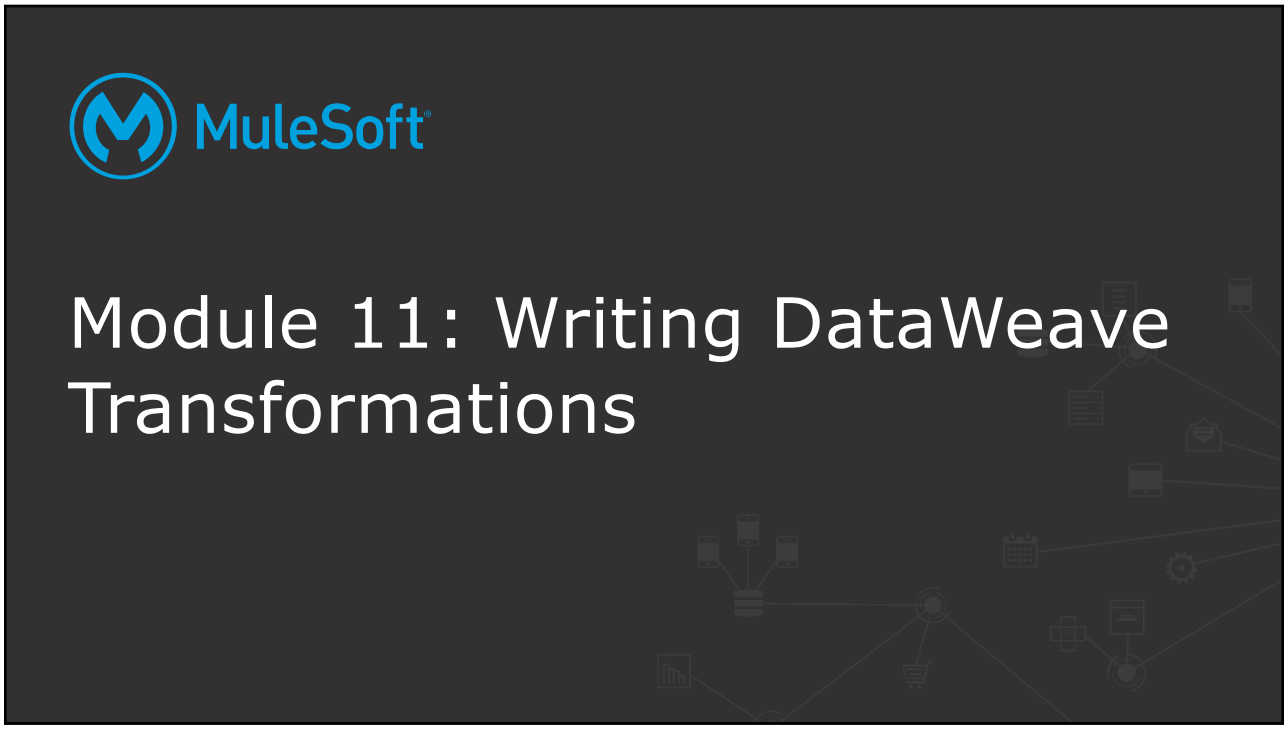


Module 11: Writing DataWeave Transformations

Goal

MuleSoft

- You have been using DataWeave throughout class
  - To write inline expressions to dynamically set the value of properties in event processors
  - To transform data – this was mostly generated by the graphical drag-and-drop editor so far

- In this module, you
  - Learn to write DataWeave transformations from scratch
  - Get familiar with the language so you can write more complicated transformations that are not possible with the drag-and-drop GUI

Goal

At the end of this module, you should be able to

- Write DataWeave expressions for basic XML, JSON, and Java transformations
- Write DataWeave transformations for complex data structures with repeated elements
- Define and use global and local variables and functions
- Use DataWeave functions
- Coerce and format strings, numbers, and dates
- Define and use custom data types
- Call Mule flows from DataWeave expressions
- Store DataWeave scripts in external files

Creating transformations with the Transform Message component

Creating transformations with the Transform Message component

MuleSoft

- To now, you have used the Transform Message component to
  - Create transformations using the visual editor
  - Define metadata for the input and output payload
  - Write basic transformation expressions
- But…
  - Where does the code go?
  - Can you save the code externally and reuse it?
  - What happens when you create sample data for live preview?
  - Does the target of a transformation have to be the payload?

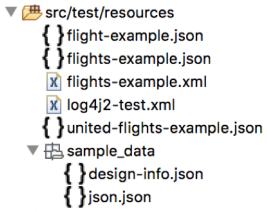
Where is the DataWeave code?

MuleSoft

- By default, the expression is placed inline in the Mule configuration file

```
<flow doc:id="a2d732ff-aaec-448e-9bdd-a25319cc55a2" name="postFlight
    <ee:transform doc:name="Transform Message" doc:id="ccda7d44-19b9
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
output application/java
---
payload]]></ee:set-payload>
        </ee:message>
        <ee:variables >
            <ee:set-variable variableName="DWoutput" ><![CDATA[%dw 2
output application/json
---
payload]]></ee:set-variable>
        </ee:variables>
    </ee:transform>
    <logger level="INFO" doc:name="Logger" doc:id="6b956315-38e9-421
</flow>
```

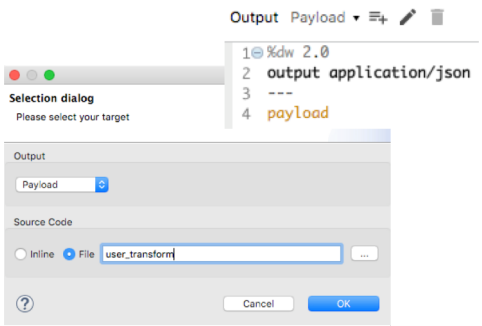
- Sample data used for live preview is stored in src/test/resources



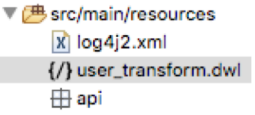
Exporting DataWeave code to a DWL file

MuleSoft

- In the Transform Message component, click the Edit current target button and set source code to file
  - Transform is saved in a DWL file
  - DWL files are stored in src/main/resources




```
<ee:transform doc:name="Transform Message" doc:id="69db8
    <ee:message >
        <ee:set-payload resource="user_transform.dwl" />
    </ee:message>
</ee:transform>
```



Reusing DataWeave code

MuleSoft

- A single script can be stored in an **external DWL file** and referenced
  - In the Transform Message component using the resource attribute
  - In an element that has an expression property (like the Choice router) using the ${file::filename} syntax

```
<ee:transform doc:name="Transform Message" doc:id="69db8
    <ee:message >
        <ee:set-payload resource="user_transform.dwl" />
    </ee:message>
</ee:transform>
```

- You can also create **modules** (libraries) of reusable DataWeave functions

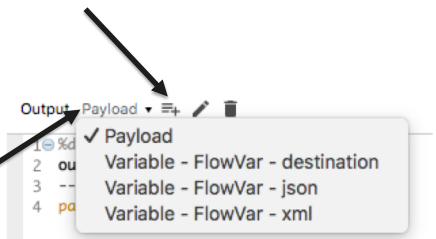
Creating multiple transformations

MuleSoft

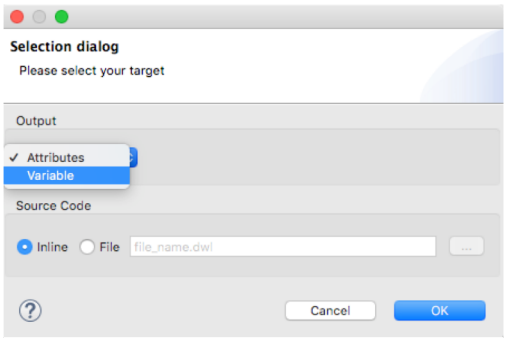
- You can also create multiple transformations with one Transform Message component

(1) Use Add new target button

(2) Set where to store result



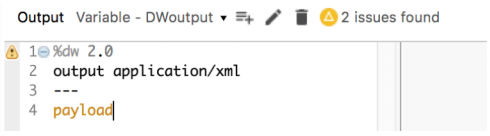
(3) Switch between multiple transformations

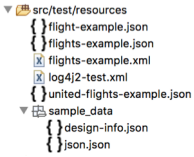


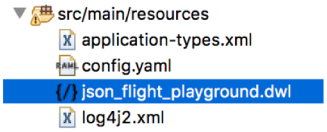
Walkthrough 11-1: Create transformations with the Transform Message component

MuleSoft

- Create a new flow that receives POST requests of JSON flight objects
- Add sample data and use live preview
- Create a second transformation that stores the output in a variable
- Save a DataWeave script in an external file
- Review DataWeave script errors





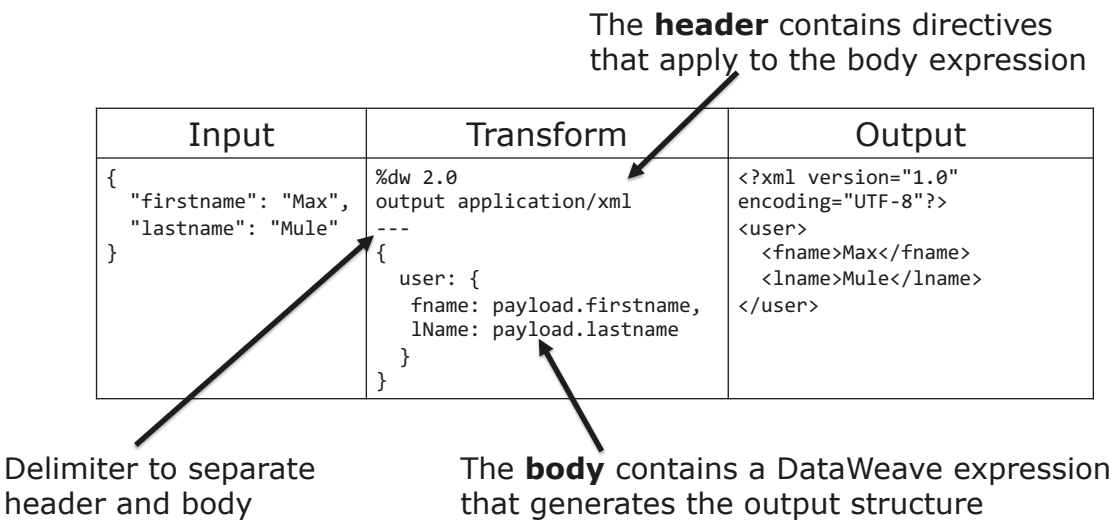


Writing DataWeave transformation expressions

The DataWeave expression is a data model for the output

- It is not dependent upon the types of the input and output, just their structures
- It's against this model that the transform executes
- The data model of the produced output can consist of three different types of data
  - **Objects**: Represented as collection of key value pairs
  - **Arrays**: Represented as a sequence of comma separated values
  - **Simple literals**

Example DataWeave transformation expression



The output directive

MuleSoft

- Specifies the mime type (format) that the script outputs

Some we have used or seen	application/json application/java application/xml text/plain
Others we will use	**application/dw** (DataWeave – for testing DataWeave expressions) application/csv
Others	application/xlsx application/flatfile (Flat File, Cobol Copybook, Fixed Width) multipart/* application/octet-stream, application/x-www-form-urlencoded

Two types of DataWeave errors

MuleSoft

- Scripting errors
  - A problem with the syntax
- Formatting errors
  - A problem with how the transformation from one format to another is written
  - For example, a script to output XML that does not specify a data structure with a single root node
- If you get an error, transform the input to **application/dw**
  - If the transformation is successful, then the error is likely a formatting error

Including comments in DataWeave expressions

MuleSoft

- Comments that use a Java-like syntax can be used

```
// My single-line comment here

/* * My multi-line
comment here. */
```


Transforming basic data structures

Writing expressions for JSON or Java input and output

MuleSoft

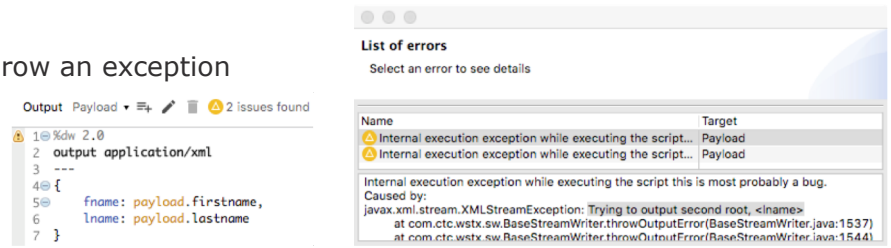
- The data model can consist of three different types of data: objects, arrays, simple literals

Input	Transform	JSON output
{ "firstname":"Max", "lastname":"Mule" }	fname: payload.firstname	{"fname": "Max"}
	{fname: payload.firstname}	{"fname": "Max“}
	user: { fname: payload.firstname, lname: payload.lastname, num: 1 }	{"user": { "fname": "Max", "lname": "Mule", "num": 1 }}
	[{fname: payload.firstname, num: 1}, {lname: payload.lastname, num: 2}]	[{"fname": "Max","num": 1}, {"lname": "Mule","num": 2}]

Writing expressions for XML output

MuleSoft

- XML can only have one top-level value and that value must be an object with one property
  - This will throw an exception

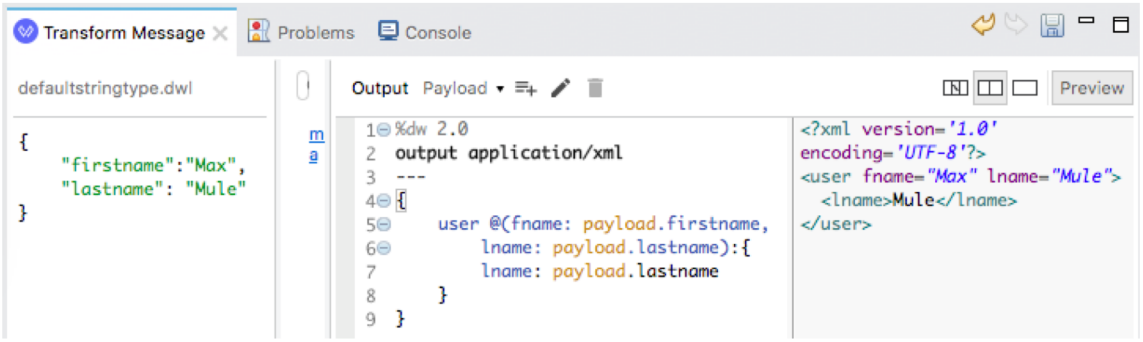


  - This will work

Specifying attributes for XML output

MuleSoft

- Use @(attName: attValue) to create an attribute



Writing expressions for XML input

MuleSoft

- By default, only XML elements and not attributes are created as JSON fields or Java object properties
- Use @ to reference attributes

Input	Transform	JSON output

<user firstname="Max">
<lastname>Mule</lastname>
</user>
	payload	{ "user": {
		"lastname": "Mule"
		}
		}

	payload.user	{"lastname": "Mule" }

	{
	fname: payload.user.@firstname,
	lname: payload.user.lastname
	}
		{"fname": "Max",
		"lname": "Mule" }

Walkthrough 11-2: Transform basic JSON, Java, and XML data structures

MuleSoft

- Write scripts to transform the JSON payload to various JSON and Java structures
- Write scripts to transform the JSON payload to various XML structures

Output Variable - DWoutput ▾ Preview

```
%dw 2.0
output application/xml
---
data: {
    hub: "MUA",
    flight @(airline: payload.airline): {
        code: payload.toAirportCode,
    }
}
```

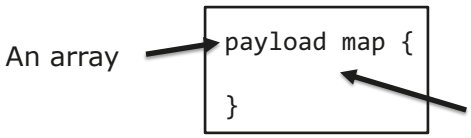
```
<?xml version='1.0' encoding='UTF-8'?>
<data>
  <hub>MUA</hub>
  <flight airline="United">
    <code>SFO</code>
  </flight>
</data>
```


Transforming complex data structures with arrays

Working with collections

MuleSoft

- Use the **map** function to apply a transformation to each element in an array
  - The input array can be JSON or Java
  - Returns an array of elements



The transformation function (or lambda) to apply to each element

A **lambda** is an anonymous function not bound to an identifier

The transformation function (or lambda)

MuleSoft

- Inside the transformation function
  - $$ refers to the index (or key)
  - $ refers to the value

Input	Transform	Output

[
{"firstname":"Max",
"lastname":"Mule"},
{"firstname":"Molly",
"lastname":"Mule"}
]
	%dw 2.0
	output application/json
	--- 	
	payload map {
	num: $$,
	fname: $.firstname,
	lname: $.lastname
	}
		[
		{"num": 0,
		"fname": "Max",
		"lname": "Mule"},
		{"num": 1,
		"fname": "Molly",
		"lname": "Mule"}
		]

	%dw 2.0
	%output application/json
	--- 	
	users: payload map {
	user: {
	fname: $.firstname,
	lname: $.lastname
	}
	}
		{
		"users": [
		{"user": {
		"fname": "Max",
		"lname": "Mule"
		}},
		{"user": {
		"fname": "Molly",
		"lname": "Mule"
		}}
		]
		}

Use explicit arguments in lambdas for more readable code

MuleSoft

```
payload map {

}
```

→

```
payload map (object, index) -> {

}
```

Input	Transform	Output

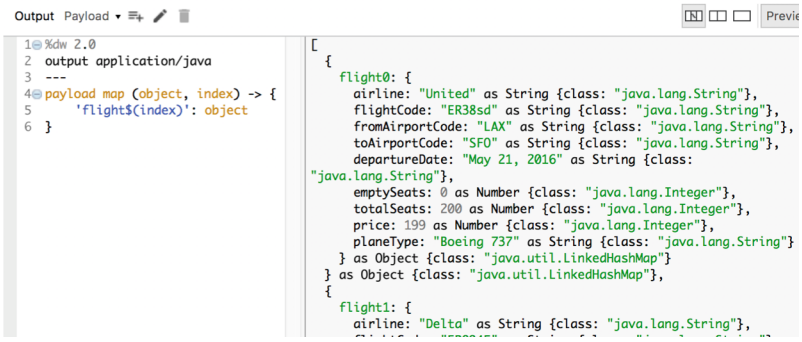
[
{"firstname":"Max",
"lastname":"Mule"},
{"firstname":"Molly",
"lastname":"Mule"}
]
	%dw 2.0
	output application/json
	--- 	
	payload **map (object, index)** -> {
	num: **index**,
	fname: **object**.firstname,
	lname: **object**.lastname
	}
		[
		{"num": 0,
		"fname": "Max",
		"lname": "Mule"},
		{"num": 1,
		"fname": "Molly",
		"lname": "Mule"}
		]

	%dw 2.0
	%output application/json
	--- 	
	users: payload map **(object, index)** -> {
	user: {
	fname: **object**.firstname,
	lname: **object**.lastname
	}
	}
		{
		"users": [
		{"user": {
		"fname": "Max",
		"lname": "Mule"
		}},
		{"user": {
		"fname": "Molly",
		"lname": "Mule"
		}}
		]
		}

Walkthrough 11-3: Transform complex data structures with arrays

MuleSoft

- Create a new flow that receives POST requests of a JSON array of flight objects
- Transform a JSON array of objects to DataWeave, JSON, and Java



Transforming complex XML data structures

Writing expressions for XML output

MuleSoft

- When mapping array elements (JSON or JAVA) to XML, wrap the map operation in {(…)}
  - {} are defining the object
  - () are transforming each element in the array as a key/value pair

Input	Transform	Output

`[ {"firstname":"Max", "lastname":"Mule"}, {"firstname":"Molly", "lastname":"Mule"} ]`	`%dw 2.0`
	`output application/xml`
	`---`
	`users: payload map (object, index) -> {`
	`  fname: object.firstname,`
	`  lname: object.lastname`
	`}`
		`Cannot coerce an array to an object`

	`users: payload map (object, index) -> {`
	`  fname: object.firstname,`
	`  lname: object.lastname`
	`})}`
		`<users>`
		`  <fname>Max</fname>`
		`  <lname>Mule</lname>`
		`  <fname>Molly</fname>`
		`  <lname>Mule</lname>`
		`</users>`

Writing expressions for XML output (cont)

MuleSoft

Input	Transform	Output
[{"firstname":"Max", "lastname":"Mule"}, {"firstname":"Molly", "lastname":"Mule"}]	users: {(payload map (object, index) -> { fname: object.firstname, lname: object.lastname })}	<users> <fname>Max</fname> <lname>Mule</lname> <fname>Molly</fname> <lname>Mule</lname> </users>

	users: **{(** payload map (object, index) -> {
	user: {
	fname: object.firstname,
	lname: object.lastname
	}
	}
	)}
		<users>
		<user>
		<fname>Max</fname>
		<lname>Mule</lname>
		</user>
		<user>
		<fname>Molly</fname>
		<lname>Mule</lname>
		</user>
		</users>

Writing expressions for XML input

MuleSoft

- Use .* selector to reference repeated elements

Input	Transform	JSON output
<users> <user firstname="Max"> <lastname>Mule</lastname> </user> <user firstname="Molly"> <lastname>Jennet</lastname> </user> </users>	payload	{"users": {"user": {"lastname": "Mule"}, "user": {"lastname": "Jennet"} } }
	payload.users	{{"user": {"lastname": "Mule"}, "user": {"lastname": "Jennet"} } }
	payload.users.user	{"lastname": "Mule" }
	payload.users.*user	[{"lastname": "Mule" }, {"lastname": "Jennet" }]

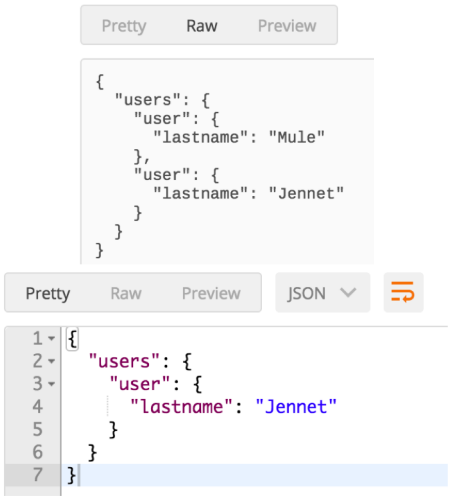
	payload.users.*user map (obj,index) -> {
	fname: obj.@firstname,
	lname: obj.lastname
	}
		[
		{"fname": "Max", "lname": "Mule" },
		{"fname": "Molly", "lname": "Jennet" }
		]

Beware of tools that "prettify" responses

MuleSoft

- In some tools (like Postman), make sure you look at the raw response and not the pretty response

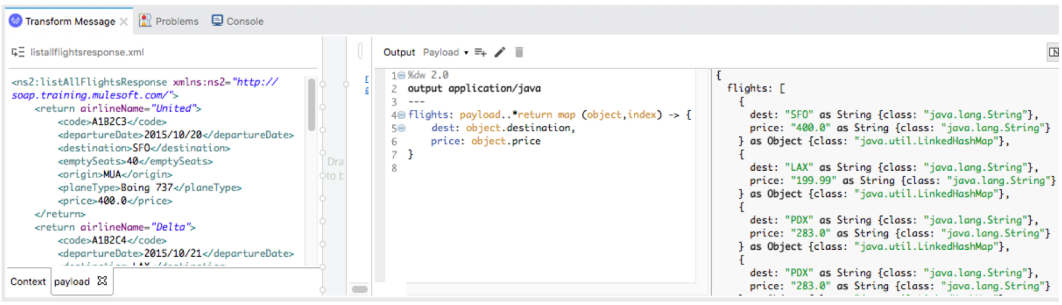
Input	Transform
<users> <user firstname="Max"> <lastname>Mule</lastname> </user> <user firstname="Molly"> <lastname>Jennet</lastname> </user> </users>	payload



Walkthrough 11-4: Transform to and from XML with repeated elements

MuleSoft

- Transform a JSON array of objects to XML
- Replace sample data associated with a transformation
- Transform XML with repeated elements to different data types



Defining and using variables and functions

Defining and using global variables

- Use the **var** directive in the header
- Assign it a constant or a lambda expression
  - DataWeave is a functional programming language where variables behave just like functions
- Global variables can be referenced anywhere in the body

Input	Transform	Output

`{"firstname":"Max", "lastname":"Mule"}`	`%dw 2.0`
	`output application/xml`
	`var mname = "the"`
	`var mname2 = () -> "other"`
	`var lname = (aString) -> upper(aString)`
	`---`
	`name: {`
	`  first: payload.firstname,`
	`  middle1: mname, middle2: mname2(),`
	`  last: lname(payload.lastname)`
	`}`
		`<name>`
		`  <first>Max</first>`
		`  <middle1>the</middle1>`
		`  <middle2>other</middle2>`
		`  <last>MULE</last>`
		`</name>`

Defining and using variables in a syntax similar to traditional functions

MuleSoft

- DataWeave includes an alternate syntax to access lambda expressions assigned to a variable as functions
  - May be more clear or easier to read for some people

Input	Transform	Output

{"firstname":"Max", "lastname":"Mule"}	%dw 2.0
	output application/xml
	var lname = (aString) -> upper(aString)
	--- 	
	name: {
	first: payload.firstname,
	last: lname(payload.lastname)
	}
		<name>
		<first>Max</first>
		<last>MULE</last>
		</name>

	%dw 2.0
	output application/xml
	fun lname(aString) = upper(aString)
	--- 	
	name: {
	first: payload.firstname,
	last: lname(payload.lastname)
	}	

Defining and using local variables

MuleSoft

- Use the **using** keyword in the body with the syntax

  `using (<variable-name> = <expression>)`

- Local variables can only be referenced from within the scope of the expression where they are initialized

Input	Transform	Output
{"firstname":"Max", "lastname":"Mule"}	using (name = payload.firstname ++ " " ++ payload.lastname) name	"Max Mule"

	using (fname = payload.firstname, lname = payload.lastname) {
	person: using (user = fname, color = "gray") {
	name1: user,
	color: color
	},
	name2: lname
	}
		{
		"person": {
		"name1": "Max",
		"color": "gray"
		},
		"name2": "Mule"
		}

	using (fname = payload.firstname, lname = payload.lastname) {
	person: using (user = fname, color = "gray") {
	name1: user,
	color: color
	},
	name2: lname,
	color: color
	}
		Unable to resolve reference of color

Walkthrough 11-5: Define and use variables and functions

MuleSoft

- Define and use a global constant
- Define and use a global variable that is equal to a lambda expression
- Define and use a lambda expression assigned to a variable as a function
- Define and use a local variable

Formatting and coercing data

Using the as operator for type coercion

```
price: payload.price as Number
price: $.price as Number {class:"java.lang.Double"}
```

- Defined types include
  - Any (the top-level type)
  - Null, String, Number, Boolean
  - Object, Array, Range
  - Date, Time, LocalTime, DateTime, LocalDateTime, TimeZone Period
  - More…

Using format patterns

MuleSoft

- Use metadata **format** schema property to format numbers and dates

```
price as Number as String {format: "###.00"},
someDate as DateTime {format: "yyyyMMddHHmm"}
```

- For formatting patterns, see
  - https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html
  - https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

Walkthrough 11-6: Coerce and format strings, numbers, and dates

MuleSoft

- Coerce data types
- Format strings, numbers, and dates

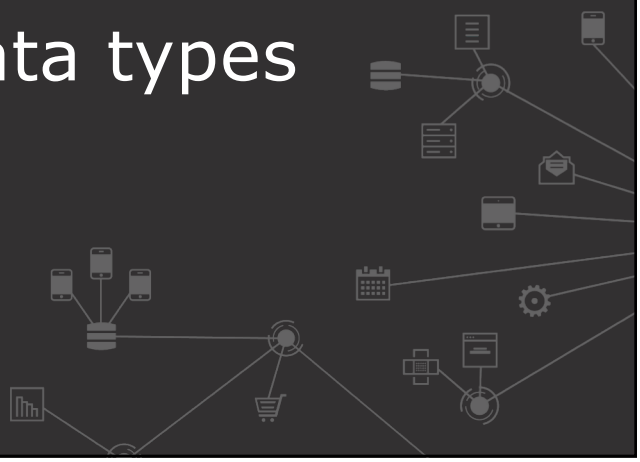
Output Payload ▾ Preview

```
using (flights =
    payload..*return map (object,index) -> {
        dest: object.destination,
        price: object.price as Number as String {format: "###.00"},
        totalSeats: getNumSeats(object.planeType as String),
        plane: upper(object.planeType as String),
        date: object.departureDate as Date {format: "yyyy/MM/dd"}
            as String {format: "MMM dd, yyyy"}
    }
)

flights
```

```
  dest: "SFO",
  price: "400.00" as String {format: "###.00"},
  totalSeats: 150,
  plane: "BOING 737",
  date: "Oct 20, 2015" as String {format: "MMM dd, yyyy"}
},
{
  dest: "LAX",
  price: "199.99" as String {format: "###.00"},
  totalSeats: 150,
  plane: "BOING 737",
  date: "Oct 21, 2015" as String {format: "MMM dd, yyyy"}
```


Using custom data types



Defining and using custom data types

MuleSoft

- Use **type** header directive
  - Name has to be all lowercase letters
    - No special characters, uppercase letters, or numbers

```
%dw 2.0
output application/json
type ourdateformat = DateTime {format: "yyyyMMddHHmm"}
---
someDate: payload.departureDate as ourdateformat
```


Transforming objects to POJOs

MuleSoft

- Specify inline

```
customer:payload.user as Object {class: "my.company.User"}
```

- Or define a custom data type to use

```
type user = Object {class: "my.company.User"}
---
customer:payload.user as user
```

Walkthrough 11-7: Define and use custom data types

MuleSoft

- Define and use custom data types
- Transform objects to POJOs

Using DataWeave functions

Review: Using DataWeave functions

- Functions are packaged in modules
- Functions in the Core module are imported automatically into DataWeave scripts
- For function reference, see
  - https://docs.mulesoft.com/mule4-user-guide/v/4.1/dataweave
- For functions with 2 parameters, an alternate syntax can be used

```
#[contains(payload,max)]
#[payload contains "max "]
```

Calling functions that have a lambda expression as a parameter

- The alternate notation can make calling functions that have a lambda expression as a parameter easier to read

```
sizeOf(filter(payload, (value) -> value.age > 30)))
sizeOf(payload filter $.age > 30)
```

- When using a series of functions, the last function in the chain is executed first: filter then orderBy

```
flights orderBy $.price filter ($.availableSeats > 30)
```

- If `flights` is actually an expression using the map function, you need to force it to be evaluated first by defining it as a local variable
  - The Using statement is at the top of the precedence order, before binary functions

Note: For precedence, see
https://docs.mulesoft.com/mule4-user-guide/v/4.1/dataweave-flow-control-precedence

Using DataWeave functions not in the Core module

MuleSoft

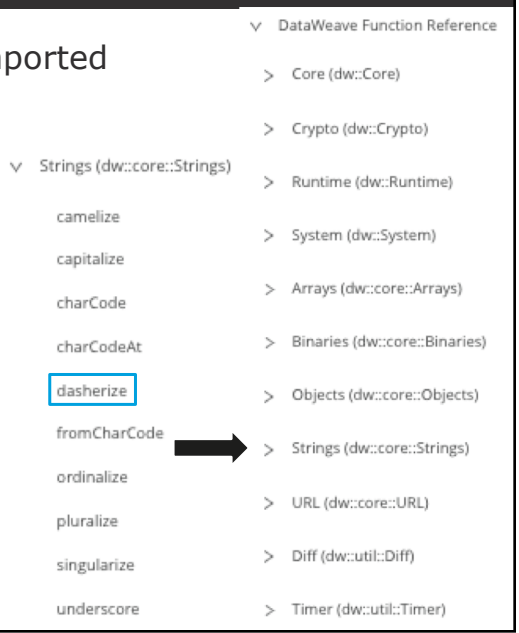
- Functions in all other modules must be imported
- Import a function in a module

```
%dw 2.0
output application/xml
import dasherize from dw::core::Strings
---
n: upper(dasherize(payload.firstname ++ payload.lastname
```

- Import a module

```
%dw 2.0
output application/xml
import dw::core::Strings
---
n: upper(Strings::dasherize(payload.firstname ++
payload.lastname))
```

MAX-MULE



Walkthrough 11-8: Use DataWeave functions

MuleSoft

- Use functions in the Core module that are imported automatically
- Replace data values using pattern matching
- Order data, remove duplicate data, and filter data
- Use a function in another module that you must explicitly import into a script to use
- Dasherize data

```
flights distinctBy $
        filter ($.availableSeats !=0)
        orderBy $.departureDate
        orderBy $.price
```

Looking up data by calling a flow

Calling flows from a DataWeave expression

MuleSoft

- Use the **lookup** function

```
{a: lookup("myFlow",{b:"Hello"}) }
```

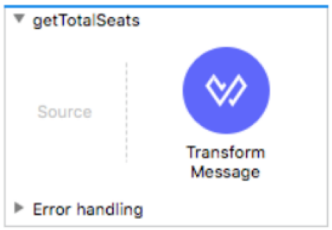
  - The first argument is the name of the flow to be called
    - Cannot call subflows
  - The second argument is the payload to send to the flow as a map
  - Whatever payload the flow returns is what the expression returns

Walkthrough 11-9: Look up data by calling a flow

MuleSoft

- Call a flow from a DataWeave expression

```
totalSeats: lookup("getTotalSeats",{planeType: object.planeType}),
```



Summary

Summary

MuleSoft

- DataWeave code can be inline, in a DWL file, or in a module of functions
- The **data model** for a transformation can consist of three different types of data: objects, arrays, and simple literals
- **Many formats** can be used as input and output including JSON, Java, XML, CSV, Excel, Flat File, Cobol Copybook, Fixed Width, and more
- The DataWeave **application/dw** format can be used to test expressions to ensure there are no scripting errors
- Use the **map** function to apply a transformation function (a lambda) to each item in an array
- A **lambda** is an anonymous function not bound to an identifier
- When mapping array elements (JSON or JAVA) to XML, wrap the map operation in {(…)}

Summary

MuleSoft

- DataWeave is a functional programming language where variables behave just like functions
- Define global variables in the header with **var**
  - Assign them a constant or a lambda expression
  - Use **fun** directive to access lambdas assigned to variables as traditional functions
- Define local variables in the body with **using()**
  - The Using statement is at the top of the precedence order, before binary functions
- Functions are packaged in modules
  - Functions in the Core module are imported automatically into DataWeave scripts
  - Use the **import** header directive to import functions in all other modules
- Functions with 2 parameters can be called with 2 different syntax

Summary

MuleSoft

- Use the metadata **format** schema property to format #'s and dates
- Use the **type** header directive to specify custom data types
- Transform objects to POJOs using: as Object {class: "com.myPOJO"}
- Use **lookup()** to get data by calling other flows