



Module 11: Writing DataWeave Transformations



- 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 •

Preview

```
1 %dw 2.0
2   output application/json
3   type Currency = String {format: "###.00"}
4   type Flight = Object {class: "com.mulesoft.training.Flight"}
5   fun getNumSeats(planeType: String) =
6     if (planeType contains ("737"))
7       150
8     else
9       300
10  import dasherize from dw::core::Strings
11  ---
12  using flights-
13  payload..* return map (object,index) -> {
14    destination: object.destination,
15    availableSeats: object.emptySeats as Number,
16    price: object.price as Number as Currency,
17    totalSeats: getNumSeats(object.planeType as String),
18    // totalSeats: lookup('getTotalSeats',{planeType: object.planeType}),
19    planeType: dasherize(replace(object.planeType,/(Boing)/) with "Boeing"),
20    departureDate: object.departureDate as Date {format: "yyyy/MM/dd"}
21      as String {format: "MMM dd, yyyy"}
22  } as Flight
23 }
24 flights orderBy $.departureDate
25   orderBy $.price
26   distinctBy $
27   filter ($.availableSeats != 0)
```

```
[{"destination": "LAX", "availableSeats": 10, "price": "199.99", "totalSeats": 150, "planeType": "boeing-737", "departureDate": "Oct 21, 2015"}, {"destination": "PDX", "availableSeats": 23, "price": "283.00", "totalSeats": 300, "planeType": "boeing-777", "departureDate": "Oct 20, 2015"}, {"destination": "PDX", "availableSeats": 30, "price": "283.00", "totalSeats": 300, "planeType": "boeing-777", "departureDate": "Oct 21, 2015"}, {"destination": "SFO", "availableSeats": 0, "price": null, "totalSeats": 300, "planeType": "boeing-777", "departureDate": null}]
```

- 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?

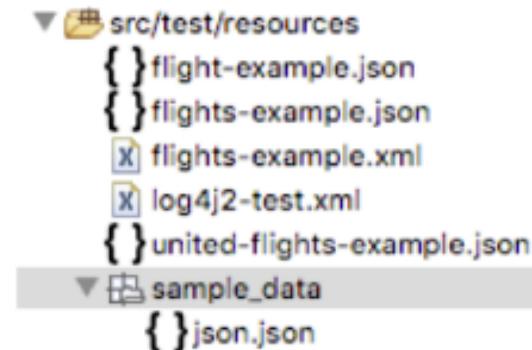
Where is the DataWeave code?



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

```
<flow name="postFlight" doc:id="847ead89-f898-4c
    <ee:transform doc:name="Transform Message" d
        <ee:message >
            <ee:set-payload ><![CDATA[%dw 2.0
output application/java
---
payload]]></ee:set-payload>
        </ee:message>
    </ee:transform>
    <logger level="INFO" doc:name="Logger" doc:i
</flow>
```

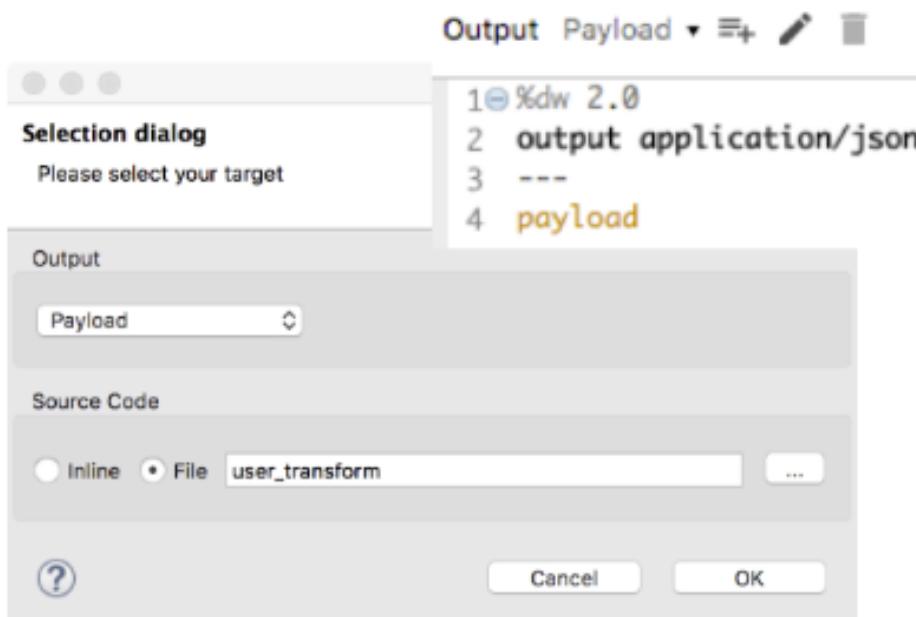
- Sample data used for live preview is stored in src/test/resources



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>
```

▼ src/main/resources
 log4j2.xml
 `{/} user_transform.dwl`
 api

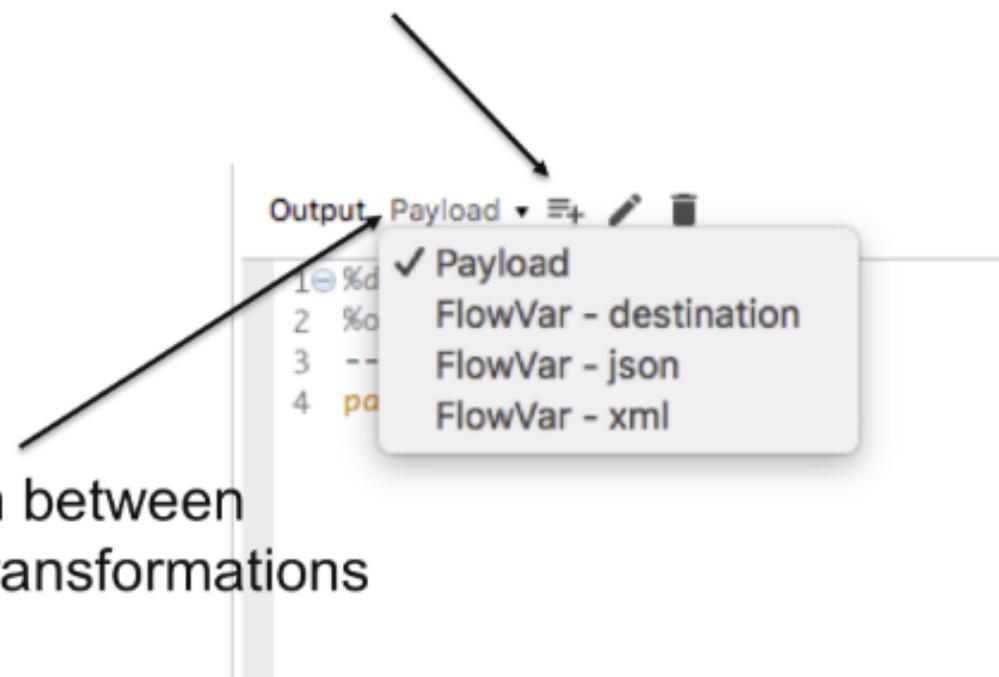
- A single script can be stored in an **external DWL file** and referenced
 - In the Transform Message component using the resource attribute
 - In an element that has an expression property (like the Choice router) using the \${file::filename} syntax
- You can also create **modules** (libraries) of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

```
<ee:transform doc:name="Transform Message" doc:id="69db8e0f-1a2c-48d3-8a2a-0a3a2a3a3a3a">
    <ee:message>
        <ee:set-payload resource="user_transform.dwl" />
    </ee:message>
</ee:transform>
```

Creating multiple transformations

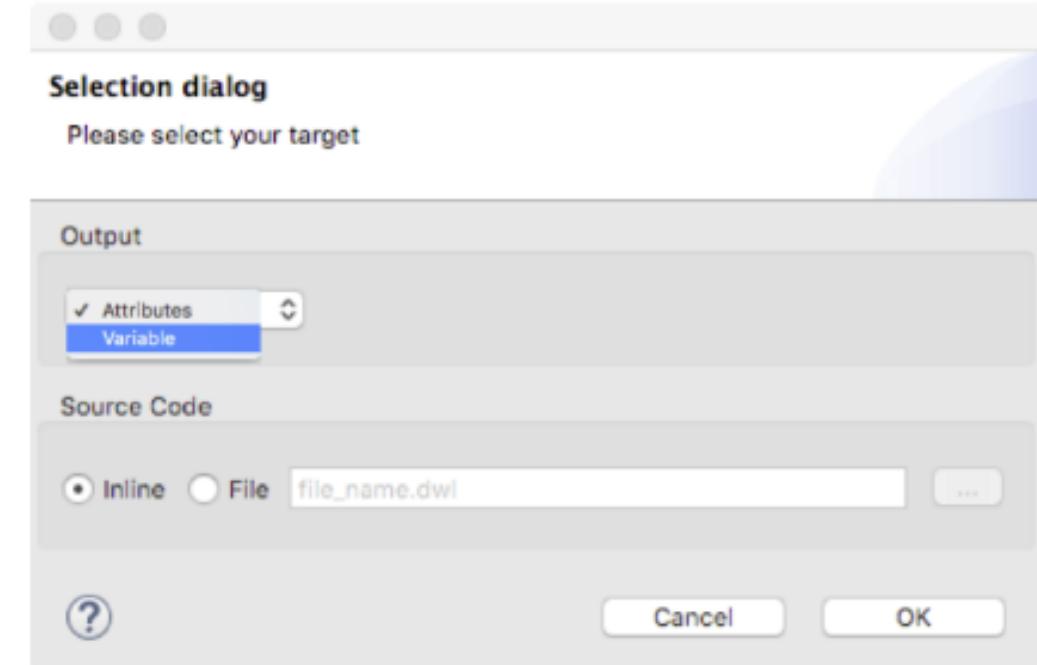
- You can also create multiple transformations with one Transform Message component

(1) Use Add new target button



(3) Switch between multiple transformations

(2) Set where to store result



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



- Create a new flow that receives POST requests of JSON flight objects
- Add and manage 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

The screenshot shows the Mule Studio interface. On the left, there is a DataWeave (DW) editor window titled "Output Variable - DWoutput". It contains the following code:

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

Below the code, a preview pane shows a JSON object:

```
{
  "airline": "United",
  "flightCode": "ER38sd",
  "fromAirportCode": "LAX",
  "toAirportCode": "JFK"
}
```

On the right, there is a file tree view:

- src/test/resources**:
 - { } flight-example.json
 - { } flights-example.json
 - [X] flights-example.xml
 - [X] log4j2-test.xml
 - { } united-flights-example.json
- src/main/resources**:
 - [X] application-types.xml
 - [X] config.yaml
 - {/} json_flight_playground.dwl
 - [X] log4j2.xml
- sample_data**:
 - { } json.json

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

In this walkthrough, you create a new flow that receives flight POST requests that you will use as the input for writing transformations in the next several walkthroughs. You will:

- 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.

The screenshot shows the Mule Studio interface. On the left, there is a code editor window titled "Output Variable - DWoutput" containing the following DataWeave script:

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

To the right of the code editor is a tree view of project resources:

- src/test/resources**: Contains `flight-example.json`, `flights-example.json`, `flights-example.xml`, `log4j2-test.xml`, and `united-flights-example.json`.
- src/main/resources**: Contains `application-types.xml`, `config.yaml`, `json_flight_playground.dwl` (which is selected), and `log4j2.xml`.
- sample_data**: Contains `json.json`.

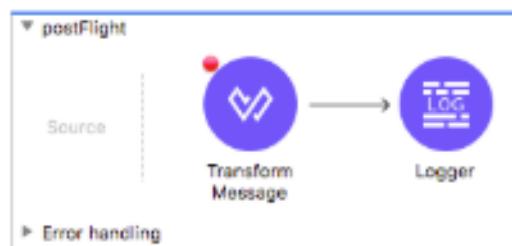
Create a new flow that receives POST requests and returns the payload as Java

1. In the apdev-flights-ws Mule project, open config.yaml.
2. Change the delta.wsdl property from /deltas?wsdl to /delta?wsdl.
3. Return to implementation.xml.
4. Right-click in the canvas and select Collapse All.
5. Drag a Transform Message component from the Mule Palette to the bottom of the canvas; a new flow should be created.
6. Change the name of the flow to postFlight.
7. In the Transform Message properties view, set the expression to the payload.

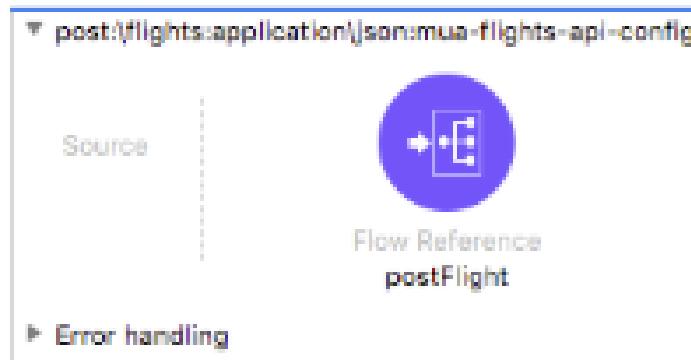
Output Payload • Preview

```
1@%dw 1.0
2 %output application/java
3 ---
4 payload
```

8. Add a breakpoint to the Transform Message component.
9. Add a Logger to the flow.



10. Return to interface.xml.
11. Locate the post:\flights\application\json:mua-flights-api-config flow.
12. Delete the Transform Message component and replace it with a Flow Reference.
13. In the Flow Reference properties view, set the flow name and display name to postFlight.



14. Save the files to redeploy the project in debug mode.
15. Close interface.xml.

Post a flight to the flow

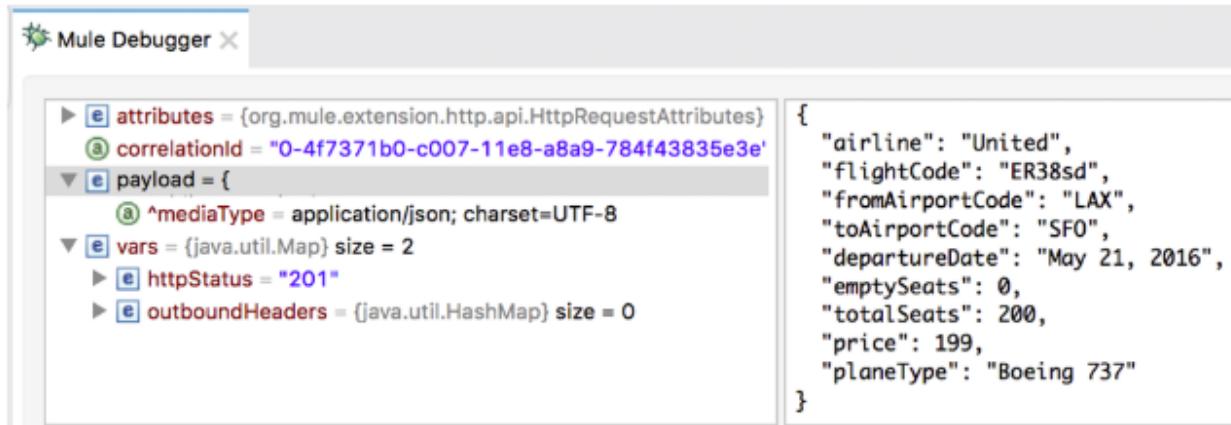
16. Open flight-example.json in src/test/resources.
17. Copy the code and close the file.
18. In Advanced REST Client, change the method to POST and remove any query parameters.
19. Add a request header called Content-Type and set it equal to application/json.
20. Set the request body to the value you copied from flight-example.json.
21. Make the request to <http://localhost:8081/api/flights>.

The screenshot shows the Advanced REST Client interface. At the top, there are fields for 'Method' (set to 'POST') and 'Request URL' (set to 'http://localhost:8081/api/flights'). Below these are buttons for 'SEND' and a three-dot menu. Underneath, a section titled 'Parameters' has an '^' icon. A table with tabs for 'Headers', 'Authorization', 'Body' (which is underlined in blue), 'Variables', and 'Actions' is shown. In the 'Body' tab, the 'Body content type' is set to 'application/json'. Below this, there are two buttons: 'FORMAT JSON' and 'MINIFY JSON'. The JSON payload is displayed in a large text area:

```
{  
    "airline": "United",  
    "flightCode": "ER38sd",  
    "fromAirportCode": "LAX",  
    "toAirportCode": "SFO",  
    "departureDate": "May 21, 2016",  
    "emptySeats": 0,  
    "totalSeats": 200,  
    "price": 199,  
    "planeType": "Boeing 737"  
}
```

22. In the Mule Debugger, expand Attributes and locate the content-type header.

23. Review the payload; you should see it has a mime-type of application/json.



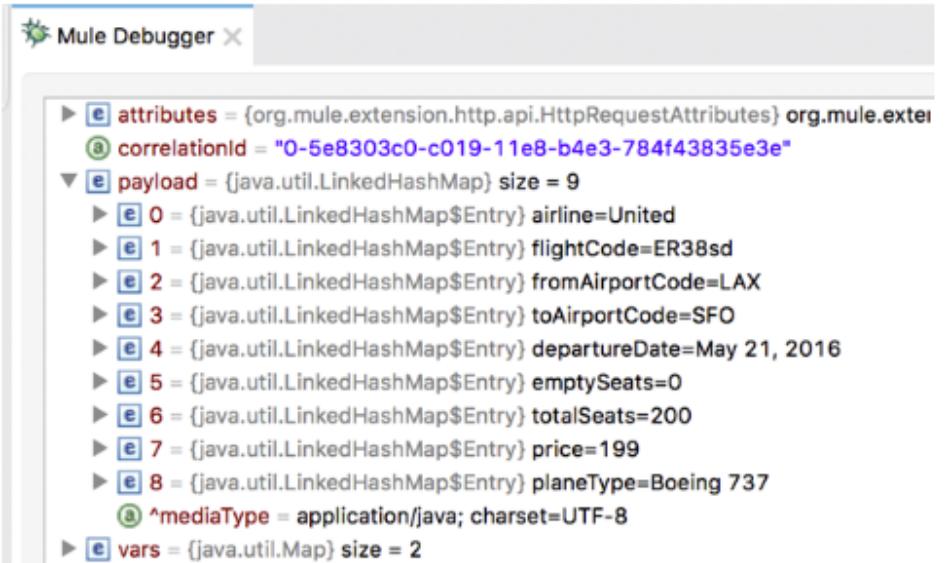
The screenshot shows the Mule Debugger interface with the title bar "Mule Debugger". In the left pane, there is a tree view of variables:

- attributes = {org.mule.extension.http.api.HttpRequestAttributes}
- correlationId = "0-4f7371b0-c007-11e8-a8a9-784f43835e3e"
- payload = {** (highlighted in gray)
- vars = {java.util.Map} size = 2
 - httpStatus = "201"
 - outboundHeaders = {java.util.HashMap} size = 0

In the right pane, the payload is displayed as a JSON object:

```
{  
    "airline": "United",  
    "flightCode": "ER38sd",  
    "fromAirportCode": "LAX",  
    "toAirportCode": "SFO",  
    "departureDate": "May 21, 2016",  
    "emptySeats": 0,  
    "totalSeats": 200,  
    "price": 199,  
    "planeType": "Boeing 737"  
}
```

24. Step to the Logger; you should see the payload now has a mime-type of application/java and is a LinkedHashMap.



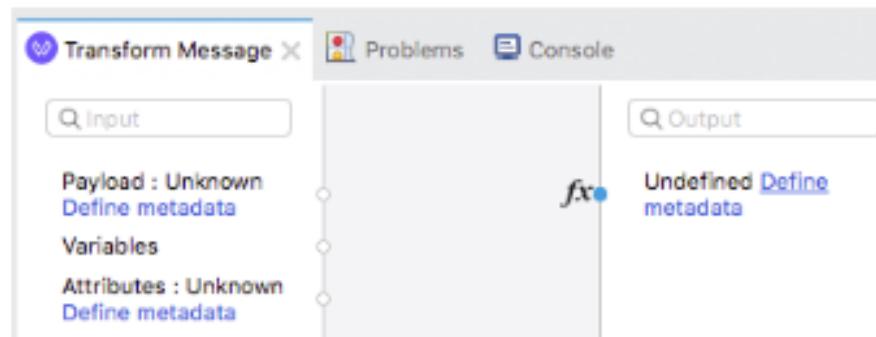
The screenshot shows the Mule Debugger interface with the title bar "Mule Debugger". In the left pane, there is a tree view of variables:

- attributes = {org.mule.extension.http.api.HttpRequestAttributes} org.mule.exter
- correlationId = "0-5e8303c0-c019-11e8-b4e3-784f43835e3e"
- payload = {java.util.LinkedHashMap} size = 9** (highlighted in gray)
- vars = {java.util.Map} size = 2
 - 0 = {java.util.LinkedHashMap\$Entry} airline=United
 - 1 = {java.util.LinkedHashMap\$Entry} flightCode=ER38sd
 - 2 = {java.util.LinkedHashMap\$Entry} fromAirportCode=LAX
 - 3 = {java.util.LinkedHashMap\$Entry} toAirportCode=SFO
 - 4 = {java.util.LinkedHashMap\$Entry} departureDate=May 21, 2016
 - 5 = {java.util.LinkedHashMap\$Entry} emptySeats=0
 - 6 = {java.util.LinkedHashMap\$Entry} totalSeats=200
 - 7 = {java.util.LinkedHashMap\$Entry} price=199
 - 8 = {java.util.LinkedHashMap\$Entry} planeType=Boeing 737

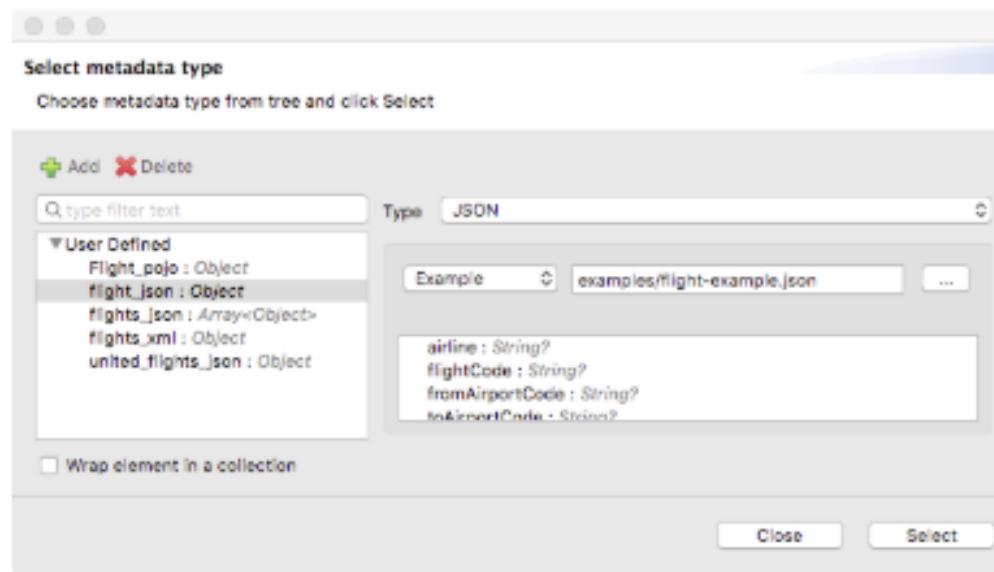
25. Step to the end of the application and switch perspectives.

Add input metadata for the transformation

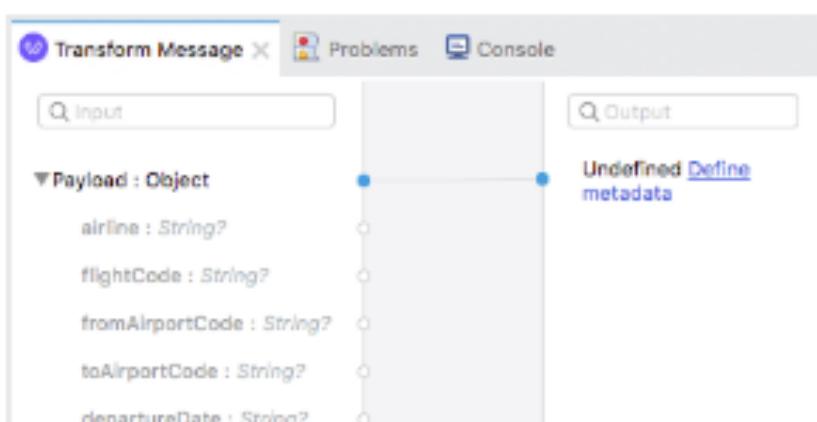
26. In the Transform Message properties view, click Define metadata in in the input section.



27. In the Select metadata type dialog box, select flight_json and click Select.



28. In the Transform Message Properties view, you should now see metadata for the input.



29. Open application-types.xml in src/main/resources.

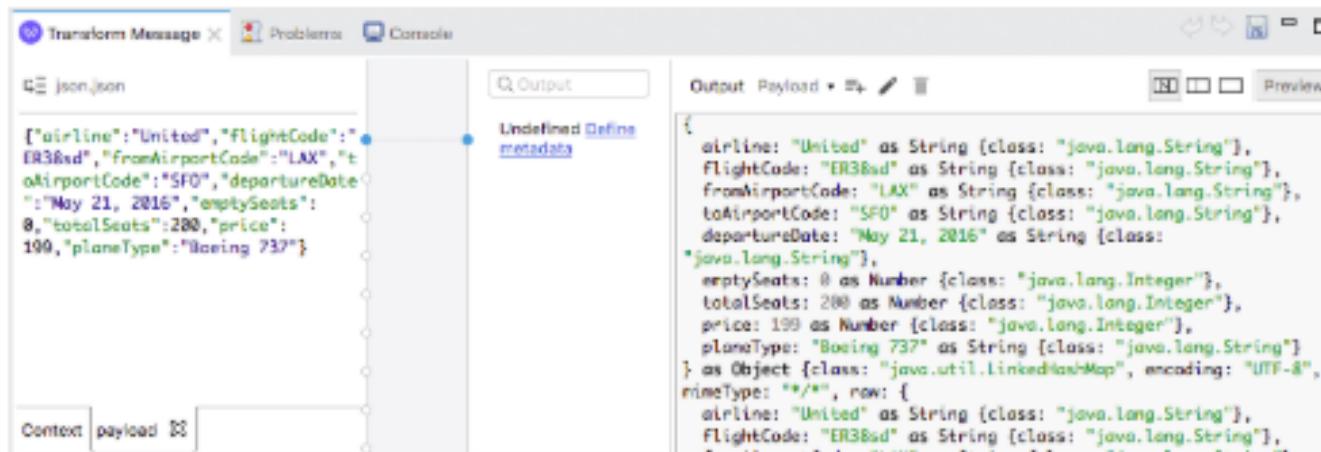
30. Review the enrichment elements.

```
63  <m>
64  </types:mule>
65  <types:enrichment select="#f963e3e4-5b3a-41bf-8c32-ef79bdce4a32">
66    <types:processor-declaration>
67      <types:input-event>
68        <types:message>
69          <types:payload type="Flight_json"/>
70        </types:message>
71      </types:input-event>
72    </types:processor-declaration>
73  </types:enrichment>
74 </types:mule>
```

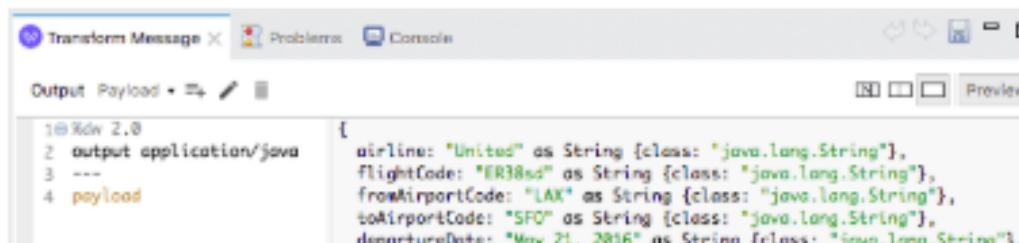
31. Close the file.

Preview sample data and sample output

32. In the Transform Message properties view, click the Preview button.
33. Click the Create required sample data to execute preview link.
34. Look at the input section; you should see a new tab called payload and it should contain the sample data.
35. Look at the preview section; you should see the sample output there of type Java.

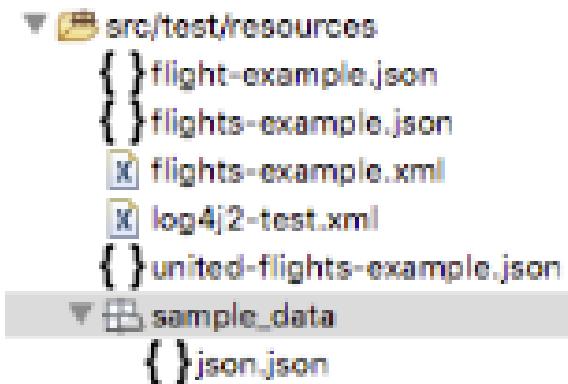


36. Click the Source Only button to the left of the Preview button.



Locate the new sample data file

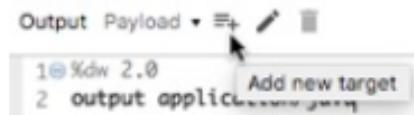
37. In the project explorer, locate the new sample_data folder in src/test/resources.



38. Open json.json.
39. Review the code and close the file.

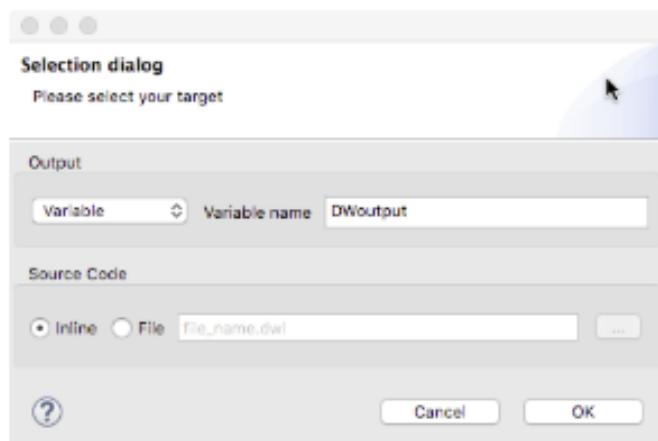
Create a second transformation with the same component

40. Click the Add new target button.



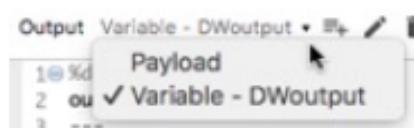
41. In the Selection dialog box, change the output to variable.

42. Set the variable name to DWoutput.



43. Click OK.

44. In the Transform Message properties view, click the drop-down menu button for the output; you should see and can switch between the two transformations.



45. For the new transformation, set the output type to json and set the expression to payload.

```
Output Variable - DWoutput ▾ + 🖊 🗑  
1@%dw 2.0  
2   output application/json  
3   ---  
4   payload
```

Test the application

46. Save the file to redeploy the project in debug mode.
47. In Advanced REST Client, post the same request to <http://localhost:8081/api/flights>.
48. In the Mule Debugger, step to the Logger.
49. Review Payload; it should be Java as before.
50. Expand Variables; you should see the new DWoutput variable.

The screenshot shows the Mule Debugger interface with two main sections: 'payload' and 'vars'.

payload:

```
{  
  "airline": "United",  
  "flightCode": "ER38sd",  
  "fromAirportCode": "LAX",  
  "toAirportCode": "SFO",  
  "departureDate": "May 21, 2016",  
  "emptySeats": 0,  
  "totalSeats": 200,  
  "price": 199,  
  "planeType": "Boeing 737"  
}
```

vars:

```
{  
  DWoutput = {n "airline": "United",n "flightCode": "ER38sd",n "from/  
  httpStatus = "201"  
  outboundHeaders = {java.util.HashMap} size = 0  
}
```

51. Step through the rest of the application and switch perspectives.

Review the Transform Message XML code

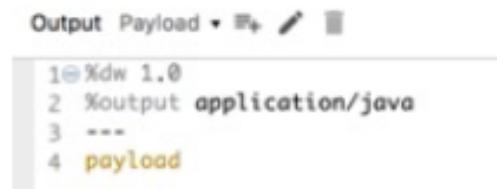
52. Right-click the Transform Message component and select Go to XML.
53. Locate and review the code for both DataWeave transformations.

```
<flow name="postFlight" doc:id="493369a6-2d8d-4dfd-a241-210504c4d5eb" >
    <ee:transform doc:name="Transform Message" doc:id="be12bc7d-22f8-491
        <ee:message >
            <ee:set-payload><![CDATA[Xdm 2.0
output application/java
---
payload]]></ee:set-payload>
        </ee:message>
        <ee:variables >
            <ee:set-variable variableName="DWoutput" ><![CDATA[Xdm 2.0
output application/json
---
payload]]></ee:set-variable>
        </ee:variables>
    </ee:transform>
    <logger level="INFO" doc:name="Logger" doc:id="03c0bf95-38e9-45d2-a.
</Flow>
```

54. Switch back to the Message Flow view.

Save a DataWeave script to an external file

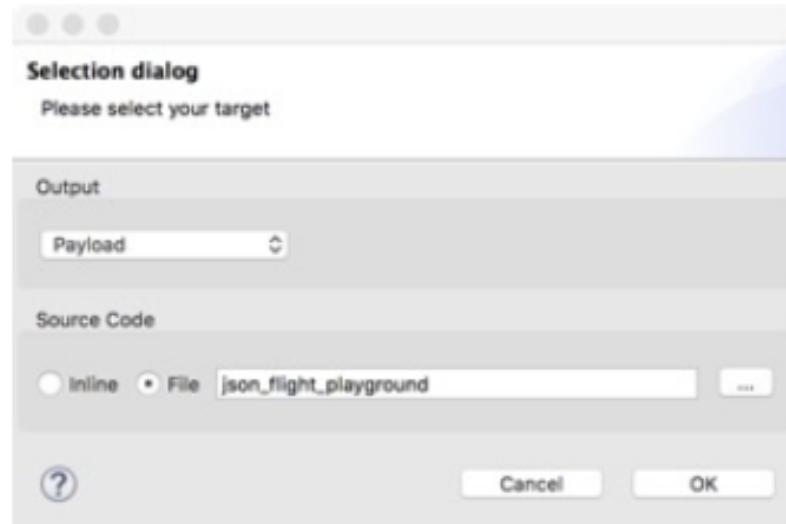
55. In the Transform Message Properties view, make sure the payload output expression is selected.
56. Click the Edit current target button (the pencil) above the code.



The screenshot shows the DataWeave editor interface. At the top, there is a toolbar with icons for Output, Payload, and a pencil. Below the toolbar, the code editor displays the following DataWeave script:

```
1@%dw 1.0
2 %output application/java
3 ---
4 payload
```

57. In the Selection dialog box, change the source code selection from inline to file.
58. Set the file name to json_flight_playground.



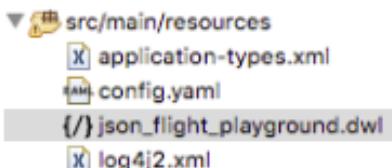
59. Click OK.

Locate and review the code that uses an external DataWeave script

60. Switch to Configuration XML view.
61. Locate the new code for the transformation in postFlight.

```
<flow name="postFlight" doc:id="847ead89-f898-4cd5-9ec0-1a1fdde0ed6d" >
    <ee:transform doc:name="Transform Message" doc:id="f963e3e4-5b3a-41f
        <ee:message >
            <ee:set-payload resource="json_flight_playground.dwl" />
        </ee:message>
        <ee:variables >
            <ee:set-variable variableName="DWoutput" ><![CDATA[%dw 2.0
output application/json
--->
payload]]></ee:set-variable>
        </ee:variables>
    </ee:transform>
    <logger level="INFO" doc:name="Logger" doc:id="360ce24e-4dd3-4cef-bc
</flow>
```

62. In the Package Explorer, expand src/main/resources.



63. Open json_flight_playground.dwl.
64. Review and then close the file.

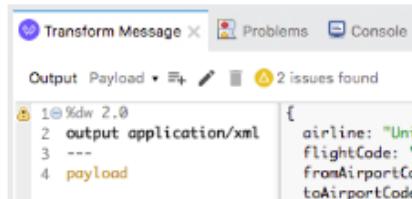
The screenshot shows the Eclipse code editor with the file 'json_flight_playground.dwl' open. The code is as follows:

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

65. Return to Message Flow view in implementation.xml.

Change the output type to XML and review errors

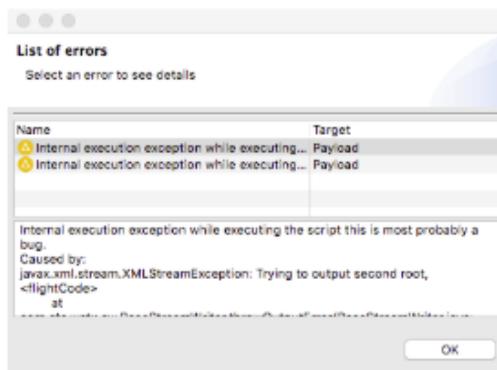
66. In the Transform Message properties view, select the Variable – DWoutput output.
67. Change the output type from application/json to application/xml.
68. Locate the warning icons indicating that there is a problem.



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

Note: If you do not see the warnings, click the Preview button. If you do not see a data preview, select the payload tab in the input section. If you don't see the payload tab, right-click payload in the input section and select Edit sample data.

69. Mouse over the icon located to the left of the code; you should see a message that there was an exception trying to output the second root <flightCode>.
70. Click the icon above the code; a List of errors dialog box should open.
71. Select and review the first issue.



72. In the List of errors dialog box, click OK.

Note: You will learn how to successfully transform to XML in the next walkthrough.

Test the application

73. Save the file to redeploy the project in debug mode.
74. In Advanced REST Client, post the same request to <http://localhost:8081/api/flights>.
75. In the Mule Debugger, step once; you should get an error.
76. Expand Exception and review the error information; you should see there is a DataWeave error when trying to output the second root.

The screenshot shows two perspectives in the Mule Studio interface: 'Mule Debugger' and 'implementation'.

Mule Debugger Perspective: This perspective displays a detailed error stack trace. The error message is: "Exception while trying to write value as Xml : Trying to output second root, <flightCode>" evaluating expression: "%dw 2.0|output applica". The stack trace includes various exception classes like ExpressionRuntimeException, MULE:EXPRESSION, and ExpressionExecutionException, along with suppressed exceptions and the cause of the error.

implementation Perspective: This perspective shows the Mule flow configuration. A flow named 'postFlight' is defined with a single route. The route starts with a 'Source' component (indicated by a dashed red box), followed by a 'Transform Message' component (highlighted with a red dashed border), and ends with a 'Logger' component.

77. Step to the end of the application and switch perspectives.

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

- 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

- 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

- 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



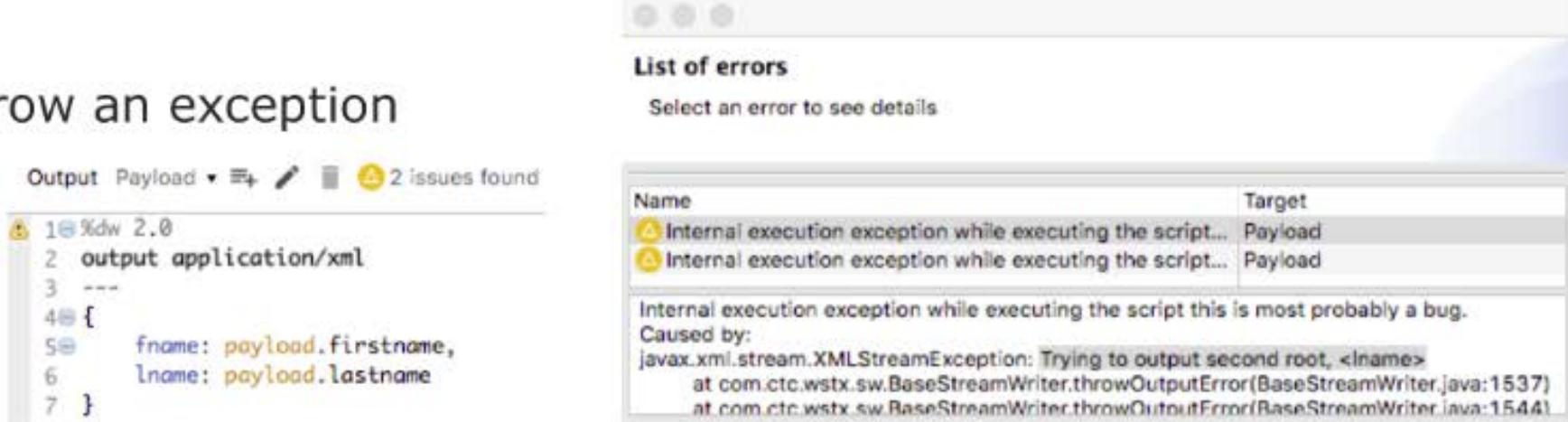
- The data model can consist of three different types of data: objects, arrays, simple literals

Input	Transform	JSON output
{ "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

- 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. On the left, the script editor displays the following code:

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

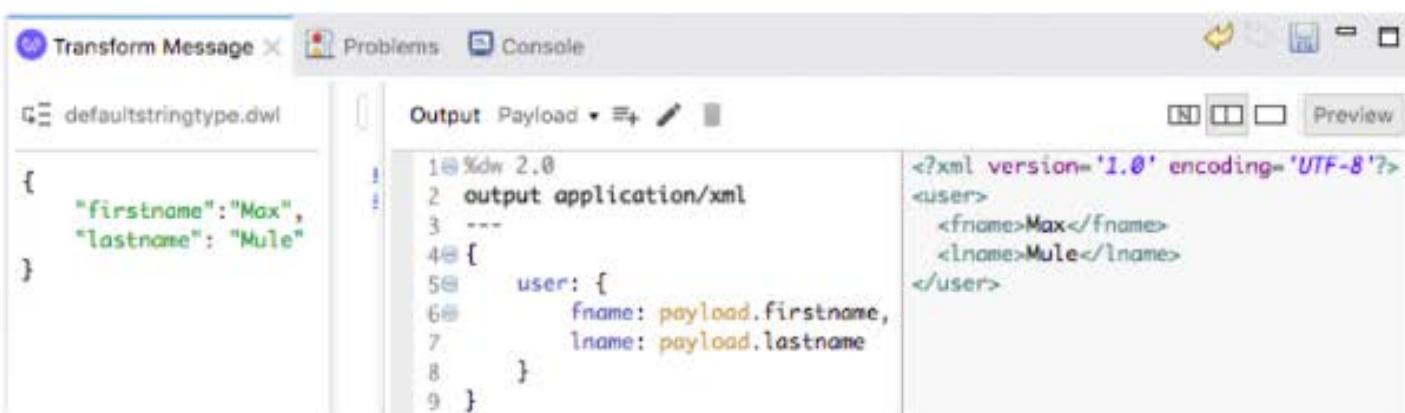
The right side shows the 'List of errors' dialog with two entries:

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

Details for the first error:

Internal execution exception while executing the script this is most probably a bug.
Caused by:
javax.xml.stream.XMLStreamException: Trying to output second root, <names>
at com.ctc.wstx.sw.BaseStreamWriter.throwOutputError(BaseStreamWriter.java:1537)
at com.ctc.wstx.sw.BaseStreamWriter.throwOutputError(BaseStreamWriter.java:1544)

- This will work



The screenshot shows the Mule Studio interface with a 'Transform Message' component. The script editor on the left contains:

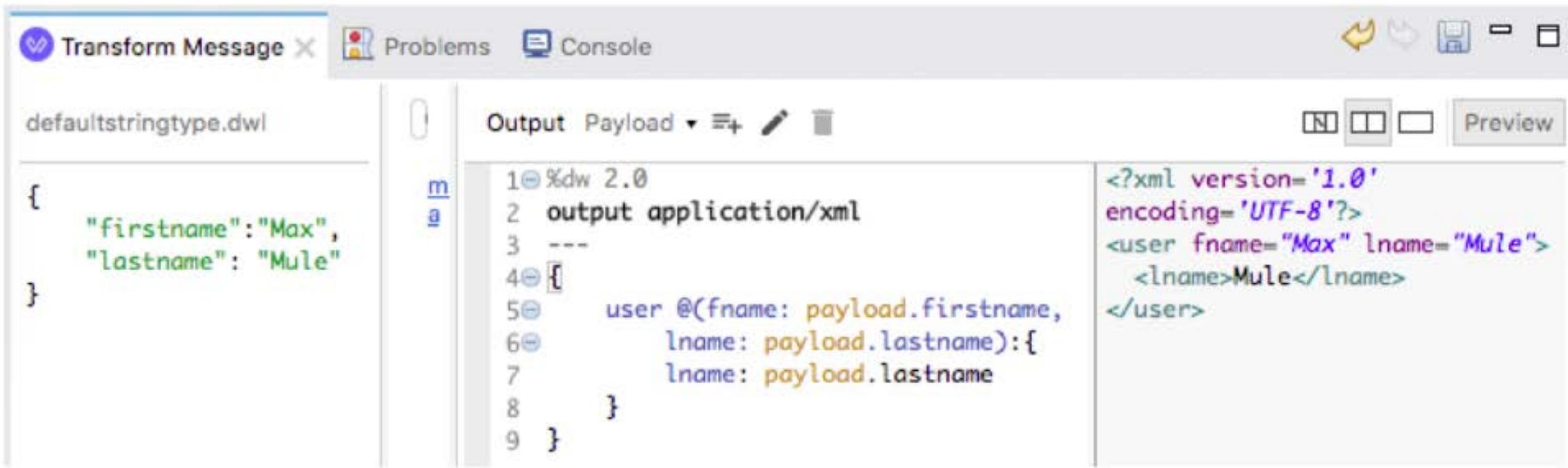
```
{ "firstname": "Max", "lastname": "Mule" }
```

The preview window on the right 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. On the left, the file 'defaultstringtype.dwl' contains the following JSON payload:

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

In the center, the DWL script is displayed:

```
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 ]
```

On the right, the generated XML output is shown:

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

- 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

Output Payload ▾ Preview

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

{
 data: {
 hub: "MUA" as String {class: "java.lang.String"},
 code: "SFO" as String {class: "java.lang.String"},
 airline: "United" as String {class: "java.lang.String"}
 } as Object {class: "java.util.LinkedHashMap"}
} as Object {class: "java.util.LinkedHashMap", encoding:
"UTF-8", mimeType: "*/*", raw: {
 ...
}}

Output Variable - DWoutput ▾ Preview

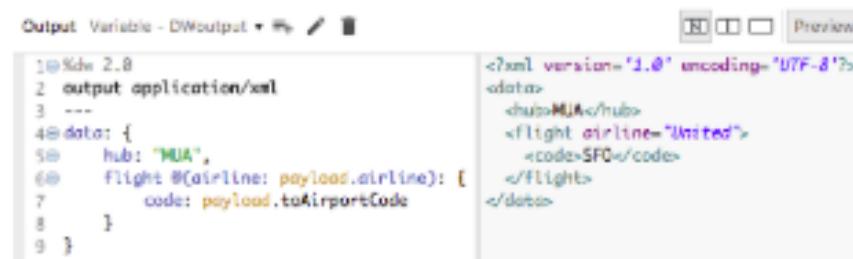
```
1@%dw 2.0
2  output application/xml
3  ---
4@ data: {
5@   hub: "MUA",
6@   flight @(airline: payload.airline): [
7@     code: payload.toAirportCode
8   ]
9 }
```

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

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

In this walkthrough, you continue to work with the JSON flight data posted to the flow. You will:

- 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 a code editor window with the following content:

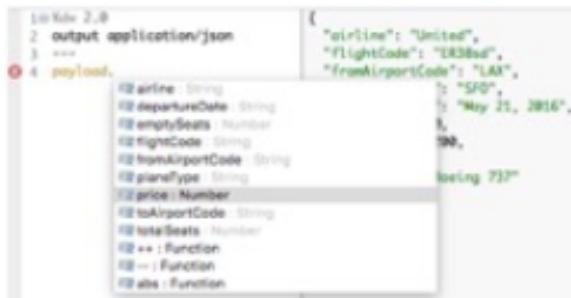
```
Output Variable - DWoutput • ⌂ ⌂ ⌂ Preview  
1@Kds 2.8  
2 output application/xml  
3 ---  
4@data: {  
5@ hub: "MIA",  
6@ flight @(airline: payload.airline): [  
7@     code: payload.toAirportCode  
8@ ]  
9 }
```

On the right side of the editor, there is a preview pane titled "Preview" which displays the generated XML output:

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

Write expressions to transform JSON to various Java structures

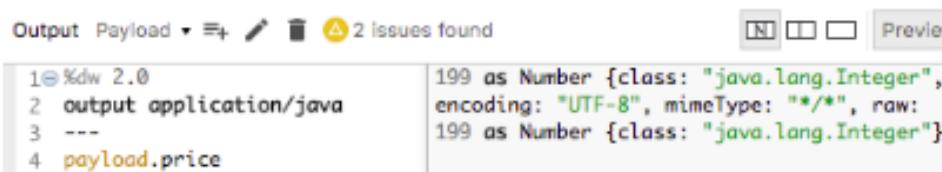
1. Return to the Transform Message properties view for the transformation in postFlight.
2. In the output drop-down menu, select Payload.
3. Change the output type from java to json.
4. Type a period after payload and select price from the auto-completion menu.



5. Look at the preview.



6. Change the output type from json to java.



7. Change the DataWeave expression to data: payload.price.

Output Payload ▾ 2 issues found

1@%dw 2.0
2 output application/java
3 ---
4 data: payload.price

{
 data: 199 as Number {class: "java.lang.Integer"}
} as Object {class: "java.util.LinkedHashMap",
encoding: "UTF-8", mimeType: "*/*", raw: {
 data: 199 as Number {class: "java.lang.Integer"}
} as Object {class: "java.util.LinkedHashMap"}}

8. Change the output type from java to json.

Output Payload ▾ 2 issues found

1@%dw 2.0
2 output application/json
3 ---
4 data: payload.price

{
 "data": 199
}

9. Change the DataWeave expression to data: payload.

Output Payload ▾ 2 issues found

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

{
 "data": {
 "airline": "United",
 "flightCode": "ER38sd",
 "fromAirportCode": "LAX",
 "toAirportCode": "SFO",
 "departureDate": "May 21, 2016",
 "emptySeats": 0,
 "totalSeats": 200,
 "price": 199,
 "planeType": "Boeing 737"
 }
}

10. Change the output type from json to java.

The screenshot shows the Mule Studio DataWeave editor with the following code:

```
1@%dw 2.0
2  output application/java
3  ---
4 @data: payload
```

The right pane displays the Java representation of the payload:

```
{
  data: {
    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"}
```

11. Change the DataWeave expression to data: {}.

12. Add a field called hub and set it to "MUA".

The screenshot shows the Mule Studio DataWeave editor with the following code:

```
1@%dw 2.0
2  output application/java
3  ---
4 @data: {
5   hub: "MUA"
6 }
```

The right pane displays the Java representation of the payload, including the new 'hub' field:

```
{
  data: {
    hub: "MUA" as String {class: "java.lang.String"}
  } as Object {class: "java.util.LinkedHashMap"}
} as Object {class: "java.util.LinkedHashMap", encoding: "UTF-8",
 mimeType: "*/*", row: {}}
```

13. Change the output type from java to json.

The screenshot shows the Mule Studio DataWeave editor with the following code:

```
1@%dw 2.0
2  output application/json
3  ---
4 @data: {
5   hub: "MUA"
6 }
```

The right pane displays the JSON representation of the payload:

```
{
  "data": {
    "hub": "MUA"
  }
}
```

14. Add a field called code and use auto-completion to set it to the toAirportCode property of the payload.

15. Add a field called airline and set it to the airline property of the payload.

The screenshot shows the Mule Studio DataWeave editor interface. The left pane displays the DataWeave script:

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

The right pane shows the preview of the JSON output:

```
{"data": {  
    "hub": "MUA",  
    "code": "SFO",  
    "airline": "United"  
}}
```

Write an expression to output data as XML

16. In the output drop-down menu, select Variable – DW output.
17. Change the DataWeave expression to data: payload.
18. Look at the preview; you should now see XML.

The screenshot shows the Mule Studio DataWeave editor interface. The left pane displays the DataWeave script:

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

The right pane shows the preview of the XML output:

```
<?xml version='1.0' encoding='UTF-8'?>
<data>
  <airline>United</airline>
  <flightCode>ER38sd</flightCode>
  <fromAirportCode>LAX</fromAirportCode>
  <toAirportCode>SFO</toAirportCode>
  <departureDate>May 21, 2016</departureDate>
  <emptySeats>0</emptySeats>
  <totalSeats>200</totalSeats>
  <price>199</price>
  <planeType>Boeing 737</planeType>
</data>
```

19. In the output drop-down menu, select Payload.
20. Copy the DataWeave expression.
21. Switch back to Variable – DWoutput and replace the DataWeave expression with the one you just copied.

The screenshot shows the Mule Studio DataWeave editor interface. The left pane displays the DataWeave script:

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

The right pane shows the preview of the XML output:

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

22. Modify the expression so the code and airline properties are child elements of a new element called flight.

Output Variable - DWoutput + Preview

```
1@ Xdw 2.0
2  output application/xml
3  ---
4@ data: {
5@   hub: "MIA",
6@   flight: {
7@     code: payload.toAirportCode,
8@     airline: payload.airline
9@   }
10 }
```

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

23. Modify the expression so the airline is an attribute of the flight element.

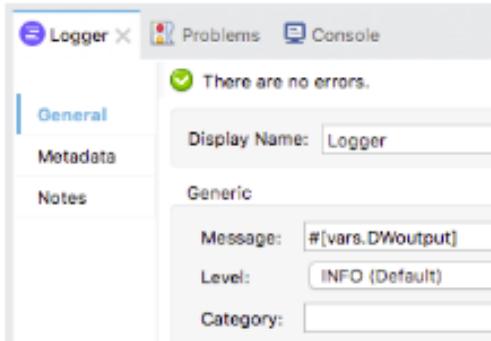
Output Variable - DWoutput + Preview

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

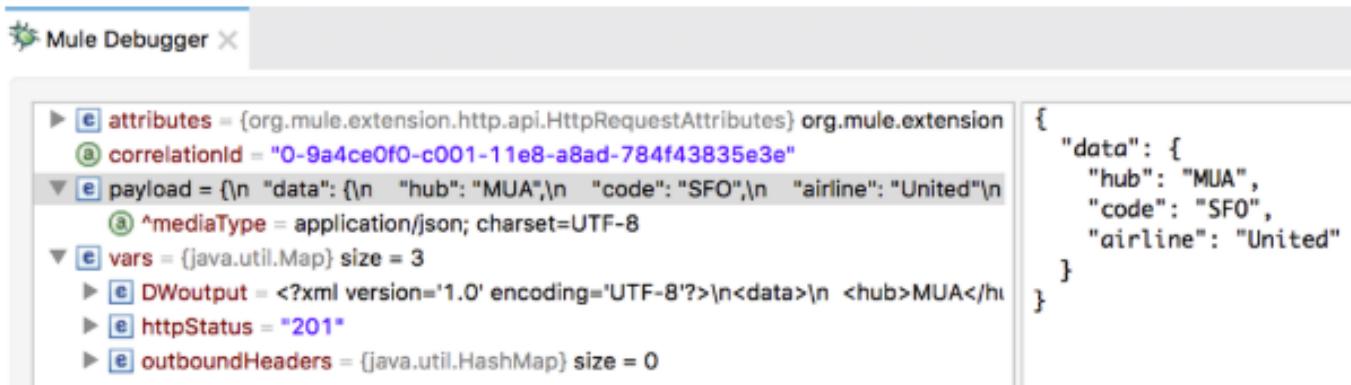
```
<?xml version='1.0' encoding='UTF-8'?>
<data>
  <hub>MIA</hub>
  <flight airline="United">
    <code>SFO</code>
  </flight>
</data>
```

Debug the application

24. In the Logger properties view, set the message to the value of the DWoutput variable.

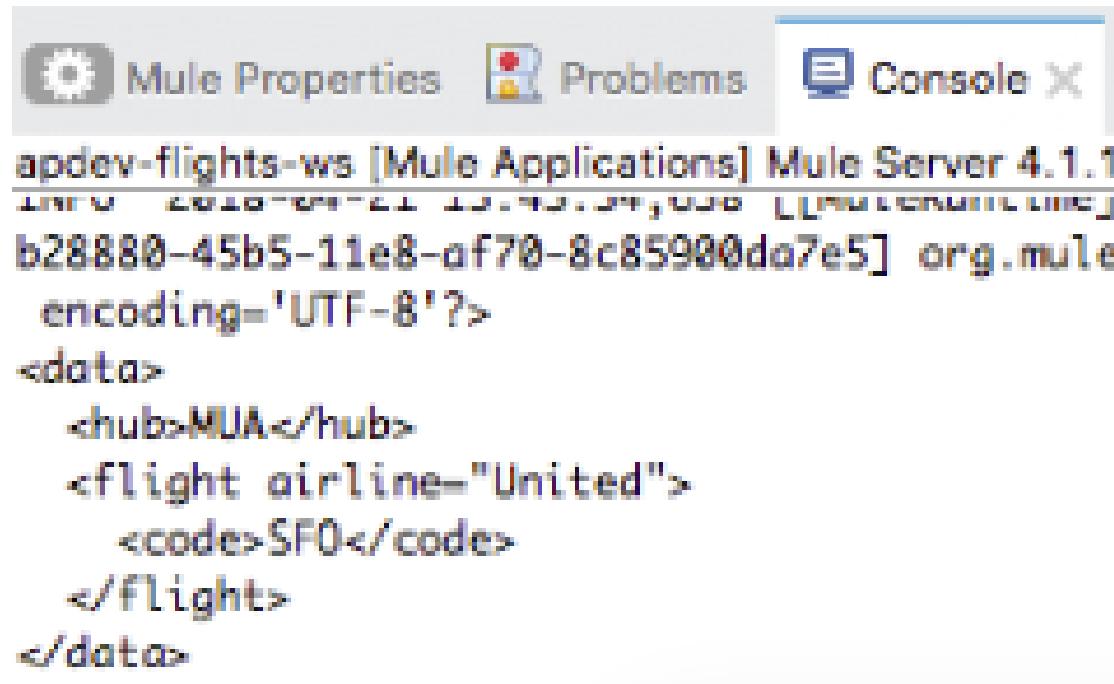


25. Save the file to redeploy the project in debug mode.
26. In Advanced REST Client, post the same request to <http://localhost:8081/api/flights>.
27. In the Mule Debugger, step to the Logger; you should see the transformed JSON payload and the DWoutput variable as a StreamProvider.



28. Step through the application and switch perspectives.

29. Look at the Logger output in the console; you should see the XML.



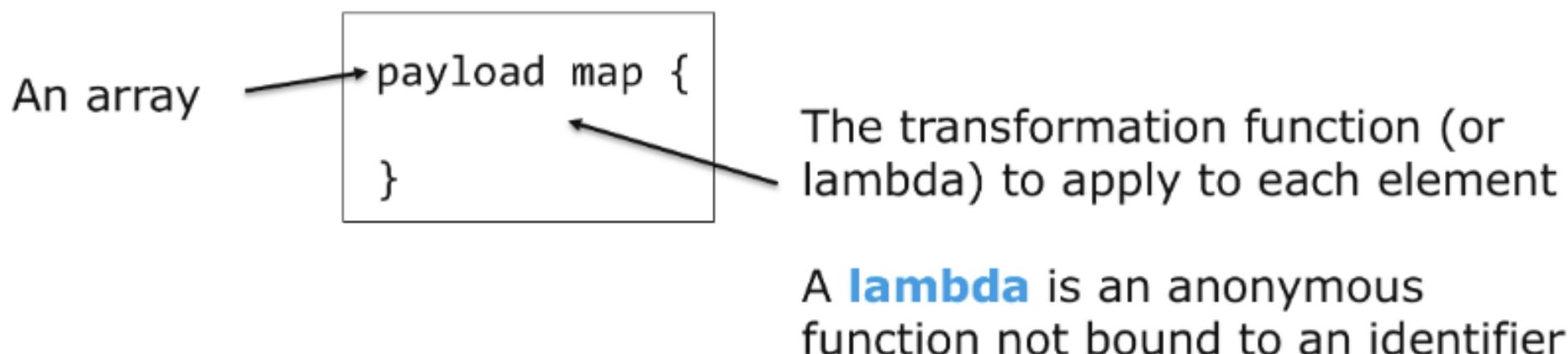
The screenshot shows the Mule Studio interface with the 'Console' tab selected. The output window displays the following XML log:

```
apdev-flights-ws [Mule Applications] Mule Server 4.1.1
b28880-45b5-11e8-af70-8c85900da7e5] org.mule
    encoding='UTF-8'?>
<data>
    <hub>MUA</hub>
    <flight airline="United">
        <code>SF0</code>
    </flight>
</data>
```

Transforming complex data structures with arrays



- 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>

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



- 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

Output Payload Preview

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

The screenshot shows the MuleSoft Anypoint Studio interface. On the left, the DataWeave code is displayed:

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

On the right, the resulting JSON output is shown:

```
[{"flight0": {"airline": "United", "flightCode": "ER38sd", "fromAirportCode": "LAX", "toAirportCode": "SFO", "departureDate": "May 21, 2016", "emptySeats": 0, "totalSeats": 200, "price": 199, "planeType": "Boeing 737"}, {"flight1": {"airline": "Delta", "flightCode": "DL1234", "fromAirportCode": "JFK", "toAirportCode": "HNL", "departureDate": "May 22, 2016", "emptySeats": 10, "totalSeats": 150, "price": 350, "planeType": "Airbus A320"}]
```

Walkthrough 11-3: Transform complex data structures with arrays

In this walkthrough, you create a new flow that allows multiple flights to be posted to it, so you have more complex data with which to work. You will:

- 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 a DataWeave transformation interface. On the left, the 'Payload' tab displays the following DW script:

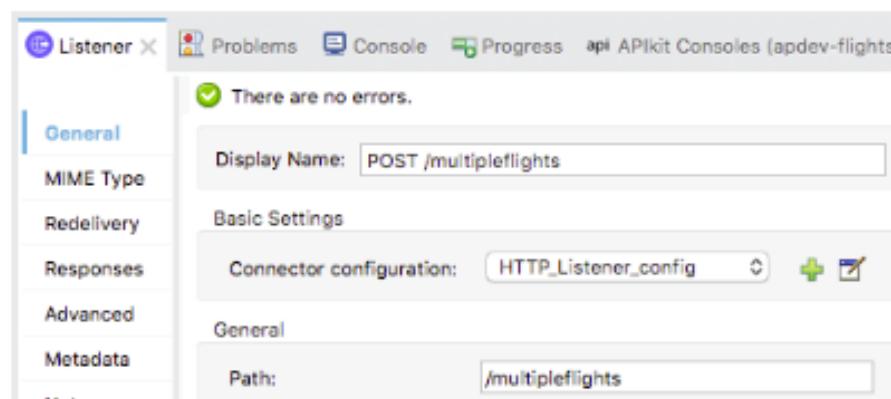
```
1@%dw 2.0
2  output application/dw
3  ---
4 @payload map (object,index) -> [
5   'flight${index}': object
6 ]
```

On the right, the 'Preview' tab shows the resulting JSON array of flight objects:

```
[{"flight0": {"airline": "United", "flightCode": "E838ad", "fromAirportCode": "LAX", "toAirportCode": "SFO", "departureDate": "May 21, 2016", "emptySeats": 0, "totalSeats": 280, "price": 199, "planeType": "Boeing 737"}, "flight1": {"airline": "Delta", "flightCode": "D838ad", "fromAirportCode": "JFK", "toAirportCode": "HNL", "departureDate": "May 22, 2016", "emptySeats": 0, "totalSeats": 280, "price": 299, "planeType": "Boeing 737"}, "flight2": {"airline": "Southwest", "flightCode": "S838ad", "fromAirportCode": "LAX", "toAirportCode": "DFW", "departureDate": "May 23, 2016", "emptySeats": 0, "totalSeats": 280, "price": 199, "planeType": "Boeing 737"}, "flight3": {"airline": "American", "flightCode": "A838ad", "fromAirportCode": "JFK", "toAirportCode": "MIA", "departureDate": "May 24, 2016", "emptySeats": 0, "totalSeats": 280, "price": 299, "planeType": "Boeing 737"}]
```

Create a new flow that receives POST requests of a JSON array of flight objects

1. Return to implementation.xml.
2. Drag out an HTTP Listener from the Mule Palette to the bottom of the canvas.
3. Change the flow name to postMultipleFlights.
4. In the HTTP Listener properties view, set the display name to POST /multipleflights.
5. Set the connector configuration to the existing HTTP_Listener_config.
6. Set the path to /multipleflights and set the allowed methods to POST.

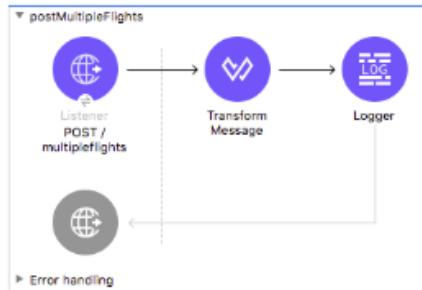


Note: You are bypassing the APIkit router because a corresponding resource/method pair is not defined in the API.

7. Add a Transform Message component to the flow.
8. In the Transform Message properties view, set the expression to the payload.

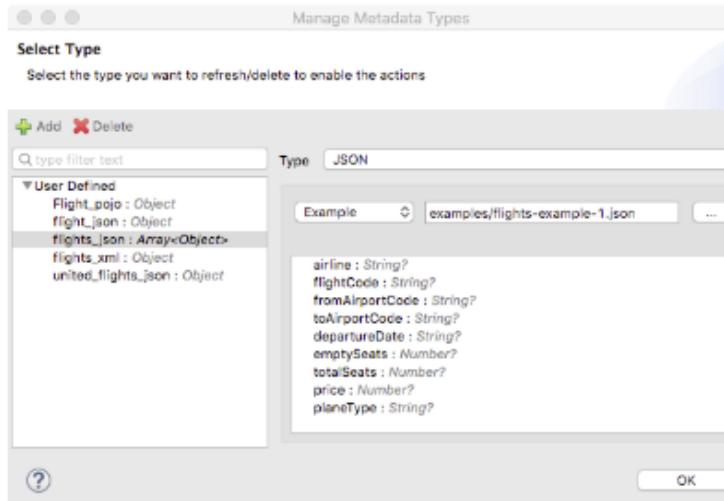


9. Add a Logger to the flow.

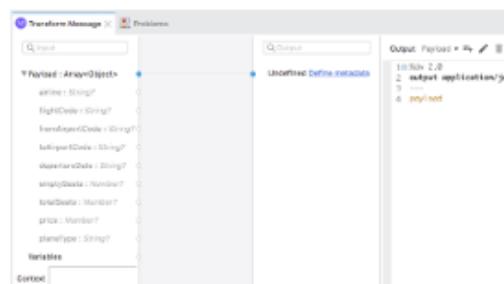


Add input metadata for the transformation

10. In the Transform Message properties view, click Define metadata in the input section.
11. In the Select metadata type dialog box, select flights_json and click Select.

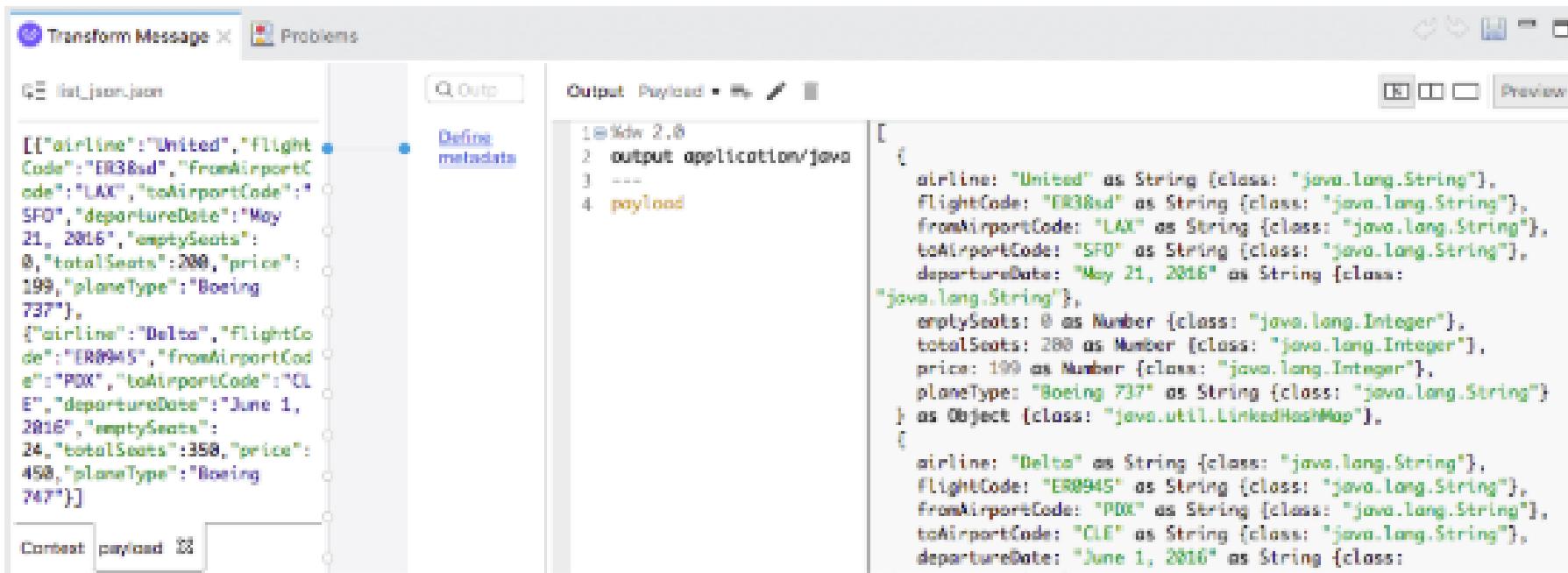


12. In the Transform Message Properties view, you should now see metadata for the input.



Preview sample data and sample output

13. Click the Preview button.
14. In the preview section, click the Create required sample data to execute preview link.
15. Look at the preview section; the sample output should be an ArrayList of LinkedHashMaps.



The screenshot shows the Mule Studio interface with a Transform Message component. On the left, the 'Input' tab displays a JSON array of flight records. The 'Output' tab shows the configuration for the output payload. The 'Payload' section contains Java code that defines the output type as application/java and specifies a payload structure. The 'Preview' tab on the right shows the resulting ArrayList of LinkedHashMaps, where each element represents a flight record with its properties mapped to their respective Java types (String, Integer, etc.).

```
1@%dw 2.0
2  output application/java
3  ---
4  payload
[{"airline": "United", "flightCode": "ER38ad", "fromAirportCode": "LAX", "toAirportCode": "SFO", "departureDate": "May 21, 2016", "emptySeats": 0, "totalSeats": 280, "price": 199, "planeType": "Boeing 737"}, {"airline": "Delta", "flightCode": "ER8945", "fromAirportCode": "PDX", "toAirportCode": "CLT", "departureDate": "June 1, 2016", "emptySeats": 24, "totalSeats": 358, "price": 458, "planeType": "Boeing 747"}]
```

16. Change the output type to application/dw.



Return values for the first object in the collection

17. Change the DataWeave expression to payload[0].

Output Payload ▾ Preview

```
1@%dw 2.0
2 output application/dw
3 ---
4 payload[0]
```

[
airline: "United",
flightCode: "ER38sd",
fromAirportCode: "LAX",
toAirportCode: "SFO",
departureDate: "May 21, 2016",
emptySeats: 0,
totalSeats: 200,
price: 199,
planeType: "Boeing 737"
]

18. Change the DataWeave expression to payload[0].price.

Output Payload ▾ Preview

```
1@%dw 2.0
2 output application/dw
3 ---
4 payload[0].price
```

199

19. Change the DataWeave expression to payload[0].*price.

Output Payload ▾ Preview

```
1@%dw 2.0
2 output application/dw
3 ---
4 payload[0].*price
```

[
199
]

Return collections of values

20. Change the DataWeave expression to return a collection of all the prices.

Output Payload   

1@%dw 2.0
2 output application/dw
3 ---
4 payload.*price

[199,
450]

21. Change the DataWeave expression to payload.price; you should get the same result.

Output Payload    Preview

1@%dw 2.0
2 output application/dw
3 ---
4 payload.price

[199,
450]

22. Change the DataWeave expression to return a collection of prices and available seats.

[payload.price, payload.emptySeats]

Output Payload    Preview

1@%dw 2.0
2 output application/dw
3 ---
4 [payload.price, payload.emptySeats]

[[199,
450],
[0,
24]]

Use the map operator to return object collections with different data structures

23. Change the DataWeave expression to payload map \$.

The screenshot shows the Mule Studio DataWeave editor interface. The code area contains the following DataWeave script:

```
1@ Kdw 2.0
2 output application/dw
3 ---
4 payload map $
```

The preview pane shows the resulting payload as a list of two objects:

```
[{"airline": "United", "flightCode": "ER38sd", "fromAirportCode": "LAX", "toAirportCode": "SFO", "departureDate": "May 21, 2016", "emptySeats": 0, "totalSeats": 200, "price": 199, "planeType": "Boeing 737"}, {"airline": "Delta", "flightCode": "DN38sd"}]
```

24. Change the DataWeave expression to set a property called flight for each object that is equal to its index value in the collection.

The screenshot shows the Mule Studio DataWeave editor interface. The code area contains the following DataWeave script:

```
1@ %dw 2.0
2 output application/dw
3 ---
4@ payload map {
5   flight: $$
6 }
```

The preview pane shows the resulting payload as a list of two objects, each with a 'flight' property set to its index (0 and 1 respectively):

```
[{"flight": 0}, {"flight": 1}]
```

25. Change the DataWeave expression to create a property called destination for each object that is equal to the value of the toAirportCode field in the payload collection.

The screenshot shows the Mule Studio DataWeave editor interface. The code area contains the following DataWeave script:

```
1@ %dw 2.0
2 output application/dw
3 ---
4@ payload map [
5@   flight: $$,
6@   destination: $.toAirportCode
7 ]
```

The preview pane shows the resulting payload as a list of two objects, each with a 'flight' property and a 'destination' property set to the 'toAirportCode' value from the original payload:

```
[{"flight": 0, "destination": "SFO"}, {"flight": 1, "destination": "CLE"}]
```

26. Change the DataWeave expression to dynamically add each of the properties present on the payload objects.

Output Payload ▾ Preview

```
1@%dw 2.0
2 output application/dw
3 ---
4@payload map {
5@   flight: $$,
6     ($$): $
7 }
```

[
 {
 flight: 0,
 "0": {
 airline: "United",
 flightCode: "ER38sd",
 fromAirportCode: "LAX",
 toAirportCode: "SFO",
 departureDate: "May 21, 2016",
 emptySeats: 0,
 totalSeats: 200,
 price: 199,
 planeType: "Boeing 737"
 }
 },
 {
 flight: 1,
 }

27. Remove the first flight property assignment.

28. Change the DataWeave expression so the name of each object is flight+index and you get flight0, flight1, and flight2.

Output Payload ▾ Preview

```
1@%dw 2.0
2 output application/dw
3 ---
4@payload map {
5   'flight$$': $  
6 }
```

[
 {
 flight0: {
 airline: "United",
 flightCode: "ER38sd",
 fromAirportCode: "LAX",
 toAirportCode: "SFO",
 departureDate: "May 21, 2016",
 emptySeats: 0,
 totalSeats: 200,
 price: 199,
 planeType: "Boeing 737"
 }
 },
 {
 flight1: {
 }

Use explicit named parameters with the map operator

29. Change the map operator so that it uses two parameters called object and index and set the field name to just the index value.

Output Payload ▾ Preview

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

[

```
{
  "0": {
    airline: "United",
    flightCode: "ER38sd",
    fromAirportCode: "LAX",
    toAirportCode: "SFO",
    departureDate: "May 21, 2016",
    emptySeats: 0,
    totalSeats: 200,
    price: 199,
    planeType: "Boeing 737"
  },
  [
    "1": {
      airline: "American",
      flightCode: "AA1234",
      fromAirportCode: "JFK",
      toAirportCode: "HNL",
      departureDate: "May 22, 2016",
      emptySeats: 10,
      totalSeats: 150,
      price: 299,
      planeType: "Boeing 777"
    }
  ]
}
```

30. Change the DataWeave expression so the name of each object is flight+index and you get flight0, flight1, and flight2.

Output Payload ▾ Preview

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

[

```
{
  Flight0: {
    airline: "United",
    flightCode: "ER38sd",
    fromAirportCode: "LAX",
    toAirportCode: "SFO",
    departureDate: "May 21, 2016",
    emptySeats: 0,
    totalSeats: 200,
    price: 199,
    planeType: "Boeing 737"
  },
  [
    Flight1: {
      airline: "American",
      flightCode: "AA1234",
      fromAirportCode: "JFK",
      toAirportCode: "HNL",
      departureDate: "May 22, 2016",
      emptySeats: 10,
      totalSeats: 150,
      price: 299,
      planeType: "Boeing 777"
    }
  ]
}
```

Note: If you want to test the application, change the output type to application/json and then in Advanced REST Client, make a request to <http://localhost:8081/multipleflights> and replace the request body with JSON from the flights-example.json file.

Change the expression to output different data types

31. Change the DataWeave expression output type from application/dw to application/json.

Output Payload ▾ ⌂ ⌂ Preview

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

[

```
{
  "flight0": {
    "airline": "United",
    "flightCode": "ER38sd",
    "fromAirportCode": "LAX",
    "toAirportCode": "SFO",
    "departureDate": "May 21, 2016",
    "emptySeats": 0,
    "totalSeats": 200,
    "price": 199,
    "planeType": "Boeing 737"
  }
}
```

32. Change the output type to application/java.

Output Payload ▾ ⌂ ⌂ Preview

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

[

```
{
  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"},
```

[

```

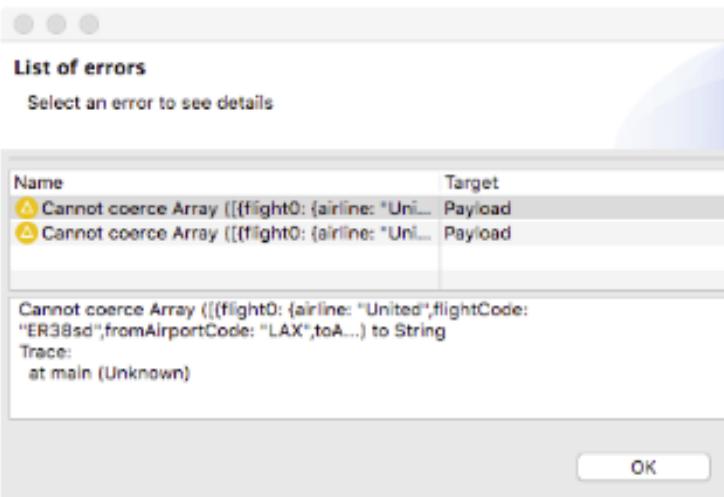
  flight1: {
```

33. Change the output type to application/xml; you should get an issue displayed.

```
Output Payload ▾ ⌂ 2 issues found
1@%dw 2.0
2   output application/xml
3   ---
4@payload map (object,index) -> {
5     'flight$(index)': object
6 }
```

```
[ 
  {
    flight0:
    airline
    flight(
    fromAir
    toAirnr
```

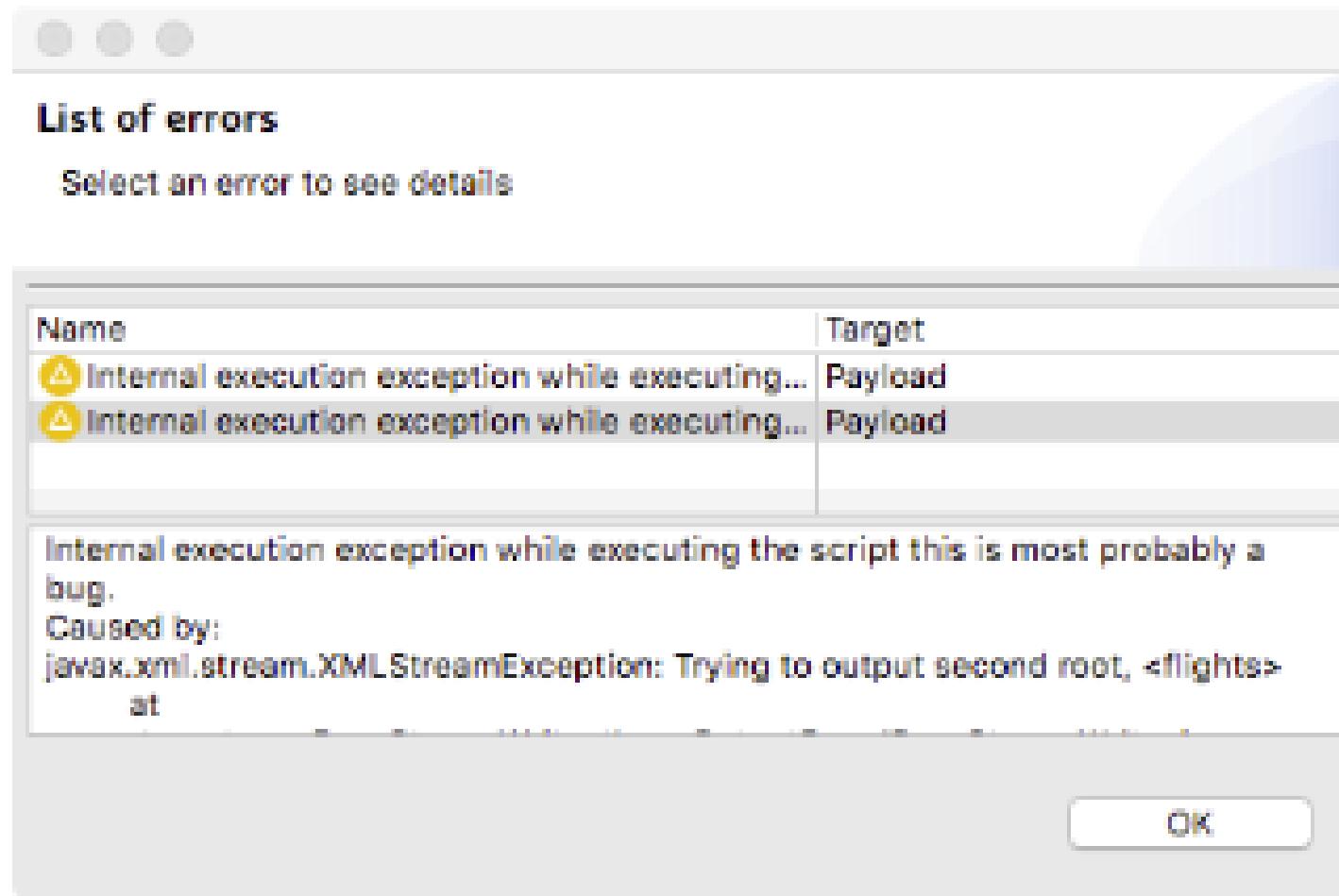
34. Click the warning icon and review the issues.



35. Change the DataWeave expression to create a root node called flights; you should still get an issue.

```
Output Payload ▾ ⌂ 2 issues found
1@%dw 2.0
2   output application/xml
3   ---
4@flights: payload map (object,index) -> {
5   'flight$(index)': object
6 }
```

36. Click the warning icon and review the issues.



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"}]	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



- 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 “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 displays the same JSON structure in a single-line, compact format:

```
1 {  
2   "users": {  
3     "user": {  
4       "lastname": "Mule"  
5     }  
6   }  
7 }
```

The Preview tab shows a visual representation of the JSON data.

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, there's an 'Input' section containing an XML file named 'listAllFlightsResponse.xml'. The XML content is as follows:

```
<ns2:listAllFlightsResponse
  xmlns:ns2="http://
  soap.training.mulesoft.com/">
  <return airlineName="United">
    <code>A1B2C3</
    code><departureDate>2015/10/20</
    departureDate>
    <destination>SFO</
    destination><emptySeats>40</
    emptySeats>
  </return>
</ns2:listAllFlightsResponse>
```

The main area is titled 'Output Payload' and contains the following MEL (Message EL) code:

```
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
```

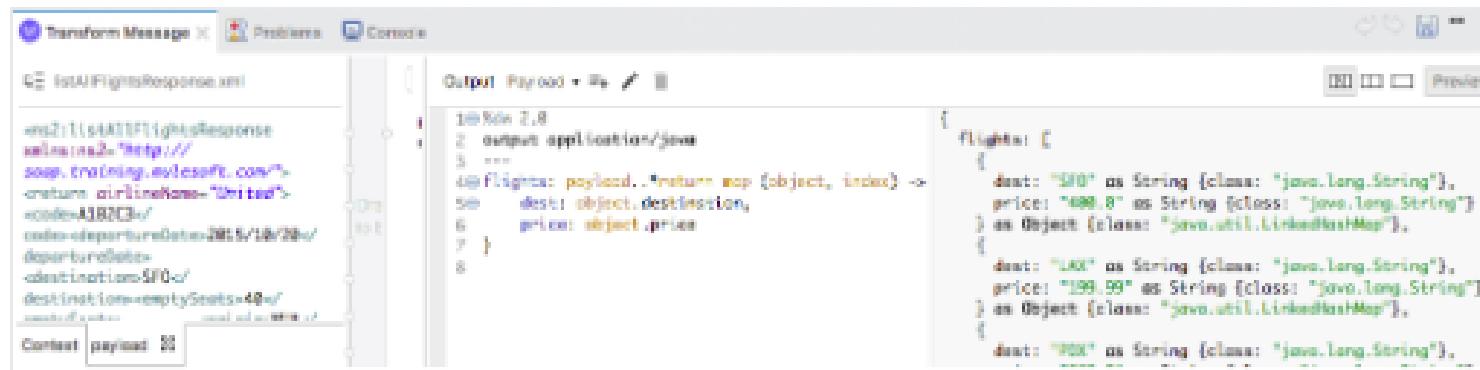
To the right of the code, the resulting JSON output is shown:

```
{
  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: "199.99" as String {class: "java.lang.String"}
    } as Object {class: "java.util.LinkedHashMap"}
  ]
}
```

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

In this walkthrough, you continue to work with the JSON data for multiple flights posted to the flow. You will:

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



Transform the JSON array of objects to XML

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Change the expression to map each item in the input array to an XML element.

```
flights: {(payload map (object,index) -> {  
    'flight${index}': object  
})}
```

3. Look at the preview; the JSON should be transformed to XML successfully.

Output Payload     Preview

1@ %dw 2.0 2 output application/xml 3 --- 4@ Flights: {(payload map (object,index) -> { 5 'flight\${index}': object 6 }) 7)}	<?xml version='1.0' encoding='UTF-8'?> <flights> <flight0> <airline>United</airline> <flightCode>ER38sd</flightCode> <fromAirportCode>LAX</fromAirportCode> <toAirportCode>SFO</toAirportCode> <departureDate>May 21, 2016</departureDate> <emptySeats>0</emptySeats> <totalSeats>200</totalSeats> <price>199</price> <planeType>Boeing 737</planeType> </flight0> <flight1> <airline>Delta</airline> <flightCode>DN12345</flightCode> <fromAirportCode>JFK</fromAirportCode> <toAirportCode>ORD</toAirportCode> <departureDate>May 22, 2016</departureDate> <emptySeats>0</emptySeats> <totalSeats>200</totalSeats> <price>229</price> <planeType>Boeing 737</planeType> </flight1> </flights>
--	---

4. Modify the expression so the flights object has a single property called flight.

The screenshot shows a Java IDE interface with two panes. The left pane displays Java code, and the right pane shows the generated XML output.

Java Code (Left Pane):

```
1@ %dw 2.0
2  output application/xml
3  ---
4@ flights: {(payload map (object,index) -> {
5    flight: object
6  })
7 })
```

XML Output (Right Pane):

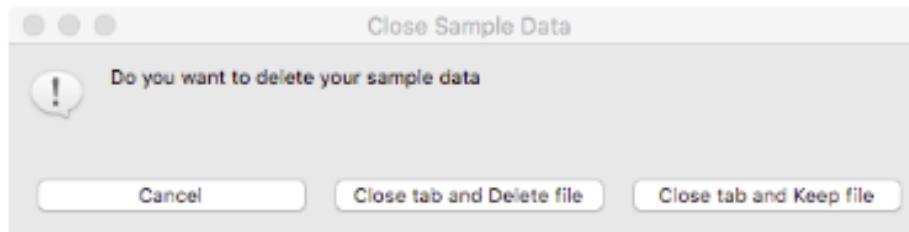
```
<?xml version='1.0' encoding='UTF-8'?>
<flights>
  <flight>
    <airline>United</airline>
    <flightCode>ER38sd</flightCode>
    <fromAirportCode>LAX</fromAirportCode>
    <toAirportCode>SFO</toAirportCode>
    <departureDate>May 21, 2016</departureDate>
    <emptySeats>0</emptySeats>
    <totalSeats>200</totalSeats>
    <price>199</price>
    <planeType>Boeing 737</planeType>
  </flight>
  <flight>
    <airline>Delta</airline>
```

Change the input metadata to XML

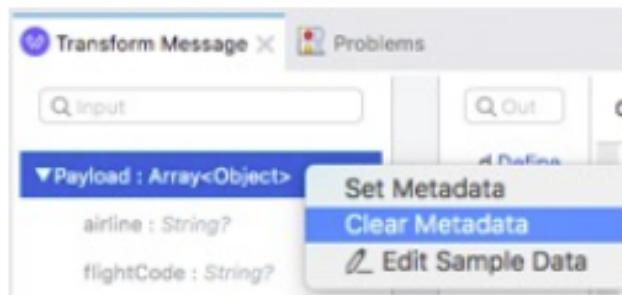
- In the input section, click the x on the payload tab to close it.



- In the Close Sample Data dialog box, click Close tab and Keep file.

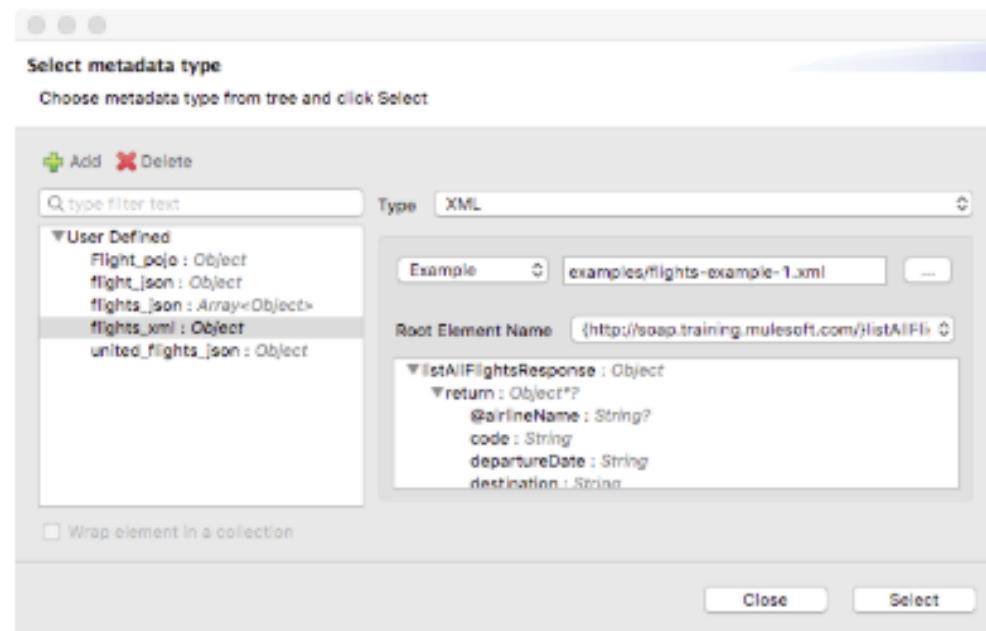


- In the input section, right-click Payload and select Clear Metadata.

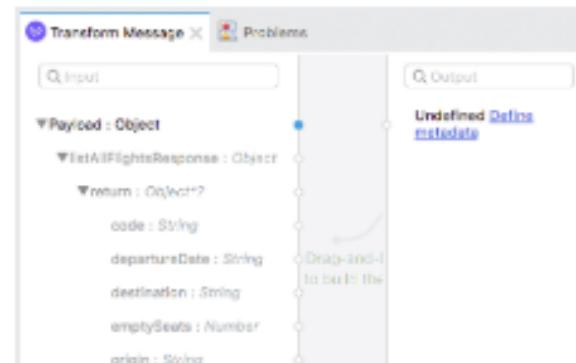


- Click the Define metadata link.

9. In the Select metadata type dialog box, select flights_xml and click Select.

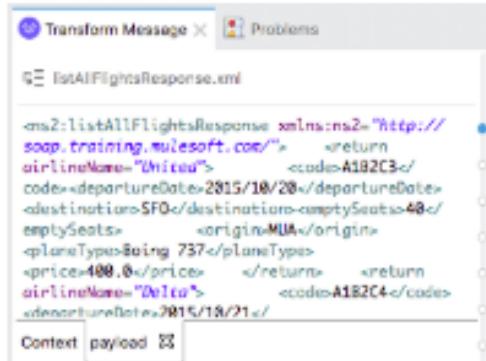


10. In the Transform Message Properties view, you should now see new metadata for the input.



Preview sample data and sample output

- In the preview section, click the Create required sample data to execute preview link.
- Look at the XML sample payload in the input section.

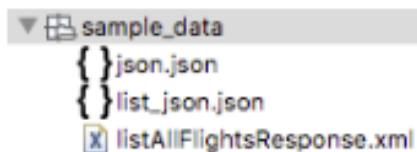


The screenshot shows the Mule Studio interface with the 'Transform Message' tab selected. A file named 'listAllFlightsResponse.xml' is open. The XML content is as follows:

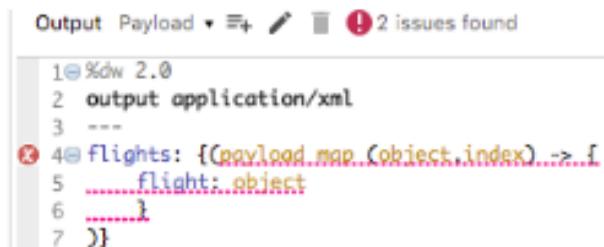
```
<ns2:listAllFlightsResponse xmlns:ns2="http://soap.training.mulesoft.com/">    <return>        <airlineName>United</airlineName>        <code>A1B2C3</code>        <departureDate>2015/10/20</departureDate>        <destination>SFO</destination>        <emptySeats>48</emptySeats>        <origin>MIA</origin>        <planeType>Boeing 737</planeType>        <price>400.0</price>    </return>    <return>        <airlineName>Delta</airlineName>        <code>A1B2C4</code>        <departureDate>2015/10/21</departureDate>    </return></ns2:listAllFlightsResponse>
```

At the bottom of the editor, there are tabs for 'Context' and 'payload'. The 'payload' tab is currently selected.

- Look at the sample_data folder in src/test/resources.



- Review the transformation expression; you should get an error.

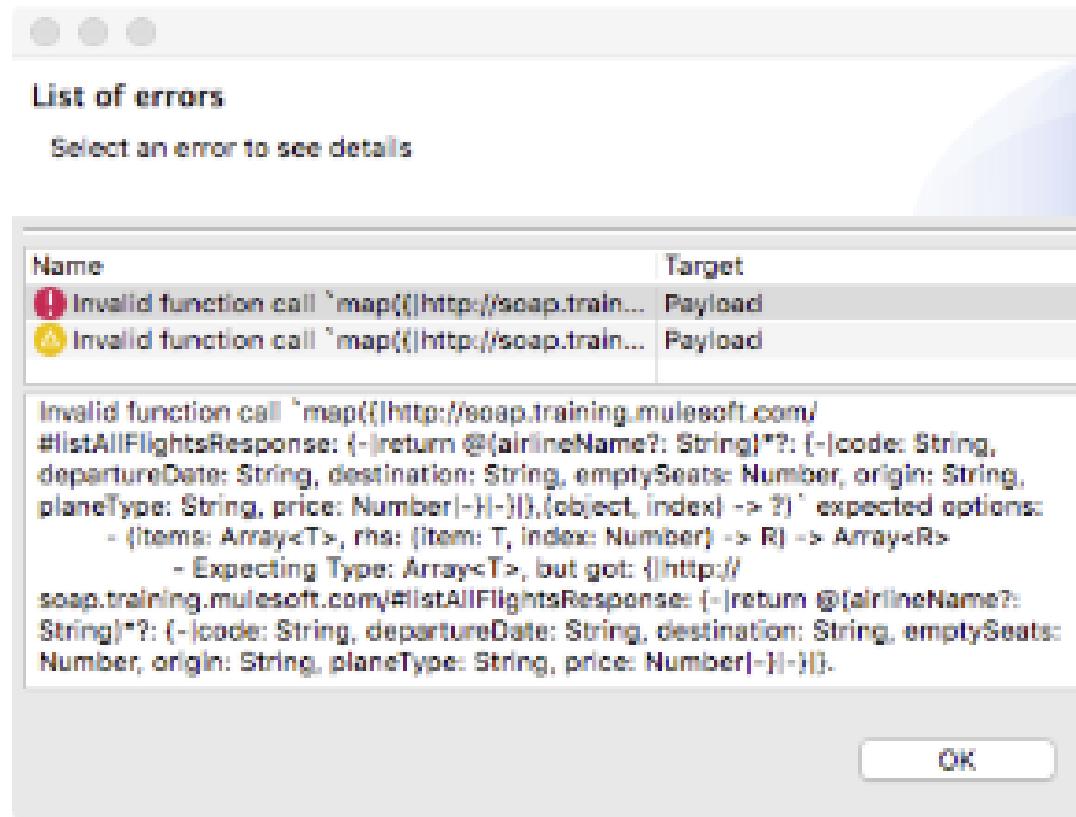


The screenshot shows the 'Output' tab of the transformation editor. The transformation code is as follows:

```
1@%dw 2.0
2  output application/xml
3  ---
4  flights: {payload map (object, index) -> {
5      flight: object
6      }
7 }}
```

A red error icon is positioned next to the line '4 flights: {payload map (object, index) -> {'. The status bar at the top right indicates '2 issues found'.

15. Look at the error.



Transform XML with repeated elements to a different XML structure

16. Look at the payload metadata in the input section.

The screenshot shows the 'Transform Message' editor in Mule Studio. The 'Input' tab is selected. The payload structure is displayed as follows:

```
Payload : Object
  listAllFlightsResponse : Object
    return : Object??
      code : String
      departureDate : String
```

17. Change the DataWeave expression so that the map is iterating over payload..return.

The screenshot shows the DataWeave editor in Mule Studio. The output type is set to 'Payload' and 'application/xml'. The transformation logic is as follows:

```
1@ Xdw 2.0
2  output application/xml
3  ---
4@ flights: {(payload..return map (object,index) -> {
5    flight: object
6    }
7 )}
```

The preview pane shows the resulting XML output:

```
<?xml version='1.0' encoding='UTF-8'?>
<flights>
  <flight>
    <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>
  </flight>
</flights>
```

18. Change the DataWeave expression so that the map is iterating over the repeated return elements of the input.

Output Payload   

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

<?xml version='1.0' encoding='UTF-8'?>
<flights>
 <flight>
 <code>A1B2C3</code>
 <departureDate>2015/10/20</departureDate>
 <destination>SFO</destination>
 <emptySeats>40</emptySeats>
 <origin>MIA</origin>
 <planeType>Boing 737</planeType>
 <price>400.0</price>
 </flight>
 <flight>
 <code>A1B2C4</code>
 <departureDate>2015/10/21</departureDate>
 </flight>
</flights>

19. Change the DataWeave expression to return flight elements with dest and price elements that map to the corresponding de values.

Output Payload   

1@%dw 2.0
2 output application/xml
3 ---
4@ flights: {payload..*return map (object,index) -> {
5 flight: {
6 dest: object.destination,
7 price: object.price
8 }
9 }
10 }
11 }

<?xml version='1.0' encoding='UTF-8'?>
<flights>
 <flight>
 <dest>SFO</dest>
 <price>400.0</price>
 </flight>
 <flight>
 <dest>LAX</dest>
 <price>199.99</price>
 </flight>
</flights>

Note: If you want to test the application with Advanced REST Client, be sure to change the content-type header to apply the request body with XML from the flights-example.xml file.

Change the expression to output different data types

20. Change the output type from application/xml to application/dw.

Output Payload ▾  

```
1@%dw 2.0
2  output application/dw
3  ---
4@ flights: {$(payload..*)return map (object,index) -> {
5@   flight: {
6@     dest: object.destination,
7@     price: object.price
8@   }
9@ }
10 }}
```

Preview

```
{ "flights": [
  "flight": {
    "dest": "SFO",
    "price": "400.0"
  },
  "flight": {
    "dest": "LAX",
    "price": "199.99"
  }
], ... }
```

21. Change the output type to application/json.

Output Payload ▾  

```
1@%dw 2.0
2  output application/json
3  ---
4@ flights: {$(payload..*)return map (object,index) -> {
5@   flight: {
6@     dest: object.destination,
7@     price: object.price
8@   }
9@ }
10 }}
```

Preview

```
{ "flights": [
  "flight": {
    "dest": "SFO",
    "price": "400.0"
  },
  "flight": {
    "dest": "LAX",
    "price": "199.99"
  }
], ... }
```

22. Change the output type to application/java.

Output Payload ▾  

```
1@%dw 2.0
2  output application/java
3  ---
4@ flights: {$(payload..*)return map (object,index) -> {
5@   flight: {
6@     dest: object.destination,
7@     price: object.price
8@   }
9@ }
10 }}
```

Preview

```
{ "flights": [
  "flight": {
    "dest": "PDX" as String {class: "java.lang.String"},  
    "price": "283.0" as String {class: "java.lang.String"}  
  } as Object {class: "java.util.LinkedHashMap"}  
} as Object {class: "java.util.LinkedHashMap"}  
} as Object {class: "java.util.LinkedHashMap", encoding:  
"UTF-8", mimetype: "*/*", raw: [
  "flights": [
    "flight": {
      "dest": "PDX" as String {class: "java.lang.String"},  
      "price": "283.0" as String {class: "java.lang.String"}  
    } as Object {class: "java.util.LinkedHashMap"}  
  } as Object {class: "java.util.LinkedHashMap"}  
} as Object {class: "java.util.LinkedHashMap"}  
]
```

23. In the DataWeave expression, remove the {{ }} around the map expression.

Output Payload

1@%dw 2.0
2 output application/java
3 ---
4@ flights: payload.*return map {object,index} -> {
5@ flight: {
6@ dest: object.destination,
7@ price: object.price
8@ }
9@ }
10

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

24. Change the DataWeave expression to remove the flight property; you should get an ArrayList with five LinkedHashMaps.

Output Payload

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"}
 } 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"}
] as Array {class: "java.util.ArrayList"}
] as Object {class: "java.util.LinkedHashMap", encoding:

Defining and using variables and functions



- 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
{"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) }	

- 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



- 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 a code editor interface with two panes. The left pane displays Java code, and the right pane shows the resulting JSON output.

Java Code:

```
Output Payload ▾ + Preview
1@ Xdw 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
19 Flights
```

Output (JSON):

```
[
  {
    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"
  }
],
```

Walkthrough 11-5: Define and use variables and functions

In this walkthrough, you continue to work with the DataWeave transformation in postMultipleFlights. You will:

- 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 a DataWeave 2.0 editor interface. On the left, the code pane displays the following DataWeave script:

```
1@ Rdw 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
19 flights
```

On the right, the preview pane shows the resulting JSON output:

```
[{"dest": "SFO", "price": "400.0", "totalSeats": 150, "plane": "Boeing 737"}, {"dest": "LAX", "price": "190.00", "totalSeats": 150, "plane": "Boeing 737"}, {"dest": "PDX", "price": "283.0", "totalSeats": 300, "plane": "Boeing 777"}]
```

Define and use a global constant

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Change the output type to application/dw.
3. In the header, define a global variable called numSeats that is equal to 400.

```
var numSeats = 400
```

4. In the body, add a totalSeats property that is equal to the numSeats constant.

```
totalSeats: numSeats
```

```
4 var numSeats = 400
5 ---
6@ flights: payload..*return map (object,index) -> {
7@   dest: object.destination,
8@   price: object.price,
9@   totalSeats: numSeats
10 }
```

5. Look at the preview.



The screenshot shows the 'Output' tab of the Transformation Message editor. The left pane displays the XSD schema (Xdw 2.0), and the right pane shows the resulting JSON payload. The payload contains two flight objects, each with destination, price, and totalSeats properties set to 400.

```
1@ Xdw 2.0
2  output application/dw
3
4 var numSeats = 400
5 ---
6@ flights: payload..*return map (object,index) -> {
7@   dest: object.destination,
8@   price: object.price,
9@   totalSeats: numSeats
10 }
11
```

```
{
  flights: [
    {
      dest: "SFO",
      price: "400.0",
      totalSeats: 400
    },
    {
      dest: "LAX",
      price: "199.99",
      totalSeats: 400
    }
  ]
}
```

6. Comment out the variable declaration in the header.

```
//var numSeats = 400
```

Define and use a global variable that is equal to a lambda expression that maps to a constant

7. In the header, define a global variable called numSeats that is equal to a lambda expression with an input parameter x equal to 400.
8. Inside the expression, set numSeats to x.

```
var numSeats = (x=400) -> x
```

9. Add parentheses after numSeats in the totalSeats property assignment.

```
totalSeats: numSeats()
```

10. Look at the preview.

The screenshot shows a JSON editor interface. On the left, there is a code editor pane containing the following Groovy-like pseudocode:

```
1@ %dw 2.0
2  output application/dw
3
4 //var numSeats = 400
5 var numSeats = (x = 400) -> x
6 ---
7@ flights: payload..*return map (object,index) -> {
8@   dest: object.destination,
9    price: object.price,
10   totalSeats: numSeats()
11 }
12
```

On the right, there is a preview pane showing the resulting JSON output:

```
{
  flights: [
    {
      dest: "SFO",
      price: "400.0",
      totalSeats: 400
    },
    {
      dest: "LAX",
      price: "199.99",
      totalSeats: 400
    }
  ]
}
```

Add a plane property to the output

11. Add a plane property to the DataWeave expression equal to the value of the input planeType values.
12. Look at the values of plane type in the preview.

Output Payload •  

1@%dw 2.0
2 output application/dw
3
4 //var numSeats = 400
5 var numSeats = (x = 400) -> x
6 ---
7@flights: payload..*return map (object,index) -> {
8@ dest: object.destination,
9 price: object.price,
10 totalSeats: numSeats(),
11 plane: object.planeType
12 }

[{"dest": "LAX", "price": "199.99", "totalSeats": 400, "plane": "Boing 737"}, {"dest": "PDX", "price": "283.0", "totalSeats": 400, "plane": "Boing 777"}]

Define and use a global variable that is equal to a lambda expression with input parameters

13. Comment out the numSeats variable declaration in the header.

14. In the header, define a global variable called numSeats that is equal to a lambda expression with an input parameter called planeType of type String

```
var numSeats = (planeType: String) ->
```

15. Inside the expression, add an if/else block that checks to see if planeType contains the string 737 and sets numSeats to 150 or 300.

```
6 var numSeats = (planeType: String) ->
7   if (planeType contains('737'))
8     150
9   else
10    300
```

16. Change the totalSeats property assignment to pass the object's planeType property to numSeats.

```
totalSeats: numSeats(object.planeType)
```

17. Force the argument to a String.

```
totalSeats: numSeats(object.planeType as String)
```

18. Look at the preview; you should see values of 150 and 300.

The screenshot shows a Dataweave editor interface with two main sections: 'Output' and 'Payload'.

Output:

```
%dw 2.0
output application/dw
//var numSeats = 400
//var numSeats = (x = 400) -> x
var numSeats = (planeType: String) ->
  if (planeType contains('737'))
    150
  else
    300
---
flights: payload..*return map (object,index) -> {
  dest: object.destination,
  price: object.price,
  totalSeats: numSeats(object.planeType as String).
```

Payload:

```
[{"dest": "LAX", "price": "199.99", "totalSeats": 150, "plane": "Boeing 737"}, {"dest": "PDX", "price": "283.0", "totalSeats": 300, "plane": "Boeing 737"}]
```

Define and use a lambda expression assigned to a variable as a function

20. In the header, use the fun keyword to define a function called getNumSeats with a parameter

```
fun getNumSeats(planeType: String)
```

21. Copy the if/else expression in the numSeats declaration.

22. Set the getNumSeats function equal to the if/else expression.

```
fun getNumSeats(planeType: String) =
```

```
    if (planeType contains('737'))
```

```
        150
```

```
    else
```

```
        300
```

23. In the body, change the totalSeats property to be equal to the result of the getNumSeats func

```
totalSeats: getNumSeats(object.planeType as String)
```

24. Look at the preview.

Output Payload •

Preview

```
6⑥ /* var numSeats = (planeType: String) ->
7      if (planeType contains('737'))
8          150
9      else
10         300
11 */
12⑥ fun getNumSeats (planeType: String) ->
13      if (planeType contains('737'))
14          150
15      else
16         300
17 ----
18⑥ flights: payload..*return map (object,index) -> {
19      dest: object.destination,
20      price: object.price,
21      totalSeats: getNumSeats(object.planeType as String),
22      plane: object.planeType
23 }
```

flights: [

{

dest: "SFO",
price: "480.0",
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"

Create and use a local variable

25. In the body, surround the existing DataWeave expression with parentheses and add the using keyword in front of it.

```
17  ===
18@using flights: payload..*return map (object,index) -> {
19@    dest: object.destination,
20    price: object.price,
21    totalSeats: getNumSeats(object.planeType as String),
22    plane: object.planeType
23 }
24 }
```

26. Change the colon after flights to an equal sign.
27. Modify the indentation to your liking.
28. After the local variable declaration, set the transformation expression to be the local flights variable.

```
18@using flights =
19@  payload..*return map (object,index) -> {
20@    dest: object.destination,
21    price: object.price,
22    totalSeats: getNumSeats(object.planeType as String),
23    plane: object.planeType
24 }
25 )
26
27 Flights
```

29. Look at the preview.

Output Payload  

   Preview

```
10      300
11  */
12 fun getNumSeats (planeType: String) =
13     if (planeType contains('737'))
14         150
15     else
16         300
17 ---
18 using (flights =
19     payload..*return map (object,index) -> {
20         dest: object.destination,
21         price: object.price,
22         totalSeats: getNumSeats(object.planeType as String),
23         plane: object.planeType
24     }
25 )
26
27 flights
```

```
[ {
  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"
}
```

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

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

- 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

Output Payload ▾ Preview

```
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
```

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"}
,

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

In this walkthrough, you continue to work with the XML flights data posted to the postMultipleFlights flow. You will:

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



The screenshot shows a Mule Studio interface with the following components:

- Output Payload:** A code editor window containing Java code for processing flight data.
- Preview:** A window showing the resulting JSON payload.

Code (Output Payload):

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

Preview (Resulting JSON):

```
[  
  {  
    dest: "SFO",  
    price: "$488.88" as String {format: "$##.##"},  
    totalSeats: 150,  
    plane: "BOING 737",  
    date: "Oct 28, 2015" as String {format: "MMM dd, yyyy"}  
  },  
  {  
    dest: "LAX",  
    price: "$199.99" as String {format: "$##.##"},  
    totalSeats: 150,  
    plane: "BOING 737",  
    date: "Oct 21, 2015" as String {format: "MMM dd, yyyy"}  
  }]
```

Format a string

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Use the upper function to return the plane value in uppercase.

```
18 using (flights -  
19   payload.*return map (object, index) -> {  
20     dest: object.destination,  
21     price: object.price,  
22     totalSeats: getNumSeats(object.planeType as String),  
23     plane: upper(object.planeType)  
24   }  
25 }
```

3. Coerce the argument to a String.

plane: upper(object.planeType as String)

4. Look at the preview.

```
Output Payload ▾ + Preview
```

```
15   else  
16     300  
17 ---  
18 using (flights -  
19   payload.*return map (object, index) -> {  
20     dest: object.destination,  
21     price: object.price,  
22     totalSeats: getNumSeats(object.planeType as String),  
23     plane: upper(object.planeType as String)  
24   }  
25 }  
26  
27 flights
```

```
[  
{  
  dest: "SFO",  
  price: "400.0",  
  totalSeats: 150,  
  plane: "BOING 737"  
},  
{  
  dest: "LAX",  
  price: "199.99",  
  totalSeats: 150,  
  plane: "BOING 737"  
},
```

Coerce a string to a number

5. In the preview, look at the data type of the prices; you should see that they are strings.
6. Change the DataWeave expression to use the as keyword to return the prices as numbers.

```
price: object.price as Number,
```

7. Look at the preview.

Output Payload •

Preview

```
15   else
16     300
17 ---
18@using (flights =
19@  payload..*return map {object,index} -> {
20@    dest: object.destination,
21    price: object.price as Number,
22    totalSeats: getNumSeats(object.planeType as String),
23    plane: upper(object.planeType as String)
24  }
25 )
26
27 flights
```

[
 {
 dest: "SFO",
 price: 400.0,
 totalSeats: 150,
 plane: "BOING 737"
 },
 {
 dest: "LAX",
 price: 199.99,
 totalSeats: 150,
 plane: "BOING 737"
 },
]

8. Change the output type from application/dw to application/java.
9. Look at the preview; you should see the prices are now either Integer or Double objects.

Output Payload •

Preview

```
12@ fun getNumSeats (planeType: String) ->
13@   if (planeType contains('737'))
14@     150
15@   else
16@     300
17 ---
18@using (flights =
19@  payload..*return map {object,index} -> {
20@    dest: object.destination,
21    price: object.price as Number,
22    totalSeats: getNumSeats(object.planeType as String),
23    plane: upper(object.planeType as String)
24  }
25 )
26
27 flights
```

[
 {
 dest: "SFO" as String {class: "java.lang.String"},
 price: 400 as Number {class: "java.lang.Integer"},
 totalSeats: 150 as Number {class: "java.lang.Integer"},
 plane: "BOING 737" as String {class: "java.lang.String"}
 } as Object {class: "java.util.LinkedHashMap"},
 {
 dest: "LAX" as String {class: "java.lang.String"},
 price: 199.99 as Number {class: "java.lang.Double"},
 totalSeats: 150 as Number {class: "java.lang.Integer"},
 plane: "BOING 737" as String {class: "java.lang.String"}
 } as Object {class: "java.util.LinkedHashMap"},
 {
 dest: "PDX" as String {class: "java.lang.String"},
 price: 283 as Number {class: "java.lang.Integer"},
 }

Coerce a string to a specific type of number object

10. Change the DataWeave expression to use the class metadata key to coerce the prices to java.lang.Double objects.

```
price: object.price as Number {class:"java.lang.Double"},
```

11. Look at the preview; you should see the prices are now all Double objects.

The screenshot shows the Mule Studio DataWeave editor. On the left, the code is displayed:

```
12@ fun getNumSeats (planeType: String) =  
13@   if (planeType contains('737'))  
14@     150  
15@   else  
16@     300  
17 ---  
18@ using (flights =  
19@   payload..*return map {object,index} -> {  
20@     dest: object.destination,  
21@     price: object.price as Number {class: "java.lang.Double"},  
22@     totalSeats: getNumSeats(object.planeType as String),  
23@     plane: upper(object.planeType as String)  
24@   }  
25 )  
26 flights
```

On the right, the preview pane shows the resulting JSON output:

```
[  
  {  
    dest: "SFO" as String {class: "java.lang.String"},  
    price: 400.0 as Number {class: "java.lang.Double"},  
    totalSeats: 150 as Number {class: "java.lang.Integer"},  
    plane: "BOING 737" as String {class: "java.lang.String"}  
  } as Object {class: "java.util.LinkedHashMap"},  
  {  
    dest: "LAX" as String {class: "java.lang.String"},  
    price: 199.99 as Number {class: "java.lang.Double"},  
    totalSeats: 150 as Number {class: "java.lang.Integer"},  
    plane: "BOING 737" as String {class: "java.lang.String"}  
  } as Object {class: "java.util.LinkedHashMap"},  
  {  
    dest: "PDX" as String {class: "java.lang.String"},  
    price: 283.0 as Number {class: "java.lang.Double"},  
  } as Object {class: "java.util.LinkedHashMap"}]
```

12. Change the output type from application/java back to application/dw.

The screenshot shows the Mule Studio DataWeave editor. The code is identical to the previous screenshot:

```
12@ fun getNumSeats (planeType: String) =  
13@   if (planeType contains('737'))  
14@     150  
15@   else  
16@     300  
17 ---  
18@ using (flights =  
19@   payload..*return map {object,index} -> {  
20@     dest: object.destination,  
21@     price: object.price as Number {class: "java.lang.Double"},  
22@     totalSeats: getNumSeats(object.planeType as String),  
23@     plane: upper(object.planeType as String)  
24@   }  
25 )  
26 flights
```

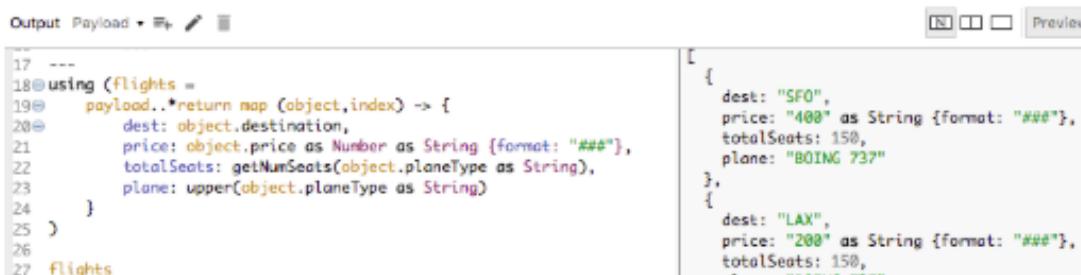
The output type is set to "application/dw" in the top bar, which changes the preview to show the raw DataWeave code instead of JSON.

Format a number

13. Remove the coercion of the price to a java.lang.Double.
14. Coerce the price to a String and use the format schema property to format the prices to zero decimal places.

```
price: object.price as Number as String {format: "##"},
```

15. Look at the preview; the prices should now be formatted.



The screenshot shows the DataWeave preview window with the following code:

```
17 ---  
18@using (flights =  
19@  payload..*return map (object,index) -> {  
20@    dest: object.destination,  
21@    price: object.price as Number as String {format: "##"},  
22@    totalSeats: getNumSeats(object.planeType as String),  
23@    plane: upper(object.planeType as String)  
24  }  
25 )  
26 flights
```

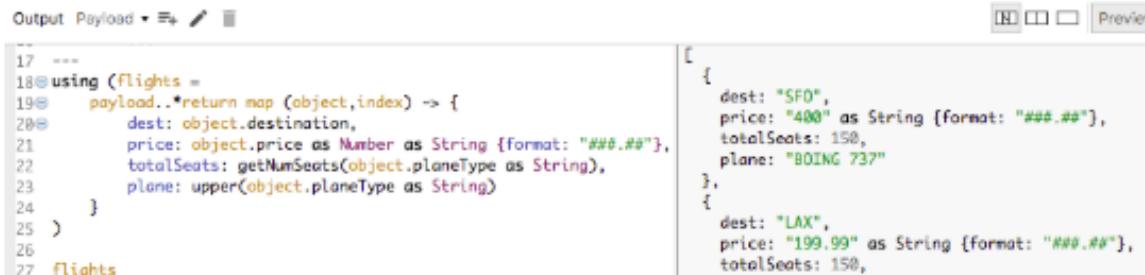
The preview pane displays the resulting JSON output:

```
[  
  {  
    dest: "SFO",  
    price: "400" as String {format: "##"},  
    totalSeats: 150,  
    plane: "BOING 737"  
  },  
  {  
    dest: "LAX",  
    price: "200" as String {format: "##"},  
    totalSeats: 150,  
    plane: "Airbus A320"  
  }]
```

16. Change the DataWeave expression to format the prices to two decimal places.

```
price: object.price as Number as String {format: "##.##"},
```

17. Look at the preview.



The screenshot shows the DataWeave preview window with the following code:

```
17 ---  
18@using (flights =  
19@  payload..*return map (object,index) -> {  
20@    dest: object.destination,  
21@    price: object.price as Number as String {format: "##.##"},  
22@    totalSeats: getNumSeats(object.planeType as String),  
23@    plane: upper(object.planeType as String)  
24  }  
25 )  
26 flights
```

The preview pane displays the resulting JSON output:

```
[  
  {  
    dest: "SFO",  
    price: "400" as String {format: "##.##"},  
    totalSeats: 150,  
    plane: "BOING 737"  
  },  
  {  
    dest: "LAX",  
    price: "199.99" as String {format: "##.##"},  
    totalSeats: 150,  
    plane: "Airbus A320"  
  }]
```

18. Change the DataWeave expression to format the prices to two minimal decimal places.

```
price: object.price as Number as String {format: "##.00"},
```

19. Look at the preview; all the prices should now be formatted to two decimal places.

Output Payload    

[
 {
 dest: "SFO",
 price: "400.00" as String {format: "###.##"},
 totalSeats: 150,
 plane: "BOING 737"
 },
 {
 dest: "LAX",
 price: "199.99" as String {format: "###.##"},
 totalSeats: 150,
 plane: "ATR 72"
 }]
flights

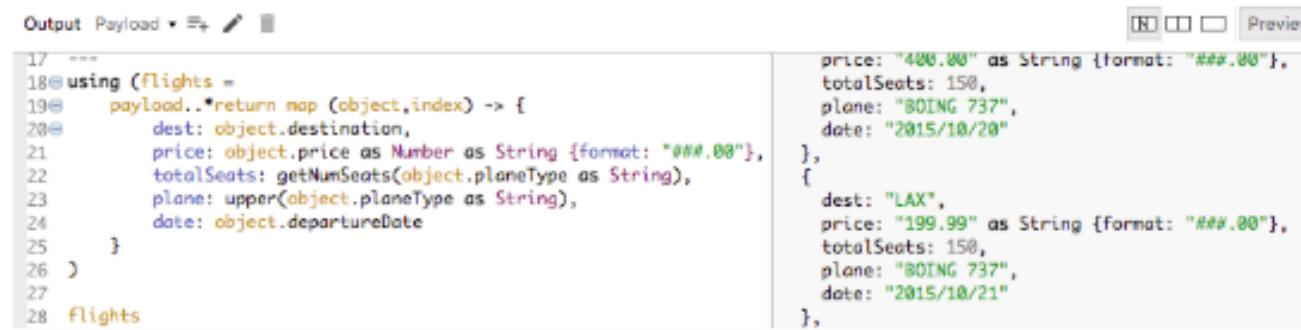
Note: If you are not a Java programmer, you may want to look at the Java documentation for the `DecimalFormatter` class to review documentation on patterns. <https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html>

Coerce a string to a date

20. Add a date field to the return object.

date: object.departureDate

21. Look at the preview; you should see the date property is a String.



The screenshot shows the Mule DataWeave editor with two panes. The left pane contains the DataWeave script:

```
17 /**
18@using (flights =
19@  payload..*return map (object,index) -> {
20@    dest: object.destination,
21@    price: object.price as Number as String {format: "###.##"}, // Coerced to String
22@    totalSeats: getNumSeats(object.planeType as String),
23@    plane: upper(object.planeType as String),
24@    date: object.departureDate
25@  }
26@)
27@ Flights
28@
```

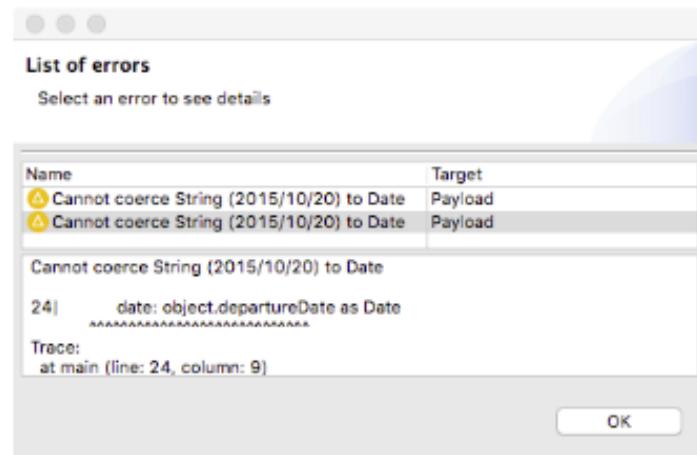
The right pane shows the preview of the transformed data:

```
price: "400.00" as String {format: "##.##"},  
totalSeats: 150,  
plane: "BOING 737",  
date: "2015/10/20"  
],  
{  
  dest: "LAX",  
  price: "199.99" as String {format: "##.##"},  
  totalSeats: 150,  
  plane: "BOING 737",  
  date: "2015/10/21"  
},
```

22. Change the DataWeave expression to use the as operator to convert departureDate to a date object; you should get an exception.

date: object.departureDate as Date

23. Look at the exception.



24. Look at the format of the dates in the preview.

25. Change the DataWeave expression to use the format schema property to specify the pattern of the input date strings.

date: object.departureDate as Date {format: "yyyy/MM/dd"}

Note: If you are not a Java programmer, you may want to look at the Java documentation for the `DateTimeFormatter` class to review documentation on pattern letters. <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

26. Look at the preview; you should see date is now a Date object.

The screenshot shows the DataWeave editor interface with the following components:

- Output Payload**: Shows the DataWeave code being run.
- Preview**: Shows the resulting JSON output.

The DataWeave code is as follows:

```
17 ---
18@using(Flights =
19@  payload.*return map {object,index} -> {
20@    dest: object.destination,
21@    price: object.price as Number {format: "##0.00"},
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  }
26 }
27
28 flights
```

The preview output shows two flight objects:

```
price: "499.99" as String {format: "##0.00"},  
totalSeats: 150,  
plane: "BOING 737",  
date: 12015-10-20 as Date {format: "yyyy/MM/dd"}  
,  
{  
  dest: "LAX",  
  price: "199.99" as String {format: "##0.00"},  
  totalSeats: 150,  
  plane: "BOING 737",  
  date: 12015-10-21 as Date {format: "yyyy/MM/dd"}  
},
```

Format a date

27. Use the as operator to convert the dates to strings, which can then be formatted.

```
date: object.departureDate as Date {format: "yyyy/MM/dd"} as String
```

28. Look at the preview section; the date is again a String – but with a different format.

Output Payload   

17 ===
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"} as String
25@ }
26@)
27@ flights

price: "199.99" as String {format: "###.00"},
totalSeats: 150,
plane: "BOING 737",
date: "2015-10-20"
},
[
 dest: "LAX",
 price: "199.99" as String {format: "###.00"},
 totalSeats: 150,
 plane: "BOING 737",
 date: "2015-10-21"
],

29. Use the format schema property with any pattern letters and characters to format the date strings.

```
date: object.departureDate as Date {format: "yyyy/MM/dd"} as String {format: "MMM dd, yyyy"}
```

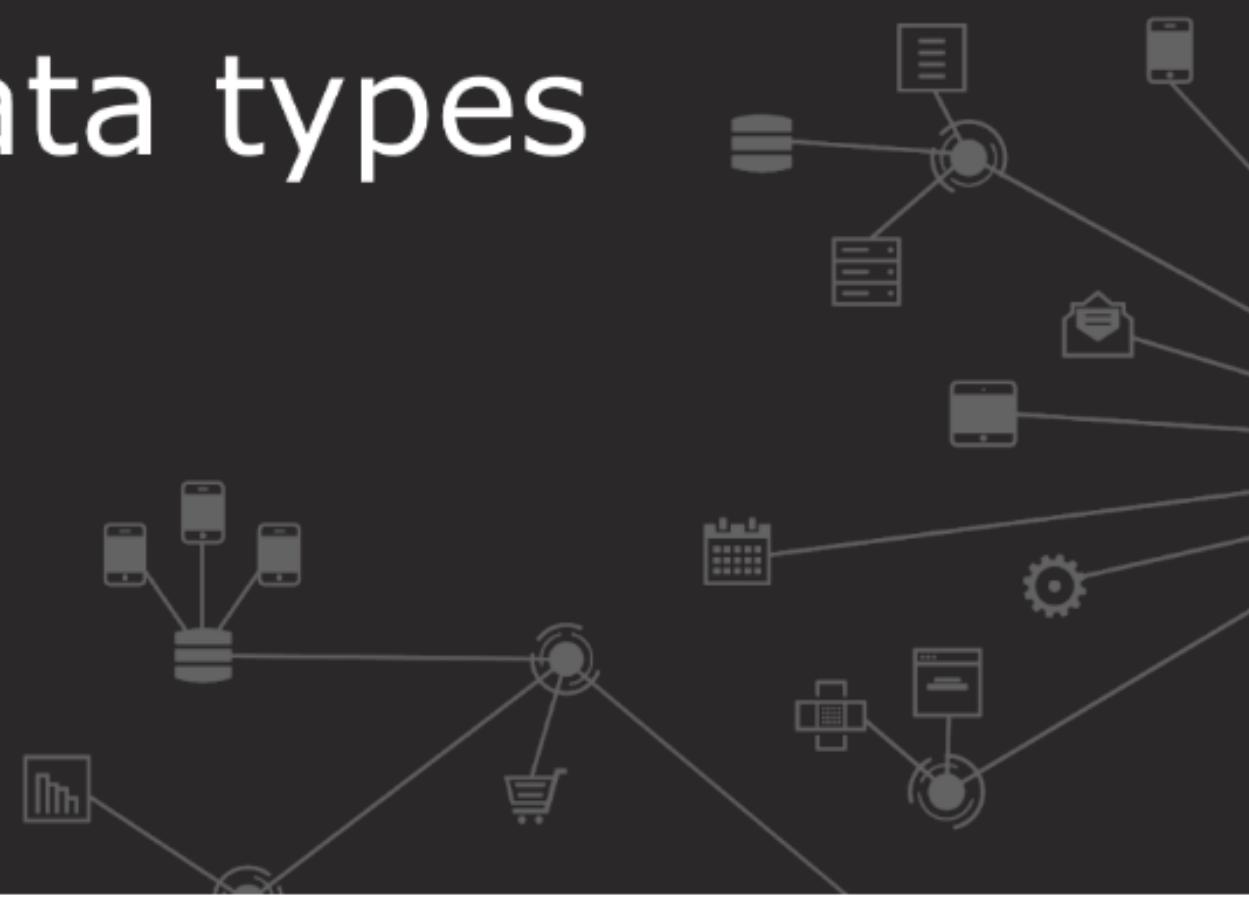
30. Look at the preview; the dates should now be formatted according to the pattern.

Output Payload   

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

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



- 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
```

- Specify inline

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

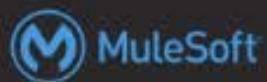
- Or define a custom data type to use

```
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

Output Payload ▾ Preview

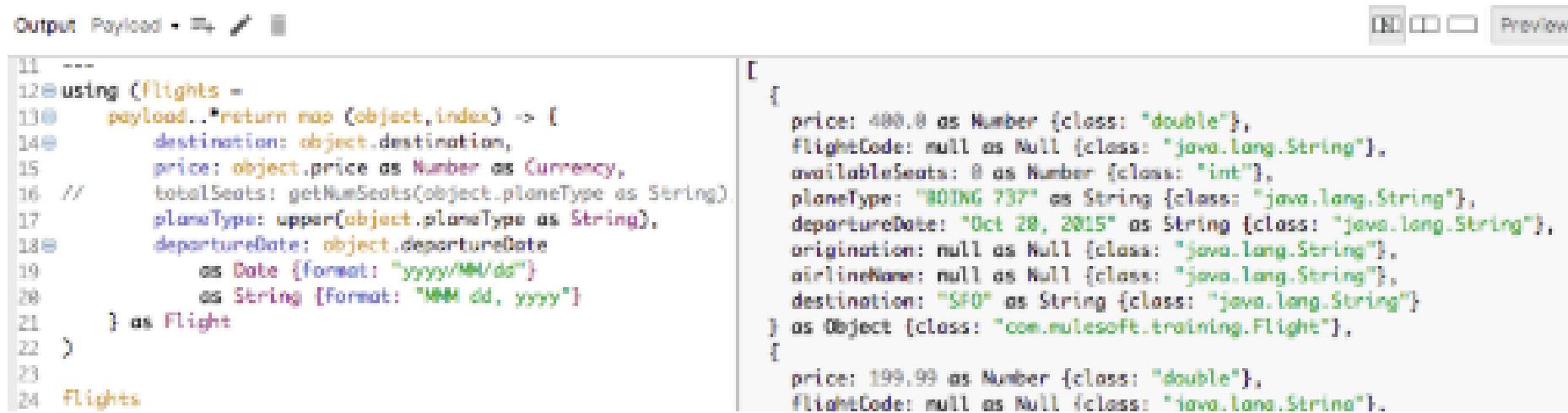
```
11 ---  
12@using (flights =  
13@    payload..*return map (object,index) -> {  
14@        destination: object.destination,  
15@        price: object.price as Number as Currency,  
16@        // totalSeats: getNumSeats(object.planeType as String)  
17@        planeType: upper(object.planeType as String),  
18@        departureDate: object.departureDate  
19@            as Date {format: "yyyy/MM/dd"}  
20@            as String {format: "MMM dd, yyyy"}  
21@    } as Flight  
22@}  
23@ flights
```

[{
 price: 400.0 as Number {class: "double"},
 flightCode: null as Null {class: "java.lang.String"},
 availableSeats: 0 as Number {class: "int"},
 planeType: "BOING 737" as String {class: "java.lang.String"},
 departureDate: "Oct 20, 2015" as String {class: "java.lang.String"},
 origination: null as Null {class: "java.lang.String"},
 airlineName: null as Null {class: "java.lang.String"},
 destination: "SFO" as String {class: "java.lang.String"}
} as Object {class: "com.mulesoft.training.Flight"},
{
 price: 199.99 as Number {class: "double"},
 flightCode: null as Null {class: "java.lang.String"}.

Walkthrough 11-7: Define and use custom data types

In this walkthrough, you continue to work with the flight JSON posted to the flow. You will:

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



The screenshot shows a Mule Studio interface with a code editor and a preview pane. The code editor contains Java code for transforming a JSON payload into a custom Flight object. The preview pane shows the resulting POJO object.

```
Output Payload • ⚡ Preview
```

```
11 ---  
12 @using(Flights =  
13     payload.*return map (object,index) -> {  
14         destination: object.destination,  
15         price: object.price as Number as Currency,  
16         // totalSeats: getNumSeats(object.planeType as String)  
17         planeType: upper(object.planeType as String),  
18         departureDate: object.departureDate  
19             as Date [format: "yyyy/MM/dd"]  
20             as String [format: "MM dd, yyyy"]  
21     } as Flight  
22 }  
23 Flights  
24
```

```
[  
{  
    price: 499.0 as Number {class: "double"},  
    flightCode: null as Null {class: "java.lang.String"},  
    availableSeats: 0 as Number {class: "int"},  
    planeType: "BOING 737" as String {class: "java.lang.String"},  
    departureDate: "Oct 28, 2015" as String {class: "java.lang.String"},  
    origination: null as Null {class: "java.lang.String"},  
    airlineName: null as Null {class: "java.lang.String"},  
    destination: "SFO" as String {class: "java.lang.String"}  
} as Object {class: "com.mulesoft.training.Flight"},  
{  
    price: 199.99 as Number {class: "double"},  
    flightCode: null as Null {class: "java.lang.String"}  
}
```

Define a custom data type

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Select and cut the string formatting expression for the prices.

```
18① using flights =
19①   payload..*return map (object,index) -> {
20①     dest: object.destination,
21     price: object.price as Number as String {format: "###.00"},
22     totalSeats: getNumSeats(object.planeType as String),
23     plane: unmapObject(planeType as String)
```

3. In the header section of the expression, define a custom data type called Currency.
4. Set it equal to the value you copied.

```
type Currency = String {format: "###.00"}
```

Output Payload ➔   ① 3 issues found

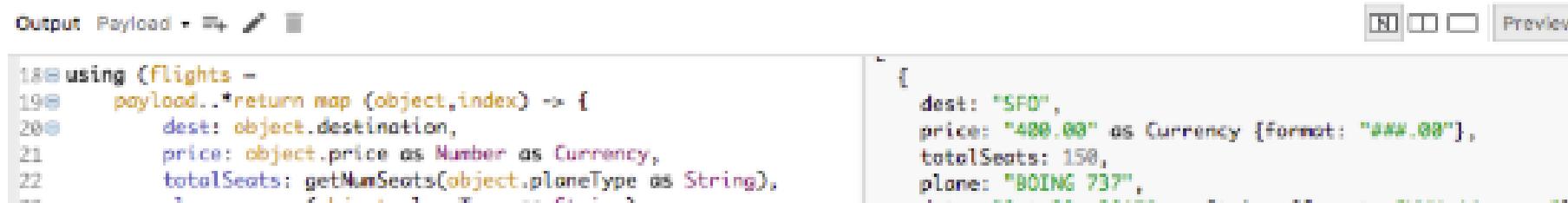
```
1① %dw 2.0
2 output application/dw
3 type Currency = String {format: "###.00"}
4 //var numSeats = 400
```

Use a custom data type

5. In the DataWeave expression, set the price to be of type currency.

```
price: object.price as Number as Currency,
```

6. Look at the preview; the prices should still be formatted as strings to two decimal places.



The screenshot shows a DataWeave editor interface. On the left, there is a code editor window with the following DataWeave code:

```
180 using (flights ->
190   payload..*return map (object,index) -> {
200     dest: object.destination,
21     price: object.price as Number as Currency,
22     totalSeats: getNumSeats(object.planeType as String),
23     plane: object.planeType as String
24   }
25 }
```

On the right, there is a preview window showing the resulting JSON output:

```
{  
  dest: "SFO",  
  price: "488.00" as Currency {format: "###.##"},  
  totalSeats: 158,  
  plane: "BOING 737",  
  ...  
}
```

Transform objects to POJOs

7. Open the Flight.java class in the project's src/main/java folder and look at the names of the properties.

```
global implementation Flight.java
1 package com.mulesoft.training;
2
3 import java.util.Comparator;
4
5 public class Flight implements java.io.Serializable,
6
7     String flightCode;
8     String origination;
9     int availableSeats;
10    String departureDate;
11    String airlineName;
12    String destination;
13    double price;
14    String planeType;
15
16    public Flight() {
17
18    }
```

8. Return to postMultipleFlights in implementation.xml.
9. In the header section of the expression, define a custom data type called flight that is of type com.mulesoft.traning.Flight.

```
type Flight = Object {class: "com.mulesoft.training.Flight"}
```

```
Output Payload • ⌂
```

```
1 %dw 2.0
2 output application/dw
3 type Currency = String {format: "###.00"}
4 type Flight = Object {class: "com.mulesoft.training.Flight"}
```

10. In the DataWeave expression, set the map objects to be of type Flight.

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

11. Look at the preview.



The screenshot shows the Mule Studio interface with the DataWeave editor and preview pane. The code in the editor is:

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

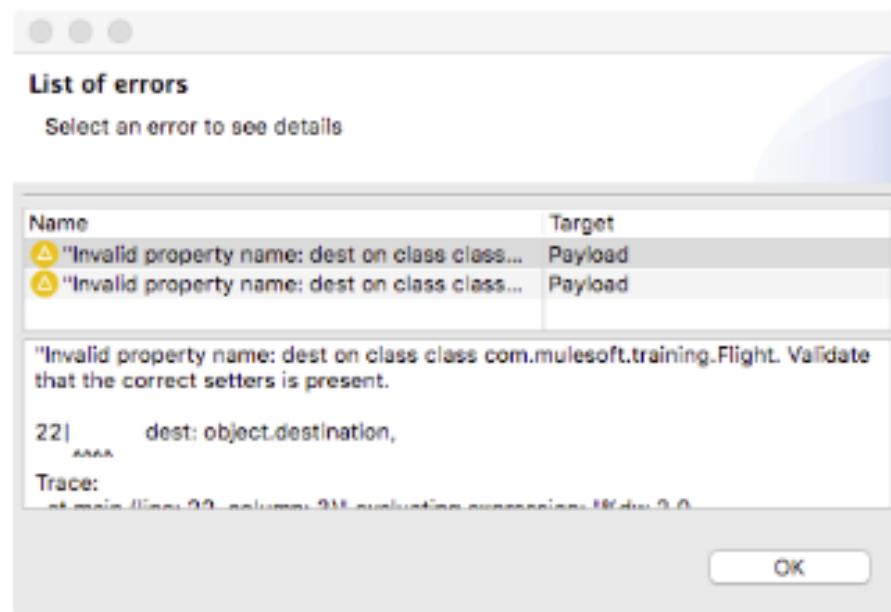
The preview pane displays the resulting JSON output:

```
[  
  {  
    dest: "SFO",  
    price: "400.00" as Currency [format: "###.00"],  
    totalSeats: 150,  
    plane: "BOING 737",  
    date: "Oct 20, 2015" as String [format: "MMM dd, yyyy"]  
  } as Flight [class: "com.mulesoft.training.Flight"],  
  {  
    dest: "LAX",  
    price: "100.99" as Currency [format: "##.##"],  
    totalSeats: 150,  
    plane: "BOING 737",  
    date: "Oct 21, 2015" as String [format: "MMM dd, yyyy"]  
  } as Flight [class: "com.mulesoft.training.Flight"]  
]
```

12. Change the output type from application/dw to application/java.

```
1@ %dw 2.0  
2@ output application/java  
3@ type Currency = String [format: "###.00"]  
4@ type Flight = Object [class: "com.mulesoft.training.Flight"]
```

13. Look at the issue you get.



14. Change the name of the dest key to destination.

destination: object.destination,

15. Change the other keys to match the names of the Flight class properties:

- plane to planeType
- date to departureDate

16. Comment out the totalSeats property.

17. Look at the preview.

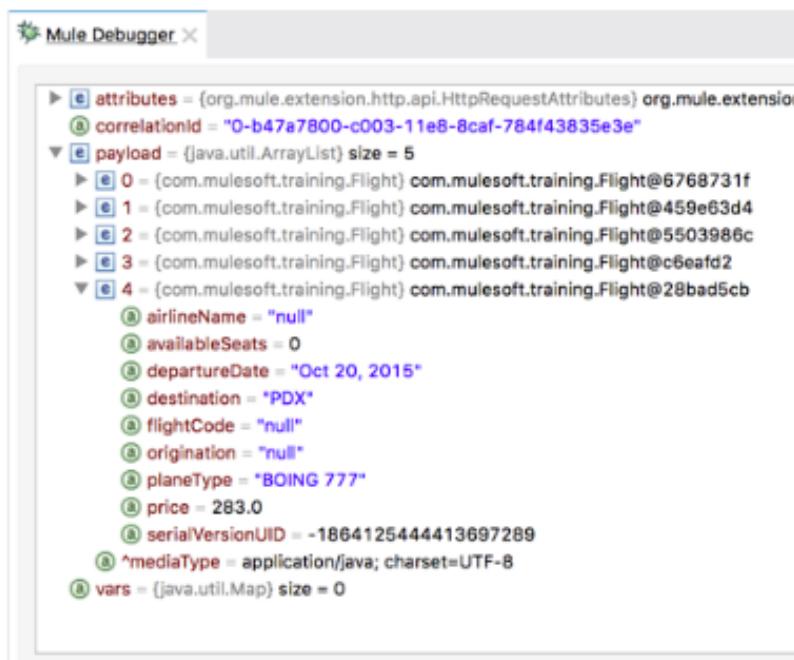
The screenshot shows the Mule Studio interface with the "Preview" tab selected. On the left, there is a code editor window containing Java code. On the right, there is a preview window showing the resulting JSON output. The Java code is as follows:

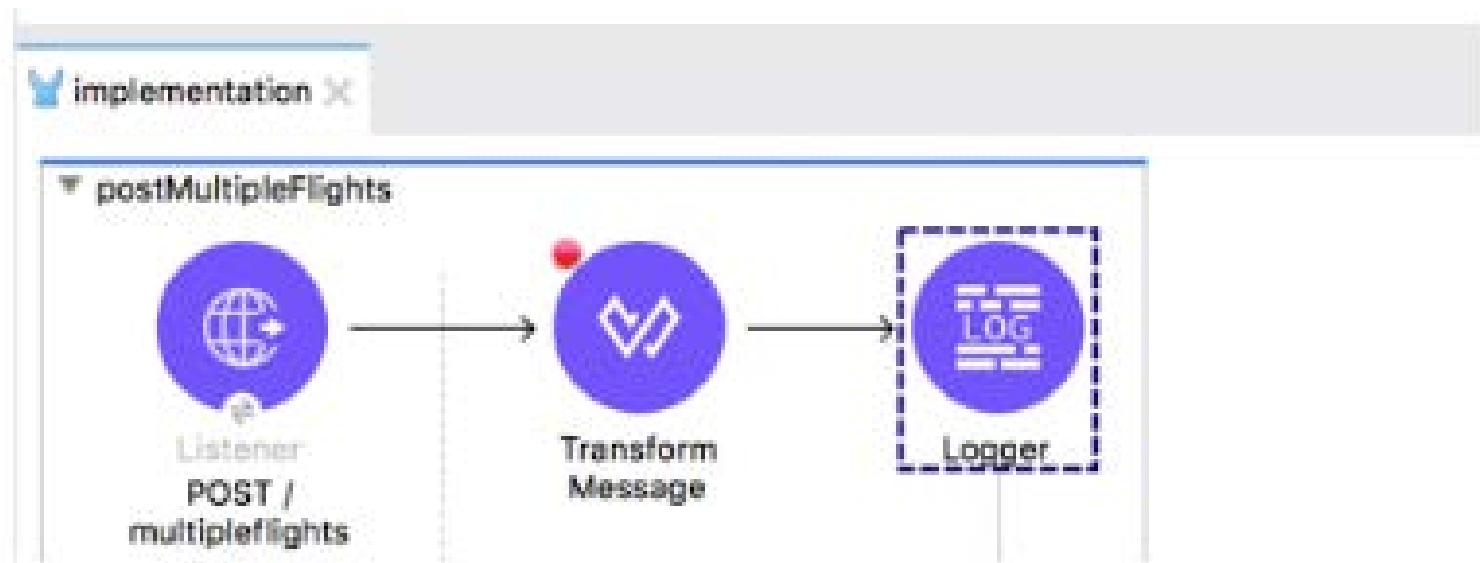
```
18    300
19 ===
20@using (Flights =
21@  payload.*return map {object,index} -> {
22@    destination: object.destination,
23@    price: object.price as Number as Currency,
24@    // totalSeats: getNumSeats(object.planeType as String,
25@    planeType: upper(object.planeType as String),
26@    departureDate: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

The JSON output is:

```
[{"price": 400.0 as Number [class: "double"], "flightCode": null as Null [class: "java.lang.String"], "availableSeats": 0 as Number [class: "int"], "planeType": "BOING 737" as String [class: "java.lang.String"], "departureDate": "Oct 20, 2015" as String [class: "java.lang.String"], "origination": null as Null [class: "java.lang.String"], "airlineName": null as Null [class: "java.lang.String"], "destination": "SFO" as String [class: "java.lang.String"]}, {"price": 199.99 as Number [class: "double"], "flightCode": null as Null [class: "java.lang.String"], "availableSeats": 0 as Number [class: "int"], "planeType": "BOING 777" as String [class: "java.lang.String"], "departureDate": "Oct 20, 2015" as String [class: "java.lang.String"], "origination": null as Null [class: "java.lang.String"], "airlineName": null as Null [class: "java.lang.String"], "destination": "PDX" as String [class: "java.lang.String"]}]
```

Note: If you want to test the application with Advanced REST Client, be sure to change the content-type header to application/XML and replace the request body with XML from the flights-example.xml file.





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	scan
--	isBlank	min	sizeOf
abs	isDecimal	minBy	splitBy
avg	isEmpty	mod	sqrt
cell	isEven	native	startsWith
contains	isInteger	now	sum
daysBetween	isLeapYear	orderBy	to
distinctBy	isOdd	pluck	trim
endsWith	joinBy	pow	typeOf
filter	log	random	unzip
filterObject	lower	randomInt	upper
find	map	read	uuid
flatMap	mapObject	readUrl	with
flatten	match	reduce	write
floor	matches	replace	zip
	max	round	49

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>

- 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

<p>▼ DataWeave Function Reference</p> <ul style="list-style-type: none">> Core (dw::Core)> Crypto (dw::Crypto)> Runtime (dw::Runtime)> System (dw::System)> Arrays (dw::core::Arrays)> Binaries (dw::core::Binaries)> Objects (dw::core::Objects)> Strings (dw::core::Strings)> URL (dw::core::URL)> Diff (dw::util::Diff)> Timer (dw::util::Timer)
<p>camelize</p> <p>capitalize</p> <p>charCode</p> <p>charCodeAt</p> <p>dasherize</p> <p>fromCharCode</p> <p>ordinalize</p> <p>pluralize</p> <p>singularize</p> <p>underscore</p>

- 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
- Dasherize data

```
34 @flights orderBy $.departureDate  
35      orderBy $.price  
36      distinctBy $  
37      filter ($.availableSeats != 0)
```

Walkthrough 11-8: Use DataWeave functions

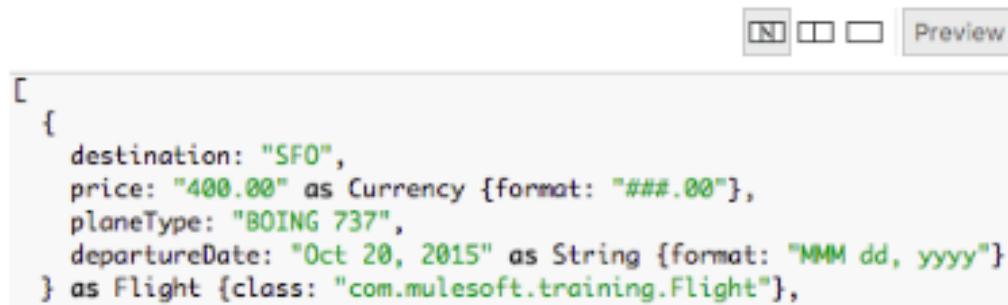
In this walkthrough, you continue to work with the flights JSON posted to the flow. You will:

- 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.

```
34 @flights orderBy $.departureDate  
35      orderBy $.price  
36      distinctBy $  
37      filter ($.availableSeats != 0)
```

Use the replace function

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Change the output type from application/java to application/dw.
3. Look at the preview and see that Boeing is misspelled.

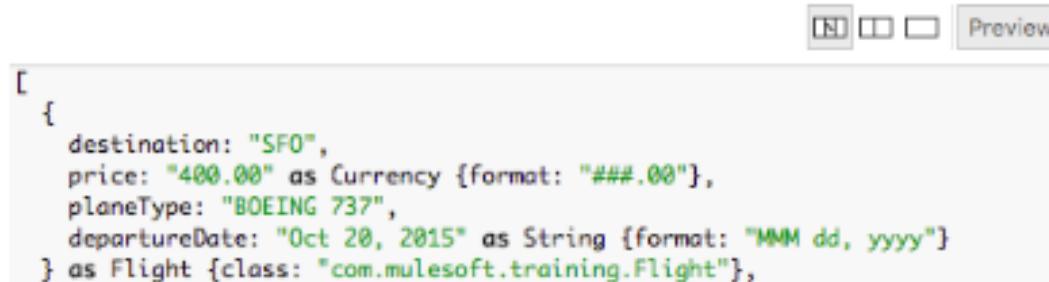


```
[  
 {  
 destination: "SFO",  
 price: "400.00" as Currency {format: "###.00"},  
 planeType: "BOING 737",  
 departureDate: "Oct 20, 2015" as String {format: "MMM dd, yyyy"}  
 } as Flight {class: "com.mulesoft.training.Flight"},
```

4. For planeType, use the replace function to replace the string Boing with Boeing.

`planeType: upper(replace(object.planeType,/(Boing)/) with "Boeing"),`

5. Look at the preview; Boeing should now be spelled correctly.



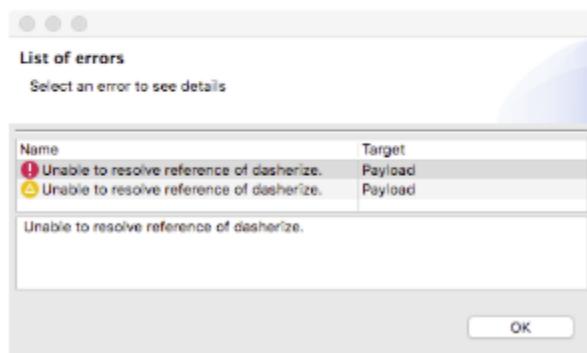
```
[  
 {  
 destination: "SFO",  
 price: "400.00" as Currency {format: "###.00"},  
 planeType: "BOEING 737",  
 departureDate: "Oct 20, 2015" as String {format: "MMM dd, yyyy"}  
 } as Flight {class: "com.mulesoft.training.Flight"},
```

Use the dasherize function in the Strings module

6. For planeType, replace the upper function with the dasherize function.

```
planeType: dasherize(replace(object.planeType,/(Boing)/) with "Boeing"),
```

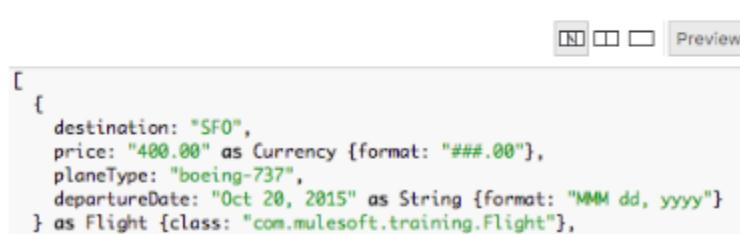
7. Look at the error you get.



8. In the script header, add an import statement to import dasherize from the Strings module.

```
import dasherize from dw::core::Strings  
1@%dw 2.0  
2 output application/dw  
3 import dasherize from dw::core::Strings  
4 type Currency = String {format: "##,.00"}  
5 type Flight = Object {class: "com.mulesoft.training.Flight"}
```

9. Look at the preview.



Use the orderBy function

10. In the preview section, look at the flight prices; the flights should not be ordered by price.
11. Use the orderBy function to order the flights object by price.

```
flights orderBy $.price
```

12. Look at the preview; the flights should now be ordered by price.

The screenshot shows the DataWeave preview window with two panes. The left pane displays the DataWeave code and its execution context:

```
Output Payload ▾ ↻ ⌂ Preview  
13      388  
14  /*  
15@ fun getNumSeats (planeType: String) =  
16@   if (planeType contains('737'))  
17@     158  
18@   else  
19@     308  
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@     planeType: dasherize(replace(object.planeType, /-/g)),  
27@     departureDate: object.departureDate  
28@       as Date [Format: "yyyy/MM/dd"]  
29@       as String [Format: "MM dd, yyyy"]  
30@   } as Flight  
31 }  
32  
33 flights orderBy $.price
```

The right pane shows the resulting JSON payload, which is a list of three flight objects. The flights are ordered by price, with the first flight having a price of 199.99 and the second and third flights having a price of 283.00.

```
[  
  {  
    destination: "LAX",  
    price: "199.99" as Currency [Format: "###.##"],  
    planeType: "boeing-737",  
    departureDate: "Oct 21, 2015" as String [Format: "MM dd, yyyy"]  
  } as Flight (class: "com.mulesoft.training.Flight"),  
  [  
    destination: "PDX",  
    price: "283.00" as Currency [Format: "###.##"],  
    planeType: "boeing-777",  
    departureDate: "Oct 21, 2015" as String [Format: "MM dd, yyyy"]  
  ] as Flight (class: "com.mulesoft.training.Flight"),  
  [  
    destination: "PDX",  
    price: "283.00" as Currency [Format: "###.##"],  
    planeType: "boeing-777",  
    departureDate: "Oct 28, 2015" as String [Format: "MM dd, yyyy"]  
  ] as Flight (class: "com.mulesoft.training.Flight"),  
  [  
    destination: "PDX",  
  ]
```

13. Look at the three \$283 flights; there is a duplicate and they are not ordered by date.
14. Change the DataWeave expression to sort flights by price and then date.

```
flights orderBy $.departureDate
```

```
orderBy $.price
```

15. Look at the preview; the flights should now be ordered by price and flights of the same price should be sorted by date.

Output Payload

```
15@ fun getNumSeats (planeType: String) =  
16@     if (planeType contains('737'))  
17@         158  
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@         planeType: dasherize(replace(object.planeType,/(Boeing-|737-)/g,  
27@             '$1$2')),  
28@         departureDate: object.departureDate  
29@             as Date {format: "yyyy/MM/dd"}  
30@             as String {format: "MMM dd, yyyy"}  
31@     } as Flight  
32 )  
33@ flights orderBy $.departureDate  
34@     orderBy $.price  
35
```

Preview

```
[{"destination": "PDX", "price": "283.00" as Currency {format: "###.##"}, "planeType": "boeing-777", "departureDate": "Oct 20, 2015" as String {format: "MMM dd, yyyy"}] as Flight (class: "com.rulesoft.training.Flight"), [{"destination": "PDX", "price": "283.00" as Currency {format: "###.##"}, "planeType": "boeing-777", "departureDate": "Oct 20, 2015" as String {format: "MMM dd, yyyy"}] as Flight (class: "com.rulesoft.training.Flight"), [{"destination": "SFO", "price": "283.00" as Currency {format: "###.##"}, "planeType": "boeing-777", "departureDate": "Oct 21, 2015" as String {format: "MMM dd, yyyy"}] as Flight (class: "com.rulesoft.training.Flight")
```

Remove duplicate data

16. Use the distinctBy function to first remove any duplicate objects.

```
flights orderBy $.departureDate  
    orderBy $.price  
    distinctBy $
```

17. Look at the preview; you should now see only two \$283 flights to PDX instead of three.

The screenshot shows a development environment with two main sections: a code editor on the left and a preview pane on the right.

Code Editor (Left):

```
Output Payload • ⓘ ☰  
15@ fun getNumSeats (planeType: String) =  
16@   if (planeType contains('737'))  
17@     158  
18@   else  
19@     388  
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@     planeType: deserializer(replace(object.planeType,/(Boin  
27@     departureDate: object.departureDate  
28@       as Date [format: "yyyy/MM/dd"]  
29@       as String [format: "MM dd, yyyy"]  
30@     ) as Flight  
31@   )  
32  
33@ flights orderBy $.departureDate  
34@   orderBy $.price  
35@   distinctBy $
```

Preview (Right):

The preview pane displays the resulting data structure for the two flights:

```
price: "199.00" as Currency {format: "###.00"},  
planeType: "boeing-737",  
departureDate: "Oct 21, 2015" as String {format: "MM dd, yyyy"}  
] as Flight {class: "com.rulesoft.training.Flight"},  
{  
  destination: "PDX",  
  price: "283.00" as Currency {format: "###.00"},  
  planeType: "boeing-777",  
  departureDate: "Oct 20, 2015" as String {format: "MM dd, yyyy"}  
} as Flight {class: "com.rulesoft.training.Flight"},  
{  
  destination: "PDX",  
  price: "283.00" as Currency {format: "###.00"},  
  planeType: "boeing-777",  
  departureDate: "Oct 21, 2015" as String {format: "MM dd, yyyy"}  
} as Flight {class: "com.rulesoft.training.Flight"},  
{  
  destination: "SFO",  
  price: "400.00" as Currency {format: "###.00"},  
  planeType: "boeing-737",  
  departureDate: "Oct 20, 2015" as String {format: "MM dd, yyyy"}  
}
```

18. Add an availableSeats field that is equal to the emptySeats field and coerce it to a number.

availableSeats: object.emptySeats as Number

19. Look at the preview; you should get three \$283 flights to PDX again.

Output Payload ↗

Preview

```
168     if (planeType.contains('737'))  
17         158  
18     else  
19         300  
20 ---  
21@ using (flights ->  
22@     payload.  
23@     .return map (object,index) -> {  
24@         destination: object.destination,  
25@         price: object.price as Number as Currency,  
26@         totalSeats: getNumSeats(object.planeType as String),  
27@         planeType: dasherize(replace(object.planeType,/-/g)),  
28@         departureDate:  
29@             object.departureDate  
30@             as Date {format: "yyyy/MM/dd"}  
31@             as String {format: "MM dd, yyyy"},  
32@         availableSeats: object.emptySeats as Number  
33@     } as Flight  
34@ )  
35@ flights orderBy $.departureDate  
36@         orderBy $.price  
37@         distinctBy $
```

[
destination: "PDX",
price: "283.00" as Currency {format: "###.##"},
planeType: "boeing-777",
departureDate: "Oct 20, 2015" as String {format: "MM dd, yyyy"},
availableSeats: 0
} as Flight {class: "com.mulesoft.training.Flight"},
{
destination: "PDX",
price: "283.00" as Currency {format: "###.##"},
planeType: "boeing-777",
departureDate: "Oct 20, 2015" as String {format: "MM dd, yyyy"},
availableSeats: 23
} as Flight {class: "com.mulesoft.training.Flight"},
{
destination: "PDX",
price: "283.00" as Currency {format: "###.##"},
planeType: "boeing-777",
departureDate: "Oct 21, 2015" as String {format: "MM dd, yyyy"},
availableSeats: 30
} as Flight {class: "com.mulesoft.training.Flight"}
See Editable Schema: <http://www.mulesoft.com/exchange/Flight>

Use the filter function

20. In the preview, look at the values of the availableSeats properties; you should see some are equal to zero.
21. Use the filter function to remove any objects that have availableSeats equal to 0.

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

22. Look at the preview; you should no longer get the flight that had no available seats.

The screenshot shows the MuleSoft Anypoint Studio interface. On the left, the code editor displays a Java-like script with numbered lines from 18 to 38. Lines 34 through 38 demonstrate the use of the `filter` function to exclude flights with zero available seats. On the right, the preview pane shows the resulting JSON output, which contains three flight records, each with a non-zero value for `availableSeats`.

```
Output Payload • ⌂ Preview  
[{"id": "1", "destination": "NYC", "price": "199.99" as Currency {format: "###.##"}, "planeType": "boeing-737", "departureDate": "Oct 21, 2015" as String {format: "MMM dd, yyyy"}, "availableSeats": 10} as Flight {class: "com.mulesoft.training.Flight"}, {"id": "2", "destination": "PDX", "price": "283.00" as Currency {format: "###.##"}, "planeType": "boeing-777", "departureDate": "Oct 20, 2015" as String {format: "MMM dd, yyyy"}, "availableSeats": 23} as Flight {class: "com.mulesoft.training.Flight"}, {"id": "3", "destination": "PDX", "price": "283.00" as Currency {format: "###.##"}, "planeType": "boeing-777", "departureDate": "Oct 21, 2015" as String {format: "MMM dd, yyyy"}, "availableSeats": 38} as Flight {class: "com.mulesoft.training.Flight"}]
```

```
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@      planeType: dasherize(replace(object.planeType,/-/c  
27@      departureDate: object.departureDate  
28@          as Date {format: "yyyy/MM/dd"}  
29@          as String {format: "MM dd, yyyy"},  
30@      availableSeats: object.emptySeats as Number  
31@  } as Flight  
32 )  
33  
34@flights orderBy $.departureDate  
35@    orderBy $.price  
36@    distinctBy $  
37@    filter ($.availableSeats !=0)  
38
```

Looking up data by calling a flow



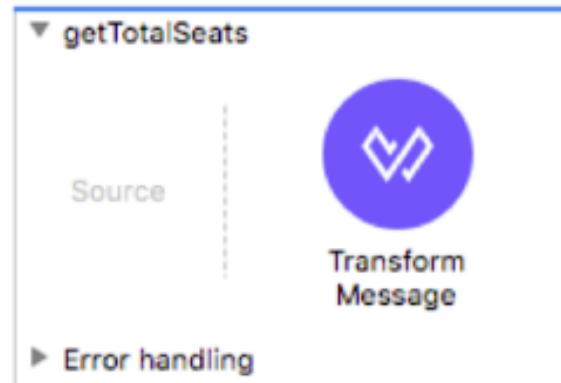
- Use the **lookup** function

```
{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

- Call a flow from a DataWeave expression

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

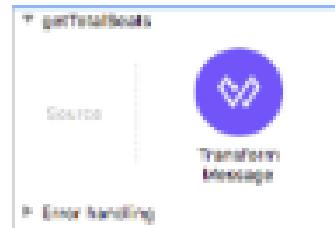


Walkthrough 11-9: Look up data by calling a flow

In this walkthrough, you continue to work with the transformation in `getMultipleFlights`. You will:

- Call a flow from a DataWeave expression.

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

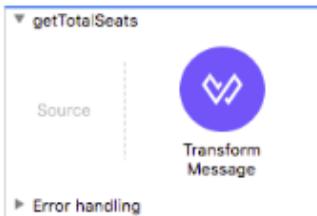


Create a lookup flow

1. Return to the Transform Message properties view for the transformation in postMultipleFlights.
2. Copy the getNumSeats function.

```
15 fun getNumSeats (planeType: String) =  
16     if (planeType contains('737'))  
17         150  
18     else  
19         300
```

3. Drag a Transform Message component from the Mule Palette and drop it at the bottom of the canvas to create a new flow.
4. Change the name of the flow to getTotalSeats.



5. In the Transform Message properties view of the component in getTotalSeats, paste the function you copied into the script header.
6. In the script body, call the function, passing to it the payload.

```
Output Payload • +    
1 %dw 2.0  
2 output application/java  
3  
4 fun getNumSeats(planeType: String) =  
5     if (planeType contains ("737"))  
6         150  
7     else  
8         300  
9  
10 ---  
11 getNumSeats(payload.planeType)
```

Call a flow from a DataWeave expression

7. Return to the Transform Message properties view of the component in postMultipleFlights.
8. Add a property called totalSeats inside the expression (keep the other one commented out).
9. Change the return type of the map function from Flight to Object.

```
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:  
27    planeType: dasherize(replace(object.planeType,/(Boin  
28@    departureDate: object.departureDate  
29        as Date {format: "yyyy/MM/dd"}  
30        as String {format: "MMM dd, yyyy"},  
31    availableSeats: object.emptySeats as Number  
32 } as Object  
33 )
```

10. Set totalSeats equal to the return value from the DataWeave lookup() function.

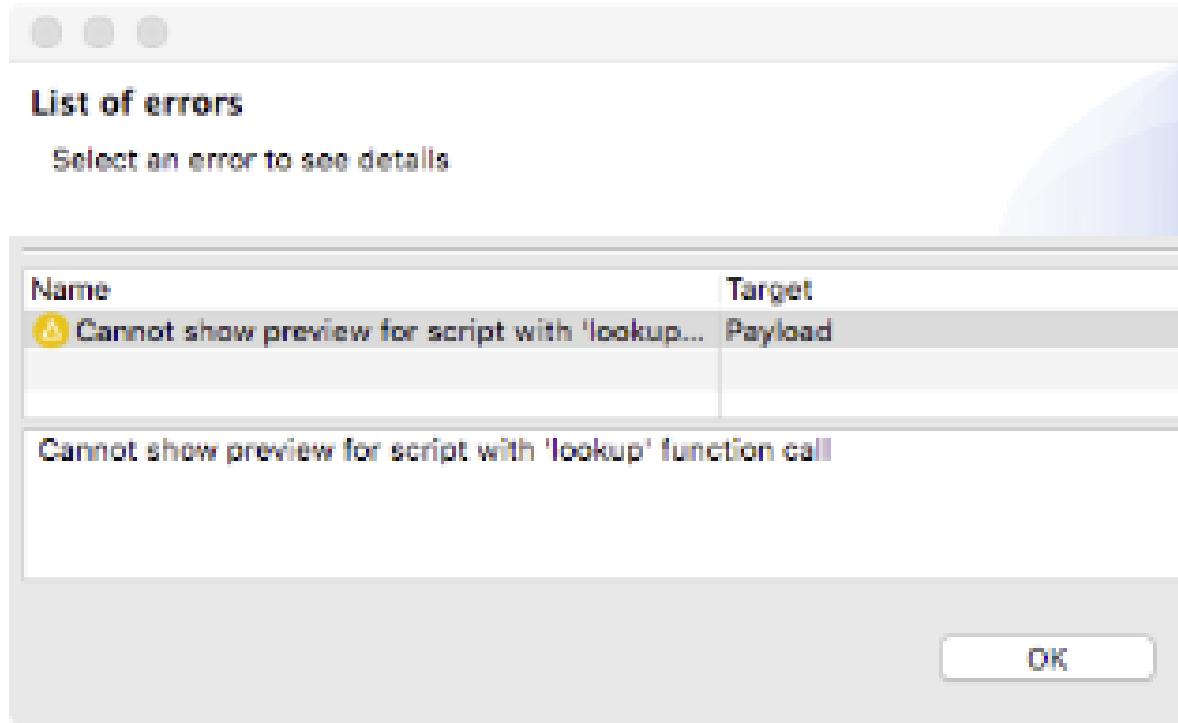
totalSeats: lookup()

11. Pass to lookup() an argument that is equal to the name of the flow to call: "getTotalSeats".
12. Pass an object to lookup() as the second argument.
13. Give the object a field called type (to match what the flow is expecting) and set it equal to the value of the planeType field.

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

14. Look at the preview.

15. Look at the warning you get.



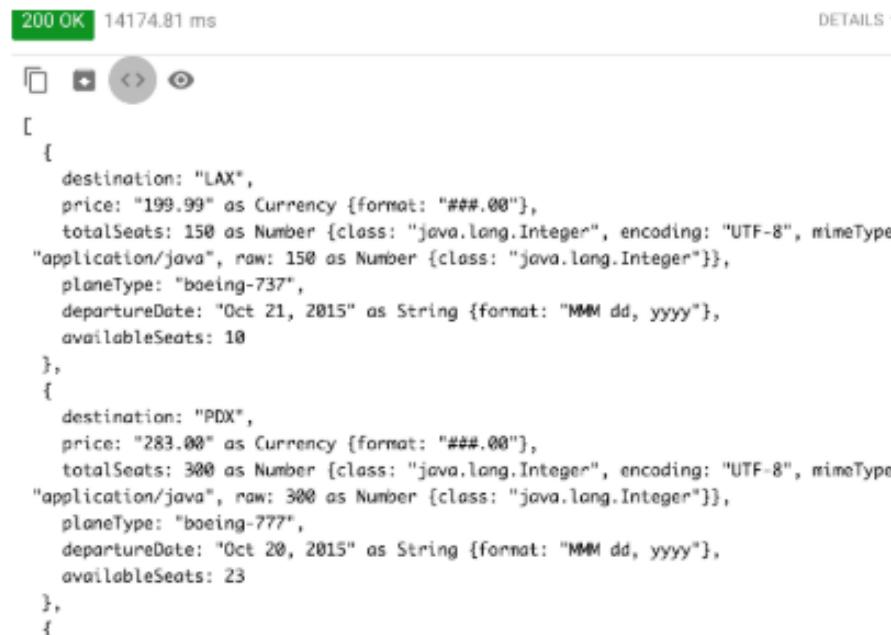
Test the application

16. Run or debug the project.
17. In Advanced REST Client, make sure the method is set to POST and the request URL is set to <http://localhost:8081/multipleflights>.
18. Set a Content-Type header to application/xml.
19. Set the request body to the value contained in the flights-example.xml file in the src/test/resources folder.

The screenshot shows the Advanced REST Client interface. At the top, there are fields for 'Method' (set to POST) and 'Request URL' (set to <http://localhost:8081/multipleflights>). Below these are buttons for 'SEND' and a more options menu. Underneath the URL field, there's a section titled 'Parameters' with a collapse/expand arrow. The 'Body' tab is selected, indicated by a blue underline. The 'Headers' tab is also visible. The 'Body' content type is set to 'application/xml'. The XML payload is displayed in a code editor-like area:

```
<ns2:listAllFlightsResponse xmlns:ns2="http://soap.training.mulesoft.com/">
    <return airlineName="United">
        <code>A1B2C3</code><departureDate>2015/10/20</departureDate>
        <destination>SFO</destination><emptySeats>40</emptySeats>
        <origin>MIA</origin>
        <planeType>Boing 737</planeType>
        <price>400.0</price>
    </return>
    <return airlineName="Delta">
        <code>A1B2C4</code>
        <departureDate>2015/10/21</departureDate><destination>LAX</destination><emptyS...
        <origin>MIA</origin>
        <planeType>Boing 737</planeType>
        <price>199.99</price>
    </return>
</ns2:listAllFlightsResponse>
```

20. Send the request; you should get the DataWeave representation of the return flight data and each flight should have a totalSeats property equal to 150 or 300.



```
[{"destination": "LAX", "price": "199.99" as Currency {format: "##.##"}, "totalSeats": 150 as Number {class: "java.lang.Integer", encoding: "UTF-8", mimeType: "application/java", raw: 150 as Number {class: "java.lang.Integer"}}, "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 as Number {class: "java.lang.Integer", encoding: "UTF-8", mimeType: "application/java", raw: 300 as Number {class: "java.lang.Integer"}}, "planeType": "boeing-777", "departureDate": "Oct 20, 2015" as String {format: "MMM dd, yyyy"}, "availableSeats": 23}]
```

21. Return to Anypoint Studio.
22. Stop the project.
23. Close the project.

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 {{ ... }}

- 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

- 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

DIY Exercise 11-1: Write DataWeave transformations

Time estimate: 3 hours

Objectives

In this exercise, you combine multiple data sources using DataWeave. You will:

- Use a map function to iterate through arrays.
- Use if else conditional statements to change output based on conditions.
- Format numbers and dates.
- Use sort and filter functions to modify data structures.
- Use the read function to translate MIME types.
- Create custom DataWeave modules to reuse and share DataWeave code

Scenario

You have been asked to enhance the flights Mule application to return the transactions and account details for each flight. You need to retrieve the results from the three online airline flights data sources (American, Delta, and United) and then join these results with results from two additional online data sources:

- A Transactions web service that returns a list of all transactions related to a particular flight:
<http://apdev-accounts-ws.cloudhub.io/api/transactions?wsdl>
- An Accounts API that returns a list of account details associated with flight transactions: <http://apdev-accounts-ws.cloudhub.io/api/accounts>

The Mule application should return a list of flight information along with transaction information and account details for each flight, organized into a collection of structured objects. A sample final output can be found in /files/module11/sample_output.xml (in the MUFundamentals_DIExercises.zip that you can download from the Course Resources).

Import the starting project

Import the /files/module11/flights-mod11-dataweave-starter.jar deployable archive file (in the MUFundamentals_DIYexercises.zip that you can download from the Course Resources) into Anypoint Studio.

Note: This is an enhanced version of the flights Mule application solution from Module 11 of the Fundamentals course and has an updated RAML file.

Retrieve transactions

Create a new flow that makes a call to the Transactions web service. The web service takes a list of flight IDs in the SOAP request, then returns a list of transactions associated with each flight id. Here is the configuration information for the web service:

```
webservice.wsdl = http://apdev-accounts-ws.cloudhub.io/api/transactions?wsdl
```

```
webservice.service = TransactionServiceService
```

```
webservice.port = TransactionServicePort
```

```
webservice.address = http://apdev-accounts-ws.cloudhub.io/api/transactions
```

```
webservice.operation = GetTransactionsforFlights
```

Here is a sample request to the web service where each flightID in the request XML is composed by combining the airlineName, flightCode, and departureDate keys from each flight response:

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

<ns0:GetTransactionsforFlights xmlns:ns0="http://trainingmulesoft.com/">

    <flightID>UNITEDER38SD2015-03-20</flightID>
    <flightID>UNITEDER39RK2015-09-11</flightID>
    <flightID>UNITEDER39RJ2015-02-12</flightID>

</ns0:GetTransactionsforFlights>
```

Here is a sample response from the web service:

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

```
<ns2:GetTransactionsforFlightsResponse xmlns:ns2="http://training.mulesoft.com/">
    <transaction>
        <amount>3177.0</amount>
        <customerRef>4456</customerRef>
        <flightID>UNITEDER39RK2015-09-11</flightID>
        <region>US</region>
        <transactionID>181948797</transactionID>
    </transaction>
    <transaction>
        <amount>6032.0</amount>
        <customerRef>4968</customerRef>
        <flightID>UNITEDER38SD2015-03-20</flightID>
        <region>US</region>
        <transactionID>181948717</transactionID>
    </transaction>
</ns2:GetTransactionsforFlightsResponse>
```

Store the transactions results in a variable so that the Transactions web service response can be combined with the flights response.

Retrieve account data for each flight transaction

Create a flow to make a request to the Accounts API to retrieve account details for each transaction record. The API has a POST api/accountList endpoint that expects a JSON-formatted array of account ids formatted as strings and returns a string that represents an array of Account objects in JSON format.

Here is the configuration information for the API:

```
accounts.host = apdev-accounts-ws.cloudhub.io  
accounts.port = 80  
accounts.basePath = /api
```

To make a request to the Accounts API, you must provide a valid input to the POST method and set the Requester-API header. The customerRef fields from the previous transactions SOAP API provide valid accountID values for the Accounts API requests. For example, you can write DataWeave expressions to extract these customerRef fields from the previous transactions response example:

```
[  
  "4456",  
  "4968"  
]
```

Note: For ease of implementation, the Accounts API is not RESTful due to the POST api/accountList endpoint. The POST method is used to make it easy to include the array of account ids in the request rather than asking you to serialize the array with escaped characters into a GET request.

After the Accounts API returns a valid response, save the response payload to a flow variable so it can be joined with the flights and transactions responses.

Hint: To transform a string into a target object, you can use the DataWeave function:

```
read(data, "application/json")
```

Note: For an additional challenge, set the HTTP Request's output target to this variable without modifying the current event payload.

Join all data into a nested object structure

The final step is to join flights, transactions, and accounts data. The Mule application contains starter code for the DataWeave transformation needed to build up this complex data structure. The essential transformation is located in the Combine all payloads and convert to XML Transform Message component in the getFlightsFlow.

Complete the join in two steps. First, join each flight object with the transaction with the same flightID value, so it creates a nested object:
airlineName: "UNITED",

flightCode: "ER39RK",

departureDate: "2015-Mar-20",

...

transactions:

{

idCustomer: "4564",

transID: "181948544",

amount: "735.0",

 ...

}

Next, join the accounts details (account name and account type) with each transaction:
airlineName: "UNITED",

flightCode: "ER39RK",

departureDate: "2015-Mar-20",

...

transactions:

{

idCustomer: 4564,

transID: "181948544",

amount: "735.0",

nameCustomer: "Max Plank",

type: "business"

...

}

Call getFlightsFlow from the corresponding API interface flow

Inside the get:/flightstrans:mua-flights-transactions-api-config flow, add a Flow Reference component and set the flow name to getFlightsFlow.

Output combined data as XML data

Add additional DataWeave code to transform the output to valid XML. Move or add some of the response data fields as XML attributes instead of XML elements as needed. Here is an example final XML response:

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

<flights>

    <flight flightCode="ER38sd">

        <airline>United</airline>

        <departureDate>2015-Mar-20</departureDate>

        <emptySeats>0</emptySeats>

        <fromAirportCode>MUA</fromAirportCode>

        <planeType>Boeing 737</planeType>

        <price>400.0</price>

        <toAirportCode>SFO</toAirportCode>

        <transactions>

            <transaction transID="181948544" amount="735.0">

                <idCustomer>4564</idCustomer>

                <nameCustomer>Max Plack</nameCustomer>

                <type>business</type>

                <amountUSFormatted>735.00</amountUSFormatted>

            
```

```
<transaction transID="181948717" amount="6032.0">  
    <idCustomer>4968</idCustomer>  
    <nameCustomer>Nick The Shoemaker</nameCustomer>  
    <type>personal</type>  
    <amountUSFormatted>6,032.00</amountUSFormatted>  
  </transaction>  
</transactions>  
</flight>  
</flights>
```

Create a reusable function to format the transaction amount

In the final DataWeave transformation, extract the logic used to format the transaction amount into a function. Save the function in a file named Formats.dwl in src/main/resources/modules and reference the module inside the final DataWeave transformation.

Verify your solution

Load the solution /files/module11/flights-mod11-dataweave-solution.jar (in the MUFundamentals_DIYexercises.zip that you can download from the Course Resources) and compare your solution.