></p>
MimeType	<code>type MimeType = String</code>	<p>Represents a MIME type, such as <code>application/json</code>.</p> <p><i>Introduced in DataWeave version 2.2.0.</i></p>
Settings	<code>type Settings = Object</code>	<p>Reader or writer configuration settings.</p> <p><i>Introduced in DataWeave version 2.2.0.</i></p>

## DataFormat Annotations

Annotation	Definition	Description
DataFormatExtension	<code>@DataFormatExtension()</code>	Registration hook that the DataWeave engine uses to discover the variable that represents the custom data format. For an example, see the <a href="#">Custom Data Formats Example README</a> .

## dw::module::Multipart

This helper module provide functions for creating MultiPart and formats and parts (including fields and boundaries) of MultiPart formats.

To use this module, you must import it into your DataWeave code, for example, by adding the line `import dw::module::Multipart` to the header of your DataWeave script.

### Functions

Name	Description
<code>field</code>	Creates a <code>MultipartPart</code> data structure using the specified part name, input content for the part, format (or mime type), and optionally, file name.
<code>file</code>	Creates a <code>MultipartPart</code> data structure from a resource file.
<code>form</code>	Creates a <code>Multipart</code> data structure using a specified array of parts.
<code>generateBoundary</code>	Helper function for generating boundaries in <code>Multipart</code> data structures.

### Types

- [Multipart Types](#)

#### field

`field(opts: { | name: String, value: Any, mime?: String, fileName?: String | }): MultipartPart`

Creates a `MultipartPart` data structure using the specified part name, input content for the part, format (or mime type), and optionally, file name.

This version of the `field` function accepts arguments as an array of objects that use the parameter names as keys, for example: `Multipart::field({name:"order",value: myOrder, mime: "application/json", fileName: "order.json"})`

#### Parameters

Name	Description
opts	<p>Array of objects that specifies:</p> <ul style="list-style-type: none"> <li>• A unique <code>name</code> (required) for the <code>Content-Disposition</code> header of the part.</li> <li>• A <code>value</code> (required) for the content of the part.</li> <li>• <code>mime</code> (optional for strings) for the mime type (for example, <code>application/json</code>) to apply to content within the part. This setting can be used to transform the input type, for example, from JSON to XML.</li> <li>• An optional <code>fileName</code> value that you can supply to the <code>filename</code> parameter in the part's <code>Content-Disposition</code> header.</li> </ul>

## Example

This example creates a `Multipart` data structure that contains parts.

## Source

```
%dw 2.0
import dw::module::Multipart
output multipart/form-data
var firstPart = "content for my first part"
var secondPart = "content for my second part"
---
{
  parts: {
    part1: Multipart:::field({name:"myFirstPart",value: firstPart}),
    part2: Multipart:::field("mySecondPart", secondPart)
  }
}
```

## Output

```
-----_Part_320_1528378161.1542639222352
Content-Disposition: form-data; name="myFirstPart"
content for my first part
-----_Part_320_1528378161.1542639222352
Content-Disposition: form-data; name="mySecondPart"
content for my second part
-----_Part_320_1528378161.1542639222352--
```

## Example

This example produces two parts. The first part (named `order`) outputs content in JSON and provides a file name for the part (`order.json`). The second (named `clients`) outputs content in XML and does not provide a file name. Also notice that in this example you need to add the function's

namespace to the function name, for example, `Multipart::field`.

## Source

```
%dw 2.0
import dw::module::Multipart
output multipart/form-data
var myOrder = [
    {
        order: 1,
        amount: 2
    },
    {
        order: 32,
        amount: 1
    }
]
var myClients = {
    clients: {
        client: {
            id: 1,
            name: "Mariano"
        },
        client: {
            id: 2,
            name: "Shoki"
        }
    }
}
---
{
    parts: {
        order: Multipart::field({name:"order",value: myOrder, mime: "application/json",
fileName: "order.json"}),
        clients: Multipart::field({name:"clients", value: myClients, mime:
"application/xml"})
    }
}
```

## Output

```

-----=_Part_8032_681891620.1542560124825
Content-Type: application/json
Content-Disposition: form-data; name="order"; filename="order.json"

[
  {
    "order": 1,
    "amount": 2
  },
  {
    "order": 32,
    "amount": 1
  }
]
-----=_Part_8032_681891620.1542560124825
Content-Type: application/xml
Content-Disposition: form-data; name="clients"

<clients>
  <client>
    <id>1</id>
    <name>Mariano</name>
  </client>
  <client>
    <id>2</id>
    <name>Shoki</name>
  </client>
</clients>
-----=_Part_8032_681891620.1542560124825--

```

## **field(name: String, value: Any, mime: String = "", fileName: String = ""): MultipartPart**

Creates a **MultipartPart** data structure using the specified part name, input content for the part, format (or mime type), and optionally, file name.

This version of the **field** function accepts arguments in a comma-separated list, for example:

```
Multipart::field("order", myOrder, "application/json", "order.json")`
```

### **Parameters**

Name	Description
<code>opts</code>	A set of parameters that specify: <ul style="list-style-type: none"> <li>• A unique <code>name</code> (required) for the <code>Content-Disposition</code> header of the part.</li> <li>• A <code>value</code> (required) for the content of the part.</li> <li>• <code>mime</code> (optional for strings) for the mime type (for example, <code>application/json</code>) to apply to content within the part. This type can be used to transform the input type.</li> <li>• An optional <code>fileName</code> value that you can supply to the <code>filename</code> parameter in the part's <code>Content-Disposition</code> header.</li> </ul>

### Example

This example produces two parts. The first part (named `order`) outputs content in JSON and provides a file name for the part (`order.json`). The second (named `clients`) outputs content in XML and does not provide a file name. The only difference between this `field` example and the previous `field` example is the way you pass in arguments to the method. Also notice that in this example you need to add the function's namespace to the function name, for example, `Multipart::field`.

### Source

```
%dw 2.0
import dw::module::Multipart
output multipart/form-data
var myOrder = [
    {
        order: 1,
        amount: 2
    },
    {
        order: 32,
        amount: 1
    }
]
var myClients = {
    clients: {
        client: {
            id: 1,
            name: "Mariano"
        },
        client: {
            id: 2,
            name: "Shoki"
        }
    }
}
---
{
    parts: {
        order: Multipart::field("order", myOrder, "application/json", "order.json"),
        clients: Multipart::field("clients", myClients, "application/xml")
    }
}
```

## Output

```

-----=_Part_4846_2022598837.1542560230901
Content-Type: application/json
Content-Disposition: form-data; name="order"; filename="order.json"

[
  {
    "order": 1,
    "amount": 2
  },
  {
    "order": 32,
    "amount": 1
  }
]
-----=_Part_4846_2022598837.1542560230901
Content-Type: application/xml
Content-Disposition: form-data; name="clients"

<clients>
  <client>
    <id>1</id>
    <name>Mariano</name>
  </client>
  <client>
    <id>2</id>
    <name>Shoki</name>
  </client>
</clients>
-----=_Part_4846_2022598837.1542560230901--

```

## file

**file(opts: {} | name: String, path: String, mime?: String, fileName?: String | {})**

Creates a **MultipartPart** data structure from a resource file.

This version of the **file** function accepts as argument an object containing key/value pairs, enabling you to enter the key/value pairs in any order, for example:

```

Multipart::file({ name: "myFile", path: "myClients.json", mime: "application/json",
  fileName: "partMyClients.json"})

```

## Parameters

Name	Description
opts	An object that specifies the following key/value pairs: <ul style="list-style-type: none"> <li>• A unique <code>name</code> (required) for the <code>Content-Disposition</code> header of the part.</li> <li>• A <code>path</code> (required) relative to the <code>src/main/resources</code> project path for the Mule app.</li> <li>• <code>mime</code> (optional for strings) for the mime type (for example, <code>application/json</code>) to apply to content within the part. This setting <i>cannot</i> be used to transform the input mime type.</li> <li>• An optional <code>fileName</code> value for the <code>filename</code> parameter in the part's <code>Content-Disposition</code> header. Defaults to the string "<code>filename</code>" if not supplied. This value does not need to match the input file name.</li> </ul>

## Example

This example creates a `MultipartPart` from a file accessible to the DataWeave function, the file name is `orders.xml` and is located in the Mule application's `/src/main/resources` folder.

The `file` function locates the external `orders.xml` file and uses key/value pairs to indicate the various parameters needed to build the `MultipartPart`.

- The `name` can be anything, but it usually coincides with the required parameter needed by the receiving server that accepts this `Multipart` payload.
- The `path` is set to `./orders.xml`, which is the path and name for the `orders.xml` file that is loaded into the `MultipartPart`.
- The `mime` parameter specifies the correct MIME type for the file. In this case, it is `application/xml`.
- The `filename` can be changed to any value, it can be different from the actual input file's filename.

Note that the output of this example is not compatible with the `multipart/form-data` output type because it is just one part of a `Multipart` structure. To create a valid `multipart/form-data` output, use the `Multipart::form()` function with one or more `Multipart` files and/or fields.

## Source

```
%dw 2.0
import dw::module::Multipart
output application/dw
var ordersFilePath = "./orders.xml"
---
Multipart::file{ name: "file", path: ordersFilePath, mime: "application/xml",
fileName: "orders.xml" }
```

## Input

A file called `orders.xml` located in `src/main/resources` with the following content:

```
<orders>
  <order>
    <item>
      <id>1001</id>
      <qty>1</qty>
      <price>\$100</price>
    </item>
    <item>
      <id>2001</id>
      <qty>2</qty>
      <price>\$50</price>
    </item>
  </order>
</orders>
```

## Output

```
{
headers: {
  "Content-Type": "application/xml",
  "Content-Disposition": {
    name: "file",
    filename: "orders.xml"
  }
},
content: "<?xml version='1.0' encoding='UTF-8'?>\n<orders>\n  <order>\n    <item>\n      <id>1001</id>\n      <qty>1</qty>\n      <price>\$100</price>\n    </item>\n    <item>\n      <id>2001</id>\n      <qty>2</qty>\n      <price>\$50</price>\n    </item>\n  </order>\n</orders>"
}
```

## Example

This example inserts file content from a `MultipartPart` into a `Multipart`, resulting in a `multipart/form-data` output. The example uses the `form` function to create the `Multipart` and uses `file` to create a part.

The `Multipart::form()` function accepts an array of `Multipart` items, where each part can be created using the `Multipart::field()` or `Multipart::file()` functions.

## Source

```
%dw 2.0
import dw::module::Multipart
output multipart/form-data
var ordersFilePath = "./orders.xml"
var myArgs = { name: "file", path: ordersFilePath, mime: "application/xml", fileName: "myorders.xml"}
---
Multipart::form([
    Multipart::file(myArgs)
])

```

## Output

```
-----_Part_5349_1228640551.1560391284935
Content-Type: application/xml
Content-Disposition: form-data; name="file"; filename="myorders.xml"
<?xml version='1.0' encoding='UTF-8'?>
<orders>
    <order>
        <item>
            <id>1001</id>
            <qty>1</qty>
            <price>$100</price>
        </item>
        <item>
            <id>2001</id>
            <qty>2</qty>
            <price>$50</price>
        </item>
    </order>
</orders>
-----_Part_5349_1228640551.1560391284935--
```

**file(fieldName: String, path: String, mime: String = 'application/octet-stream', sentFileName: String = 'filename')**

Creates a **MultipartPart** data structure from a resource file.

This version of the **file** function accepts String arguments in a comma-separated list, for example:

```
Multipart::field("myFile", myClients, 'application/json', "partMyClients.json")
```

## Parameters

Name	Description
fieldName	A unique name (required) for the <b>Content-Disposition</b> header of the part.

Name	Description
path	A path (required) relative to the <code>src/main/resources</code> project path for the Mule app.
mime	The mime type (optional for strings), for example, <code>application/json</code> , to apply to content within the part. This setting <i>cannot</i> be used to transform the input mime type.
sentFileName	An optional file name value for the <code>filename</code> parameter in the part's <code>Content-Disposition</code> header. Defaults to the string <code>"filename"</code> if not specified. This value does not need to match the input file name.

## Example

This example inserts file content from a `MultipartPart` into a `Multipart` data structure. It uses the `form` function to create the `Multipart` type and uses `file` to create a part named `myClient` with JSON content from an external file `myClients.json`. It also specifies `partMyClients.json` as the value for to the `filename` parameter.

## Source

```
%dw 2.0
import dw::module::Multipart
var myClients = "myClients.json"
output multipart/form-data
---
Multipart::form([
  Multipart::file("myFile", myClients, 'application/json', "partMyClients.json")
])
```

## Input

A file called `myClients.json` and located in `src/main/resources` with the following content.

```
clients: {
  client: {
    id: 1,
    name: "Mariano"
  },
  client: {
    id: 2,
    name: "Shoki"
  }
}
```

## Output

```

-----=_Part_1586_1887987980.1542569342438
Content-Type: application/json
Content-Disposition: form-data; name="myFile"; filename="partMyClients.json"

{
  clients: {
    client: {
      id: 1,
      name: "Mariano"
    },
    client: {
      id: 2,
      name: "Shoki"
    }
  }
}
-----=_Part_1586_1887987980.1542569342438--

```

## form

### form(parts: Array<MultipartPart>): Multipart

Creates a **Multipart** data structure using a specified array of parts.

#### Parameters

Name	Description
parts	An array of parts ( <b>MultipartPart</b> data structures).

#### Example

This example creates a **Multipart** data structure that contains three parts.

The first part uses the **Multipart::file()** function to import an external file named **orders.xml**. The file is located in the internal **src/main/resources** folder of the Mule application. See the **file** function documentation for more details on this example.

The second part uses the **Multipart::field()** function version that accepts field names as input parameters in the form of an object with key/value pairs, enabling you to pass the keys in any order. This part also does not specify the optional **fileName** parameter. When specified, **fileName** is part of the **Content-Distribution** element of the part. The **mime** field is also optional. When included, the field sets the **Content-Type** element to the **mime** value. In this case the **Content-Type** is set to **text/plain**.

The third part uses the more compact version of the **Multipart::field()** function which sets the required and optional parameters, in the correct order, as input parameters. The first three parameters **name**, **value**, and **mime** are required. The **fileName** parameters is optional, use it only if the content is read from a file or is written to a file. In this version, the **mime** parameter is output as the

`Content-Type` element, and the `fileName` is output as the `filename` parameter of the `Content-Distribution` element.

## Source

```
%dw 2.0
import dw::module::Multipart
output multipart/form-data
var myOrders = "./orders.xml"
var fileArgs = { name: "file", path: myOrders, mime: "application/xml", fileName:
"myorders.xml"}
var fieldArgs = {name:"userName",value: "Annie Point", mime: "text/plain"}
---
Multipart::form([
    Multipart::file(fileArgs),
    Multipart::field(fieldArgs),
    Multipart::field("myJson", {"user": "Annie Point"}, "application/json",
"userInfo.json")
])
```

## Output

```

-----=_Part_146_143704079.1560394078604
Content-Type: application/xml
Content-Disposition: form-data; name="file"; filename="myorders.xml"
<?xml version='1.0' encoding='UTF-8'?>
<orders>
  <order>
    <item>
      <id>1001</id>
      <qty>1</qty>
      <price>$100</price>
    </item>
    <item>
      <id>2001</id>
      <qty>2</qty>
      <price>$50</price>
    </item>
  </order>
</orders>
-----=_Part_146_143704079.1560394078604
Content-Type: text/plain
Content-Disposition: form-data; name="userName"
Annie Point
-----=_Part_146_143704079.1560394078604
Content-Type: application/json
Content-Disposition: form-data; name="myJson"; filename="userInfo.json"
{
  "user": "Annie Point"
}
-----=_Part_146_143704079.1560394078604--

```

## Example

This example constructs a data structure using DataWeave code that is compatible with the **multipart/form-data** output format, demonstrating how you can manually construct a data structure compatible with **multipart/form-data** output, without using the **form** function.

In the following structure, the part keys **part1** and **part2** are stripped out in the **multipart/form-data** output.

## Source

```
%dw 2.0
import dw::module::Multipart
output multipart/form-data
var firstPart = "content for my first part"
var secondPart = "content for my second part"
---
{
  parts: {
    part1: Multipart::field({name:"myFirstPart",value: firstPart}),
    part2: Multipart::field("mySecondPart", secondPart)
  }
}
```

## Output

```
-----=_Part_320_1528378161.1542639222352
Content-Disposition: form-data; name="myFirstPart"

content for my first part
-----=_Part_320_1528378161.1542639222352
Content-Disposition: form-data; name="mySecondPart"

content for my second part
-----=_Part_320_1528378161.1542639222352--
```

## generateBoundary

**generateBoundary(len: Number = 70): String**

Helper function for generating boundaries in **Multipart** data structures.

## Multipart Types

Type	Definition	Description
Multipart	<code>type Multipart = { preamble?: String, parts: { _:? MultipartPart } }</code>	<b>Multipart</b> type, a data structure for a complete <b>Multipart</b> format. See the output example for the <b>Multipart</b> <b>form</b> function <a href="#">documentation</a> .
MultipartPart	<code>type MultipartPart = { headers?: { "Content-Disposition"?:{ name: String, filename?: String }, "Content-Type"?:{ String }, content: Any }</code>	<b>MultipartPart</b> type, a data structure for a part within a <b>Multipart</b> format. See the output examples for the <b>Multipart</b> <b>field</b> function <a href="#">documentation</a> .

# dw::Mule

This DataWeave module contains functions for interacting with Mule runtime.

## Functions

Name	Description
<a href="#">causedBy</a>	This function matches an error by its type, like an error handler does.
<a href="#">lookup</a>	This function enables you to execute a flow within a Mule app and retrieve the resulting payload.
<a href="#">p</a>	This function returns a string that identifies the value of one of these input properties: Mule property placeholders, System properties, or Environment variables.

## Types

- [Mule Types](#)

### causedBy

**causedBy(@DesignOnlyType error: Error, errorType: String): Boolean**

This function matches an error by its type, like an error handler does.

**causedBy** is useful when you need to match by a super type, but the specific sub-type logic is also needed. It can also be useful when handling a COMPOSITE\_ROUTING error that contains child errors of different types.

#### Parameters

Name	Description
<b>error</b>	Optional. An <b>Error</b> type.
<b>errorType</b>	A string that identifies the error, such as HTTP:UNAUTHORIZED.

#### Example

This XML example calls **causedBy** from a **when** expression in a Mule error handling component to handle a SECURITY error differently depending on whether it was caused by an HTTP:UNAUTHORIZED or HTTP:FORBIDDEN error. Notice that the first expression passes in the **error** (an **Error** type) explicitly, while the second one passes it implicitly, without specifying the value of the parameter. Note that **error** is the variable that DataWeave uses for errors associated with a Mule message object (see [DataWeave Variables for Mule Runtime](#)).

#### Source

```

<error-handler name="securityHandler">
  <on-error-continue type="SECURITY">
    <!-- general error handling for all SECURITY errors -->
    <choice>
      <when expression="#[Mule::causedBy(error, 'HTTP:UNAUTHORIZED')]">
        <!-- specific error handling only for HTTP:UNAUTHORIZED errors -->
      </when>
      <when expression="#[Mule::causedBy('HTTP:FORBIDDEN')]">
        <!-- specific error handling only for HTTP:FORBIDDEN errors -->
      </when>
    </choice>
  </on-error-continue>
</error-handler>

```

## lookup

**lookup(flowName: String, payload: Any, timeoutMillis: Number = 2000)**

This function enables you to execute a flow within a Mule app and retrieve the resulting payload.

It works in Mule apps that are running on Mule Runtime version 4.1.4 and later.

Similar to the Flow Reference component (recommended), the `lookup` function enables you to execute another flow within your app and to retrieve the resulting payload. It takes the flow's name and an input payload as parameters. For example, `lookup("anotherFlow", payload)` executes a flow named `anotherFlow`.

The function executes the specified flow using the current attributes, variables, and any error, but it only passes in the payload without any attributes or variables. Similarly, the called flow will only return its payload.

Note that the `lookup` function *does not* support calling subflows.



Always keep in mind that a functional language like DataWeave expects the invocation of the `lookup` function to *not* have side effects. So, the internal workings of the DataWeave engine might cause a `lookup` function to be invoked in parallel with other `lookup` functions, or not to be invoked at all.

MuleSoft recommends that you invoke flows with the Flow Ref (`flow-ref`) component, using the `target` attribute to put the result of the flow in a `var` and then referencing that `var` from within the DataWeave script.

*This function is **Deprecated**.*

### Parameters

Name	Description
<code>flowName</code>	A string that identifies the target flow.

Name	Description
<code>payload</code>	The payload to send to the target flow, which can be any ( <a href="#">Any</a> ) type.
<code>timeoutMillis</code>	Optional. Timeout (in milliseconds) for the execution of the target flow. Defaults to <code>2000</code> milliseconds (2 seconds) if the thread that is executing is <code>CPU_LIGHT</code> or <code>CPU_INTENSIVE</code> , or 1 minute when executing from other threads. If the lookup takes more time than the specified <code>timeoutMillis</code> value, an error is raised.

## Example

This example shows XML for two flows. The `lookup` function in `flow1` executes `flow2` and passes the object `{test:'hello '}` as its payload to `flow2`. The Set Payload component (`<set-payload/>`) in `flow2` then concatenates the value of `{test:'hello '}` with the string `world` to output and log `hello world`.

## Source

```

<flow name="flow1">
    <http:listener doc:name="Listener" config-ref="HTTP_Listener_config"
        path="/source"/>
    <ee:transform doc:name="Transform Message" >
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
output application/json
---
Mule::lookup('flow2', {test:'hello '})]]></ee:set-payload>
        </ee:message>
    </ee:transform>
</flow>
<flow name="flow2" >
    <set-payload value='#[payload.test ++ "world"]' doc:name="Set Payload" />
    <logger level="INFO" doc:name="Logger" message='#[payload]'/>
</flow>

```

## p

### `p(propertyName: String): String`

This function returns a string that identifies the value of one of these input properties: Mule property placeholders, System properties, or Environment variables.

For more on this topic, see [Configure Properties](#).

## Parameters

Name	Description
<code>propertyName</code>	A string that identifies property.

## Example

This example logs the value of the property `http.port` in a Logger component.

## Source

```
<flow name="simple">
  <logger level="INFO" doc:name="Logger"
    message="#[Mule:::p('http.port')]" />
</flow>
```

## Mule Types

Type	Definition	Description
Error	<pre>type Error = { description?: String, detailedDescription?: String, errorType?: ErrorType, childErrors?: Array&lt;Error&gt; }</pre>	<p>A complex type that represents an Error:</p> <ul style="list-style-type: none"><li>• <code>description?: String</code>: Concise description of the error.</li><li>• <code>detailedDescription?: String</code>: Detailed description of the error. This message can include information specific to a Java exception.</li><li>• <code>errorType?: ErrorType</code>: Returns the type of the error.</li><li>• <code>childErrors?: Array&lt;Error&gt;</code>: Lists child Errors, if any. For example, the Scatter-Gather router might throw an error aggregating all of its routing errors as children.</li></ul> <p>Not all failing components aggregate errors, so this type can return an empty collection.</p>

Type	Definition	Description
ErrorType	<pre>type ErrorType = {     identifier?: String,     namespace?: String,     parentErrorType?: ErrorType }</pre>	<p>A type of error that a Mule component can throw.</p> <ul style="list-style-type: none"> <li>The error type has a <code>identifier</code> string that end users can provide in the Mule configuration.</li> <li>Every error belongs to a <code>namespace</code> to avoid collisions with errors that have the same <code>identifier</code> string but belong to different namespace.</li> <li>Error types can be a specialization of a more general error type, in which case the <code>parentErrorType</code> should return the more general error type. This type is used during error type matching within error handlers. So when selecting the general error type for error handling, it also handles the more specialized error types.</li> </ul>
Message	<pre>type Message = Object {class: "org.mule.runtime.api.message.Message"}</pre>	Type that represents a Mule message.

## dw::Runtime

This module contains functions for interacting with the DataWeave runtime, which executes the language.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::Runtime` to the header of your DataWeave script.

## Functions

Name	Description
<code>dataFormatsDescriptor</code>	<p>Returns an array of all <code>DataFormatDescriptor</code> values that are installed in the current instance of DataWeave.</p> <p><i>Experimental:</i> This function is an experimental feature that is subject to change or removal from future versions.</p>
<code>eval</code>	<p>Evaluates a script with the specified context and returns the result of that evaluation.</p> <p><i>Experimental:</i> This function is an experimental feature that is subject to change or removal from future versions.</p>

Name	Description
evalUrl	Evaluates the script at the specified URL.  <i>Experimental:</i> This function is an experimental feature that is subject to change or removal from future versions.
fail	Throws an exception with the specified message.
failIf	Produces an error with the specified message if the expression in the evaluator returns <code>true</code> . Otherwise, the function returns the value.
location	Returns the location of a given value, or <code>null</code> if the location can't be traced back to a DataWeave file.
locationString	Returns the location string of a given value.
orElse	Returns the result of the <code>orElse</code> argument if the <code>previous</code> argument to <code>try</code> fails. Otherwise, the function returns the value of the <code>previous</code> argument.
orElseTry	Function to use with <code>try</code> to chain multiple <code>try</code> requests.
prop	Returns the value of the property with the specified name or <code>null</code> if the property is not defined.
props	Returns all the properties configured for the DataWeave runtime, which executes the language.
run	Runs the input script under the provided context and executes the script in the current runtime.  <i>Experimental:</i> This function is an experimental feature that is subject to change or removal from future versions.
runUrl	Runs the script at the specified URL.  <i>Experimental:</i> This function is an experimental feature that is subject to change or removal from future versions.
try	Evaluates the delegate function and returns an object with <code>success: true</code> and <code>result</code> if the delegate function succeeds, or an object with <code>success: false</code> and <code>error</code> if the delegate function throws an exception.
version	Returns the DataWeave version that is currently running.
wait	Stops the execution for the specified timeout period (in milliseconds).

## Types

- [Runtime Types](#)

## dataFormatsDescriptor

### dataFormatsDescriptor(): Array<DataFormatDescriptor>

Returns an array of all `DataFormatDescriptor` values that are installed in the current instance of

DataWeave.

*Experimental:* This function is an experimental feature that is subject to change or removal from future versions of DataWeave.

### Example

This example shows how `dataFormatsDescriptor` behaves with different inputs.

### Source

```
import * from dw::Runtime
---
dataFormatsDescriptor()
```

### Output

```
[{"id": "json", "binary": false, "defaultEncoding": "UTF-8", "extensions": [".json"], "defaultMimeType": "application/json", "acceptedMimeTypes": ["application/json"], "readerProperties": [{"name": "streaming", "optional": true, "defaultValue": false, "description": "Used for streaming input (use only if entries are accessed sequentially).", "possibleValues": [true, false]}, {"name": "writeAttributes", "optional": true, "defaultValue": false, "description": "Indicates that if a key has attributes, they are going to be added as children key-value pairs of the key that contains them. The attribute new key name will start with @."}]}
```

```

    "possibleValues": [
        true,
        false
    ],
},
{
    "name": "skipNullOn",
    "optional": true,
    "defaultValue": "None",
    "description": "Indicates where is should skips null values if any or
not. By default it doesn't skip.",
    "possibleValues": [
        "arrays",
        "objects",
        "everywhere"
    ]
}
],
{
    "id": "xml",
    "binary": false,
    "extensions": [
        ".xml"
    ],
    "defaultMimeType": "application/xml",
    "acceptedMimeTypes": [
        "application/xml"
    ],
    "readerProperties": [
        {
            "name": "supportDtd",
            "optional": true,
            "defaultValue": true,
            "description": "Whether DTD handling is enabled or disabled; disabling
means both internal and external subsets will just be skipped unprocessed."
        },
        "possibleValues": [
            true,
            false
        ]
    ],
{
    "name": "streaming",
    "optional": true,
    "defaultValue": false,
    "description": "Used for streaming input (use only if entries are
accessed sequentially).",
    "possibleValues": [
        true,
        false
    ]
}
]

```

```

        },
        {
            "name": "maxEntityCount",
            "optional": true,
            "defaultValue": 1,
            "description": "The maximum number of entity expansions. The limit is in place to avoid Billion Laughs attacks.",
            "possibleValues": [
                ]
        }
    ],
    "writerProperties": [
        {
            "name": "writeDeclaration",
            "optional": true,
            "defaultValue": true,
            "description": "Indicates whether to write the XML header declaration or not.",
            "possibleValues": [
                true,
                false
            ]
        },
        {
            "name": "indent",
            "optional": true,
            "defaultValue": true,
            "description": "Indicates whether to indent the code for better readability or to compress it into a single line.",
            "possibleValues": [
                true,
                false
            ]
        }
    ]
}
]

```

## eval

**eval(fileToExecute: String, fs: Dictionary<String>, readerInputs: Dictionary<ReaderInput> = {}, inputValues: Dictionary<Any> = {}, configuration: RuntimeExecutionConfiguration = {}): EvalSuccess | ExecutionFailure**

Evaluates a script with the specified context and returns the result of that evaluation.

*Experimental:* This function is an experimental feature that is subject to change or removal from future versions of DataWeave.

## Parameters

Name	Description
fileToExecute	Name of the file to execute.
fs	An object that contains the file to evaluate.
readerInputs	Reader inputs to bind.
inputValues	Additional literal values to bind
configuration	The runtime configuration.

## Example

This example shows how `eval` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::Runtime

var jsonValue = {
    value: '{"name": "Mariano"}' as Binary {encoding: "UTF-8"},
    encoding: "UTF-8",
    properties: {},
    mimeType: "application/json"
}

var jsonValue2 = {
    value: '{"name": "Mariano", "lastName": "achaval"}' as Binary {encoding: "UTF-8"},
    encoding: "UTF-8",
    properties: {},
    mimeType: "application/json"
}

var invalidJsonValue = {
    value: '{"name": "Mariano' as Binary {encoding: "UTF-8"},
    encoding: "UTF-8",
    properties: {},
    mimeType: "application/json"
}

var Utils = "fun sum(a,b) = a +b"
output application/json
---
{
    "execute_ok" : run("main.dwl", {"main.dwl": "{a: 1}"}, {"payload": jsonValue}),
    "logs" : do {
        var execResult = run("main.dwl", {"main.dwl": "{a: log(1)}"}, {"payload": jsonValue})
        ---
    }
}
```

```

{
    m: execResult.logs.message,
    l: execResult.logs.level
}
},
"grant" : eval("main.dwl", {"main.dwl": "{a: readUrl('http://google.com')}"}, {"payload": jsonValue }, {},{ securityManager: (grant, args) -> false }),
"library" : eval("main.dwl", {"main.dwl": "Utils::sum(1,2)", "/Utils.dwl": Utils }, {"payload": jsonValue }),
"timeout" : eval("main.dwl", {"main.dwl": "(1 to 1000000000000) map \$ + 1" }, {"payload": jsonValue }, {},{timeOut: 2}).success,
"execFail" : eval("main.dwl", {"main.dwl": "dw::Runtime::fail('My Bad')"}, {"payload": jsonValue }),
"parseFail" : eval("main.dwl", {"main.dwl": "(1 + " }, {"payload": jsonValue }),
"writerFail" : eval("main.dwl", {"main.dwl": "output application/xml --- 2" }, {"payload": jsonValue }),
"defaultOutput" : eval("main.dwl", {"main.dwl": "payload" }, {"payload": jsonValue2}, {},{outputMimeType: "application/csv", writerProperties: {"separator": "|"}}),
"onExceptionFail": do {
    dw::Runtime::try( () ->
        eval("main.dwl", {"main.dwl": "dw::Runtime::fail('Failing Test')"}, {"payload": jsonValue2}, {},{onException: "FAIL"})
    ).success
},
"customLogger":
    eval(
"main.dwl",
    {"main.dwl": "log(1234)" },
    {"payload": jsonValue2},
    {},
{
    loggerService: {
        initialize: () -> {token: "123"},
        log: (level, msg, context) -> log("${level} ${msg}", context)
    }
}
)
}

```

## Output

```
{
"execute_ok": {
    "success": true,
    "value": "{\n    a: 1\n}",
    "mimeType": "application/dw",
    "encoding": "UTF-8",
    "logs": [

```

```

        ],
    },
    "logs": {
        "m": [
            "1"
        ],
        "l": [
            "INFO"
        ]
    },
    "grant": {
        "success": false,
        "message": "The given required permissions: 'Resource' are not being granted for this execution.\nTrace:\n  at readUrl (Unknown)\n  at main::main (line: 1, column: 5)",
        "location": {
            "start": {
                "index": 0,
                "line": 0,
                "column": 0
            },
            "end": {
                "index": 0,
                "line": 0,
                "column": 0
            },
            "content": "Unknown location"
        },
        "stack": [
            "readUrl (anonymous:0:0)",
            "main (main:1:5)"
        ],
        "logs": [
        ]
    },
    "library": {
        "success": true,
        "value": 3,
        "logs": [
        ]
    },
    "timeout": true,
    "execFail": {
        "success": false,
        "message": "My Bad\nTrace:\n  at fail (Unknown)\n  at main::main (line: 1, column: 1)",
        "location": {
            "start": {
                "index": 0,

```

```

        "line": 0,
        "column": 0
    },
    "end": {
        "index": 0,
        "line": 0,
        "column": 0
    },
    "content": "Unknown location"
},
"stack": [
    "fail (anonymous:0:0)",
    "main (main:1:1)"
],
"logs": [
]
},
"parseFail": {
    "success": false,
    "message": "Invalid input \"1 + \\", expected parameter or parenEnd (line 1, column 2):\n\n1| (1 + \n      ^^^^\\nLocation:\\nmain (line: 1, column:2)",
    "location": {
        "start": {
            "index": 0,
            "line": 1,
            "column": 2
        },
        "end": {
            "index": 4,
            "line": 1,
            "column": 6
        },
        "content": "\n1| (1 + \n      ^^^^"
    },
    "logs": [
]
},
"writerFail": {
    "success": true,
    "value": 2,
    "logs": [
]
},
"defaultOutput": {
    "success": true,
    "value": {
        "name": "Mariano",
        "lastName": "achaval"
    }
}

```

```

},
"logs": [
],
},
"onExceptionFail": false,
"customLogger": {
  "success": true,
  "value": 1234,
  "logs": [
]
}
}

```

## evalUrl

**evalUrl(url: String, readerInputs: Dictionary<ReaderInput> = {}, inputValues: Dictionary<Any> = {}, configuration: RuntimeExecutionConfiguration = {}): EvalSuccess | ExecutionFailure**

Evaluates the script at the specified URL.

*Experimental:* This function is an experimental feature that is subject to change or removal from future versions of DataWeave.

### Parameters

Name	Description
<code>url</code>	Name of the file execute.
<code>readerInputs</code>	Inputs to read and bind to the execution.
<code>inputValues</code>	Inputs to bind directly to the execution.
<code>configuration</code>	The runtime configuration.

### Example

This example shows how `evalUrl` behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::Runtime
var jsonValue = {
    value: '{"name": "Mariano"}' as Binary {encoding: "UTF-8"},
    encoding: "UTF-8",
    properties: {},
    mimeType: "application/json"
}

var Utils = "fun sum(a,b) = a +b"
output application/json
---
{
    "execute_ok" :
evalUrl("classpath://org/mule/weave/v2/engine/runtime_evalUrl/example.dwl",
{"payload": jsonValue }),
    "execute_okWithValue" :
evalUrl("classpath://org/mule/weave/v2/engine/runtime_evalUrl/example.dwl", {}),
    {"payload": {"name": "Mariano"}}
}
```

## Output

```
{
    "execute_ok": {
        "success": true,
        "value": "Mariano",
        "logs": [
            ]
    },
    "execute_okWithValue": {
        "success": true,
        "value": "Mariano",
        "logs": [
            ]
    }
}
```

## fail

**fail(message: String = 'Error'): Nothing**

Throws an exception with the specified message.

### Parameters

Name	Description
message	An error message ( <code>String</code> ).

## Example

This example returns a failure message `Data was empty` because the expression `(sizeOf(myVar) <= 0)` is `true`. A shortened version of the error message is shown in the output.

## Source

```
%dw 2.0
import * from dw::Runtime
var result = []
output application/json
---
if(sizeOf(result) <= 0) fail('Data was empty') else result
```

## Output

```
ERROR 2018-07-29 11:47:44,983 ...
*****
Message : "Data was empty"
...
```

## failIf

**failIf<T>(value: T, evaluator: (value: T) -> Boolean, message: String = 'Failed'): T**

Produces an error with the specified message if the expression in the evaluator returns `true`. Otherwise, the function returns the value.

## Parameters

Name	Description
value	The value to return only if the <code>evaluator</code> expression is <code>false</code> .
evaluator	Expression that returns <code>true</code> or <code>false</code> .

## Example

This example produces a runtime error (instead of a SUCCESS message) because the expression `isEmpty(result)` is `true`. It is `true` because an empty object is passed through variable `result`.

## Source

```
%dw 2.0
import failIf from dw::Runtime
var result = {}
output application/json
---
{ "result" : "SUCCESS" failIf (isEmpty(result)) }
```

## Output

```
ERROR 2018-07-29 11:56:39,988 ...
*****
Message : "Failed"
```

## location

### location(value: Any): Location

Returns the location of a given value, or `null` if the location can't be traced back to a DataWeave file.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<code>value</code>	A value of any type.

#### Example

This example returns the location that defines the function `sqrt` in the `dw::Core` module.

#### Source

```
%dw 2.0
import location from dw::Runtime
output application/json
---
location(sqrt)
```

## Output

```
{  
  "uri": "/dw/Core.dwl",  
  "nameIdentifier": "dw::Core",  
  "startLine": 5797,  
  "startColumn": 36,  
  "endLine": 5797,  
  "endColumn": 77  
}
```

## locationString

### locationString(value: Any): String

Returns the location string of a given value.

#### Parameters

Name	Description
value	A value of any type.

#### Example

This example returns the contents of the line (the location) that defines variable `a` in the header of the DataWeave script.

#### Source

```
%dw 2.0  
import * from dw::Runtime  
var a = 123  
output application/json  
---  
locationString(a)
```

#### Output

```
"var a = 123"
```

## orElse

### orElse<T, R>(previous: TryResult<T>, orElse: () -> R): T | R

Returns the result of the `orElse` argument if the `previous` argument to `try` fails. Otherwise, the function returns the value of the `previous` argument.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
previous	Result from a previous call to <code>try</code> .
orElse	Argument to return if the <code>previous</code> argument fails.

## Example

This example waits shows how to chain different try

## Source

```
%dw 2.0
import * from dw::Runtime
var user = {}
var otherUser = {name: "DW"}
output application/json
---
{
    a: try(() -> user.name!) orElse "No User Name",
    b: try(() -> otherUser.name) orElse "No User Name"
}
```

## Output

```
{
    "a": "No User Name",
    "b": "DW"
}
```

## orElseTry

`orElseTry<T, R>(previous: TryResult<T>, orElse: () -> R): TryResult<T | R>`

Function to use with `try` to chain multiple `try` requests.

*Introduced in DataWeave version 2.2.0.*

## Parameters

Name	Description
previous	Result from a previous call to <code>try</code> .
orElseTry	Argument to try if the <code>previous</code> argument fails.

## Example

This example waits shows how to chain different try

## Source

```
%dw 2.0
import * from dw::Runtime
var user = {}
var otherUser = {}
output application/json
---
{
    a: try(() -> user.name!) orElseTry otherUser.name!,
    b: try(() -> user.name!) orElseTry "No User Name"
}
```

## Output

```
{
  "a": {
    "success": false,
    "error": {
      "kind": "KeyNotFoundException",
      "message": "There is no key named 'name'",
      "location": "\n|      a: try(() -> user.name!) orElseTry otherUser.name!,\n^^^^^^^^^^^^^",
      "stack": [
        "main (org::mule::weave::v2::engine::transform:9:40)"
      ]
    }
  },
  "b": {
    "success": true,
    "result": "No User Name"
  }
}
```

## prop

**prop(propertyName: String): String | Null**

Returns the value of the property with the specified name or **null** if the property is not defined.

### Parameters

Name	Description
propertyName	The property to retrieve.

### Example

This example gets the **user.timezone** property.

## Source

```
%dw 2.0
import * from dw::Runtime
output application/dw
---
{ "props" : prop("user.timezone") }
```

## Output

```
{ props: "America/Los_Angeles" as String {class: "java.lang.String"} }
```

## props

### props(): Dictionary<String>

Returns all the properties configured for the DataWeave runtime, which executes the language.

#### Example

This example returns all properties from the [java.util.Properties](#) class.

## Source

```
%dw 2.0
import * from dw::Runtime
output application/dw
---
{ "props" : props() }
```

## Output

```
{
  props: {
    "java.vendor": "Oracle Corporation" as String {class: "java.lang.String"},  

    "sun.java.launcher": "SUN_STANDARD" as String {class: "java.lang.String"},  

    "sun.management.compiler": "HotSpot 64-Bit Tiered Compilers" as String ..., *  

    "os.name": "Mac OS X" as String {class: "java.lang.String"},  

    "sun.boot.class.path": "/Library/Java/JavaVirtualMachines/ ....,  

    "org.glassfish.grizzly.nio.transport.TCPNIOTransport...": "1048576" ....,  

    "java.vm.specification.vendor": "Oracle Corporation" as String ...,  

    "java.runtime.version": "1.8.0_111-b14" as String {class: "java.lang.String"},  

    "wrapper.native_library": "wrapper" as String {class: "java.lang.String"},  

    "wrapper.key": "XlI14YartmfEU3oKu7o81kNQbwhveXi-" as String ...,  

    "user.name": "me" as String {class: "java.lang.String"},  

    "mvel2.disable.jit": "TRUE" as String {class: "java.lang.String"},  

    "user.language": "en" as String {class: "java.lang.String"} ...,
```

```
"sun.boot.library.path": "/Library/Java/JavaVirtualMachines ...",
"xpath.provider": "com.mulesoft.licm.DefaultXPathProvider" ....,
"wrapper.backend": "pipe" as String {class: "java.lang.String"}, 
"java.version": "1.8.0_111" as String {class: "java.lang.String"}, 
"user.timezone": "America/Los_Angeles" as String {class: "java.lang.String"}, 
"java.net.preferIPv4Stack": "TRUE" as String {class: "java.lang.String"}, 
"sun.arch.data.model": "64" as String {class: "java.lang.String"}, 
"java.endorsed.dirs": "/Library/Java/JavaVirtualMachines/..., 
"sun.cpu.isalist": "" as String {class: "java.lang.String"}, 
"sun.jnu.encoding": "UTF-8" as String {class: "java.lang.String"}, 
"mule.testingMode": "" as String {class: "java.lang.String"}, 
"file.encoding.pkg": "sun.io" as String {class: "java.lang.String"}, 
"file.separator": "/" as String {class: "java.lang.String"}, 
"java.specification.name": "Java Platform API Specification" ...., 
"java.class.version": "52.0" as String {class: "java.lang.String"}, 
"jetty.git.hash": "82b8fb23f757335bb3329d540ce37a2a2615f0a8" ...., 
"user.country": "US" as String {class: "java.lang.String"}, 
"mule.agent.configuration.folder": "/Applications/AnypointStudio.app/ ..., 
"log4j.configurationFactory": "org.apache.logging.log4j.core...", 
"java.home": "/Library/Java/JavaVirtualMachines/..., 
"java.vm.info": "mixed mode" as String {class: "java.lang.String"}, 
"wrapper.version": "3.5.34-st" as String {class: "java.lang.String"}, 
"os.version": "10.13.4" as String {class: "java.lang.String"}, 
"org.eclipse.jetty.LEVEL": "WARN" as String {class: "java.lang.String"}, 
"path.separator": ":" as String {class: "java.lang.String"}, 
"java.vm.version": "25.111-b14" as String {class: "java.lang.String"}, 
"wrapper.pid": "5212" as String {class: "java.lang.String"}, 
"java.util.prefs.PreferencesFactory": "com.mulesoft.licm..."}, 
"wrapper.java.pid": "5213" as String {class: "java.lang.String"}, 
"mule.home": "/Applications/AnypointStudio.app/..., 
"java.awt.printerjob": "sun.lwawt.macosx.CPrinterJob" ...., 
"sun.io.unicode.encoding": "UnicodeBig" as String {class: "java.lang.String"}, 
"awt.toolkit": "sun.lwawt.macosx.LWCToolkit" ...., 
"org.glassfish.grizzly.nio.transport...": "1048576" ...., 
"user.home": "/Users/me" as String {class: "java.lang.String"}, 
"java.specification.vendor": "Oracle Corporation" ...., 
"java.library.path": "/Applications/AnypointStudio.app/..., 
"java.vendor.url": "http://java.oracle.com/" as String ...., 
"java.vm.vendor": "Oracle Corporation" as String {class: "java.lang.String"}, 
gopherProxySet: "false" as String {class: "java.lang.String"}, 
"wrapper.jvmid": "1" as String {class: "java.lang.String"}, 
"java.runtime.name": "Java(TM) SE Runtime Environment" ...., 
"mule.encoding": "UTF-8" as String {class: "java.lang.String"}, 
"sun.java.command": "org.mule.runtime.module.reboot....", 
"java.class.path": "%MULE_LIB%:/Applications/AnypointStudio.app...", 
"log4j2.loggerContextFactory": "org.mule.runtime.module.launcher...", 
"java.vm.specification.name": "Java Virtual Machine Specification" , 
"java.vm.specification.version": "1.8" as String {class: "java.lang.String"}, 
"sun.cpu.endian": "little" as String {class: "java.lang.String"}, 
"sun.os.patch.level": "unknown" as String {class: "java.lang.String"}, 
"com.ning.http.client.AsyncHttpClientConfig.useProxyProperties": "true" ...,
```

```

"wrapper.cpu.timeout": "10" as String {class: "java.lang.String"},  

"java.io.tmpdir": "/var/folders/42/dd7313rx7qz0n625hr29kty80000gn/T/" ...,  

"anypoint.platform.analytics_base_uri": ...,  

"java.vendor.url.bug": "http://bugreport.sun.com/bugreport/" ...,  

"os.arch": "x86_64" as String {class: "java.lang.String"},  

"java.awt.graphicsenv": "sun.awt.CGraphicsEnvironment" ...,  

"mule.base": "/Applications/AnypointStudio.app...",  

"java.ext.dirs": "/Users/staceyduke/Library/Java/Extensions: ..." ,  

"user.dir": "/Applications/AnypointStudio.app/..."},  

"line.separator": "\n" as String {class: "java.lang.String"},  

"java.vm.name": "Java HotSpot(TM) 64-Bit Server VM" ...,  

"org.quartz.scheduler.skipUpdateCheck": "true" ...,  

"file.encoding": "UTF-8" as String {class: "java.lang.String"},  

"mule.forceConsoleLog": "" as String {class: "java.lang.String"},  

"java.specification.version": "1.8" as String {class: "java.lang.String"},  

"wrapper.arch": "universal" as String {class: "java.lang.String"}  

} as Object {class: "java.util.Properties"}

```

## run

**run(fileToExecute: String, fs: Dictionary<String>, readerInputs: Dictionary<ReaderInput> = {}, inputValues: Dictionary<Any> = {}, configuration: RuntimeExecutionConfiguration = {}): RunSuccess | ExecutionFailure**

Runs the input script under the provided context and executes the script in the current runtime.

*Experimental:* This function is an experimental feature that is subject to change or removal from future versions of DataWeave.

### Parameters

Name	Description
fileToExecute	Name of the file to execute.
fs	File system that contains the file to execute.
readerInput	Inputs to read and bind to the execution.
inputValues	Inputs to bind directly to the execution.
configuration	The runtime configuration.

### Example

This example shows how **run** behaves with different inputs.

### Source

```

import * from dw::Runtime
var jsonValue = {
  value: '{"name": "Mariano"}' as Binary {encoding: "UTF-8"},  

  encoding: "UTF-8",
}

```

```

properties: {},
mimeType: "application/json"
}

var jsonValue2 = {
  value: '{"name": "Mariano", "lastName": "achaval"}' as Binary {encoding: "UTF-8"},
  encoding: "UTF-8",
  properties: {},
  mimeType: "application/json"
}

var invalidJsonValue = {
  value: '{"name": "Mariano' as Binary {encoding: "UTF-8"}, // note the missing quote
  encoding: "UTF-8",
  properties: {},
  mimeType: "application/json"
}

var Utils = "fun sum(a,b) = a +b"
---
{
  "execute_ok" : run("main.dwl", {"main.dwl": "{a: 1}"}, {"payload": jsonValue }),
  "logs" : do {
    var execResult = run("main.dwl", {"main.dwl": "{a: log(1)}"}, {"payload": jsonValue })
    ---
    {
      m: execResult.logs.message,
      l: execResult.logs.level
    }
  },
  "grant" : run("main.dwl", {"main.dwl": "{a: readUrl('http://google.com')}"}, {"payload": jsonValue }, { securityManager: (grant, args) -> false }),
  "library" : run("main.dwl", {"main.dwl": "Utils::sum(1,2)", "/Utils.dwl": Utils }, {"payload": jsonValue }),
  "timeout" : run("main.dwl", {"main.dwl": "(1 to 1000000000000) map \$ + 1" }, {"payload": jsonValue }, {timeOut: 2}).success,
  "execFail" : run("main.dwl", {"main.dwl": "dw::Runtime::fail('My Bad')" }, {"payload": jsonValue }),
  "parseFail" : run("main.dwl", {"main.dwl": "(1 + " }, {"payload": jsonValue }),
  "writerFail" : run("main.dwl", {"main.dwl": "output application/xml --- 2" }, {"payload": jsonValue }),
  "readerFail" : run("main.dwl", {"main.dwl": "output application/xml --- payload" }, {"payload": invalidJsonValue }),
  "defaultOutput" : run("main.dwl", {"main.dwl": "payload" }, {"payload": jsonValue2}, {outputMimeType: "application/csv", writerProperties: {"separator": "|"}}),
}

```

## Output

```
{
  "execute_ok": {
    "success": true,
    "value": "{\n  a: 1\n}",
    "mimeType": "application/dw",
    "encoding": "UTF-8",
    "logs": [
      ]
  },
  "logs": {
    "m": [
      "1"
    ],
    "l": [
      "INFO"
    ]
  },
  "grant": {
    "success": false,
    "message": "The given required permissions: 'Resource' are not being granted for this execution.\nTrace:\n  at readUrl (Unknown)\n  at main::main (line: 1, column: 5)",
    "location": {
      "start": {
        "index": 0,
        "line": 0,
        "column": 0
      },
      "end": {
        "index": 0,
        "line": 0,
        "column": 0
      },
      "content": "Unknown location"
    },
    "stack": [
      "readUrl (anonymous:0:0)",
      "main (main:1:5)"
    ],
    "logs": [
      ]
  },
  "library": {
    "success": true,
    "value": "3",
    "mimeType": "application/dw",
    "encoding": "UTF-8",
    "logs": [
      ]
  }
}
```

```

        ],
    },
    "timeout": false,
    "execFail": {
        "success": false,
        "message": "My Bad\nTrace:\n  at fail (Unknown)\n  at main::main (line: 1, column: 1)",
        "location": {
            "start": {
                "index": 0,
                "line": 0,
                "column": 0
            },
            "end": {
                "index": 0,
                "line": 0,
                "column": 0
            },
            "content": "Unknown location"
        },
        "stack": [
            "fail (anonymous:0:0)",
            "main (main:1:1)"
        ],
        "logs": [
        ]
    },
    "parseFail": {
        "success": false,
        "message": "Invalid input \"1 + \\", expected parameter or parenEnd (line 1, column 2):\n\n1| (1 + \n      ^^^^\\nLocation:\\nmain (line: 1, column:2)",
        "location": {
            "start": {
                "index": 0,
                "line": 1,
                "column": 2
            },
            "end": {
                "index": 4,
                "line": 1,
                "column": 6
            },
            "content": "\n1| (1 + \n      ^^^^"
        },
        "logs": [
        ]
    },
    "writerFail": {
        "success": false,

```

```

    "message": "Trying to output non-whitespace characters outside main element tree
(in prolog or epilog), while writing Xml at .",
    "location": {
        "content": ""
    },
    "stack": [
        ],
    "logs": [
        ]
},
"readerFail": {
    "success": false,
    "message": "Unexpected end-of-input at payload@[1:18] (line:column), expected
'\\'', while reading `payload` as Json.\n \n1| {\"name\": \"Mariano\n
^",
    "location": {
        "content": "\n1| {\"name\": \"Mariano\n
^"
    },
    "stack": [
        ],
    "logs": [
        ]
},
"defaultOutput": {
    "success": true,
    "value": "name|lastName\nMariano|achaval\n",
    "mimeType": "application/csv",
    "encoding": "UTF-8",
    "logs": [
        ]
}
}

```

## runUrl

**runUrl(url: String, readerInputs: Dictionary<ReaderInput> = {}, inputValues: Dictionary<Any> = {}, configuration: RuntimeExecutionConfiguration = {}): RunSuccess | ExecutionFailure**

Runs the script at the specified URL.

*Experimental:* This function is an experimental feature that is subject to change or removal from future versions of DataWeave.

### Parameters

Name	Description
url	The name of the file to execute.
readerInputs	Inputs to read and bind to the execution.
inputValues	Inputs to be bind directly to the execution.
configuration	The runtime configuration.

## Example

This example shows how `runUrl` behaves with different inputs.

## Source

```
import * from dw::Runtime
var jsonValue = {
    value: '{"name": "Mariano"}' as Binary {encoding: "UTF-8"},
    encoding: "UTF-8",
    properties: {},
    mimeType: "application/json"
}

var Utils = "fun sum(a,b) = a +b"
---
{
    "execute_ok" :
    runUrl("classpath://org/mule/weave/v2/engine/runtime_runUrl/example.dwl", {"payload":
    jsonValue })
}
```

## Output

```
{
  "execute_ok": {
    "success": true,
    "value": "\"Mariano\"",
    "mimeType": "application/dw",
    "encoding": "UTF-8",
    "logs": [
      ]
  }
}
```

## try

`try<T>(delegate: () -> T): TryResult<T>`

Evaluates the delegate function and returns an object with `success: true` and `result` if the delegate function succeeds, or an object with `success: false` and `error` if the delegate function throws an exception.

The `orElseTry` and `orElse` functions will also continue processing if the `try` function fails. See the `orElseTry` and `orElse` documentation for more complete examples of handling failing `try` function expressions.

Note: Instead of using the `orElseTry` and `orElse` functions, based on the output of the `try` function, you can add conditional logic to execute when the result is `success: true` or `success: false`.

## Parameters

Name	Description
<code>delegate</code>	The function to evaluate.

## Example

This example calls the `try` function using the `randomNumber` function as argument. The function `randomNumber` generates a random number and calls `fail` if the number is greater than 0.5. The declaration of this function is in the script's header.

## Source

```
%dw 2.0
import try, fail from dw::Runtime
output application/json
fun randomNumber() =
  if(random() > 0.5)
    fail("This function is failing")
  else
    "OK"
---
try(() -> randomNumber())
```

## Output

When `randomNumber` fails, the output is:

```
{  
  "success": false,  
  "error": {  
    "kind": "UserException",  
    "message": "This function is failing",  
    "location": "Unknown location",  
    "stack": [  
      "fail (anonymous:0:0)",  
      "myFunction (anonymous:1:114)",  
      "main (anonymous:1:179)"  
    ]  
  }  
}
```

When `randomNumber` succeeds, the output is:

```
{  
  "success": true,  
  "result": "OK"  
}
```

## version

### version(): String

Returns the DataWeave version that is currently running.

*Introduced in DataWeave version 2.5.0.*

#### Example

This example returns the DataWeave version.

#### Source

```
%dw 2.0  
import * from dw::Runtime  
output application/json  
---  
version()
```

#### Output

```
"2.5"
```

## wait

`wait<T>(value: T, timeout: Number): T`

Stops the execution for the specified timeout period (in milliseconds).



Stopping the execution blocks the thread, potentially causing slowness, low performance and potentially freezing of the entire runtime. This operation is intended for limited functional testing purposes. Do not use this function in a production application, performance testing, or with multiple applications deployed.

### Parameters

Name	Description
<code>value</code>	Input of any type.
<code>timeout</code>	The number of milliseconds to wait.

### Example

This example waits 2000 milliseconds (2 seconds) to execute.

### Source

```
%dw 2.0
import * from dw::Runtime
output application/json
---
{ "user" : 1 } wait 2000
```

### Output

```
{ "user": 1 }
```

## Runtime Types

Type	Definition	Description
DataFormatDescriptor	<pre>type DataFormatDescriptor = {     name: String, binary: Boolean,     defaultEncoding?: String,     extensions: Array&lt;String&gt;,     defaultMimeType: String,     acceptedMimeTypes:         Array&lt;String&gt;,     readerProperties:         Array&lt;DataFormatProperty&gt;,     writerProperties:         Array&lt;DataFormatProperty&gt; }</pre>	Description of a <code>DataFormat</code> that provides all metadata information.  <i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.
DataFormatProperty	<pre>type DataFormatProperty = {     name: String, optional: Boolean, defaultValue?: Any,     description: String,     possibleValues: Array&lt;Any&gt; }</pre>	Type that describes a data format property. The fields include a <code>name</code> , <code>description</code> , array of possible values ( <code>possibleValues</code> ), an optional default value ( <code>defaultValue</code> ), and an <code>optional</code> flag that indicates whether the property is required or not.  <i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.
EvalSuccess	<pre>type EvalSuccess = { success: true, value: Any, logs: Array&lt;LogEntry&gt; }</pre>	Data type of the data that returns when an <code>eval</code> function executes successfully.  <i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.
ExecutionFailure	<pre>type ExecutionFailure = {     success: false, message: String, kind: String, stack?: Array&lt;String&gt;, location: Location, logs: Array&lt;LogEntry&gt; }</pre>	Data type of the data that returns when a <code>run</code> or <code>eval</code> function fails.  <i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.
Location	<pre>type Location = { start?: Position, end?: Position, locationString: String, text?: String, sourceIdentifier?: String }</pre>	Type that represents the location of an expression in a DataWeave file.  <i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.

Type	Definition	Description
LogEntry	<pre>type LogEntry = { level: LogLevel, timestamp: String, message: String }</pre>	<p>Type for a log entry, which consists of a <code>level</code> for a <code>LogLevel</code> value, a <code>timestamp</code>, and <code>message</code>.</p> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>
LogLevel	<pre>type LogLevel = "INFO"   "ERROR"   "WARN"</pre>	<p>Identifies the different kinds of log levels (<code>INFO</code>, <code>ERROR</code>, or <code>WARN</code>).</p> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>
LoggerService	<pre>type LoggerService = { initialize?: () -&gt; Object, log: (level: LogLevel, msg: String, context: Object) -&gt; Any, shutdown?: () -&gt; Boolean }</pre>	<p>Service that handles all logging:</p> <ul style="list-style-type: none"> <li>• <code>initialize</code>: Function called when the execution starts. DataWeave sends the result to every <code>log</code> call through the <code>context</code> parameter, so that, for example, a logging header can be sent at initialization and recovered in each log.</li> <li>• <code>log</code>: Function that is called on every log message.</li> <li>• <code>shutdown</code>: Function called when the execution completes, which is a common time to flush any buffer or to log out gracefully.</li> </ul> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>
MimeType	<pre>type MimeType = String</pre>	<p>A String representation of a MIME type.</p> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>
Position	<pre>type Position = { index: Number, line: Number, column: Number }</pre>	<p>Type that represents a position in a file by its index and its line and column.</p> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>

Type	Definition	Description
ReaderInput	<pre>type ReaderInput = { value: Binary, encoding?: String, properties?: Dictionary&lt;SimpleType&gt;, mimeType: MimeType }</pre>	<p>Input to the DataWeave reader created for the specified MIME type, which includes the Binary input and MIME type, as well as optional encoding and properties values.</p> <ul style="list-style-type: none"> <li>• <b>value</b>: The input, in Binary format.</li> <li>• <b>encoding</b>: The encoding for the reader to use.</li> <li>• <b>properties</b>: The reader properties used to parse the input.</li> <li>• <b>mimeType</b>: The MIME type of the input.</li> </ul> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>
RunSuccess	<pre>type RunSuccess = { success: true, value: Binary, mimeType: MimeType, encoding?: String, logs: Array&lt;LogEntry&gt; }</pre>	<p>Data type of the data that returns when a <code>run</code> function executes successfully.</p> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>

Type	Definition	Description
RuntimeExecutionConfiguration	<pre>type RuntimeExecutionConfiguration =   { timeOut?: Number, outputMimeType?: MimeType, writerProperties?: Dictionary&lt;SimpleType&gt;, onException?: "HANDLE"   "FAIL", securityManager?: SecurityManager, loggerService?: LoggerService, maxStackSize?: Number, onUnhandledTimeout?: (threadName: String, javaStackTrace: String, code: String) -&gt; Any }</pre>	<p>Configuration of the runtime execution that has advanced parameters.</p> <ul style="list-style-type: none"> <li>• <b>timeOut</b>: Maximum amount of time the DataWeave script takes before timing out.</li> <li>• <b>outputMimeType</b>: Default output MIME type if not specified in the DataWeave script.</li> <li>• <b>writerProperties</b>: Writer properties to use with the specified the <b>outputMimeType</b> property.</li> <li>• <b>onException</b> Specifies the behavior that occurs when the execution fails: <ul style="list-style-type: none"> <li>◦ <b>HANDLE</b> (default value) returns <b>ExecutionFailure</b>.</li> <li>◦ <b>FAIL</b> propagates an exception.</li> </ul> </li> <li>• <b>securityManager</b>: Identifies the <b>SecurityManager</b> to use in this execution. This security manager is composed by the current <b>SecurityManager</b>.</li> <li>• <b>loggerService</b>: The <b>LoggerService</b> to use in this execution.</li> <li>• <b>maxStackSize</b>: The maximum stack size.</li> <li>• <b>onUnhandledTimeout</b>: Callback that is called when the watchdog was not able to stop the execution after a timeout, which is useful for logging or reporting the problem. The callback is called with the following: <ul style="list-style-type: none"> <li>◦ <b>threadName</b>: Name of the thread that hanged.</li> <li>◦ <b>javaStackTrace</b>: Java stack trace where the hang occurred.</li> <li>◦ <b>code</b>: The DataWeave code that caused the hang.</li> </ul> </li> </ul> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>

Type	Definition	Description
SecurityManager	<code>type SecurityManager = (grant: String, args: Array&lt;Any&gt;) -&gt; Boolean</code>	<p>Function that is called when a privilege must be granted to the current execution.</p> <ul style="list-style-type: none"> <li>• <code>grant</code> is the name of the privilege, such as <code>Resource</code>.</li> <li>• <code>args</code> provides a list of parameters that the function requesting the privilege calls.</li> </ul> <p><i>Experimental:</i> This type is an experimental feature that is subject to change or removal from future versions of DataWeave.</p>
TryResult	<code>type TryResult = { success: Boolean, result?: T, error?: { kind: String, message: String, stack?: Array&lt;String&gt;, stackTrace?: String, location?: String } }</code>	<p>Object with a result or error message. If <code>success</code> is <code>false</code>, data type provides the <code>error</code>. If <code>true</code>, the data type provides the <code>result</code>.</p> <p>Starting in Mule 4.4.0, if the stack is not present, the <code>stackTrace</code> field is available with the native Java stack trace.</p>

## dw::System

This module contains functions that allow you to interact with the underlying system.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::System` to the header of your DataWeave script.

### Functions

Name	Description
<code>envVar</code>	Returns an environment variable with the specified name or <code>null</code> if the environment variable is not defined.
<code>envVars</code>	Returns all the environment variables defined in the host system as an array of strings.

### envVar

`envVar(variableName: String): String | Null`

Returns an environment variable with the specified name or `null` if the environment variable is not defined.

#### Parameters

Name	Description
variableName	String that provides the name of the environment variable.

## Example

This example returns a Mac command console (**SHELL**) path and returns **null** on **FAKE\_ENV\_VAR** (an undefined environment variable). **SHELL** is one of the standard Mac environment variables. Also notice that the **import** command enables you to call the function without prepending the module name to it.

## Source

```
%dw 2.0
import * from dw::System
output application/json
---
{
  "envVars" : [
    "real" : envVar("SHELL"),
    "fake" : envVar("FAKE_ENV_VAR")
  ]
}
```

## Output

```
"envVars": [
  {
    "real": "/bin/bash"
  },
  {
    "fake": null
  }
]
```

## envVars

### envVars(): Dictionary<String>

Returns all the environment variables defined in the host system as an array of strings.

## Example

This example returns a Mac command console (**SHELL**) path. **SHELL** is one of the standard Mac environment variables. To return all the environment variables, you can use **dw::System::envVars()**.

## Source

```
%dw 2.0
import dw::System
output application/json
---
{ "envVars" : dw::System::envVars().SHELL }
```

## Output

```
{ "envVars": "/bin/bash" }
```

# dw::util::Coercions

This utility module assists with type coercions.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::util::Coercions` to the header of your DataWeave script.

*Introduced in DataWeave version 2.4.0.*

## Functions

Name	Description
<code>toArray</code>	Splits a <code>String</code> value into an <code>Array</code> of characters.
<code>toBinary</code>	Transform a <code>String</code> value into a <code>Binary</code> value using the specified encoding.
<code>toBoolean</code>	Transform a <code>String</code> value into a <code>Boolean</code> value.
<code>toDate</code>	Transforms a <code>String</code> value into a <code>Date</code> value and accepts a format and locale.
<code>toDateTime</code>	Transforms a <code>Number</code> value into a <code>DateTime</code> value using <code>milliseconds</code> or <code>seconds</code> as the unit.
<code>toLocalDateTime</code>	Transforms a <code>String</code> value into a <code>LocalDateTime</code> value and accepts a format and locale.
<code>toLocalTime</code>	Transforms a <code>String</code> value into a <code>LocalTime</code> value and accepts a format and locale.
<code>toNumber</code>	A variant of <code>toNumber</code> that transforms a <code>DateTime</code> value into a number of seconds or milliseconds, depending on the selected unit.
<code>toPeriod</code>	Transform a <code>String</code> value into a <code>Period</code> value.
<code>toRegex</code>	Transforms a <code>String</code> value into a <code>Regex</code> value.
<code>toString</code>	A variant of <code>toString</code> that transforms a <code>Number</code> value (whole or decimal) into a <code>String</code> value and accepts a format, locale, and rounding mode value.

Name	Description
<code>toTime</code>	Transforms a <code>String</code> value into a <code>Time</code> value and accepts a format and locale.
<code>toTimeZone</code>	Transform a <code>String</code> value into a <code>TimeZone</code> value.
<code>toUri</code>	Transforms a <code>String</code> value into a <code>Uri</code> value.

## Types

- [Coercions Types](#)

## toArray

`toArray(@StreamCapable text: String): Array<String>`

Splits a `String` value into an `Array` of characters.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>text</code>	The <code>String</code> value to transform into an <code>Array</code> of characters (a <code>Array&lt;String&gt;</code> type).

### Example

This example shows how `toArray` behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json indent=false
---
{
  a: toArray(""),
  b: toArray("hola")
}
```

### Output

```
{"a": [], "b": ["h", "o", "l", "a"]}
```

## toBinary

## **toBinary(str: String, encoding: String): Binary**

Transform a **String** value into a **Binary** value using the specified encoding.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<b>str</b>	The <b>String</b> value to transform into a <b>Binary</b> value.
<b>encoding</b>	The encoding to apply to the <b>String</b> value. Accepts encodings that are supported by your JDK. For example, <b>encoding</b> accepts Java canonical names and aliases for the basic and extended encoding sets in Oracle JDK 8 and JDK 11.

### Example

This example shows how **toBinary** behaves with different inputs. It produces output in the **application/dw** format.

### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    'UTF-16Ex': toBinary("DW", "UTF-16"),
    'utf16Ex': toBinary("DW", "utf16"),
    'UnicodeBigEx': toBinary("DW", "UnicodeBig"),
    'UTF-32Ex': toBinary("DW", "UTF-32"),
    'UTF_32Ex': toBinary("DW", "UTF_32")
}
```

### Output

```
{
    "UTF-16Ex": "/v8ARABX" as Binary {base: "64"},
    utf16Ex: "/v8ARABX" as Binary {base: "64"},
    UnicodeBigEx: "/v8ARABX" as Binary {base: "64"},
    "UTF-32Ex": "AAAARAAAAFc=" as Binary {base: "64"},
    UTF_32Ex: "AAAARAAAAFc=" as Binary {base: "64"}
}
```

## **toBoolean**

### **toBoolean(str: String): Boolean**

Transform a **String** value into a **Boolean** value.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
str	The <b>String</b> value to transform into a <b>Boolean</b> value.

#### Example

This example shows how **toBoolean** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
  a: toBoolean("true"),
  b: toBoolean("false"),
  c: toBoolean("FALSE"),
  d: toBoolean("TrUe")
}
```

#### Output

```
{
  "a": true,
  "b": false,
  "c": false,
  "d": true
}
```

## toDate

**toDate(str: String, format: String | Null = null, locale: String | Null = null): Date**

Transforms a **String** value into a **Date** value and accepts a format and locale.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
str	The <b>String</b> value to transform into a <b>Date</b> value.

Name	Description
format	The formatting to use on the <code>Date</code> value. A <code>null</code> value has no effect on the <code>Date</code> value. This parameter accepts Java character patterns based on ISO-8601. A <code>Date</code> value, such as <code>2011-12-03</code> , has the format <code>uuuu-MM-dd</code> .
locale	Optional ISO 3166 country code to use, such as <code>US</code> , <code>AR</code> , or <code>ES</code> . A <code>null</code> or absent value uses your JVM default.

## Example

This example shows how `toDate` behaves with different inputs. It produces output in the `application/dw` format.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
  a: toDate("2015-10-01"),
  b: toDate("2003/10/01","uuuu/MM/dd")
}
```

## Output

```
{
  a: |2015-10-01|,
  b: |2003-10-01| as Date {format: "uuuu/MM/dd"}
}
```

## toDateTime

**toDateTime(number: Number, unit: MillisOrSecs | Null = null): DateTime**

Transforms a `Number` value into a `DateTime` value using `milliseconds` or `seconds` as the unit.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
number	The <code>Number</code> value to transform into a <code>DateTime</code> value.
unit	The unit to use for the conversion: <code>"milliseconds"</code> or <code>"seconds"</code> . A <code>null</code> value for the <code>unit</code> field defaults to <code>"seconds"</code> .

## Example

This example shows how `toDateTime` behaves with different inputs. It produces output in the `application/dw` format.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    fromEpoch: toDateTime(1443743879),
    fromMillis: toDateTime(1443743879000, "milliseconds")
}
```

## Output

```
{
    fromEpoch: |2015-10-01T23:57:59Z|,
    fromMillis: |2015-10-01T23:57:59Z| as DateTime {unit: "milliseconds"}
}
```

## toDateTime(str: String, format: String | Null = null, locale: String | Null = null): DateTime

Transforms a `String` value into a `DateTime` value and accepts a format and locale.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>str</code>	The <code>String</code> value to transform into a <code>DateTime</code> value.
<code>format</code>	The formatting to use on the <code>DateTime</code> value. A <code>null</code> value has no effect on the <code>DateTime</code> value. This parameter accepts Java character patterns based on ISO-8601. A <code>DateTime</code> value, such as <code>2011-12-03T10:15:30.000000+01:00</code> , has the format <code>uuuu-MM-dd HH:mm:ssz</code> .
<code>locale</code>	Optional ISO 3166 country code to use, such as <code>US</code> , <code>AR</code> , or <code>ES</code> . A <code>null</code> or absent value uses your JVM default.

## Example

This example shows how `toDateTime` behaves with different inputs. It produces output in the `application/dw` format.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    a: toDate("2015-10-01T23:57:59Z"),
    b: toDate("2003-10-01 23:57:59Z", "uuuu-MM-dd HH:mm:ssz")
}
```

## Output

```
{
    a: |2015-10-01T23:57:59|,
    b: |2003-10-01T23:57:59Z| as DateTime {format: "uuuu-MM-dd HH:mm:ssz"}
}
```

## toLocalDateTime

**toLocalDateTime(str: String, format: String | Null = null, locale: String | Null = null): LocalDateTime**

Transforms a **String** value into a **LocalDateTime** value and accepts a format and locale.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<b>str</b>	The <b>String</b> value to transform into a <b>LocalDateTime</b> value.
<b>format</b>	The formatting to use on the <b>LocalDateTime</b> value. A <b>null</b> value has no effect on the <b>LocalDateTime</b> value. This parameter accepts Java character patterns based on ISO-8601. A <b>LocalDateTime</b> value, such as <b>2011-12-03T10:15:30.000000</b> has the format <b>uuuu-MM-dd HH:mm:ss</b> .
<b>locale</b>	Optional ISO 3166 country code to use, such as <b>US</b> , <b>AR</b> , or <b>ES</b> . A <b>null</b> or absent value uses your JVM default.

### Example

This example shows how **toLocalDateTime** behaves with different inputs. It produces output in the **application/dw** format.

### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
  a: toLocalDateTime("2015-10-01T23:57:59"),
  b: toLocalDateTime("2003-10-01 23:57:59", "uuuu-MM-dd HH:mm:ss")
}
```

## Output

```
{
  a: |2015-10-01T23:57:59|,
  b: |2003-10-01T23:57:59| as LocalDateTime {format: "uuuu-MM-dd HH:mm:ss"}
}
```

## toLocalTime

**toLocalTime(str: String, format: String | Null = null, locale: String | Null = null): LocalTime**

Transforms a **String** value into a **LocalTime** value and accepts a format and locale.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<b>str</b>	The <b>String</b> value to transform into a <b>LocalTime</b> value.
<b>format</b>	The formatting to use on the <b>LocalTime</b> value. A <b>null</b> value has no effect on the <b>LocalTime</b> value. This parameter accepts Java character patterns based on ISO-8601. A <b>LocalTime</b> value, such as <b>22:15:30.000000</b> , has the format <b>HH:mm:ss.n</b> .
<b>locale</b>	Optional ISO 3166 country code to use, such as <b>US</b> , <b>AR</b> , or <b>ES</b> . A <b>null</b> or absent value uses your JVM default.

### Example

This example shows how **toLocalTime** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
    toLocalTimeEx: toLocalTime("23:57:59"),
    toLocalTimeEx2: toLocalTime("13:44:12.283", "HH:mm:ss.n")
}
```

## Output

```
{
    "toLocalTimeEx": "23:57:59",
    "toLocalTimeEx2": "13:44:12.283"
}
```

## toNumber

**toNumber(dateTime: DateTime, unit: MillisOrSecs | Null = null): Number**

A variant of `toNumber` that transforms a `DateTime` value into a number of seconds or milliseconds, depending on the selected unit.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>dateTime</code>	The <code>DateTime</code> value to transform into a <code>Number</code> value.
<code>unit</code>	The unit of time ("milliseconds" or "seconds") to use Given a <code>null</code> value, the function uses "seconds".

### Example

This example shows how `toNumber` behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
    epoch: toNumber(|2015-10-01T23:57:59Z|),
    millis: toNumber(|2015-10-01T23:57:59Z|, "milliseconds")
}
```

## Output

```
{  
    "epoch": 1443743879,  
    "millis": 1443743879000  
}
```

## toNumber(period: Period, unit: PeriodUnits | Null = null): Number

A variant of `toNumber` that transforms a `Period` value into a number of hours, minutes, seconds, milliseconds or nanoseconds (`nanos`).

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>period</code>	The <code>Period</code> value to transform into a <code>Number</code> value.
<code>unit</code>	The unit to apply to the specified <code>period</code> : <code>hours</code> , <code>minutes</code> , <code>seconds</code> , <code>milliseconds</code> , or <code>nanos</code> .

### Example

This example shows how `toNumber` behaves with different inputs.

### Source

```
%dw 2.0  
import * from dw::util::Coercions  
output application/json  
---  
{  
    toSecondsEx1: toNumber(|PT1H10M|, "seconds"),  
    toSecondsEx2: toNumber(|PT1M7S|, "milliseconds")  
}
```

## Output

```
{  
    "toSecondsEx1": 4200,  
    "toSecondsEx2": 67000  
}
```

## toNumber(value: String | Key, format: String | Null = null, locale: String | Null = null): Number

A variant of `toNumber` that transforms a `String` or `Key` value into a `Number` value and that accepts a

format and locale.

Introduced in DataWeave version 2.4.0.

## Parameters

Name	Description
value	The <b>String</b> or <b>Key</b> value to transform into a <b>Number</b> value.
format	Optional formatting to apply to the <b>value</b> . A <b>format</b> accepts <b>#</b> or <b>0</b> (but not both) as placeholders for <i>decimal</i> values and a single whole number that is less than <b>10</b> . Only one decimal point is permitted. A <b>null</b> or empty <b>String</b> value has no effect on the <b>Number</b> value. Other characters produce an error.
locale	Optional ISO 3166 country code to use, such as <b>US</b> , <b>AR</b> , or <b>ES</b> . A <b>null</b> or absent value uses your JVM default.

## Example

This example shows how **toNumber** behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Coercions
var myKey = keysOf({"123" : "myValue"})
output application/json
---
{
    "default": toNumber("1.0"),
    "withFormat": toNumber("0.005",".00"),
    "withLocal": toNumber("1,25","###","ES"),
    "withExtraPlaceholders": toNumber("5.55","####.#####"),
    "keyToNumber": toNumber(myKey[0])
}
```

## Output

```
{
    "default": 1.0,
    "withFormat": 0.005,
    "withLocal": 1.25,
    "withExtraPlaceholders": 5.55,
    "keyToNumber": 123
}
```

## toPeriod

### toPeriod(str: String): Period

Transform a **String** value into a **Period** value.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
str	The <b>String</b> value to transform into a <b>Period</b> value.

#### Example

This example shows how **toPeriod** behaves with different inputs. It produces output in the **application/dw** format.

#### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    toPeriodEx1: toPeriod("P1D"),
    toPeriodEx2: toPeriod("PT1H1M")
}
```

#### Output

```
{
    toPeriodEx1: |P1D|,
    toPeriodEx2: |PT1H1M|
}
```

## toRegex

### toRegex(str: String): Regex

Transforms a **String** value into a **Regex** value.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
str	The <code>String</code> value to transform into a <code>Regex</code> value.

## Example

This example shows how `toRegex` behaves with different inputs. It produces output in the `application/dw` format.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    toRegexEx1: toRegex("a-Z"),
    toRegexEx2: toRegex("0-9+")
}
```

## Output

```
{
    toRegexEx1: /a-Z/,
    toRegexEx2: /0-9+/
}
```

## toString

`toString(number: Number, format: String | Null = null, locale: String | Null = null, roundMode: RoundingMode | Null = null): String`

A variant of `toString` that transforms a `Number` value (whole or decimal) into a `String` value and accepts a format, locale, and rounding mode value.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
number	The <code>Number</code> value to format.

Name	Description
<code>format</code>	The formatting to apply to the <code>Number</code> value. A <code>format</code> accepts <code>#</code> or <code>0</code> (but not both) as placeholders for <i>decimal</i> values, and only one decimal point is permitted. A <code>null</code> or empty <code>String</code> value has no effect on the <code>Number</code> value. Most other values are treated as literals, but you must escape special characters, such as a dollar sign (for example, <code>\\$</code> ). Inner quotations must be closed and differ from the surrounding quotations.
<code>locale</code>	Optional ISO 3166 country code to use, such as <code>US</code> , <code>AR</code> , or <code>ES</code> . A <code>null</code> or absent value uses your JVM default. When you pass a translatable format, such as <code>eeee</code> and <code>MMMM</code> , a <code>locale</code> (such as <code>"ES</code> ) transforms the corresponding numeric values to a localized string.

Name	Description
roundMode	<p>Optional parameter for rounding decimal values when the formatting presents a rounding choice, such as a format of <code>0.#</code> for the decimal <code>0.15</code>. The default is <code>HALF_UP</code>, and a <code>null</code> value behaves like <code>HALF_UP</code>. Only one of the following values is permitted:</p> <ul style="list-style-type: none"> <li>• <code>UP</code>: Always rounds away from zero (for example, <code>0.01</code> to <code>"0.1"</code> and <code>-0.01</code> to <code>"-0.1"</code>). Increments the preceding digit to a non-zero fraction and never decreases the magnitude of the calculated value.</li> <li>• <code>DOWN</code>: Always rounds towards zero (for example, <code>0.19</code> to <code>"0.1"</code> and <code>-0.19</code> to <code>"-0.1"</code>). Never increments the digit before a discarded fraction (which truncates to the preceding digit) and never increases the magnitude of the calculated value.</li> <li>• <code>CEILING</code>: Rounds towards positive infinity and behaves like <code>UP</code> if the result is positive (for example, <code>0.35</code> to <code>"0.4"</code>). If the result is negative, this mode behaves like <code>DOWN</code> (for example, <code>-0.35</code> to <code>"-0.3"</code>). This mode never decreases the calculated value.</li> <li>• <code>FLOOR</code>: Rounds towards negative infinity and behaves like <code>DOWN</code> if the result is positive (for example, <code>0.35</code> to <code>"0.3"</code>). If the result is negative, this mode behaves like <code>UP</code> (for example, <code>-0.35</code> to <code>"-0.4"</code>). The mode never increases the calculated value.</li> <li>• <code>HALF_UP</code>: Default mode, which rounds towards the nearest "neighbor" unless both neighbors are equidistant, in which case, this mode rounds up. For example, <code>0.35</code> rounds to <code>"0.4"</code>, <code>0.34</code> rounds to <code>"0.3"</code>, and <code>0.36</code> rounds to <code>"0.4"</code>. Negative decimals numbers round similarly. For example, <code>-0.35</code> rounds to <code>"-0.4"</code>.</li> <li>• <code>HALF_DOWN</code>: Rounds towards the nearest numeric "neighbor" unless both neighbors are equidistant, in which case, this mode rounds down. For example, <code>0.35</code> rounds to <code>"0.3"</code>, <code>0.34</code> rounds to <code>"0.3"</code>, and <code>0.36</code> rounds to <code>"0.4"</code>. Negative decimals numbers round similarly. For example, <code>-0.35</code> rounds to <code>"-0.3"</code>.</li> <li>• <code>HALF_EVEN</code>: For decimals that end in a <code>5</code> (such as, <code>1.125</code> and <code>1.135</code>), the behavior depends on the number that precedes the <code>5</code>. <code>HALF_EVEN</code> rounds up when the next-to-last digit before the <code>5</code> is an odd number but rounds down when the next-to-last digit is even. For example, <code>0.225</code> rounds to <code>"0.22"</code>, <code>0.235</code> and <code>0.245</code> round to <code>"0.24"</code>, and <code>0.255</code> rounds to <code>"0.26"</code>. Negative decimals round similarly, for example, <code>-0.225</code> to <code>"-0.22"</code>. When the last digit is not <code>5</code>, the setting behaves like <code>HALF_UP</code>. Rounding of monetary values sometimes follows the <code>HALF_EVEN</code> pattern.</li> </ul>

## Example

This example shows how `toString` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
    a: toString(1.0),
    b: toString(0.005, ".00"),
    c: toString(0.035, "#.##", "ES"),
    d: toString(0.005, "#.##", "ES", "HALF_EVEN"),
    e: toString(0.035, "#.00", null, "HALF_EVEN"),
    f: toString(1.1234, "\$.## 'in my account'")
}
```

## Output

```
{
    "a": "1",
    "b": ".01",
    "c": "0,04",
    "d": "0",
    "e": ".04",
    "f": "$1.12 in my account"
}
```

**toString(date: Date | DateTime | LocalDateTime | LocalTime | Time, format: String | Null = null, locale: String | Null = null): String**

A variant of [toString](#) that transforms a [Date](#), [DateTime](#), [LocalTime](#), [LocalDateTime](#), or [Time](#) value into a [String](#) value.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
date	The <a href="#">Date</a> , <a href="#">DateTime</a> , <a href="#">LocalTime</a> , <a href="#">LocalDateTime</a> , or <a href="#">Time</a> value to coerce to a <a href="#">String</a> type.

Name	Description
format	<p>The ISO-8601 formatting to use on the date or time. For example, this parameter accepts character patterns based on the Java 8 <code>java.time.format</code>. A <code>null</code> value has no effect on the value. Defaults:</p> <ul style="list-style-type: none"> <li>• <code>Date</code> example: <code>2011-12-03</code> (equivalent format: <code>uuuu-MM-dd</code>)</li> <li>• <code>DateTime</code> example: <code>2011-12-03T10:15:30.000000+01:00</code> (equivalent format: <code>uuuu-MM-dd HH:mm:ssz</code>)</li> <li>• <code>LocalDateTime</code> example: <code>2011-12-03T10:15:30.000000</code> (equivalent format: <code>uuuu-MM-dd HH:mm:ss</code>)</li> <li>• <code>LocalTime</code> example: <code>10:15:30.000000</code> (equivalent format: <code>HH:mm:ss.n</code>)</li> <li>• <code>Time</code> example: <code>10:15:30.000000Z</code> (equivalent format: <code>HH:mm:ss.nxxxz</code>)</li> </ul>
locale	Optional ISO 3166 country code to use, such as <code>US</code> , <code>AR</code> , or <code>ES</code> . A <code>null</code> or absent value uses your JVM default. When you pass a translatable format, such as <code>eeee</code> and <code>MMMM</code> , a <code>locale</code> (such as <code>"ES"</code> ) transforms the corresponding numeric values to a localized string.

## Example

This example shows how `toString` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
    aDate: toString(|2003-10-01|, "uuuu/MM/dd"),
    aDateTime: toString(|2018-09-17T22:13:00-03:00|),
    aLocalTime: toString(|23:57:59|, "HH-mm-ss"),
    aLocalDateTime : toString(|2015-10-01T23:57:59|),
    aLocalDateTimeFormatted: toString(|2003-10-01T23:57:59|, "uuuu-MM-dd HH:mm:ss a"),
    aLocalDateTimeFormattedAndLocalizedSpain: toString(|2003-01-01T23:57:59|, "eeee, dd
MMMM, uuuu HH:mm:ss a", "ES"),
    aTime: typeOf(|22:10:18Z|),
    aTimeZone: toString(|-03:00|)
```

## Output

```
{
  "aDate": "2003/10/01",
  "aDateTime": "2018-09-17T22:13:00-03:00",
  "aLocalTime": "23-57-59",
  "aLocalDateTime": "2015-10-01T23:57:59",
  "aLocalDateTimeFormatted": "2003-10-01 23:57:59 PM",
  "aLocalDateTimeFormattedAndLocalizedSpain": "miércoles, 01 enero, 2003 23:57:59 p.m.",
  "aTime": "Time",
  "aTimeZone": "-03:00"
}
```

## **toString(binary: Binary, encoding: String): String**

A variant of `toString` that transforms a `Binary` value into a `String` value with the specified encoding.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>binary</code>	The <code>Binary</code> value to coerce to a <code>String</code> value.
<code>encoding</code>	The encoding to apply to the <code>String</code> value. Accepts encodings that are supported by your JDK. For example, <code>encoding</code> accepts Java canonical names and aliases for the basic and extended encoding sets in Oracle JDK 8 and JDK 11.

### Example

This example shows how `toString` behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::util::Coercions
var binaryData= "DW Test" as Binary {encoding: "UTF-32"}
output application/json
---
{
  a: toString(binaryData, "UTF-32"),
}
```

### Output

```
{  
  "a": "DW Test"  
}
```

## toString(data: TimeZone | Uri | Boolean | Period | Regex | Key): String

A variant of `toString` that transforms a `TimeZone`, `Uri`, `Boolean`, `Period`, `Regex`, or `Key` value into a string.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>data</code>	The <code>TimeZone</code> , <code>Uri</code> , <code>Boolean</code> , <code>Period</code> , <code>Regex</code> , or <code>Key</code> value to coerce to a <code>String</code> value.

### Example

This example shows how `toString` behaves with different inputs.

### Source

```
%dw 2.0  
import * from dw::util::Coercions  
output application/json  
---  
{  
  transformTimeZone: toString(|Z|),  
  transformBoolean: toString(true),  
  transformPeriod: toString(|P1D|),  
  transformRegex: toString(/a-Z/),  
  transformPeriod: toString(|PT8M10S|),  
  transformUri: toString("https://docs.mulesoft.com/" as Uri)  
} ++  
{ transformKey : toString((keysOf({ "aKeyToString" : "aValue" })[0])) }
```

### Output

```
{
  "transformTimeZone": "Z",
  "transformBoolean": "true",
  "transformPeriod": "P1D",
  "transformRegex": "a-Z",
  "transformPeriod": "PT8M10S",
  "transformUri": "https://docs.mulesoft.com/",
  "transformKey": "aKeyToString"
}
```

## **toString(arr: Array<String>): String**

A variant of `toString` that joins an `Array` of characters into a single `String` value.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<code>arr</code>	The <code>Array</code> of characters to transform into a <code>String</code> value.

### Example

This example shows how `toString` behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
  a: toString([]),
  b: toString(["h", "o", "l", "a"])
}
```

### Output

```
{
  "a": "",
  "b": "hola"
}
```

## **toTime**

### **toTime(str: String, format: String | Null = null, locale: String | Null = null): Time**

Transforms a **String** value into a **Time** value and accepts a format and locale.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
<b>str</b>	The <b>String</b> value to transform into a <b>Time</b> value.
<b>format</b>	The formatting to use on the <b>Time</b> value. A <b>null</b> value has no effect on the <b>Time</b> value. This parameter accepts Java character patterns based on ISO-8601. A <b>Time</b> value, such as <b>10:15:30.000000</b> , has the format <b>HH:mm:ss.nxxx</b> .
<b>locale</b>	Optional ISO 3166 country code to use, such as <b>US</b> , <b>AR</b> , or <b>ES</b> . A <b>null</b> or absent value uses your JVM default.

## Example

This example shows how **toTime** behaves with different inputs. It produces output in the **application/dw** format.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    a: toTime("23:57:59Z"),
    b: toTime("13:44:12.283-08:00", "HH:mm:ss.nxxx")
}
```

## Output

```
{
    a: |23:57:59Z|,
    b: |13:44:12.000000283-08:00| as Time {format: "HH:mm:ss.nxxx"}
}
```

## toTimeZone

### toTimeZone(str: String): TimeZone

Transform a **String** value into a **TimeZone** value.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
str	The <b>String</b> value to transform into a <b>TimeZone</b> value.

## Example

This example shows how **toTimeZone** behaves with different inputs. It produces output in the **application/dw** format.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/dw
---
{
    timeZoneOffset: toTimeZone("-03:00"),
    timeZoneAbbreviation: toTimeZone("Z"),
    timeZoneName: toTimeZone("America/Argentina/Buenos_Aires")
}
```

## Output

```
{
    timeZoneOffset: |-03:00|,
    timeZoneAbbreviation: |Z|,
    timeZoneName: |America/Argentina/Buenos_Aires|
}
```

## toUri

### toUri(str: String): Uri

Transforms a **String** value into a **Uri** value.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
str	The <b>String</b> value to transform into a <b>Uri</b> value.

## Example

This example shows how **toUri** behaves.

## Source

```
%dw 2.0
import * from dw::util::Coercions
output application/json
---
{
  toUriExample: toUri("https://www.google.com/")
}
```

## Output

```
{
  "toUriExample": "https://www.google.com/"
}
```

## Coercions Types

Type	Definition	Description
MillisOrSecs	type MillisOrSecs = "milliseconds"   "seconds"	Type used for setting units to "milliseconds" or "seconds".
PeriodUnits	type PeriodUnits = "hours"   "minutes"   "seconds"   "milliseconds"   "nanos"	Type used for setting units of a <b>Period</b> value to "hours", "minutes", "seconds", "milliseconds", or "nanos".
RoundingMode	type RoundingMode = "UP"   "DOWN"   "CEILING"   "FLOOR"   "HALF_UP"   "HALF_DOWN"   "HALF_EVEN"	Type used when rounding decimal values up or down.

## dw::util::Diff

This utility module calculates the difference between two values and returns a list of differences.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::util::Diff` to the header of your DataWeave script.

## Functions

Name	Description
diff	Returns the structural differences between two values.

## Types

- [Diff Types](#)

## diff

**diff(actual: Any, expected: Any, diffConfig: { unordered?: Boolean } = {}, path: String = "(root)": Diff)**

Returns the structural differences between two values.

Differences between objects can be ordered (the default) or unordered. Ordered means that two objects do not differ if their key-value pairs are in the same order. Differences are expressed as [Difference](#) type.

### Parameters

Name	Description
<code>actual</code>	The actual value. Can be any data type.
<code>expected</code>	The expected value to compare to the actual. Can be any data type.
<code>diffConfig</code>	Setting for changing the default to unordered using `{"unordered": true}` (explained in the introduction).

### Example

This example shows a variety of uses of [diff](#).

### Source

```

import diff from dw::util::Diff
ns ns0 http://localhost.com
ns ns1 http://acme.com
output application/dw
---
{
  "a": diff({a: 1}, {b:1}),
  "b": diff({ns0#a: 1}, {ns1#a:1}),
  "c": diff([1,2,3], []),
  "d": diff([], [1,2,3]),
  "e": diff([1,2,3], [1,2,3, 4]),
  "f": diff([{a: 1}], [{a: 2}]),
  "g": diff({a @({c: 2}): 1}, {a @({c: 3}): 1}),
  "h": diff(true, false),
  "i": diff(1, 2),
  "j": diff("test", "other test"),
  "k": diff({a: 1}, {a:1}),
  "l": diff({ns0#a: 1}, {ns0#a:1}),
  "m": diff([1,2,3], [1,2,3]),
  "n": diff([], []),
  "o": diff([{a: 1}], [{a: 1}]),
  "p": diff({a @({c: 2}): 1}, {a @({c:2}): 1}),
  "q": diff(true, true),
  "r": diff(1, 1),
  "s": diff("other test", "other test"),
  "t": diff({a:1 ,b: 2},{b: 2, a:1}, {unordered: true}),
  "u": [{format: "ssn",data: "ABC"}] diff [{ format: "ssn",data: "ABC"}]
}

```

## Output

```

ns ns0 http://localhost.com
ns ns1 http://acme.com
---
{
  a: {
    matches: false,
    diffs: [
      {
        expected: "Entry (root).a with type Number",
        actual: "was not present in object.",
        path: "(root).a"
      }
    ]
  },
  b: {
    matches: false,
    diffs: [
      {

```

```
    expected: "Entry (root).ns0#a with type Number",
    actual: "was not present in object.",
    path: "(root).ns0#a"
}
]
},
c: {
  matches: false,
  diff: [
    {
      expected: "Array size is 0",
      actual: "was 3",
      path: "(root)"
    }
  ]
},
d: {
  matches: false,
  diff: [
    {
      expected: "Array size is 3",
      actual: "was 0",
      path: "(root)"
    }
  ]
},
e: {
  matches: false,
  diff: [
    {
      expected: "Array size is 4",
      actual: "was 3",
      path: "(root)"
    }
  ]
},
f: {
  matches: false,
  diff: [
    {
      expected: "1" as String {mimeType: "application/dw"},
      actual: "2" as String {mimeType: "application/dw"},
      path: "(root)[0].a"
    }
  ]
},
g: {
  matches: false,
  diff: [
    {
      expected: "3" as String {mimeType: "application/dw"},
```

```
        actual: "2" as String {mimeType: "application/dw"},  
        path: "(root).a.@.c"  
    }  
]  
,  
h: {  
    matches: false,  
    diffs: [  
        {  
            expected: "false",  
            actual: "true",  
            path: "(root)"  
        }  
    ]  
,  
i: {  
    matches: false,  
    diffs: [  
        {  
            expected: "2",  
            actual: "1",  
            path: "(root)"  
        }  
    ]  
,  
j: {  
    matches: false,  
    diffs: [  
        {  
            expected: "\"other test\"",  
            actual: "\"test\"",  
            path: "(root)"  
        }  
    ]  
,  
k: {  
    matches: true,  
    diffs: []  
},  
l: {  
    matches: true,  
    diffs: []  
},  
m: {  
    matches: true,  
    diffs: []  
},  
n: {  
    matches: true,  
    diffs: []  
},  
o: {  
    matches: true,  
    diffs: []  
},
```

```

o: {
  matches: true,
  diffs: []
},
p: {
  matches: true,
  diffs: []
},
q: {
  matches: true,
  diffs: []
},
r: {
  matches: true,
  diffs: []
},
s: {
  matches: true,
  diffs: []
},
t: {
  matches: true,
  diffs: []
},
u: {
  matches: true,
  diffs: []
}
}

```

## Diff Types

Type	Definition	Description
Diff	<pre>type Diff = { matches: Boolean,          diffs: Array&lt;Difference&gt; }</pre>	<p>Describes the entire difference between two values.</p> <ul style="list-style-type: none"> <li><i>Example with no differences:</i> { "matches": true, "diffs": [ ] }</li> <li><i>Example with differences:</i> { "matches": true, "diffs": [ { "expected": "4", "actual": "2", "path": "(root).a.@.d" } ] }</li> </ul> <p>See the <code>diff</code> function for another example.</p>
Difference	<pre>type Difference = { expected: String, actual: String, path: String }</pre>	Describes a single difference between two values at a given structure.

# dw::util::Math

A utility module that provides mathematical functions.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::util::Math` to the header of your DataWeave script.

*Introduced in DataWeave version 2.4.0.*

## Functions

Name	Description
<code>acos</code>	Returns an arc cosine value that can range from <code>0.0</code> through pi.
<code>asin</code>	Returns an arc sine value that can range from <code>-pi/2</code> through <code>pi/2</code> .
<code>atan</code>	Returns an arc tangent value that can range from <code>-pi/2</code> through <code>pi/2</code> .
<code>cos</code>	Returns the trigonometric cosine of an angle from a given number of radians.
<code>log10</code>	Returns the logarithm base 10 of a number.
<code>logn</code>	Returns the natural logarithm (base <code>e</code> ) of a number.
<code>sin</code>	Returns the trigonometric sine of an angle from a given number of radians.
<code>tan</code>	Returns the trigonometric tangent of an angle from a given number of radians.
<code>toDegrees</code>	Converts an angle measured in radians to an approximately equivalent number of degrees.
<code>toRadians</code>	Converts a given number of degrees in an angle to an approximately equivalent number of radians.

## Variables

- [Math Variables](#)

### acos

`acos(angle: Number): Number | NaN`

Returns an arc cosine value that can range from `0.0` through pi.

If the absolute value of the input is greater than `1`, the result is `null`.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
angle	Number to convert into its arc cosine value.

## Example

This example shows how `acos` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
    "acos0": acos(0),
    "acos13": acos(0.13),
    "acos-1": acos(-1),
    "acos1": acos(1),
    "acos1.1": acos(1.1)
}
```

## Output

```
{
    "acos0": 1.5707963267948966,
    "acos13": 1.440427347091751,
    "acos-1": 3.141592653589793,
    "acos1": 0.0,
    "acos1.1": null
}
```

## asin

**asin(angle: Number): Number | NaN**

Returns an arc sine value that can range from `-pi/2` through `pi/2`.

If the absolute value of the input is greater than 1, the result is `null`.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
angle	Number to convert into its arc sine value.

## Example

This example shows how `asin` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
  "asin0": asin(0),
  "asin13": asin(0.13),
  "asin-1": asin(-1),
  "asin1.1": asin(1.1)
}
```

## Output

```
{
  "asin0": 0.0,
  "asin13": 0.1303689797031455,
  "asin-1": -1.5707963267948966,
  "asin1.1": null
}
```

## atan

**atan(angle: Number): Number**

Returns an arc tangent value that can range from  $-\pi/2$  through  $\pi/2$ .

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
angle	Number to convert into its arc tangent value.

## Example

This example shows how `atan` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
  "atan0": atan(0),
  "atan13": atan(0.13),
  "atan-1": atan(-1)
}
```

## Output

```
{
  "atan0": 0.0,
  "atan13": 0.12927500404814307,
  "atan-1": -0.7853981633974483
}
```

## cos

### cos(angle: Number): Number

Returns the trigonometric cosine of an angle from a given number of radians.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
angle	Number of radians in an angle.

#### Example

This example shows how `cos` behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
  "cos0": cos(0),
  "cos13": cos(0.13),
  "cos-1": cos(-1)
}
```

## Output

```
{  
  "cos0": 1.0,  
  "cos13": 0.9915618937147881,  
  "cos-1": 0.5403023058681398  
}
```

## log10

**log10(a: Number): Number | NaN**

Returns the logarithm base 10 of a number.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
a	A <b>Number</b> value that serves as input to the function.

### Example

This example shows how **log10** behaves with different inputs.

### Source

```
%dw 2.0  
import * from dw::util::Math  
output application/json  
---  
{  
  "log1010": log10(10),  
  "log1013": log10(0.13),  
  "log10-20": log10(-20)  
}
```

## Output

```
{  
  "log1010": 1.0,  
  "log1013": -0.8860566476931632,  
  "log10-20": null  
}
```

## logn

**logn(a: Number): Number | NaN**

Returns the natural logarithm (base **e**) of a number.

If the input value is less than or equal to zero, the result is **NaN** (or **null**).

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
a	Number to convert into its natural logarithm.

### Example

This example shows how **logn** behaves with different inputs.

### Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
  "logn10": logn(10),
  "logn13": logn(0.13),
  "logn-20": logn(-20)
}
```

### Output

```
{
  "logn10": 2.302585092994046,
  "logn13": -2.0402208285265546,
  "logn-20": null
}
```

## sin

**sin(angle: Number): Number**

Returns the trigonometric sine of an angle from a given number of radians.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
angle	Number of radians in an angle.

## Example

This example shows how `sin` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
  "sin0": sin(0),
  "sin13": sin(0.13),
  "sin-1": sin(-1)
}
```

## Output

```
{
  "sin0": 0.0,
  "sin13": 0.12963414261969486,
  "sin-1": -0.8414709848078965
}
```

## tan

### `tan(angle: Number): Number`

Returns the trigonometric tangent of an angle from a given number of radians.

*Introduced in DataWeave version 2.4.0.*

## Parameters

Name	Description
angle	Number of radians in an angle.

## Example

This example shows how `tan` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
  "tan0": tan(0),
  "tan13": tan(0.13),
  "tan-1": tan(-1)
}
```

## Output

```
{
  "tan0": 0.0,
  "tan13": 0.13073731800446006,
  "tan-1": -1.5574077246549023
}
```

## toDegrees

### toDegrees(angrad: Number): Number

Converts an angle measured in radians to an approximately equivalent number of degrees.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
angrad	Number of radians to convert to degrees.

#### Example

This example shows how `toDegrees` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
    "toDegrees0.17": toDegrees(0.174),
    "toDegrees0": toDegrees(0),
    "toDegrees-20": toDegrees(-0.20)
}
```

## Output

```
{
    "toDegrees0.17": 9.969465635276323832571267395889251,
    "toDegrees0": 0E+19,
    "toDegrees-20": -11.45915590261646417536927286883822
}
```

## toRadians

### **toRadians(*angdeg*: Number): Number**

Converts a given number of degrees in an angle to an approximately equivalent number of radians.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<i>angdeg</i>	Number of degrees to convert into radians.

#### Example

This example shows how **toRadians** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::util::Math
output application/json
---
{
    "toRadians10": toRadians(10),
    "toRadians013": toRadians(0.13),
    "toRadians-20": toRadians(-20)
}
```

## Output

```
{  
    "toRadians10": 0.174532925199432957692222222222222,  
    "toRadians013": 0.0022689280275926284499988888888889,  
    "toRadians-20": -0.3490658503988659153844444444444444  
}
```

## Math Variables

Variable	Definition	Description
E	E	Variable E sets the value of mathematical constant e, the base of natural logarithms.  <i>Introduced in DataWeave version 2.4.0.</i>
PI	PI	Variable PI sets the value of constant value pi, the ratio of the circumference of a circle to its diameter.  <i>Introduced in DataWeave version 2.4.0.</i>

## dw::util::Timer

This utility module contains functions for measuring time.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::util::Timer` to the header of your DataWeave script.

## Functions

Name	Description
currentMilliseconds	Returns the current time in milliseconds.
duration	Executes the input function and returns an object with execution time in milliseconds and result of that function.
time	Executes the input function and returns a TimeMeasurement object that contains the start and end time for the execution of that function, as well the result of the function.
toMilliseconds	Returns the representation of a specified date-time in milliseconds.

## Types

- [Timer Types](#)

## currentMilliseconds

### currentMilliseconds(): Number

Returns the current time in milliseconds.

#### Example

This example shows the time in milliseconds when the function executed.

#### Source

```
%dw 2.0
import * from dw::util::Timer
output application/json
---
{ "currentMilliseconds" : currentMilliseconds() }
```

#### Output

```
{ "currentMilliseconds": 1532923168900 }
```

## duration

### duration<T>(valueToMeasure: () -> T): DurationMeasurement<T>

Executes the input function and returns an object with execution time in milliseconds and result of that function.

#### Parameters

Name	Description
valueToMeasure	A function to pass to <b>duration</b> .

#### Example

This example passes a `wait` function (defined in the header), which returns the execution time and result of that function in a `DurationMeasurement` object.

#### Source

```
%dw 2.0
output application/json
fun myFunction() = dw::Runtime::wait("My result",100)
---
dw::util::Timer::duration(() -> myFunction())
```

## Output

```
{  
    "time": 101,  
    "result": "My result"  
}
```

## time

**time<T>(valueToMeasure: () -> T): TimeMeasurement<T>**

Executes the input function and returns a **TimeMeasurement** object that contains the start and end time for the execution of that function, as well the result of the function.

### Parameters

Name	Description
valueToMeasure	A function to pass to <b>time</b> .

### Example

This example passes **wait** and **sum** functions (defined in the header), which return their results in **TimeMeasurement** objects.

```
%dw 2.0  
output application/json  
fun myFunction() = dw::Runtime::wait("My result",100)  
fun myFunction2() = sum([1,2,3,4])  
---  
{ testing: [  
    dw::util::Timer::time(() -> myFunction()),  
    dw::util::Timer::time(() -> myFunction2())  
]
```

## Output

```
{
  "testing": [
    {
      "start": "2018-10-05T19:23:01.49Z",
      "result": "My result",
      "end": "2018-10-05T19:23:01.591Z"
    },
    {
      "start": "2018-10-05T19:23:01.591Z",
      "result": 10,
      "end": "2018-10-05T19:23:01.591Z"
    }
  ]
}
```

## toMilliseconds

### toMilliseconds(date: DateTime): Number

Returns the representation of a specified date-time in milliseconds.

#### Parameters

Name	Description
date	A <a href="#">DateTime</a> to evaluate.

#### Example

This example shows a date-time in milliseconds.

#### Source

```
%dw 2.0
import * from dw::util::Timer
output application/json
---
{ "toMilliseconds" : toMilliseconds(|2018-07-23T22:03:04.829Z|) }
```

#### Output

```
{ "toMilliseconds": 1532383384829 }
```

## Timer Types

Type	Definition	Description
DurationMeasurement	<code>type DurationMeasurement = { time: Number, result: T }</code>	A return type that contains the execution time and result of a function call.
TimeMeasurement	<code>type TimeMeasurement = { start: DateTime, result: T, end: DateTime }</code>	A return type that contains a start time, end time, and result of a function call.

## dw::util::Tree

This utility module provides functions for handling values as tree-data structures.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::util::Tree` to the header of your DataWeave script.

*Introduced in DataWeave version 2.2.2.*

## Functions

Name	Description
<code>asExpressionString</code>	Transforms a <code>Path</code> value into a string representation of the path.
<code>filterArrayLeafs</code>	Applies a filtering expression to leaf or <code>Path</code> values of an array.
<code>filterObjectLeafs</code>	Applies a filtering expression to leaf or <code>Path</code> values of keys in an object.
<code>filterTree</code>	Filters the value or path of nodes in an input based on a specified <code>criteria</code> .
<code>isArrayType</code>	Returns <code>true</code> if the provided <code>Path</code> value is an <code>ARRAY_TYPE</code> expression.
<code>isAttributeType</code>	Returns <code>true</code> if the provided <code>Path</code> value is an <code>ATTRIBUTE_TYPE</code> expression.
<code>isObjectType</code>	Returns <code>true</code> if the provided <code>Path</code> value is an <code>OBJECT_TYPE</code> expression.
<code>mapLeafValues</code>	Maps the terminal (leaf) nodes in the tree.
<code>nodeExists</code>	Returns <code>true</code> if any node in a given tree validates against the specified criteria.

## Variables

- [Tree Variables](#)

## Types

- [Tree Types](#)

## asExpressionString

`asExpressionString(path: Path): String`

Transforms a **Path** value into a string representation of the path.

*Introduced in DataWeave version 2.2.2.*

#### Parameters

Name	Description
path	The <b>Path</b> value to transform into a <b>String</b> value.

#### Example

This example transforms a **Path** value into a **String** representation of a selector for an attribute of an object.

#### Source

```
%dw 2.0
import * from dw::util::Tree
output application/json
---
asExpressionString([
    {kind: OBJECT_TYPE, selector: "user", namespace: null},
    {kind: ATTRIBUTE_TYPE, selector: "name", namespace: null}
])
```

#### Output

```
".user.@name"
```

## filterArrayLeafs

**filterArrayLeafs(value: Any, criteria: (value: Any, path: Path) -> Boolean): Any**

Applies a filtering expression to leaf or **Path** values of an array.

The leaf values in the array must be **SimpleType** or **Null** values. See [Core Types](#) for descriptions of the types.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
value	An input value of <b>Any</b> type.
criteria	Boolean expression to apply to <b>SimpleType</b> or <b>Null</b> leaf values of all arrays in the input <b>value</b> . If the result is <b>true</b> , the array retains the leaf value. If not, the function removes the leaf value from the output.

## Example

This example shows how `filterArrayLeafs` behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Tree
var myArray = [1, {name: "", true}, test: 213}, "123", null]
output application/json
---
{
    a: myArray filterArrayLeafs ((value, path) ->
        !(value is Null or value is String)),
    b: myArray filterArrayLeafs ((value, path) ->
        (value is Null or value == 1)),
    c: { a : [1,2] } filterArrayLeafs ((value, path) ->
        (value is Null or value == 1)),
    d: myArray filterArrayLeafs ((value, path) ->
        !isArrayType(path))
}
```

## Output

```
{  
  "a": [  
    1,  
    {  
      "name": [  
        true  
      ],  
      "test": 213  
    }  
  ],  
  "b": [  
    1,  
    {  
      "name": [  
  
      ],  
      "test": 213  
    },  
    null  
  ],  
  "c": {  
    "a": [  
      1  
    ]  
  },  
  "d": [  
    {  
      "name": [  
  
      ],  
      "test": 213  
    }  
  ]  
}
```

## filterObjectLeafs

**filterObjectLeafs(value: Any, criteria: (value: Any, path: Path) -> Boolean): Any**

Applies a filtering expression to leaf or **Path** values of keys in an object.

The leaf values in the object must be **SimpleType** or **Null** values. See [Core Types](#) for descriptions of the types.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
value	An input value of <b>Any</b> type.
criteria	Boolean expression to apply to <b>SimpleType</b> or <b>Null</b> leaf values of all objects in the input <b>value</b> . If the result is <b>true</b> , the object retains the leaf value and its key. If not, the function removes the leaf value and key from the output.

## Example

This example shows how **filterObjectLeafs** behaves with different inputs.

## Source

```
%dw 2.0
import * from dw::util::Tree
var myArray = [{name @(mail: "me@me.com", test:123 ): "", id:"test"}, {name: "Me", id:null}]
output application/json
---
{
  a: {
    name: "Mariano",
    lastName: null,
    age: 123,
    friends: myArray
  } filterObjectLeafs ((value, path) ->
    !(value is Null or value is String)),
  b: { c : null, d : "hello" } filterObjectLeafs ((value, path) ->
    (value is Null and isObjectType(path)))
}
```

## Output

```
{
  "a": {
    "age": 123,
    "friends": [
      {
        },
        {
          }
        ]
    },
    "b": {
      "c": null
    }
}
```

## filterTree

**filterTree(value: Any, criteria: (value: Any, path: Path) -> Boolean): Any**

Filters the value or path of nodes in an input based on a specified **criteria**.

The function iterates through the nodes in the input. The **criteria** can apply to the value or path in the input. If the **criteria** evaluates to **true**, the node remains in the output. If **false**, the function filters out the node.

*Introduced in DataWeave version 2.4.0.*

### Parameters

Name	Description
<b>value</b>	The value to filter.
<b>criteria</b>	The expression that determines whether to filter the node.

### Example

This example shows how **filterTree** behaves with different inputs. The output is **application/dw** for demonstration purposes.

### Source

```
%dw 2.0
import * from dw::util::Tree
output application/dw
---
{
  a: {
    name : "",
    lastName @(foo: ""): "Achaval",
    friends @(id: 123): [{id: "", test: true}, {age: 123}, ""]
  } filterTree ((value, path) ->
    value match {
      case s is String -> !isEmpty(s)
      else -> true
    }
  ),
  b: null filterTree ((value, path) -> value is String),
  c: [
    {name: "Mariano", friends: []},
    {test: [1,2,3]},
    {dw: ""}
  ] filterTree ((value, path) ->
    value match {
      case a is Array -> !isEmpty(a as Array)
      else -> true
    })
}
}
```

## Output

```
{
  a: {
    lastName: "Achaval",
    friends @({id: 123}): [
      {
        test: true
      },
      {
        {
          age: 123
        }
      ]
    ],
    b: null,
    c: [
      {
        name: "Mariano"
      },
      {
        test: [
          1,
          2,
          3
        ]
      },
      {
        dw: ""
      }
    ]
  }
}
```

## isArrayType

### isArrayType(path: Path): Boolean

Returns **true** if the provided **Path** value is an **ARRAY\_TYPE** expression.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
path	The <b>Path</b> value to validate.

#### Example

This example shows how **isArrayType** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::util::Tree
output application/json
---
{
  a: isArrayType([{kind: OBJECT_TYPE, selector: "user", namespace: null},
                  {kind: ATTRIBUTE_TYPE, selector: "name", namespace: null}]),
  b: isArrayType([{kind: OBJECT_TYPE, selector: "user", namespace: null},
                  {kind: ARRAY_TYPE, selector: 0, namespace: null}]),
  c: isArrayType([{kind: ARRAY_TYPE, selector: 0, namespace: null}])
}
```

## Output

```
{
  "a": false,
  "b": true,
  "c": true
}
```

## isAttributeType

### isAttributeType(path: Path): Boolean

Returns **true** if the provided **Path** value is an **ATTRIBUTE\_TYPE** expression.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<b>path</b>	The <b>Path</b> value to validate.

#### Example

This example shows how **isAttributeType** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::util::Tree
output application/json
---
{
    a: isAttributeType([{"kind": OBJECT_TYPE, selector: "user", namespace: null},
                        {"kind": ATTRIBUTE_TYPE, selector: "name", namespace: null}]),
    b: isAttributeType([{"kind": OBJECT_TYPE, selector: "user", namespace: null},
                        {"kind": ARRAY_TYPE, selector: "name", namespace: null}]),
    c: isAttributeType([{"kind": ATTRIBUTE_TYPE, selector: "name", namespace: null}])
}
```

## Output

```
{
    "a": true,
    "b": false,
    "c": true
}
```

## isObjectType

### isObjectType(path: Path): Boolean

Returns **true** if the provided **Path** value is an **OBJECT\_TYPE** expression.

*Introduced in DataWeave version 2.4.0.*

#### Parameters

Name	Description
<b>path</b>	The <b>Path</b> value to validate.

#### Example

This example shows how **isObjectType** behaves with different inputs.

#### Source

```
%dw 2.0
import * from dw::util::Tree
output application/json
---
{
    a: isObjectType([{kind: OBJECT_TYPE, selector: "user", namespace: null},
                    {kind: ATTRIBUTE_TYPE, selector: "name", namespace: null}]),
    b: isObjectType([{kind: OBJECT_TYPE, selector: "user", namespace: null},
                    {kind: OBJECT_TYPE, selector: "name", namespace: null}]),
    c: isObjectType([{kind: OBJECT_TYPE, selector: "user", namespace: null}])
}
```

## Output

```
{
    "a": false,
    "b": true,
    "c": true
}
```

## mapLeafValues

**mapLeafValues(value: Any, callback: (value: Any, path: Path) -> Any): Any**

Maps the terminal (leaf) nodes in the tree.

Leafs nodes cannot have an object or an array as a value.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
value	The value to map.
callback	The mapper function.

### Example

This example transforms all the string values to upper case.

### Source

```
%dw 2.0
import * from dw::util::Tree
output application/json
---
{
    user: [
        {
            name: "mariano",
            lastName: "achaval"
        }],
    group: "data-weave"
} mapLeafValues (value, path) -> upper(value)
```

## Output

```
{
    "user": [
        {
            "name": "MARIANO",
            "lastName": "ACHAVAL"
        }
    ],
    "group": "DATA-WEAVE"
}
```

## Example

This example returns a new value for an object, array, or attribute.

## Source

```
%dw 2.0
output application/json
import * from dw::util::Tree
---
{
    name: "Mariano",
    test: [1,2,3]
} mapLeafValues ((value, path) -> if(isObjectType(path))
    "***"
    else if(isArrayType(path))
        "In an array"
    else "Is an attribute")
```

## Output

```
{  
  "name": "***",  
  "test": [  
    "In an array",  
    "In an array",  
    "In an array"  
  ]  
}
```

## nodeExists

**nodeExists(value: Any, callback: (value: Any, path: Path) -> Boolean): Boolean**

Returns **true** if any node in a given tree validates against the specified criteria.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
<b>value</b>	The value to search.
<b>callback</b>	The criteria to apply to the input <b>value</b> .

### Example

This example checks for each user by name and last name. Notice that you can also reference a **value** with **\$** and the **path** with **\$\$**.

### Source

```
%dw 2.0
import * from dw::util::Tree
var myObject =  {
    user: [{{
        name: "mariano",
        lastName: "achaval",
        friends: [
            {
                {
                    name: "julian"
                },
                {
                    name: "tom"
                }
            ]
        },
        {
            name: "leandro",
            lastName: "shokida",
            friends: [
                {
                    {
                        name: "peter"
                    },
                    {
                        name: "robert"
                    }
                ]
            }
        ]
    }]
}
output application/json
---
{
    mariano : myObject nodeExists ((value, path) -> path[-1].selector == "name" and value == "mariano"),
    julian : myObject nodeExists ((value, path) -> path[-1].selector == "name" and value == "julian"),
    tom : myObject nodeExists ($$[-1].selector == "name" and $ == "tom"),
    leandro : myObject nodeExists ($$[-1].selector == "name" and $ == "leandro"),
    peter : myObject nodeExists ($$[-1].selector == "name" and $ == "peter"),
    wrongField: myObject nodeExists ($$[-1].selector == "wrongField"),
    teo: myObject nodeExists ($$[-1].selector == "name" and $ == "teo")
}
```

## Output

```
{
  "mariano": true,
  "julian": true,
  "tom": true,
  "leandro": true,
  "peter": true,
  "wrongField": false,
  "teo": false
}
```

## Tree Variables

Variable	Definition	Description
ARRAY_TYPE	ARRAY_TYPE	Variable used to identify a <a href="#">PathElement</a> value as an array.  <i>Introduced in DataWeave version 2.2.2.</i>
ATTRIBUTE_TYPE	ATTRIBUTE_TYPE	Variable used to identify a <a href="#">PathElement</a> value as an attribute.  <i>Introduced in DataWeave version 2.2.2.</i>
OBJECT_TYPE	OBJECT_TYPE	Variable used to identify a <a href="#">PathElement</a> value as an object.  <i>Introduced in DataWeave version 2.2.2.</i>

## Tree Types

Type	Definition	Description
Path	type Path = Array<PathElement>	Type that consists of an array of <a href="#">PathElement</a> values that identify the location of a node in a tree. An example is <code>[{kind: OBJECT_TYPE, selector: "user", namespace: null}, {kind: ATTRIBUTE_TYPE, selector: "name", namespace: null}]</code> as Path.  <i>Introduced in DataWeave version 2.2.2.</i>

Type	Definition	Description
PathElement	<pre>type PathElement = {   kind: "Object"   "Attribute"   "Array", selector: String   Number, namespace: Namespace   Null  }</pre>	Type that represents a selection of a node in a path. An example is <code>{kind: ARRAY_TYPE, selector: "name", namespace: null} as PathElement.</code>  <i>Introduced in DataWeave version 2.2.2.</i>

## dw::util::Values

This utility module simplifies changes to values.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::util::Values` to the header of your DataWeave script.

*Introduced in DataWeave version 2.2.2.*

## Functions

Name	Description
attr	This function creates a <code>PathElement</code> to use for selecting an XML attribute and populates the type's <code>selector</code> field with the given string.
field	This function creates a <code>PathElement</code> data type to use for selecting an <i>object field</i> and populates the type's <code>selector</code> field with the given string.
index	This function creates a <code>PathElement</code> data type to use for selecting an <i>array element</i> and populates the type's <code>selector</code> field with the specified index.
mask	This <code>mask</code> function replaces all <i>simple</i> elements that match the specified criteria.
update	This <code>update</code> function updates a field in an object with the specified string value.

## Types

- [Values Types](#)

### attr

**attr(namespace: Namespace | Null = null, name: String): PathElement**

This function creates a `PathElement` to use for selecting an XML attribute and populates the type's `selector` field with the given string.

Some versions of the `update` and `mask` functions accept a `PathElement` as an argument.

*Introduced in DataWeave version 2.2.2.*

## Parameters

Name	Description
namespace	The namespace of the attribute to select. If not specified, a null value is set.
name	The string that names the attribute to select.

## Example

This example creates an attribute selector for a specified namespace (`ns0`) and sets the selector's value to `"myAttr"`. In the output, also note that the value of the `"kind"` key is `"Attribute"`.

## Source

```
%dw 2.0
output application/json
import * from dw::util::Values
ns ns0 http://acme.com/fo
---
attr(ns0 , "myAttr")
```

## Output

```
{
  "kind": "Attribute",
  "namespace": "http://acme.com/foo",
  "selector": "myAttr"
}
```

## field

**field(namespace: Namespace | Null = null, name: String): PathElement**

This function creates a `PathElement` data type to use for selecting an *object field* and populates the type's `selector` field with the given string.

Some versions of the `update` and `mask` functions accept a `PathElement` as an argument.

*Introduced in DataWeave version 2.2.2.*

## Parameters

Name	Description
namespace	The namespace of the field to select. If not specified, a null value is set.
name	A string that names the attribute to select.

## Example

This example creates an object field selector for a specified namespace (`ns0`) and sets the selector's value to `"myFieldName"`. In the output, also note that the value of the `"kind"` key is `"Object"`.

## Source

```
%dw 2.0
output application/json
import * from dw::util::Values
ns ns0 http://acme.com/foo
---
field(ns0 , "myFieldName")
```

## Output

```
{
  "kind": "Object",
  "namespace": "http://acme.com/foo",
  "selector": "myFieldName"
}
```

# index

## index(index: Number): PathElement

This function creates a `PathElement` data type to use for selecting an *array element* and populates the type's `selector` field with the specified index.

Some versions of the `update` and `mask` functions accept a `PathElement` as an argument.

*Introduced in DataWeave version 2.2.2.*

## Parameters

Name	Description
<code>index</code>	The index.

## Example

This example creates an selector for a specified index. It sets the selector's value to `0`. In the output, also note that the value of the `"kind"` key is `"Array"`.

## Source

```
%dw 2.0
output application/json
import * from dw::util::Values
ns ns0 http://acme.com/foo
---
index(0)
```

## Output

```
{
  "kind": "Array",
  "namespace": null,
  "selector": 0
}
```

## mask

**mask(value: Null, fieldName: String | Number | PathElement): (newValueProvider: (oldValue: Any, path: Path) -> Any) -> Null**

Helper function that enables `mask` to work with a `null` value.

*Introduced in DataWeave version 2.2.2.*

**mask(value: Any, selector: PathElement): (newValueProvider: (oldValue: Any, path: Path) -> Any) -> Any**

This `mask` function replaces all *simple* elements that match the specified criteria.

Simple elements do not have child elements and cannot be objects or arrays.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
<code>value</code>	A value to use for masking. The value can be any DataWeave type.
<code>selector</code>	The <code>PathElement</code> selector.

### Example

This example shows how to mask the value of a `password` field in an array of objects. It uses `field("password")` to return the `PathElement` that it passes to `mask`. It uses `with "" to specify the value ()` to use for masking.

### Source

```
%dw 2.0
output application/json
import * from dw::util::Values
---
[{"name": "Peter Parker", "password": "spiderman"}, {"name": "Bruce Wayne", "password": "batman"}] mask field("password") with "*****"
```

## Output

```
[
{
  "name": "Peter Parker",
  "password": "*****"
},
{
  "name": "Bruce Wayne",
  "password": "*****"
}
]
```

**mask(value: Any, fieldName: String): (newValueProvider: (oldValue: Any, path: Path) -> Any) -> Any**

This **mask** function selects a field by its name.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
<b>value</b>	The value to use for masking. The value can be any DataWeave type.
<b>fieldName</b>	A string that specifies the name of the field to mask.

### Example

This example shows how to perform masking using the name of a field in the input. It modifies the values of all fields with that value.

### Source

```
%dw 2.0
output application/json
import * from dw::util::Values
---
[{"name": "Peter Parker", "password": "spiderman"}, {"name": "Bruce Wayne", "password": "batman"}] mask "password" with "*****"
```

## Output

```
[  
 {  
   "name": "Peter Parker",  
   "password": "*****"  
 },  
 {  
   "name": "Bruce Wayne",  
   "password": "*****"  
 }  
 ]
```

**mask(value: Any, i: Number): (newValueProvider: (oldValue: Any, path: Path) -> Any) -> Any**

This **mask** function selects an element from array by its index.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
value	The value to mask. The value can be any DataWeave type.
index	The index to mask. The index must be a number.

### Example

This example shows how **mask** acts on all elements in the nested arrays. It changes the value of each element at index **1** to **false**.

### Source

```
%dw 2.0  
output application/json  
import * from dw::util::Values  
---  
[[123, true], [456, true]] mask 1 with false
```

## Output

```
[  
  [  
    123,  
    false  
  ],  
  [  
    456,  
    false  
  ]  
]
```

## update

**update(objectValue: Object, fieldName: String): UpdaterValueProvider<Object>**

This **update** function updates a field in an object with the specified string value.

The function returns a new object with the specified field and value.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
objectValue	The object to update.
fieldName	A string that provides the name of the field.

### Example

This example updates the `name` field in the object `{name: "Mariano"}` with the specified value.

### Source

```
%dw 2.0  
import * from dw::util::Values  
output application/json  
--  
{name: "Mariano"} update "name" with "Data Weave"
```

### Output

```
{  
  "name": "Data Weave"  
}
```

**update(objectValue: Object, fieldName: PathElement): UpdaterValueProvider<Object>**

This **update** function updates an object field with the specified **PathElement** value.

The function returns a new object with the specified field and value.

*Introduced in DataWeave version 2.2.2.*

## Parameters

Name	Description
<b>objectValue</b>	The object to update.
<b>fieldName</b>	A <b>PathElement</b> that specifies the field name.

## Example

This example updates the value of a **name** field in the object `{name: "Mariano"}`. It uses `field("name")` to return the **PathElement** that it passes to **update**. It uses `with "Data Weave"` to specify the value (**Data Weave**) of **name**.

## Source

```
%dw 2.0
import * from dw::util::Values
output application/json
---
{name: "Mariano"} update field("name") with "Data Weave"
```

## Output

```
{
  "name": "Data Weave"
}
```

## update(arrayValue: Array, indexToUpdate: Number): UpdaterValueProvider<Array>

Updates an array index with the specified value.

This **update** function returns a new array that changes the value of the specified index.

*Introduced in DataWeave version 2.2.2.*

## Parameters

Name	Description
<b>objectValue</b>	The array to update.
<b>indexToUpdate</b>	The index of the array to update. The index must be a number.

## Example

This example replaces the value `2` (the index is `1`) with `5` in the the input array `[1,2,3]`.

## Source

```
%dw 2.0
import * from dw::util::Values
output application/json
---
[1,2,3] update 1 with 5
```

## Output

```
[
  1,
  5,
  3
]
```

### update(arrayValue: Array, indexToUpdate: String): UpdaterValueProvider<Array>

This `update` function updates all objects within the specified array with the given string value.

*Introduced in DataWeave version 2.2.2.*

## Parameters

Name	Description
<code>objectValue</code>	The array of objects to update.
<code>indexToUpdate</code>	A string providing the name of the field to update.

## Example

This example updates value of the `role` fields in the array of objects.

## Source

```
%dw 2.0
import * from dw::util::Values
output application/json
---
[{"role: "a", name: "spiderman"}, {"role: "b", name: "batman"}] update "role" with
"Super Hero"
```

## Output

```
[{  
    "role": "Super Hero",  
    "name": "spiderman"  
},  
{  
    "role": "Super Hero",  
    "name": "batman"  
}]
```

## update(arrayValue: Array, indexToUpdate: PathElement): UpdaterValueProvider<Array>

This `update` function updates the specified index of an array with the given `PathElement` value.

The function returns a new array that contains given value at the specified index.

*Introduced in DataWeave version 2.2.2.*

### Parameters

Name	Description
<code>objectValue</code>	The array to update.
<code>indexToUpdate</code>	The index of the array to update. The index must be specified as a <code>PathElement</code> .

### Example

This example updates the value of an element in the input array. Notice that it uses `index(1)` to return the index as a `PathElement`, which it passes to `update`. It replaces the value `2` at that index with `5`.

### Source

```
%dw 2.0  
import * from dw::util::Values  
output application/json  
---  
[1,2,3] update index(1) with 5
```

### Output

```
[  
  1,  
  5,  
  3  
]
```

**update(value: Array | Object | Null, path: Array<String | Number | PathElement>): UpdaterValueProvider<Array | Object | Null>**

Updates the value at the specified path with the given value.

*Introduced in DataWeave version 2.2.2.*

#### Parameters

Name	Description
objectValue	The value to update. Accepts an array, object, or null value.
path	The path to update. The path must be an array of strings, numbers, or `PathElement`'s.

#### Example

This example updates the name of the user.

#### Source

```
%dw 2.0
import * from dw::util::Values
output application/json
---
{user: {name: "Mariano"}} update ["user", field("name")] with "Data Weave"
```

#### Output

```
{
  "user": {
    "name": "Data Weave"
  }
}
```

#### Example

This example updates one of the recurring fields.

#### Input

```
<users>
  <user>
    <name>Phoebe</name>
    <language>French</language>
  </user>
  <user>
    <name>Joey</name>
    <language>English</language>
  </user>
</users>
```

## Source

```
%dw 2.0
import * from dw::util::Values
output application/xml
---
payload update ["users", "user", "language"] with (if ($ == "English") "Gibberish"
else $)
```

## Output

```
<users>
  <user>
    <name>Phoebe</name>
    <language>French</language>
  </user>
  <user>
    <name>Joey</name>
    <language>Gibberish</language>
  </user>
</users>
```

## update(value: Null, toUpdate: Number | String | PathElement): UpdaterValueProvider<Null>

Helper function that enables `update` to work with a `null` value.

*Introduced in DataWeave version 2.2.2.*

## Values Types

Type	Definition	Description
UpdaterValueProvider	<code>type UpdaterValueProvider = (newValueProvider: (oldValue: Any, index: Number) -&gt; Any) -&gt; ReturnType</code>	Type that represents the output type of the update function.  <i>Introduced in DataWeave version 2.4.0.</i>

## dw::xml::Dtd

This module contains helper functions for working with XML doctype declarations.

To use this module, you must import it to your DataWeave code, for example, by adding the line `import * from dw::xml::Dtd` to the header of your DataWeave script.

*Introduced in DataWeave version 2.5.0.*

## Functions

Name	Description
<code>docTypeAsString</code>	Transforms a <code>DocType</code> value to a string representation.

## Types

- [Dtd Types](#)

### docTypeAsString

`docTypeAsString(docType: DocType): String`

Transforms a `DocType` value to a string representation.

*Introduced in DataWeave version 2.5.0.*

#### Parameters

Name	Type	Description
<code>docType</code>	<code>DocType</code>	The <code>DocType</code> value to transform to a string.

#### Example

This example transforms a `DocType` value that includes a `systemId` to a string representation.

#### Source

```
%dw 2.0
import * from dw::xml::Dtd
output application/json
---
docTypeAsString({rootName: "cXML", systemId:
"http://xml.cxml.org/schemas/cXML/1.2.014/cXML.dtd"})
```

## Output

```
"cXML SYSTEM http://xml.cxml.org/schemas/cXML/1.2.014/cXML.dtd"
```

## Example

This example transforms a `DocType` value that includes a `publicId` and `systemId` to a string representation.

## Source

```
%dw 2.0
import * from dw::xml::Dtd
output application/json
---
docTypeAsString({rootName: "html", publicId: "-//W3C//DTD XHTML 1.0 Transitional//EN",
systemId: "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"})
```

## Output

```
"html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
```

## Dtd Types

Type	Definition	Description
DocType	<pre>type DocType = { rootName: String, publicId?: String, systemId?: String, internalSubset?: String }</pre>	<p>DataWeave type for representing a doctype declaration that is part of an XML file. Supports the following fields:</p> <ul style="list-style-type: none"> <li>• <code>rootName</code>: Root element of the declaration.</li> <li>• <code>publicId</code>: Publicly available standard (optional).</li> <li>• <code>systemId</code>: Local URL (optional).</li> <li>• <code>internalSubset</code>: Internal DTD subset (optional).</li> </ul> <p><i>Introduced in DataWeave version 2.5.0.</i></p>