

hello



Module 11: Writing DataWeave Transformations



Goal

- 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

Output Payload ▾  

```

1@%dw 2.0
2 output application/dw
3 import dasherize from dw::core::Strings
4 type Currency = String {format: "##.##"}
5 type Flight = Object {class: "com.mulesoft.training.Flight"}
6
7 //var numSeats = 400
8 //var numSeats = (x<400) -> x
9@/* var numSeats = (planeType: String) ->
10     if (planeType contains('737'))
11         150
12     else
13         300
14 */
15@ fun getNumSeats(planeType: String) =
16@     if (planeType contains('737'))
17         150
18     else
19         300
20 ---
21@ using (flights =
22@     payload..*return map (object,index) -> {
23@         destination: object.destination,
24@         price: object.price as Number as Currency,
25@         totalSeats: getNumSeats(object.planeType as String),
26@         |totalSeats: lookup("getTotalSeats",{planeType: object.planeType}),
27@         planeType: dasherize(replace(object.planeType,/(Boing)/) with "Boeing"),
28@         departureDate: object.departureDate as Date {format: "yyyy/MM/dd"}
29@             as String {format: "MMM dd, yyyy"},
30@         availableSeats: object.emptySeats as Number
31@     } as Object
32 )
33
34@ flights distinctBy $
35     filter ($.availableSeats !=0)
36     orderBy $.departureDate
37     orderBy $.price

```

[

```

{
    destination: "LAX",
    price: "199.99" as Currency {format: "##.##"},
    totalSeats: 150,
    planeType: "boeing-737",
    departureDate: "Oct 21, 2015" as String {format: "MMM dd, yyyy"},
    availableSeats: 10
},
{
    destination: "PDX",
    price: "283.00" as Currency {format: "##.##"},
    totalSeats: 300,
    planeType: "boeing-777",
    departureDate: "Oct 20, 2015" as String {format: "MMM dd, yyyy"},
    availableSeats: 23
},
{
    destination: "PDX",
    price: "283.00" as Currency {format: "##.##"},
    totalSeats: 300,
    planeType: "boeing-777",
    departureDate: "Oct 21, 2015" as String {format: "MMM dd, yyyy"},
    availableSeats: 30
},
{
    destination: "SFO",
    price: "400.00" as Currency {format: "##.##"},
    totalSeats: 150,
    planeType: "boeing-737",
    departureDate: "Oct 20, 2015" as String {format: "MMM dd, yyyy"},
    availableSeats: 40
}
]
```

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



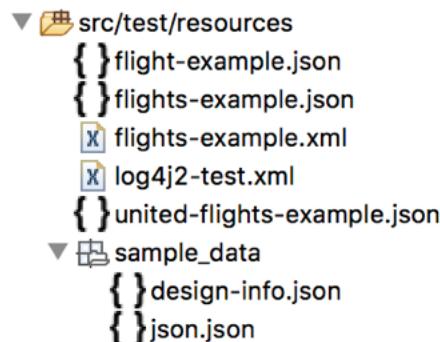
- To now, you have used the Transform Message component to
 - Create transformations using the visual editor
 - Define metadata for the input and output payload
 - Write basic transformation expressions
- But...
 - Where does the code go?
 - Can you save the code externally and reuse it?
 - What happens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

Where is the DataWeave code?

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

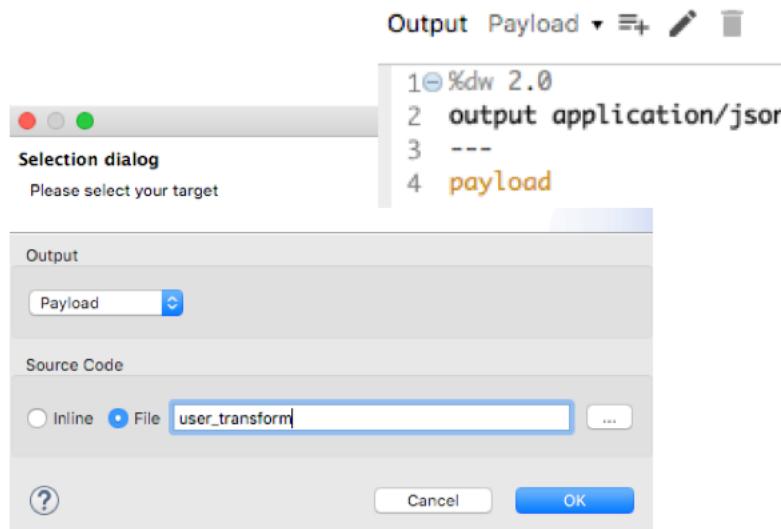
```
<flow doc:id="a2d732ff-aaec-448e-9bdd-a25319cc55a2" name="postFlight">
    <ee:transform doc:name="Transform Message" doc:id="ccda7d44-19b9"
        <ee:message>
            <ee:set-payload><![CDATA[%dw 2.0
output application/java
--->
payload]]></ee:set-payload>
        </ee:message>
        <ee:variables>
            <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
                </ee:transform>
            </ee:variables>
        </ee:variables>
    </ee:transform>
</flow>
```

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

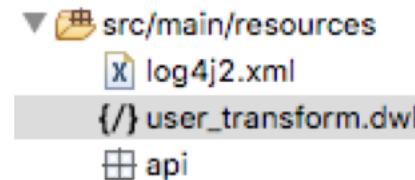


Exporting DataWeave code to a DWL file

- 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="69db8e">
    <ee:message>
        <ee:set-payload resource="user_transform.dwl" />
    </ee:message>
</ee:transform>
```



Reusing DataWeave code

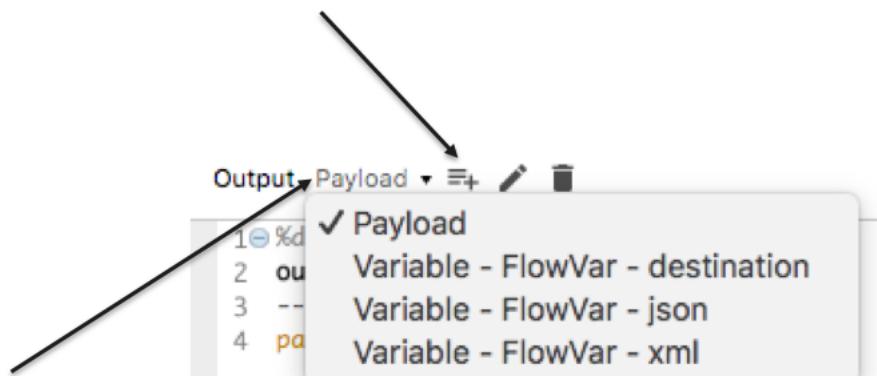
- A single script can be stored in an **external DWL file** and referenced
 - In the Transform Message component using the resource attribute
 - In an element that has an expression property (like the Choice router) using the \${file::filename} syntax
- You can also create **modules** (libraries) of reusable DataWeave functions

```
<ee:transform doc:name="Transform Message" doc:id="69db8e0f-0000-40d0-8000-000000000000">
    <ee:message>
        <ee:set-payload resource="user_transform.dwl" />
    </ee:message>
</ee:transform>
```

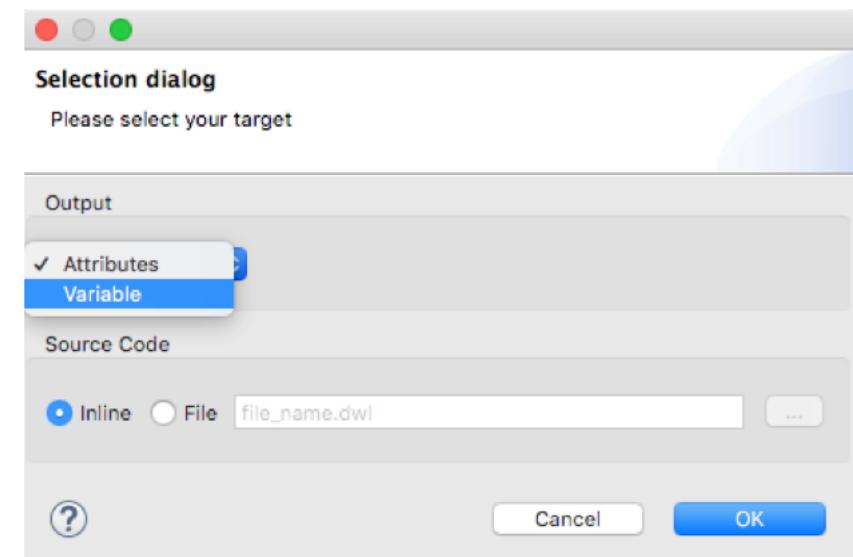
Creating multiple transformations

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

(1) Use Add new target button



(2) Set where to store result



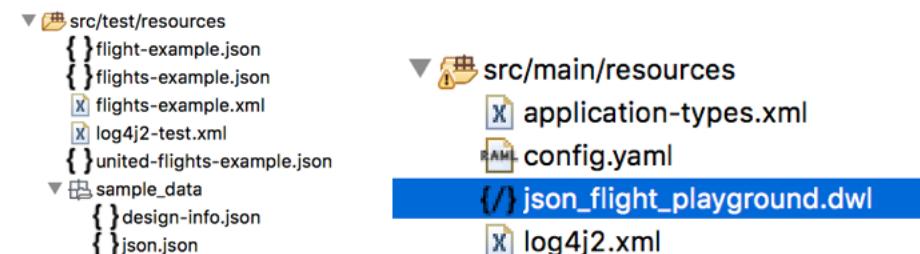
(3) Switch between multiple transformations

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

- 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

Output Variable - DWoutput ▾ + 🖊️ 🗑️ ⚠️ 2 issues found

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



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

Input	Transform	Output
{ "firstname": "Max", "lastname": "Mule" }	%dw 2.0 output application/xml --- { user: { fname: payload.firstname, lName: payload.lastname } }	<?xml version="1.0" encoding="UTF-8"?> <user> <fname>Max</fname> <lname>Mule</lname> </user>

Delimiter to separate header and body

The **header** contains directives that apply to the body expression

The **body** contains a DataWeave expression that generates the output structure

The output directive

- 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

- 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

- 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

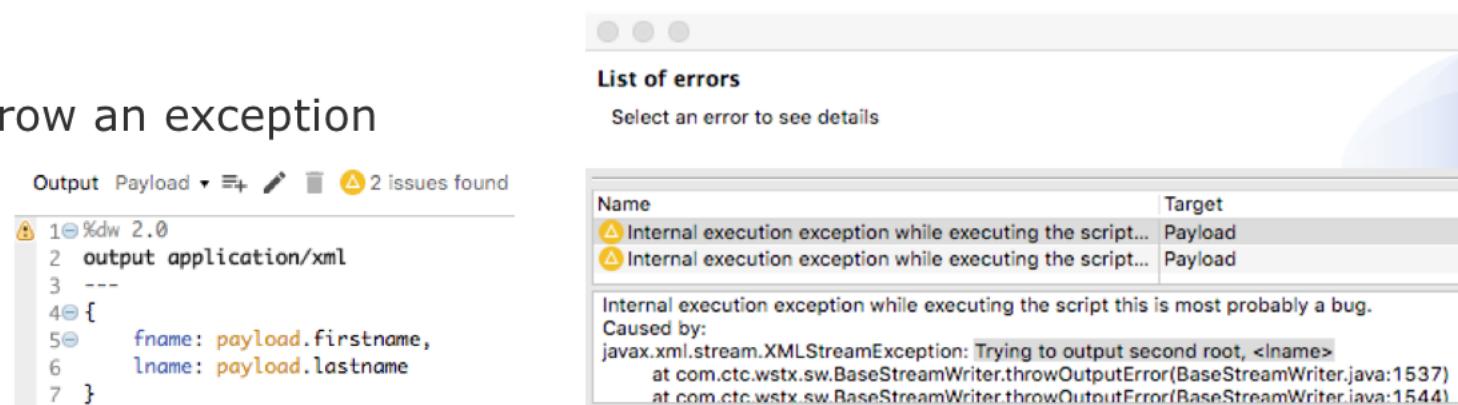
- The data model can consist of three different types of data: objects, arrays, simple literals

Input	Transform	JSON output
<pre>{ "firstname": "Max", "lastname": "Mule" }</pre>	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

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

- This will throw an exception



The screenshot shows the Mule Studio interface with the DataWeave editor open. The code is:

```

1 %dw 2.0
2 output application/xml
3 ---
4 {
5     fname: payload.firstname,
6     lname: payload.lastname
7 }

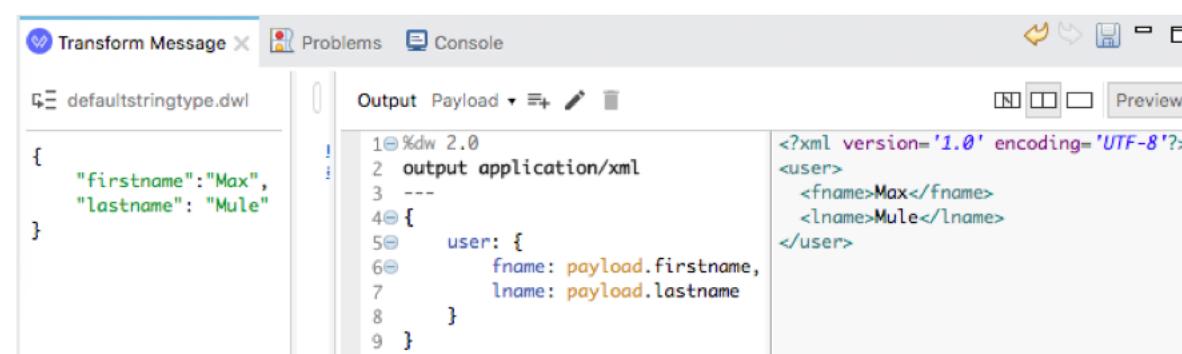
```

Two errors are listed in the 'List of errors' panel:

Name	Target
Internal execution exception while executing the script...	Payload
Internal execution exception while executing the script...	Payload

The details pane shows the exception message: "Internal execution exception while executing the script this is most probably a bug. Caused by: javax.xml.stream.XMLStreamException: Trying to output second root, <lname>".

- This will work



The screenshot shows the Mule Studio interface with the DataWeave editor open. The code is:

```

1 defaultstringtype.dwl
2 {
3     "firstname": "Max",
4     "lastname": "Mule"
5 }
6
7 %dw 2.0
8 output application/xml
9 ---
10 {
11     user: {
12         fname: payload.firstname,
13         lname: payload.lastname
14     }
15 }

```

The right panel shows the resulting XML output:

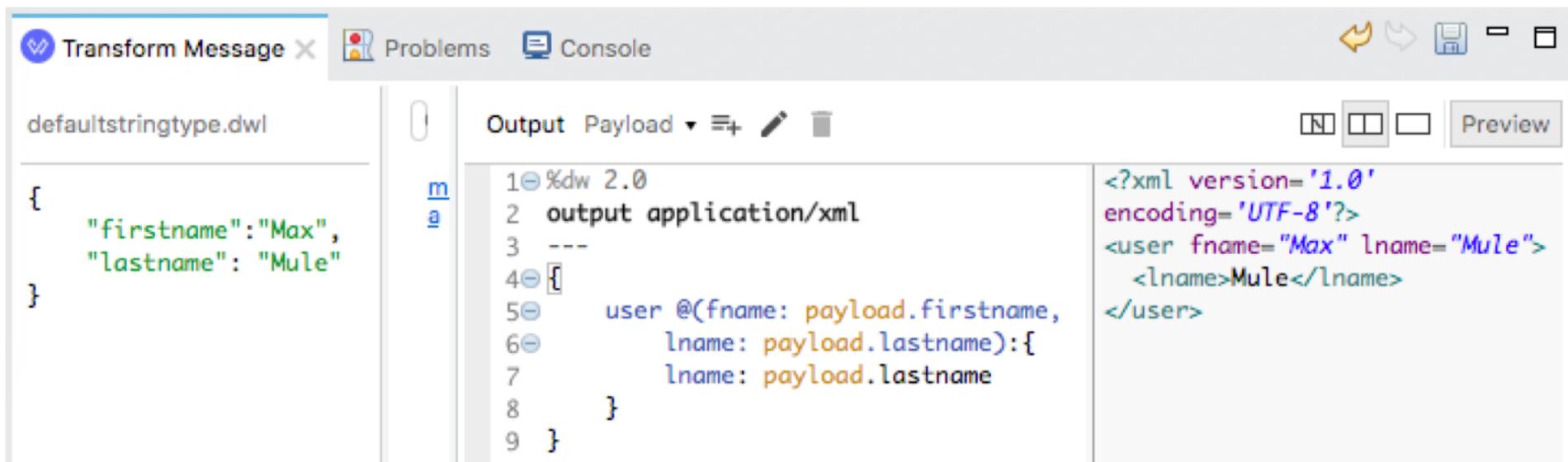
```

<?xml version='1.0' encoding='UTF-8'?>
<user>
    <fname>Max</fname>
    <lname>Mule</lname>
</user>

```

Specifying attributes for XML output

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



The screenshot shows the Mule Studio interface with the 'Transform Message' tab selected. The left pane displays a JSON payload:

```
{  
  "firstname": "Max",  
  "lastname": "Mule"  
}
```

The right pane shows the DataWeave code and its resulting XML output.

DataWeave (Payload):

```
1 %dw 2.0  
2 output application/xml  
3 ---  
4 [  
5   user @(fname: payload.firstname,  
6     lname: payload.lastname):{  
7       lname: payload.lastname  
8     }  
9 ]
```

XML Output (Preview):

```
<?xml version='1.0'  
encoding='UTF-8'?>  
<user fname="Max" lname="Mule">  
  <lname>Mule</lname>  
</user>
```

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 JSON, Java, and XML data structures

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



The screenshot shows the MuleSoft DataWeave editor interface. The top bar includes tabs for "Output" (selected), "Variable - DWoutput", and "Preview". There are also icons for edit, delete, and preview.

The left pane displays the DataWeave script:

```
1 %dw 2.0
2   output application/xml
3   ---
4   data: {
5     hub: "MUA",
6     flight @airline: payload.airline: {
7       code: payload.toAirportCode,
8     }
9   }
```

The right pane shows the generated XML output:

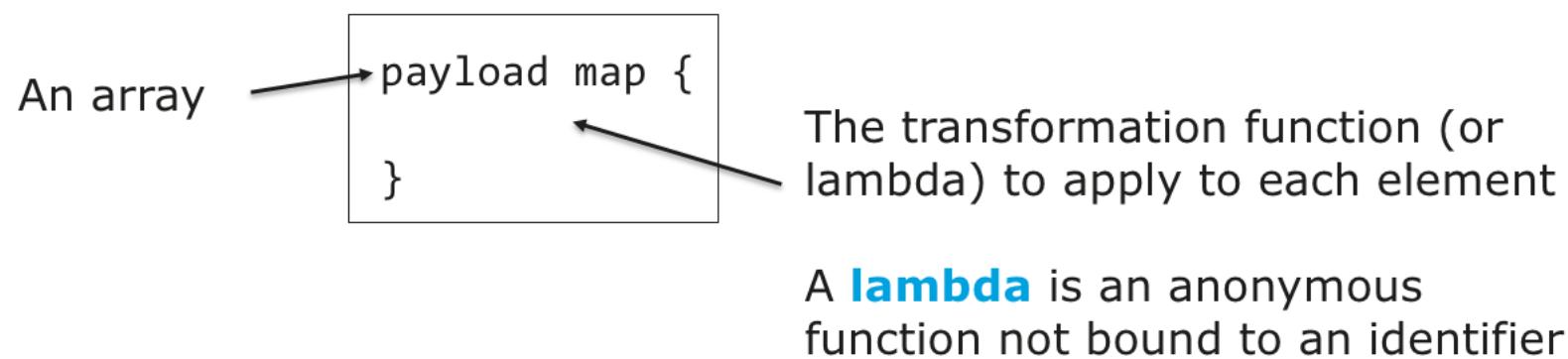
```
<?xml version='1.0' encoding='UTF-8'?>
<data>
  <hub>MUA</hub>
  <flight airline="United">
    <code>SF0</code>
  </flight>
</data>
```

Transforming complex data structures with arrays



Working with collections

- 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)

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

Input	Transform	Output
<pre>[{"firstname": "Max", "lastname": "Mule"}, {"firstname": "Molly", "lastname": "Mule"}]</pre>	<pre>%dw 2.0 output application/json --- payload map { num: \$\$, fname: \$.firstname, lname: \$.lastname }</pre>	<pre>[{"num": 0, "fname": "Max", "lname": "Mule"}, {"num": 1, "fname": "Molly", "lname": "Mule"}]</pre>
	<pre>%dw 2.0 %output application/json --- users: payload map { user: { fname: \$.firstname, lname: \$.lastname } }</pre>	<pre>{ "users": [{"user": { "fname": "Max", "lname": "Mule" }}, {"user": { "fname": "Molly", "lname": "Mule" }}] }</pre>

All contents © MuleSoft Inc.

26

Use explicit arguments in lambdas for more readable code

```
payload map {  
}
```

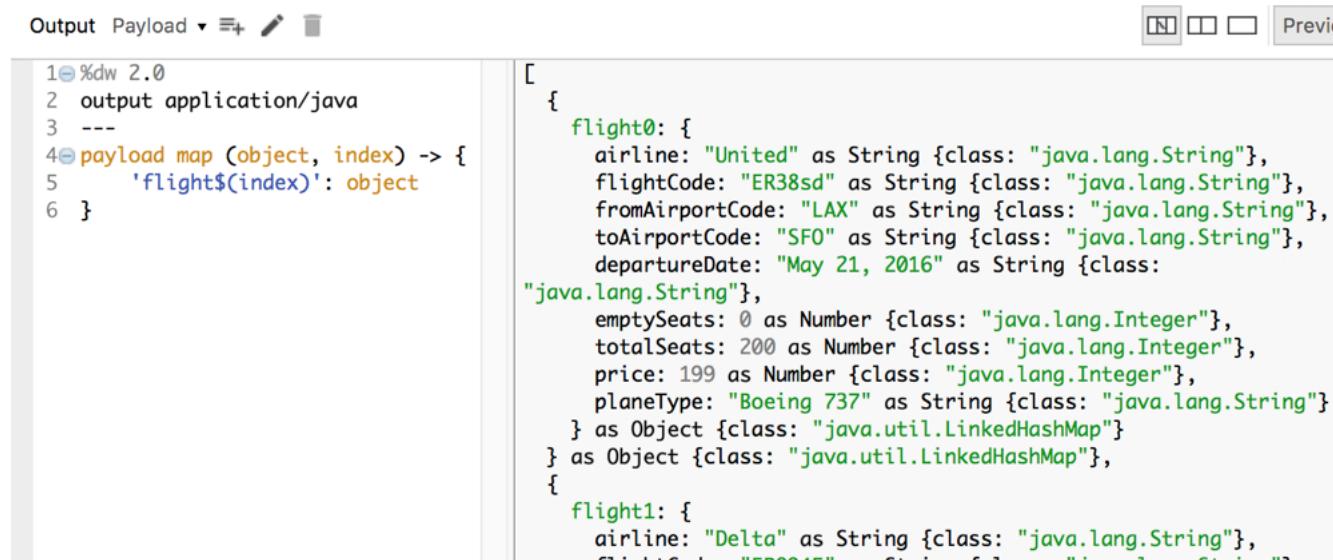


```
payload map (object, index) -> {  
}
```

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

Walkthrough 11-3: Transform complex data structures with arrays

- 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



The screenshot shows the MuleSoft DataWeaver interface. On the left, the 'Output Payload' pane displays the following DataWeave (dw) script:

```
1 %dw 2.0
2 output application/java
3 ---
4 payload map (object, index) -> {
5   'flight$(index)': object
6 }
```

On the right, the 'Preview' pane shows the resulting JSON output. The JSON structure is an array of flight objects:

```
[{"flight0": {"airline": "United" as String {class: "java.lang.String"}, "flightCode": "ER38sd" as String {class: "java.lang.String"}, "fromAirportCode": "LAX" as String {class: "java.lang.String"}, "toAirportCode": "SFO" as String {class: "java.lang.String"}, "departureDate": "May 21, 2016" as String {class: "java.lang.String"}, "emptySeats: 0 as Number {class: "java.lang.Integer"}, "totalSeats: 200 as Number {class: "java.lang.Integer"}, "price: 199 as Number {class: "java.lang.Integer"}, "planeType: "Boeing 737" as String {class: "java.lang.String"} } as Object {class: "java.util.LinkedHashMap"} } as Object {class: "java.util.LinkedHashMap"}, {"flight1": {"airline": "Delta" as String {class: "java.lang.String"}, "flightCode": "DL1234" as String {class: "java.lang.String"}, "fromAirportCode": "JFK" as String {class: "java.lang.String"}, "toAirportCode": "ATL" as String {class: "java.lang.String"}, "departureDate": "May 22, 2016" as String {class: "java.lang.String"}, "emptySeats: 10 as Number {class: "java.lang.Integer"}, "totalSeats: 150 as Number {class: "java.lang.Integer"}, "price: 250 as Number {class: "java.lang.Integer"}, "planeType: "Airbus A320" as String {class: "java.lang.String"} } as Object {class: "java.util.LinkedHashMap"} } as Object {class: "java.util.LinkedHashMap"}, {"flight2": {"airline": "Southwest" as String {class: "java.lang.String"}, "flightCode": "SW123" as String {class: "java.lang.String"}, "fromAirportCode": "DFW" as String {class: "java.lang.String"}, "toAirportCode": "HOU" as String {class: "java.lang.String"}, "departureDate": "May 23, 2016" as String {class: "java.lang.String"}, "emptySeats: 50 as Number {class: "java.lang.Integer"}, "totalSeats: 180 as Number {class: "java.lang.Integer"}, "price: 180 as Number {class: "java.lang.Integer"}, "planeType: "Boeing 737" as String {class: "java.lang.String"} } as Object {class: "java.util.LinkedHashMap"} } as Object {class: "java.util.LinkedHashMap"} ]
```

Transforming complex XML data structures



Writing expressions for XML output

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

Writing expressions for XML output (cont)

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

Writing expressions for XML input

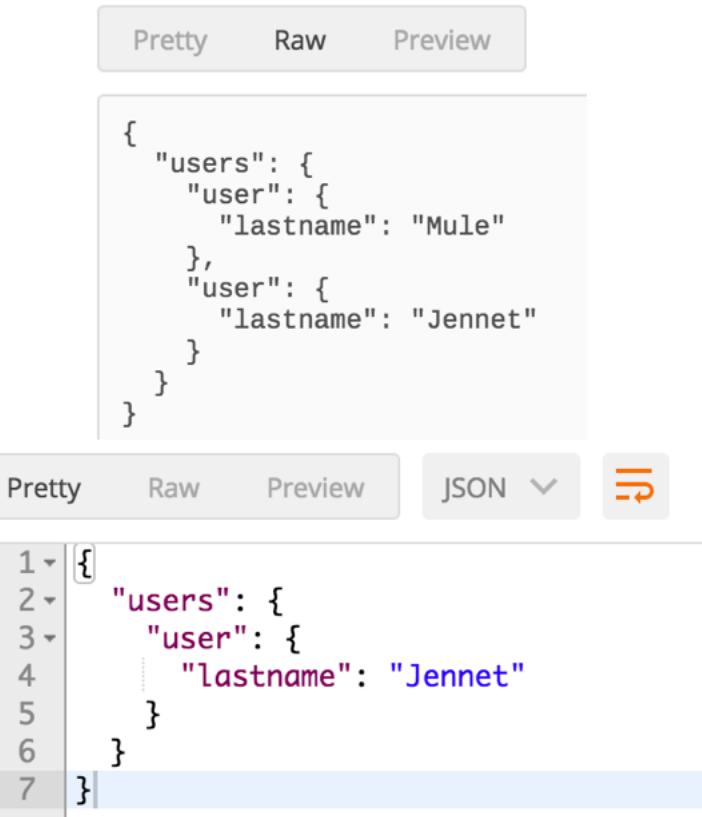
- Use `.*` selector to reference repeated elements

Input	Transform	JSON output
<pre><users> <user firstname="Max"> <lastname>Mule</lastname> </user> <user firstname="Molly"> <lastname>Jennet</lastname> </user> </users></pre>	payload	<pre>{"users": [{"user": {"lastname": "Mule"}, "user": {"lastname": "Jennet"}}]}</pre>
	payload.users	<pre>{["user": {"lastname": "Mule"}, "user": {"lastname": "Jennet"}]}</pre>
	payload.users.user	<pre>{"lastname": "Mule" }</pre>
	payload.users.*user	<pre>[{"lastname": "Mule" }, {"lastname": "Jennet" }]</pre>
	<pre>payload.users.*user map (obj,index) -> { fname: obj.firstname, lname: obj.lastname }</pre>	<pre>[{"fname": "Max", "lname": "Mule" }, {"fname": "Molly", "lname": "Jennet" }]</pre>

Beware of tools that “prettyify” responses

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

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



The screenshot shows a JSON editor interface with three tabs: Pretty, Raw, and Preview. The Pretty tab displays a readable JSON structure:

```
{
  "users": [
    {
      "user": {
        "lastname": "Mule"
      }
    },
    {
      "user": {
        "lastname": "Jennet"
      }
    }
  ]
}
```

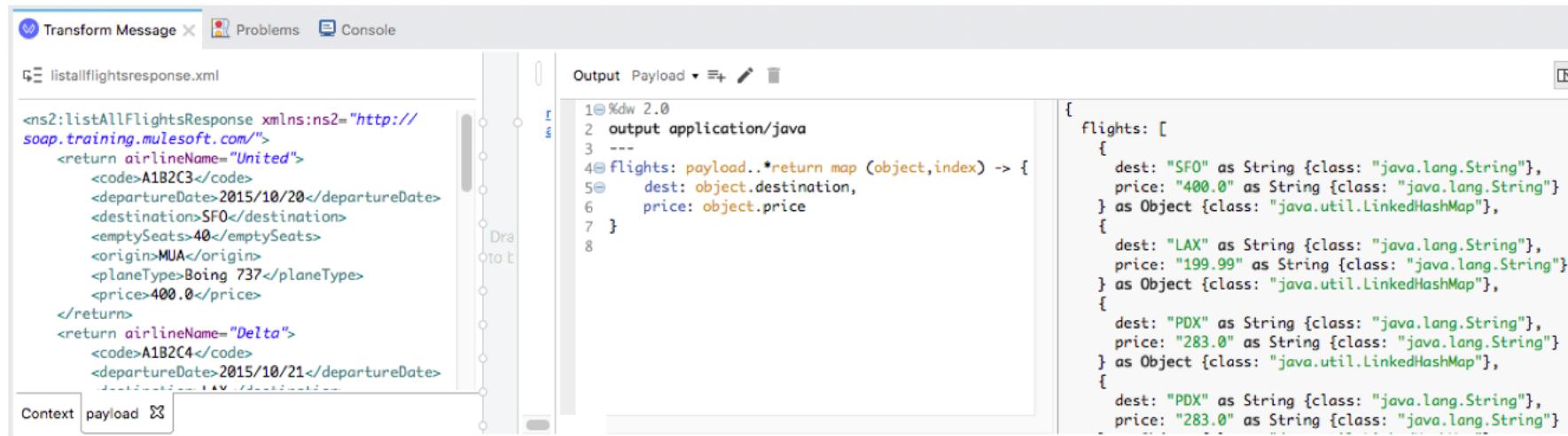
The Raw tab shows the raw JSON data:

```
1  {
2   "users": [
3     {
4       "user": {
5         "lastname": "Mule"
6       }
7     },
8     {
9       "user": {
10        "lastname": "Jennet"
11      }
12     }
13   ]
14 }
```

The Preview tab shows the raw JSON data again, with the last closing brace highlighted in blue.

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

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



The screenshot shows the Mule Studio interface with the 'Transform Message' tab selected. On the left, the 'listAllflightsresponse.xml' message is displayed as an XML document. It contains flight information for United and Delta airlines, including departure dates, destination cities, and prices. On the right, the 'Output' section shows the resulting JSON payload. The JSON structure maps the XML elements to objects, specifically creating an array of flight objects where each object has properties for destination and price.

```
<ns2:listAllflightsResponse xmlns:ns2="http://soap.training.mulesoft.com/">
    <return>
        <airlineName>United</airlineName>
        <code>A1B2C3</code>
        <departureDate>2015/10/20</departureDate>
        <destination>SFO</destination>
        <emptySeats>40</emptySeats>
        <origin>MUA</origin>
        <planeType>Boing 737</planeType>
        <price>400.0</price>
    </return>
    <return>
        <airlineName>Delta</airlineName>
        <code>A1B2C4</code>
        <departureDate>2015/10/21</departureDate>
    </return>
</ns2:listAllflightsResponse>
```

```
1@ %dw 2.0
2  output application/java
3  ---
4@ flights: payload..*return map (object,index) -> {
5@   dest: object.destination,
6@   price: object.price
7}
8

{
    flights: [
        {
            dest: "SFO" as String {class: "java.lang.String"}, 
            price: "400.0" as String {class: "java.lang.String"} 
        } as Object {class: "java.util.LinkedHashMap"}, 
        {
            dest: "LAX" as String {class: "java.lang.String"}, 
            price: "199.99" as String {class: "java.lang.String"} 
        } as Object {class: "java.util.LinkedHashMap"}, 
        {
            dest: "PDX" as String {class: "java.lang.String"}, 
            price: "283.0" as String {class: "java.lang.String"} 
        } as Object {class: "java.util.LinkedHashMap"}, 
        {
            dest: "PDX" as String {class: "java.lang.String"}, 
            price: "283.0" as String {class: "java.lang.String"} 
        }
    ]
}
```

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

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

Defining and using local variables

- 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"}	<pre>using (name = payload.firstname ++ " " ++ payload.lastname) name</pre>	"Max Mule"
	<pre>using (fname = payload.firstname, lname = payload.lastname) { person: using (user = fname, color = "gray") { name1: user, color: color }, name2: lname }</pre>	{ "person": { "name1": "Max", "color": "gray" }, "name2": "Mule" }
	<pre>using (fname = payload.firstname, lname = payload.lastname) { person: using (user = fname, color = "gray") { name1: user, color: color }, name2: lname, color: color }</pre>	Unable to resolve reference of color

Walkthrough 11-5: Define and use variables and functions

- 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



The screenshot shows the MuleSoft Anypoint Studio interface with the Dataweaver editor open. The left pane displays the MEL (Message Expression Language) code:

```
1@ %dw 2.0
2 output application/dw
3
4@ fun getNumSeats(planeType: String) =
5@   if (planeType contains('737'))
6@     150
7@   else
8@     300
9 ---
10@ using (flights =
11@   payload.*return map {object,index} -> {
12@     dest: object.destination,
13@     price: object.price,
14@     totalSeats: getNumSeats(object.planeType as String),
15@     plane: object.planeType
16@   }
17@ )
18@ flights
```

The right pane shows the resulting JSON output:

```
[{"dest": "SFO", "price": "400.0", "totalSeats": 150, "plane": "Boing 737"}, {"dest": "LAX", "price": "199.99", "totalSeats": 150, "plane": "Boing 737"}, {"dest": "PDX", "price": "283.0", "totalSeats": 300, "plane": "Boing 777"}, {"...}]
```

All contents © MuleSoft Inc.

39

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...

Any
Array
Binary
Boolean
CData
Comparable
Date and Time
Dictionary
Enum
Iterator
Key
Namespace
Nothing
Null
Number
Object
Range
Regex
SimpleType
String
TryResult
Type
Uri

Using format patterns

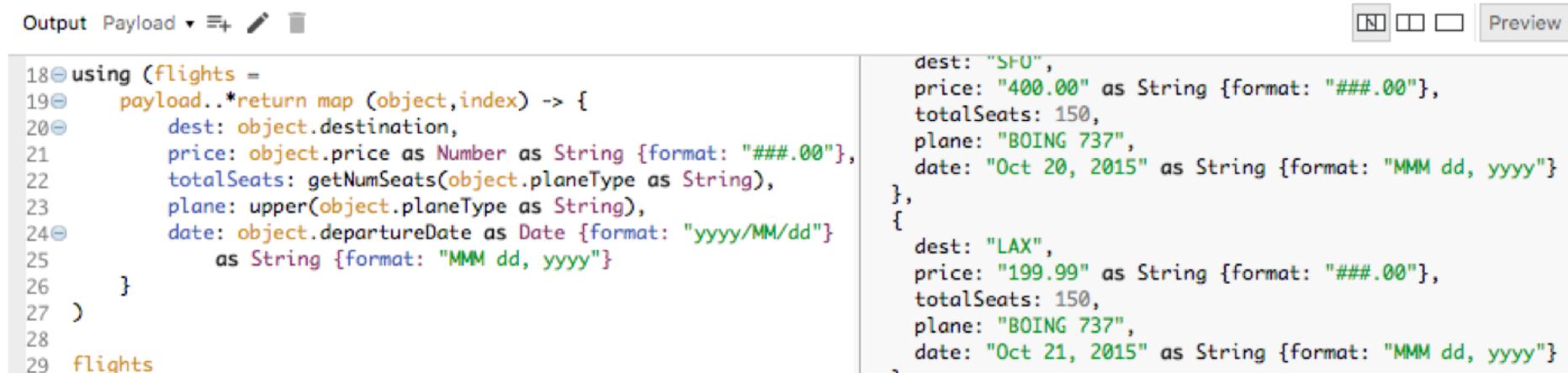
- Use metadata **format** schema property to format numbers and dates

```
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

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



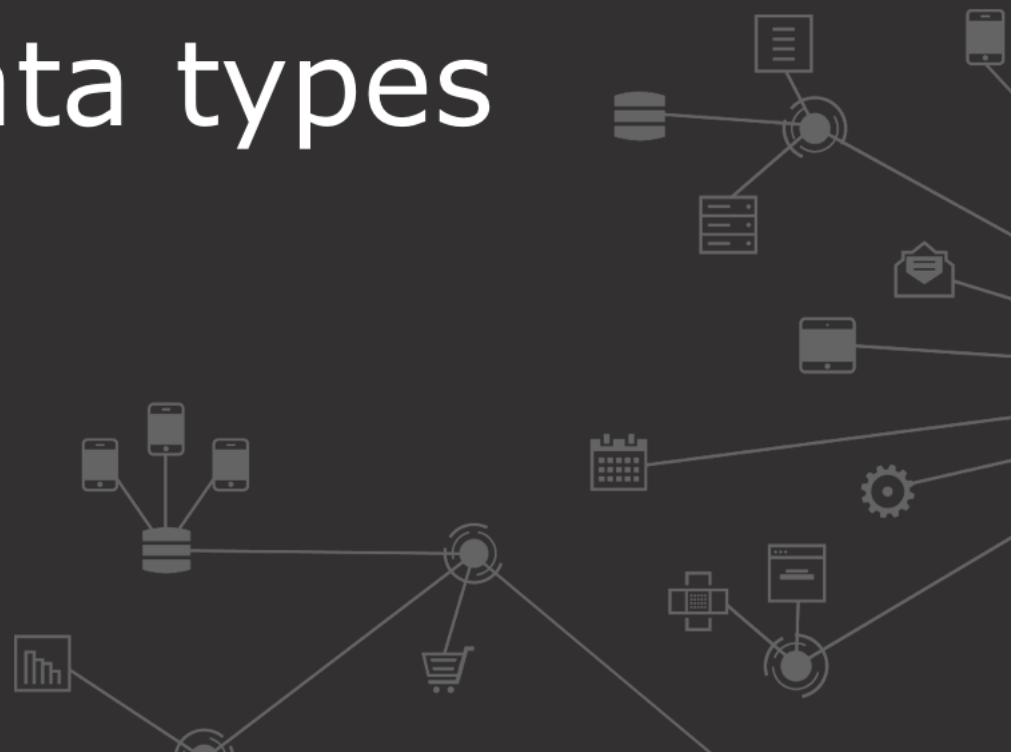
The screenshot shows the MuleSoft Dataweaver editor interface. The top bar includes tabs for "Output" and "Payload", and icons for "New", "Open", "Save", and "Preview". The main area displays the following Groovy script:

```
18@ using (flights =  
19@   payload..*return map (object,index) -> {  
20@     dest: object.destination,  
21@     price: object.price as Number as String {format: "###.00"},  
22@     totalSeats: getNumSeats(object.planeType as String),  
23@     plane: upper(object.planeType as String),  
24@     date: object.departureDate as Date {format: "yyyy/MM/dd"}  
25@       as String {format: "MMM dd, yyyy"}  
26@   }  
27@ )  
28@ flights
```

The output pane shows the resulting JSON-like data structure:

```
dest: "SFU",  
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

- 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

- Specify inline

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

- Or define a custom data type to use

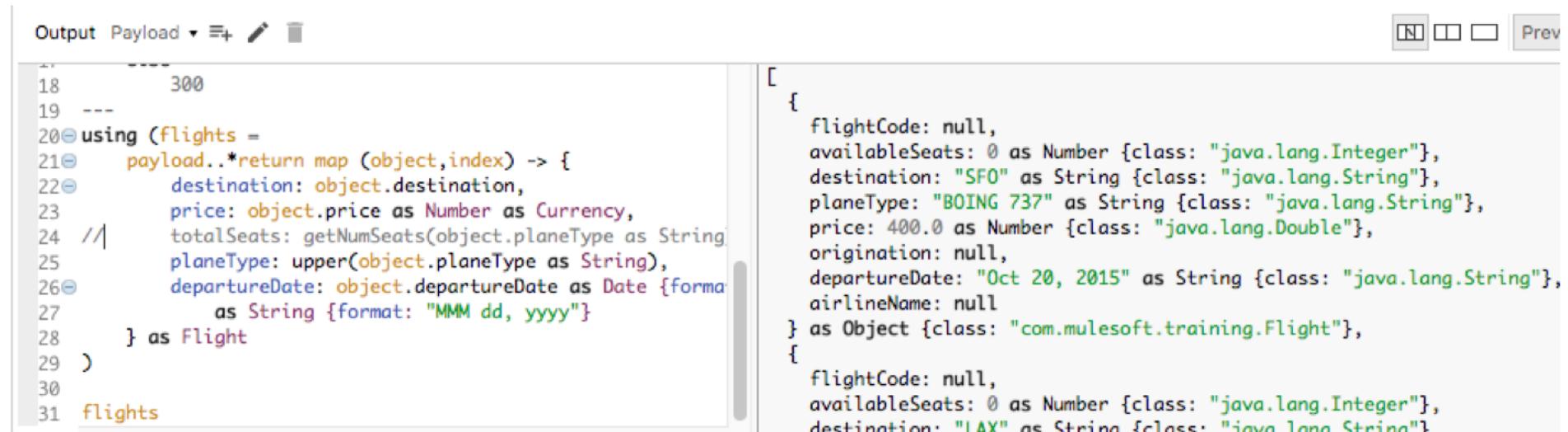
```
type user = Object {class: "my.company.User"}
```

```
---
```

```
customer:payload.user as user
```

Walkthrough 11-7: Define and use custom data types

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



The screenshot shows the MuleSoft Anypoint Studio Dataweaver editor interface. On the left, the 'Payload' tab displays Java code for transforming a list of flights into a list of Flight objects. On the right, the 'Preview' tab shows the resulting JSON output.

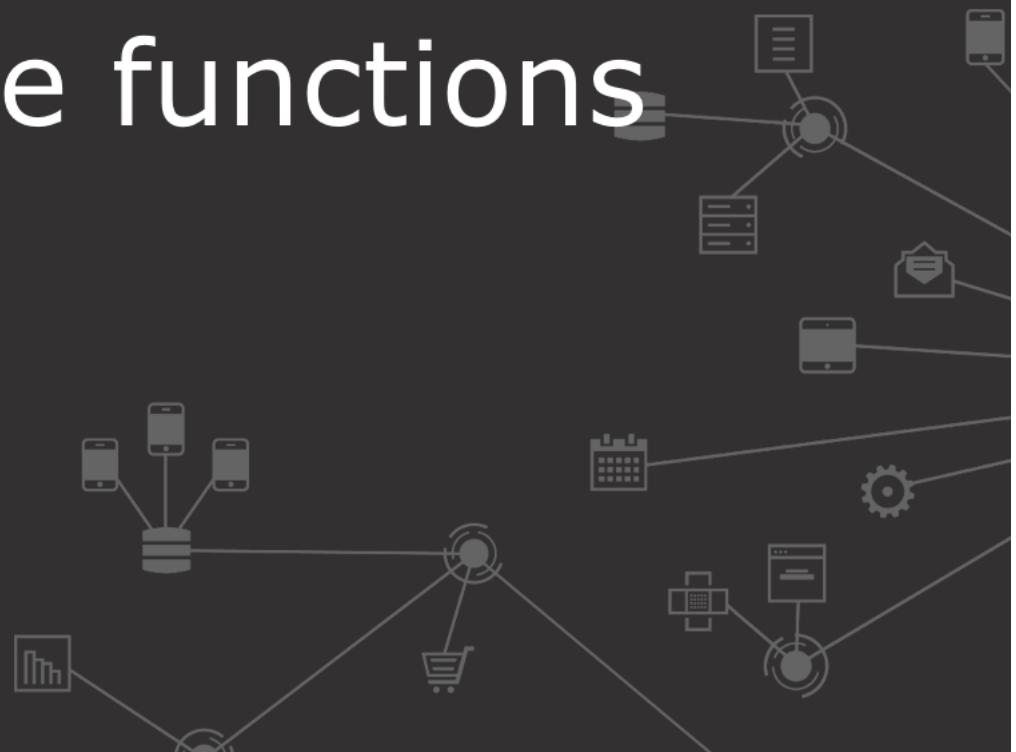
Payload (Java Code):

```
18      300
19  ---
20  using (flights =
21    payload.*return map (object,index) -> {
22      destination: object.destination,
23      price: object.price as Number as Currency,
24    // totalSeats: getNumSeats(object.planeType as String)
25    planeType: upper(object.planeType as String),
26    departureDate: object.departureDate as Date {format:
27        as String [format: "MMM dd, yyyy"]}
28    } as Flight
29  )
30
31 flights
```

Preview (JSON Output):

```
[{"flightCode": null, "availableSeats": 0 as Number [class: "java.lang.Integer"], "destination": "SFO" as String [class: "java.lang.String"], "planeType": "BOING 737" as String [class: "java.lang.String"], "price": 400.0 as Number [class: "java.lang.Double"], "origination": null, "departureDate": "Oct 20, 2015" as String [class: "java.lang.String"], "airlineName": null}, {"flightCode": null, "availableSeats": 0 as Number [class: "java.lang.Integer"], "destination": "IAX" as String [class: "java.lang.String"]}]
```

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 "]
```

DataWeave Function Reference			
Core (dw::Core)		groupBy	maxBy
++		isBlank	min
''		isDecimal	minBy
abs	isEmpty	mod	sizeOf
avg	isEven	native	startsWith
ceil	isInteger	now	sum
contains	isLeapYear	orderBy	to
daysBetween	isOdd	pluck	trim
distinctBy	joinBy	pow	typeOf
endsWith	log	random	unzip
filter	lower	randomInt	upper
filterObject	map	read	uuid
find	mapObject	readUrl	with
flatMap	match	reduce	write
flatten	matches	replace	zip
floor	max	round	49

All contents © MuleSoft Inc.

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



▼ DataWeave Function Reference

> Core (dw::Core)

> Crypto (dw::Crypto)

> Runtime (dw::Runtime)

▼ Strings (dw::core::Strings)

camelize

> System (dw::System)

capitalize

> Arrays (dw::core::Arrays)

charCode

> Binaries (dw::core::Binaries)

charCodeAt

> Objects (dw::core::Objects)

dasherize

> Strings (dw::core::Strings)

fromCharCode

> URL (dw::core::URL)

ordinalize

> Diff (dw::util::Diff)

pluralize

> Timer (dw::util::Timer)

singularize

underscore

> Timer (dw::util::Timer)

- 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

- 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

- Use the **lookup** function

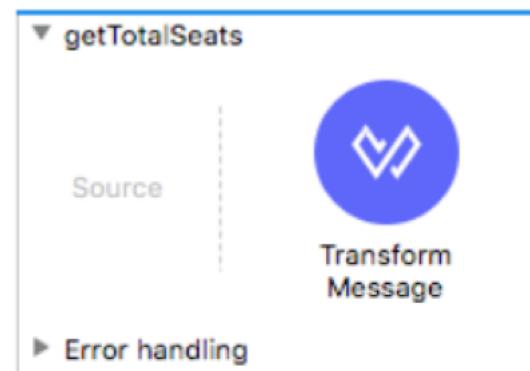
```
{a: lookup("myFlow", {b:"Hello"}) }
```

- 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

- Call a flow from a DataWeave expression

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



Summary



Summary

- 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

- 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

- 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