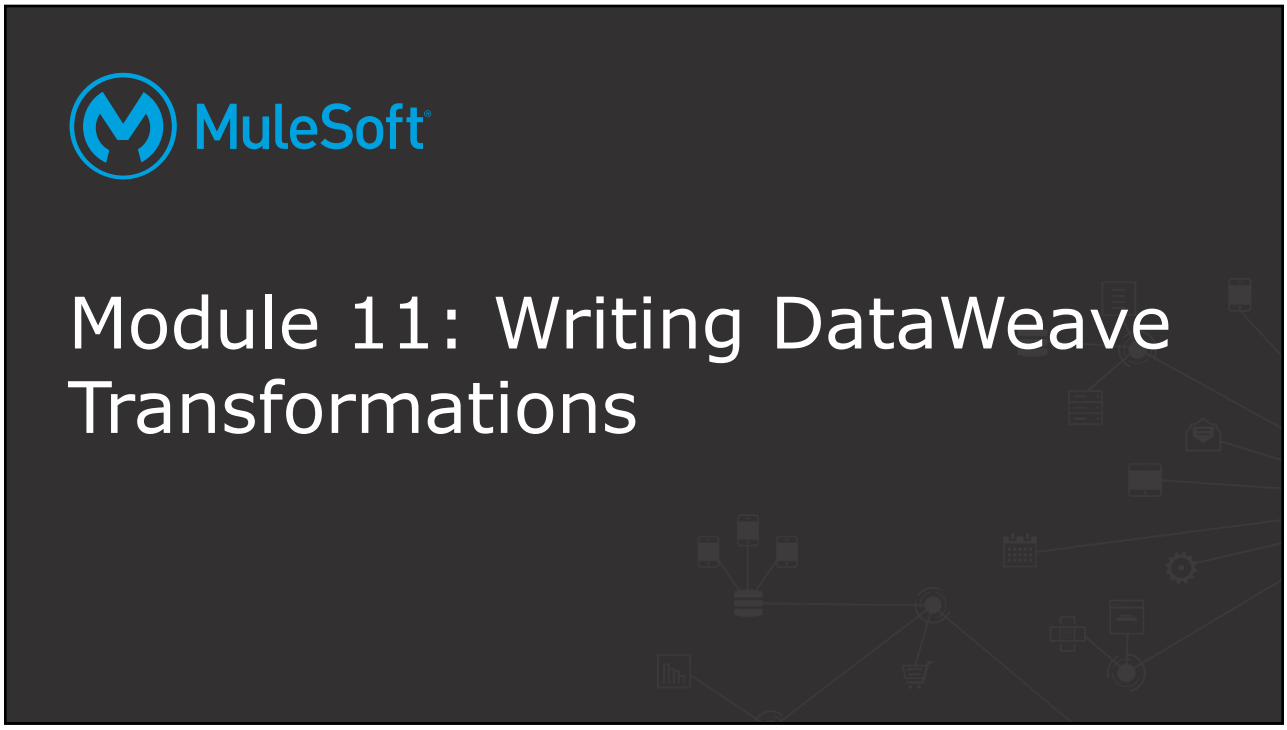


Module 11: Writing DataWeave Transformations

Goal

MuleSoft

- You have been using DataWeave to transform data throughout class – but (mostly) only using the graphical drag-and-drop editor

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

Goal

MuleSoft

Output Payload ▾ Preview

```
%dw 1.0
%output application/java
%namespace ns0 http://soap.training.mulesoft.com/
%type currency = :string {format: "###.00"}
%type flight = :object {class: "com.mulesoft.training.Flight"}
---
flights: payload.ns0#listAllFlightsResponse.*return map {
    destination: $.destination,
    price: $.price as :number as :currency,
    planeType: upper ($.planeType replace /(Boing)/ with "Boeing"),
    departureDate: $.departureDate as :date {format: "yyyy/MM/dd"}
    as :string {format: "MMM dd, yyyy"},
    availableSeats: $.emptySeats as :number,
    //totalSeats: getNumSeats($.planeType)
    totalSeats: lookup("getTotalSeatsFlow",{type: $.planeType})
}
```

Name	Value
root : *LinkedHashMap*	
flights : *ArrayList*	
[0] : *LinkedHashMap*	
destination : *String*	SFO
price : *String*	400.00
planeType : *String*	BOEING 737
departureDate : *String*	Oct 20, 2015
availableSeats : *Integer*	40
Unknown	
[1] : *LinkedHashMap*	
destination : *String*	LAX
price : *String*	199.99
planeType : *String*	BOEING 737

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

MuleSoft

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

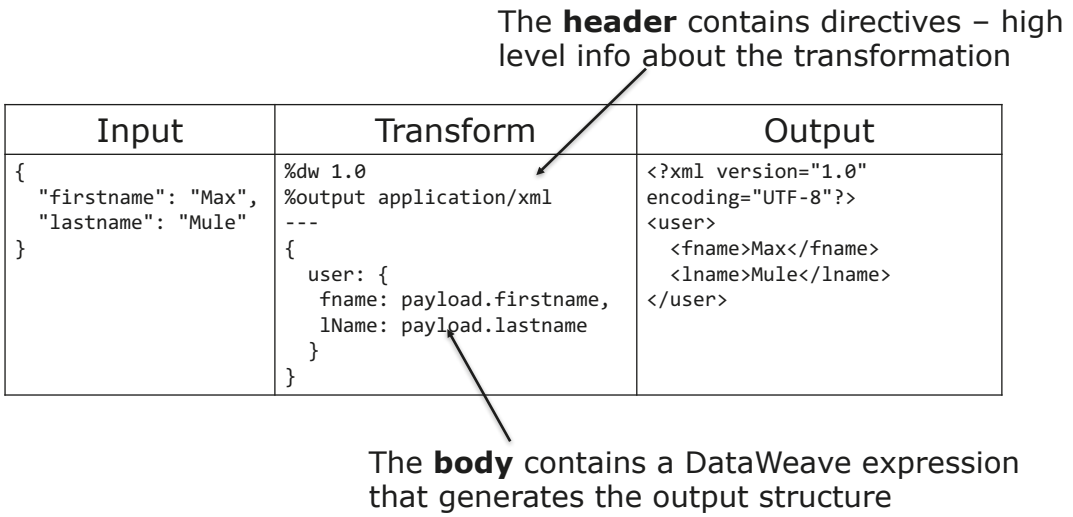
Introducing the DataWeave data transformation language

DataWeave data transformation language

MuleSoft

- A universal, simple, JSON-like language for transforming and querying data
- Easy to write, easy to maintain, and capable of supporting simple to complex mappings for any data type
  - Supports XML, JSON, Java, CSV, EDI, fixed width, flat file, and COBOL copybook out-of-the-box
- More elegant and re-usable than custom code
  - Data transformations can be stored in external DWL files and used across applications

Example DataWeave transformation expression



DataWeave 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**

The output directive

MuleSoft

- Sets the output type of the transformation
- Specified using content/type
  - application/json, text/json
  - application/xml, text/xml
  - application/java, text/java
  - application/csv, text/csv
  - application/dw

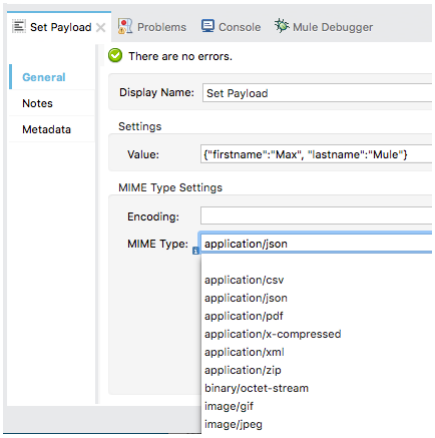
```
%dw 1.0
%output application/xml
---
{
  a: payload
}
```

- The structure of the output is defined in the DataWeave body

Setting input data MIME types

MuleSoft

- When you run your application, you may get an error unless the MIME type for the input data has been set
  - It may be set automatically if the data is posted to the flow
    - See inbound properties content-type
  - Or you can define the input metadata type as you have been doing throughout class
  - Or, you can set the mimeType property of a preceding message processor



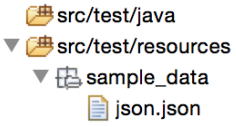
Where is the code?

MuleSoft

- By default, it is placed inline

```
<flow name="json2xml">
    <http:listener config-ref="HTTP_Listener_Configurati
    <set-payload value="{&quot;firstname&quot;:&quot;Max
    <dw:transform-message doc:name="Transform Message">
        <dw:input-payload doc:sample="json.json"/>
        <dw:set-payload><![CDATA[%dw 1.0
%output application/xml
---
{
    user: {
        lName: payload.lastname,
        fname: payload.firstname
    }
}]]></dw:set-payload>
    </dw:transform-message>
    <logger level="INFO" doc:name="Logger"/>
</flow>
```

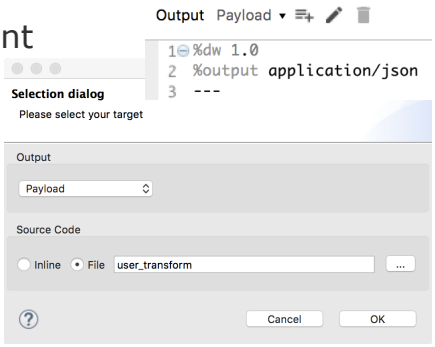
- Any added sample data is stored in src/test/resources



Reusing transformations

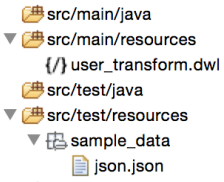
MuleSoft

- To place it an external file, 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



- To reference an existing DWL, specify it in code

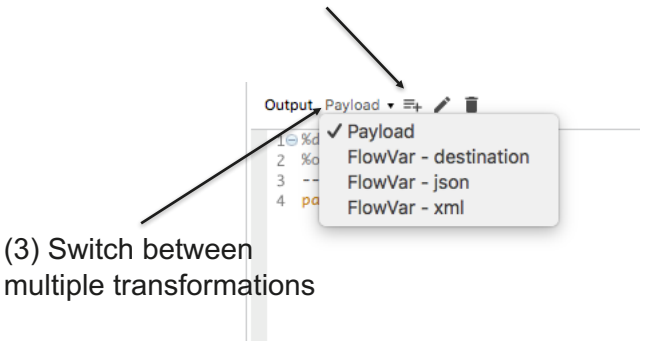
```
<flow name="json2xml">
    <http:listener config-ref="HTTP_Listener_Configuration" path="/json" allowedMe
    <set-payload value="{&quot;firstname&quot;:&quot;Max&quot;, &quot;lastname&quo
    <dw:transform-message doc:name="Transform Message">
        <dw:input-payload doc:sample="json.json"/>
        <dw:set-payload resource="classpath:user_transform.dwl"></dw:set-payload>
    </dw:transform-message>
    <logger level="INFO" doc:name="Logger"/>
</flow>
```



Creating multiple transformations

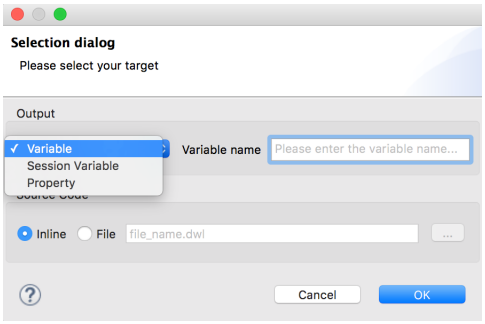
- You can also create multiple transformations with one Transform Message component

(1) Use Add new target button



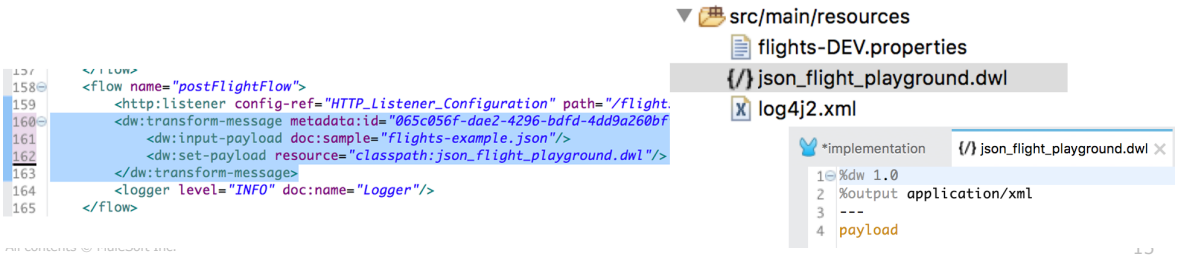
(3) Switch between multiple transformations

(2) Set where to store results: Variable/property type and name



Walkthrough 11-1: Write your first DataWeave transformation

- Create a new flow that receives POST requests
- Write a DataWeave expression to transform the data to Java, XML, or JSON
- Add sample data and use live preview
- Save the transformation in an external file



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

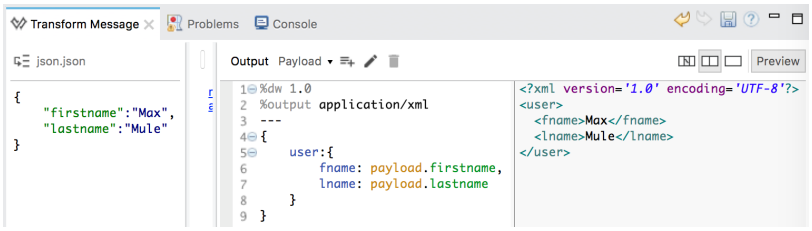
```
%dw 1.0
%output application/xml
---
{
    fname: payload.firstname,
    lname: payload.lastname
}
```

List of errors

Select an error to see details below

Name	Target
javax.xml.stream.XMLStreamException: Trying to output second root, <lname>	Payload

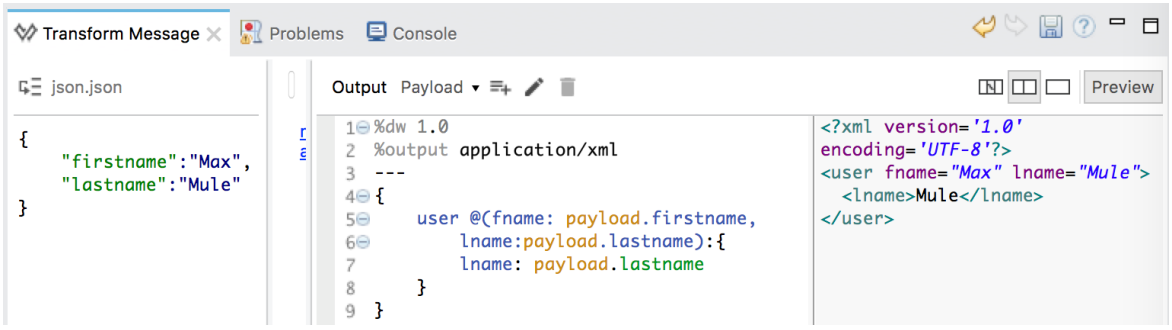
  - This will work



Specifying attributes for XML output

MuleSoft

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



Writing expressions for XML input

- 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 Java, JSON, and XML data structures

- Write expressions to transform the JSON payload to various Java structures
- Create a second transformation to store a transformation output in a flow variable
- Write expressions to transform the JSON payload to various XML structures

Output FlowVar - xml ▾ Preview

```
%dw 1.0
%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** operator to apply a transformation to each element in a collection
  - A collection can be JSON or Java arrays or XML repeated elements

The collection → `payload map {` `}` ← The transformation function (or lambda) to apply to each element

- The map operator
  - Returns an array of elements
  - Can be applied to each element in an array or each value in an object

The transformation function

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 1.0
	%output application/json
	--- 	
	payload map {
	num: $$,
	fname: $.firstname,
	lname: $.lastname
	}
		[
		{"num": 0,
		"fname": "Max",
		"lname": "Mule"},
		{"num": 1,
		"fname": "Molly",
		"lname": "Mule"}
		]

	%dw 1.0
	%output application/json
	--- 	
	users: payload map {
	user: {
	fname: $.firstname,
	lname: $.lastname
	}
	}
		{
		"users": [
		{"user": {
		"fname": "Max",
		"lname": "Mule"
		}},
		{"user": {
		"fname": "Molly",
		"lname": "Mule"
		}}
		]
		}

Using the index as a key in a transformation function

MuleSoft

- To set the index as a new key, surround it with () or ''

Input	Transform	Output

[
{"firstname":"Max",
"lastname":"Mule"},
{"firstname":"Molly",
"lastname":"Mule"}
]
	%dw 1.0
	%output application/json
	--- 	
	payload map {
	num: $$,
	($$): $
	}
		[
		{
		"num": 0,
		"0": {
		"firstname": "Max",
		"lastname": "Mule"
		}
		},
		{
		"num": 1,
		"1": {
		"firstname": "Molly",
		"lastname": "Mule"
		}
		}
		]

	payload map {
	num: $$,
	'$$': $
	}	

	payload map {
	'num$$': $
	}
		[
		{
		"num0": {
		"firstname": "Max",
		"lastname": "Mule"
		}
		},
		{
		"num1": {
		"firstname": "Molly",
		"lastname": "Mule"
		}
		}
		]

Walkthrough 11-3: Transform complex data structures

MuleSoft

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

Output Payload ▾ Preview

```
%dw 1.0
%output application/java
---
payload map {
    flight: $$,
    'flight$$': $
}
```

Name	Value
root : *ArrayList*	
[0] : *LinkedHashMap*	
flight : *Integer*	0
flight0 : *LinkedHashMap*	
airline : *String*	United
flightCode : *String*	ER38sd
fromAirportCode : *String*	LAX
toAirportCode : *String*	SFO
departureDate : *String*	May 21, 2016
emptySeats : *Integer*	0
totalSeats : *Integer*	200
price : *Integer*	199
planeType : *String*	Boeing 737
[1] : *LinkedHashMap*	
flight : *Integer*	1

26

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 1.0
	%output application/xml
	--- 	
	users: payload map {
	fname: $.firstname,
	lname: $.lastname
	}
		Cannot coerce an array to an object

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

	users: **{(** payload map {
	user: {
	fname: $.firstname,
	lname: $.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 * 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" }]

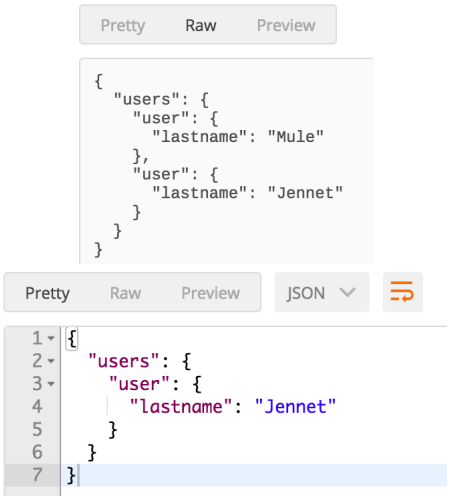
	payload.users.*user map {
	fname: $.@firstname,
	lname: $.lastname
	}
		[
		{"fname": "Max", "lname": "Mule" },
		{"fname": "Molly", "lname": "Jennet" }
		]

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

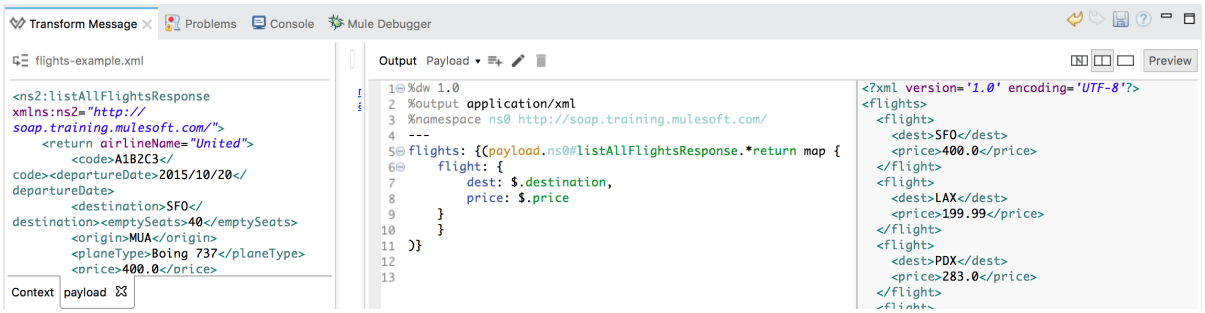
Input	Transform
<users> <user firstname="Max"> <lastname>Mule</lastname> </user> <user firstname="Molly"> <lastname>Jennet</lastname> </user> </users>	payload



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

MuleSoft

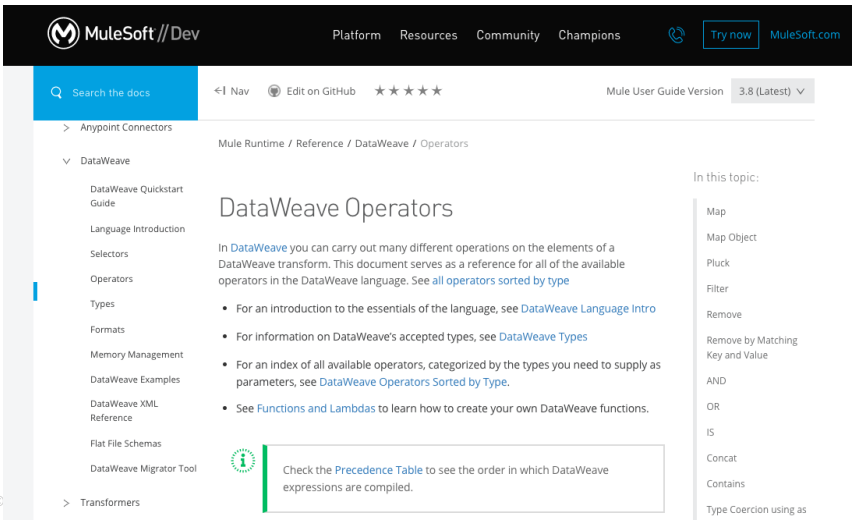
- Transform a JSON array of objects to XML
- Transform XML with repeated elements to Java



Formatting and coercing data

DataWeave documentation

- https://docs.mulesoft.com/mule-user-guide/v/3.8/dataweave



34

Formatting operators

Input	Transform	Output

	`%dw 1.0`
	`%output application/xml`
	`---`	

`{"name": "max_mule"}`	`n: upper` `(dasherize payload.name)`	`<n>MAX-MULE</n>`

- Upper
- Lower
- Camelize
- Capitalize
- Dasherize
- Underscore
- Pluralize
- Singularize
- Trim
- Ordinalize

35

Using the as operator for type coercion

MuleSoft

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

- Defined types include
  - :string
  - :number
  - :boolean
  - :object
  - :array
  - :date, :time, :timezone, :datetime, :localdatetime, :period
  - :regex, more…
- See what conversions between what types are allowed in DataWeave
  - https://docs.mulesoft.com/mule-user-guide/v/3.8/dataweave-types#type-coercion-table

Using format patterns

MuleSoft

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

```
someDate as :datetime {format: "yyyyMMddHHmm"}
```

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

Walkthrough 11-5: Coerce and format strings, numbers, and dates

MuleSoft

- Explore the DataWeave documentation
- Coerce data types
- Format strings, numbers, and dates

Output Payload ▾ Preview

```
1  %dw 1.0
2  %output application/java
3  %namespace ns0 http://soap.training.mulesoft.com/
4  ---
5  flights: payload.ns0#listAllFlightsResponse.*return map {
6       dest: $.destination,
7       price: $.price as :number as :string {format: "###.00"},
8       plane: upper $.planeType,
9       date: $.departureDate as :date {format: "yyyy/MM/dd"}
10      as :string {format: "MMM dd, yyyy"}
11 }
12
13
14
```

Name	Value
root : *LinkedHashMap*	
flights : *ArrayList*	
[0] : *LinkedHashMap*	
dest : *String*	SFO
price : *String*	400.00
plane : *String*	BOING 737
date : *String*	Oct 20, 2015
[1] : *LinkedHashMap*	
dest : *String*	LAX
price : *String*	199.99
plane : *String*	BOING 737
date : *String*	Oct 21, 2015

Using DataWeave operators

Math operators

- +
- -
- *
- /

Basic Math Operations

Max

Min

Round

Sqrt

Pow

Ceil

Floor

Abs

Mod

Conditional logic operators

- default
- when
- unless
- otherwise

- In expressions, you can use
  - ==
  - !=
  - ~= (equal regardless of type)

Input	Transform	Output

	`%dw 1.0`
	`%output application/xml`
	`---`	

`{"firstname":"Max", "lastname":"Mule"}`	`n: payload.nickname default payload.firstname`	`<n>Max</n>`
`{"firstname":"Max", "lastname":"Mule", "nickname":""}`	`n: payload.nickname` `when payload.nickname != "" otherwise payload.firstname`	`<n>Max</n>`

Additional operators

- flatten
- orderBy
- concat
- distinctBy
- groupBy
- replace
- matches
- regex
- More…

Join By
Split By
Order By
Group By
Distinct By
Zip Arrays
Unzip Array
Replace
Matches
Starts With
Ends With
Find
Match
Scan
Similar

Walkthrough 11-6: Use DataWeave operators

MuleSoft

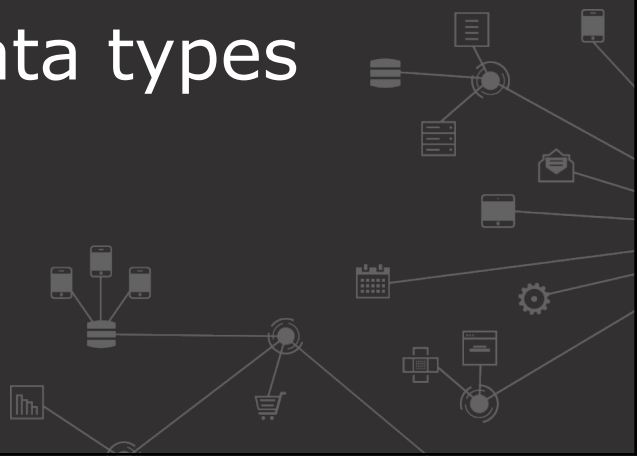
- Replace data values using pattern matching
- Order data, remove duplicate data, and filter data

Output Payload ▾ Preview

```
1  %dw 1.0
2  %output application/java
3  %namespace ns0 http://soap.training.mulesoft.com/
4  ---
5  flights: payload.ns0#listAllFlightsResponse.*return map {
6      dest: $.destination,
7      price: $.price as :number as :string {format: "###.00"},
8      plane: upper ($.planeType replace /(Boing)/ with "Boeing"),
9      date: $.departureDate as :date {format: "yyyy/MM/dd"}
10     as :string {format: "MMM dd, yyyy"},
11     seats: $.emptySeats as :number
12 } orderBy $.date orderBy $.price distinctBy $ filter ($.seats !=0)
13
14
```

Name	Value
root : *LinkedHashMap*	
flights : *ArrayList*	
[0] : *LinkedHashMap*	
[1] : *LinkedHashMap*	
[2] : *LinkedHashMap*	
[3] : *LinkedHashMap*	
dest : *String*	SFO
price : *String*	400.00
plane : *String*	BOEING 737
date : *String*	Oct 20, 2015
seats : *Integer*	40

Using custom data types



Specifying custom data types

MuleSoft

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

```
%dw 1.0
%output application/json
%type ourdateformat = :datetime {format: "yyyyMMddHHmm"}
---
someDate: payload.departureDate as :ourdateformat
```

Transforming objects to POJOs

MuleSoft

- Use as :object
- 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

Output Payload ▾ Preview

```
1  %dw 1.0
2  %output application/java
3  %namespace ns0 http://soap.training.mulesoft.com/
4  %type currency = :string {format: "###.00"}
5  %type flight = :object {class: "com.mulesoft.training.Flight"}
6  ---
7  flights: payload.ns0#listAllFlightsResponse.*return map {
8      destination: $.destination,
9      price: $.price as :number as :currency,
10     planeType: upper ($.planeType replace /(Boing)/ with "Boeing"),
11     departureDate: $.departureDate as :date {format: "yyyy/MM/dd"}
12     as :string {format: "MMM dd, yyyy"},
13     availableSeats: $.emptySeats as :number
14 } as :flight orderBy $.departureDate orderBy $.price distinctBy $
15 filter ($.availableSeats !=0)
```

Name	Value
root : *LinkedHashMap*	
flights : *ArrayList*	
[0] : *Flight*	
flightCode : *String*	
availableSeats : *Integer*	10
origination : *String*	
price : *Double*	199.99
destination : *String*	LAX
departureDate : *String*	Oct 21, 2015
airlineName : *String*	
planeType : *String*	BOEING 737
[1] : *Flight*	
flightCode : *String*	

Referencing other data besides the payload

Referencing other data besides the payload

- Up to now, we have only referenced the message payload in the DataWeave expression
- You can also
  - Reference message variables and properties
  - Call global MEL functions
  - Call other Mule flows

Referencing message variables and properties

MuleSoft

- In addition to payload, you can also reference
  - flowVars
  - inboundProperties
  - outboundProperties
  - p (System or Spring properties)

```
{n: flowVars.username}
```

- This is not MEL!
  - Do not preface these values by "message." or use #[]

Calling global MEL functions

MuleSoft

- Define global MEL functions in the configuration element you used to specify a default global exception handler

```
<configuration doc:name="Configuration" defaultExceptionStrategy-ref="..">
  <expression-language>
       <global-functions>
            def newUser() { return ["name" : "max"] }
            def upperName(user) {
                 return user.name.toUpperCase()
            }
       </global-functions>
  </expression-language>
</configuration>
```

- Call the MEL functions from DataWeave

```
{"foo" : newUser(),
 "bar": upperName(newUser())}
```

Calling external flows

- From a DataWeave transform, you can call a different flow in your Mule application
  - Whatever the flow returns is what the expression returns

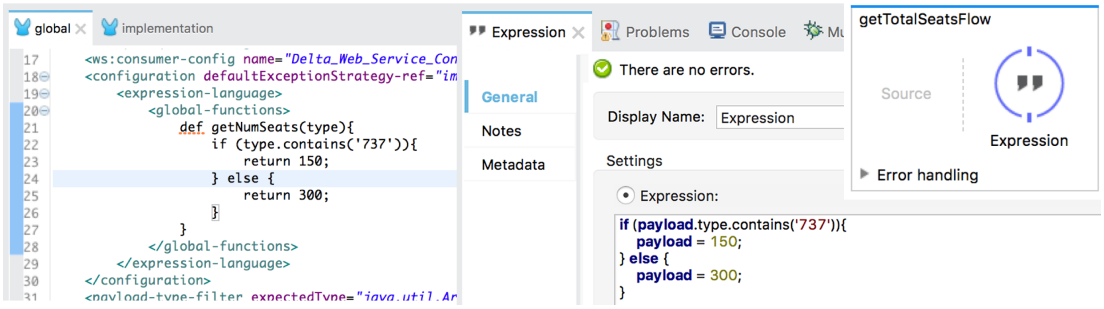
```
{a: lookup("mySecondFlow",{b:"Hello"}) }
```

  - The first argument is the name of the flow to be called
  - The second argument is the payload to send to the flow as a map

Walkthrough 11-8: Call MEL functions and other flows

- Define a global MEL function in global.xml
- Call a global MEL function in a DataWeave expression
- Call a flow in a DataWeave expression

```
//totalSeats: getNumSeats($.planeType)
totalSeats: lookup("getTotalSeatsFlow",{type: $.planeType})
```



Summary

Summary

MuleSoft

- DataWeave is a JSON-like language
- The data model for the transformation can consist of three different types of data: objects, arrays, and simple literals
- DataWeave supports XML, JSON, Java, CSV, EDI, fixed width, flat file, and COBOL copybook out-of-the-box
- Transformations can be written inline or stored in external DWL files
- There is a large set of operators that can be used in the expressions
  - upper, replace, as, orderBy, distinctBy, filter, flatten

Summary

MuleSoft

- Use the metadata format schema property to format numbers and dates
- Use the type header directive to specify custom data types
- Transform objects to POJOs using as:object {class: "com.myPOJO"}
- Transformations can reference the message payload, flow variables, inbound properties, and outbound properties
- Transformations can call global MEL functions
- Use lookup() to get data by calling other flows