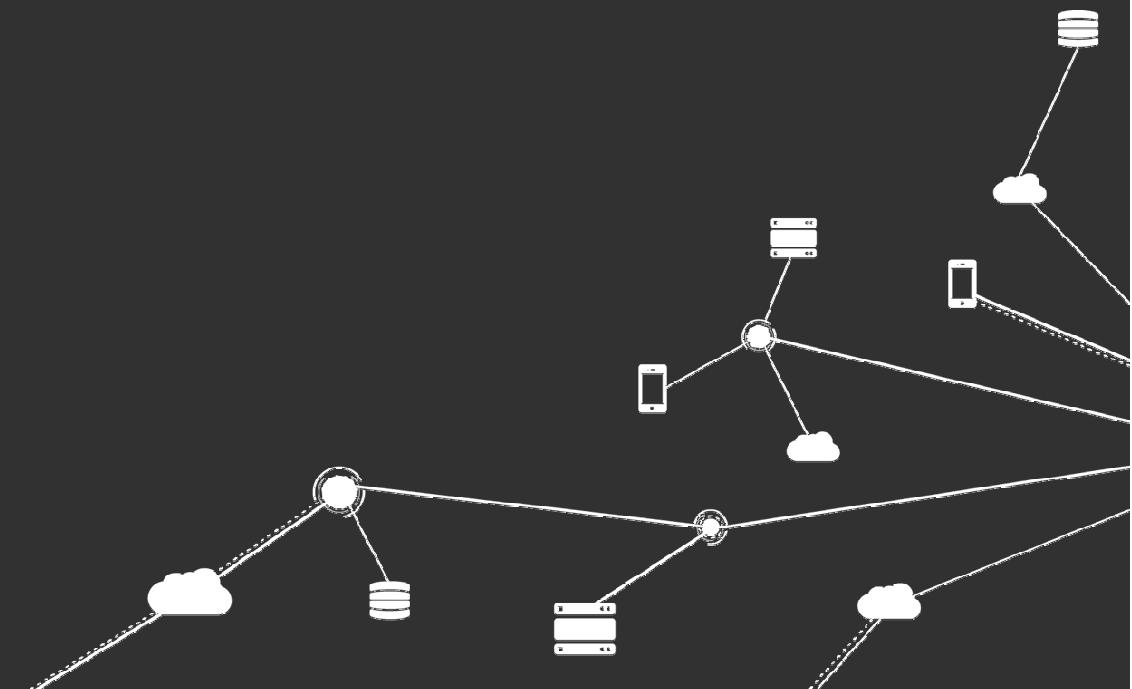




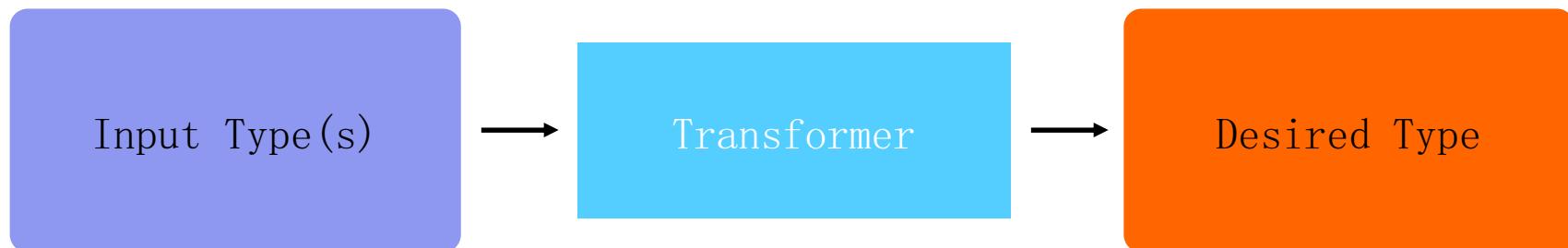
MuleSoft®

Module 5: Transforming Data



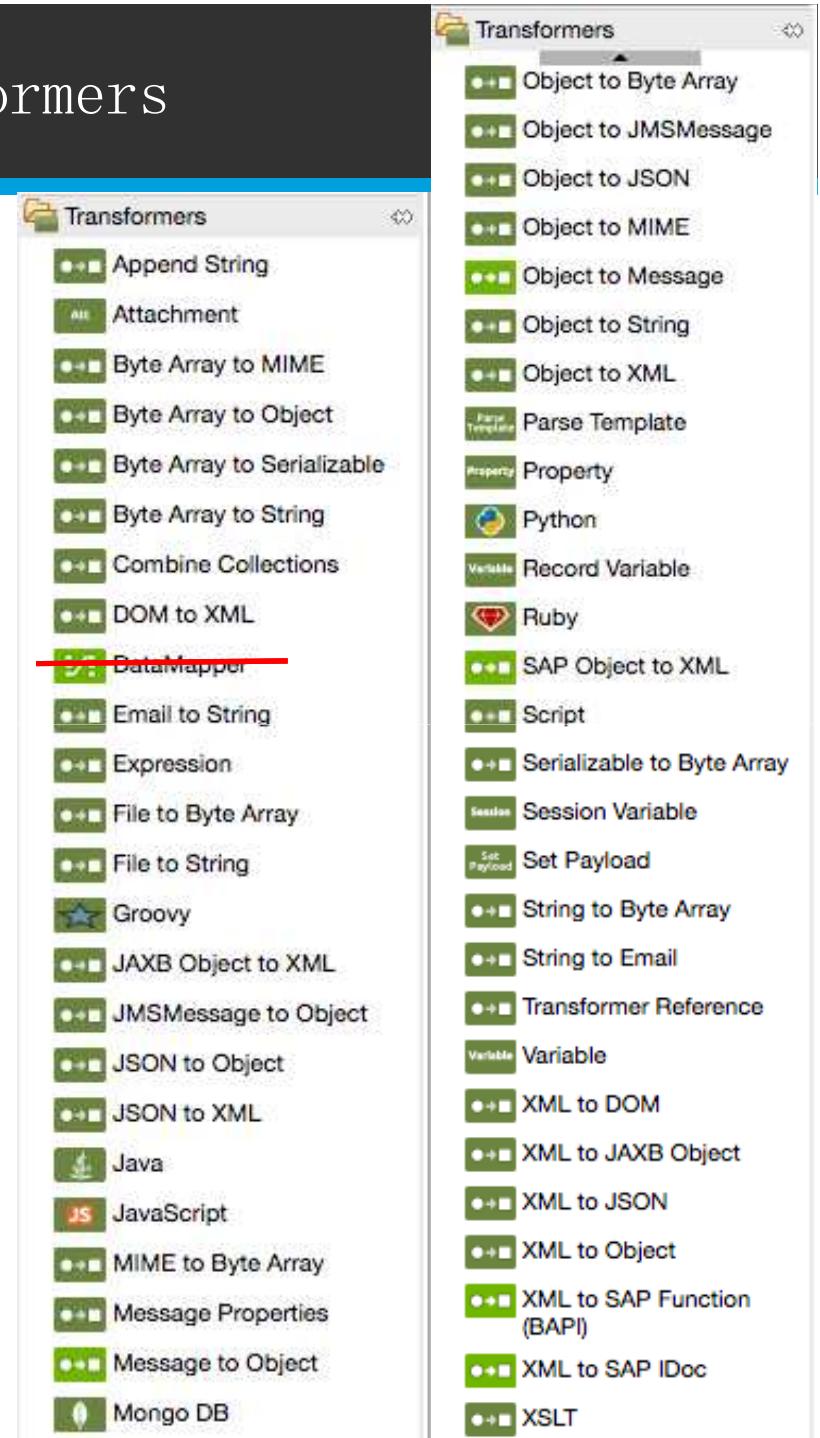
Transforming data using transformers

- Anypoint Studio provides a set of transformers to handle the most common data transformation scenarios
- Up to Mule 3.6, this was the main way to transform messages



We've already used some transformers

- DOM to XML (Delta)
 - Object to String (American DB)
 - File to String (CSV)
 - Object to JSON (Salesforce)
-
- Variable
 - Property
 - Session



About the DataMapper transformer

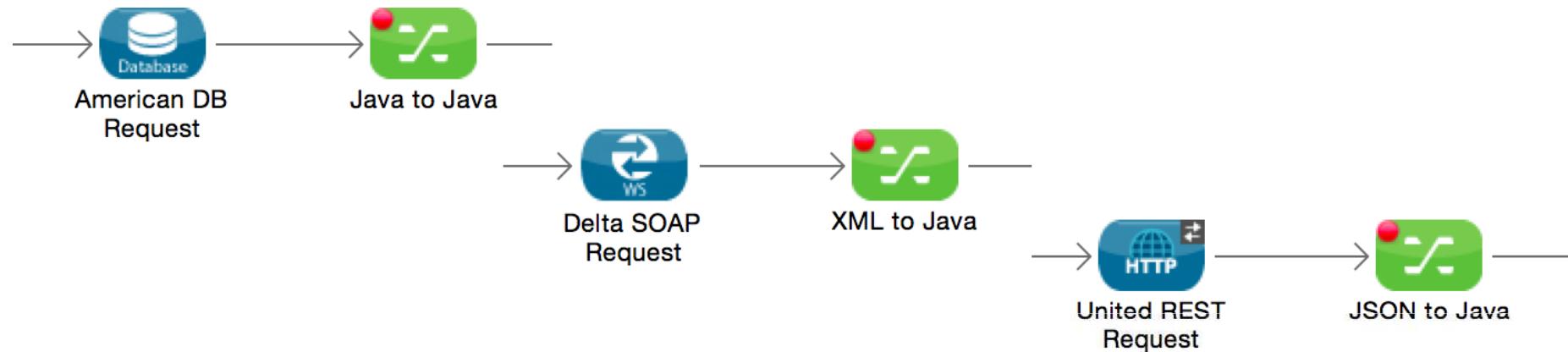


- Introduced in Mule 3.3 (2012)
- Provided a way for users to graphically manipulate data and work with many different data formats
- Customers loved the direction, but wanted much more in terms of capabilities, performance, and ease of use
- DataWeave is the new solution for the future
- DataMapper will not be supported from Mule 4.0 on
 - There will be a migration tool later this year



- DataWeave is a full-featured and fully native framework for querying and transforming data on Anypoint Platform
- Powered by the DataWeave data transformation language
 - A JSON-like language that's built just for data transformation use cases
- Powered by the core Mule runtime
 - Provides 5x performance vs previous approaches
- Fully integrated with Anypoint Studio and DataSense
 - Graphical interface with payload-aware development

Goal



Screenshot of the Mule Studio interface showing the configuration for the "XML to Java" component.

XML to Java (selected tab)

Input (Payload section):

```
%dw 1.0
%output application/java
---
payload.findFlightResponse.*return map {
    airlineName: $.airlineName,
    departureDate: $.departureDate,
    destination: $.destination,
    emptySeats: $.emptySeats as :number,
    flightCode: $.code,
    origination: $.origin,
    planeType: $.planeType replace /(Boing)/ with "Boeing"
    price: $.price as :number {format: "###.##"}
}
```

Output (Payload section):

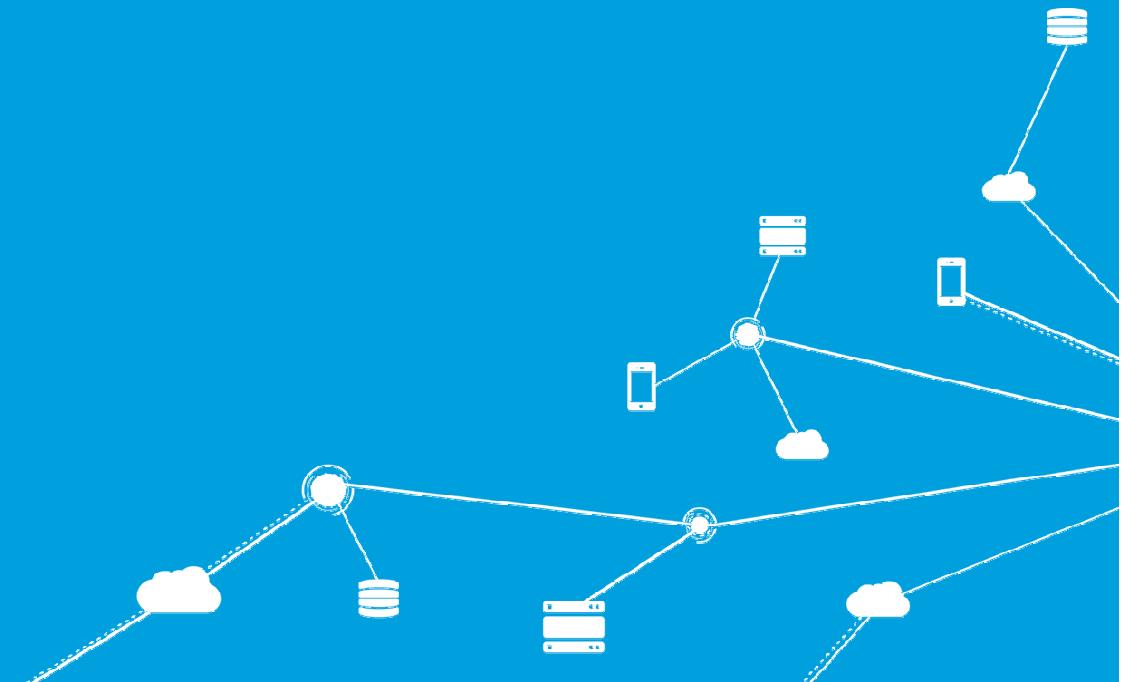
Name	Value
root : ArrayList	
e [0] : LinkedHashMap	
airlineName : String	????
departureDate : String	????
destination : String	????
emptySeats : Integer	1
flightCode : String	????
origination : String	????
planeType : String	????
price : Integer	2

Context, **Payload**, **Structure**, **Preview** tabs are visible at the bottom.

Objectives

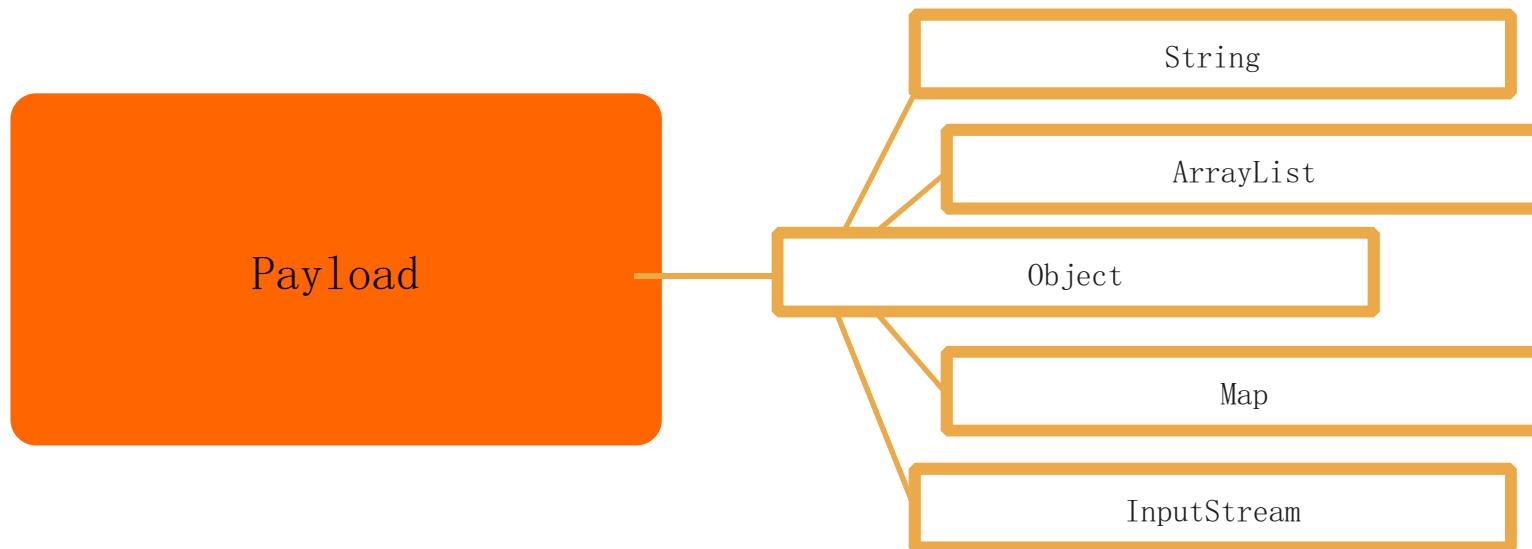
- In this module, you will learn:
 - About the different types of transformers
 - To use the DataWeave Transform Message component
 - To write DataWeave expressions for basic and complex XML, JSON, and Java transformations
 - To use DataWeave with data sources that have associated metadata
 - To add custom metadata to data sources

Introducing transformers



Transformers

- Transformers prepare a message to be processed through a flow by enhancing or altering the message header or message payload
- Remember payloads are Java objects and can be any Java type



Transformer categories

- Java object transformers
 - Transform a Java object into another Java object or some other data type (like HTTP request) or vice versa
- Message and variable transformers
 - Do not modify the message directly, but make special info available as a message makes its way through a Mule app
- Content transformers
 - Modify messages by adding to, deleting from, or converting a message payload (or a message header)
- Script transformers
 - Use Groovy, JavaScript, Python, or Ruby to perform the transformation

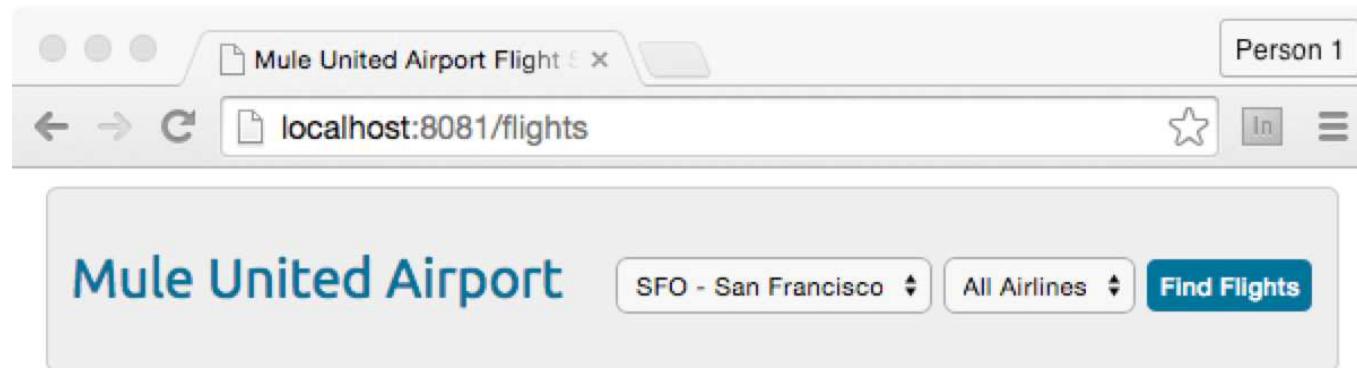
Parse Template transformer



- A content transformer
- Loads the content of an external file into a Mule flow
- Commonly used to return customized HTML responses
 - The file can have MEL expressions in it that are evaluated

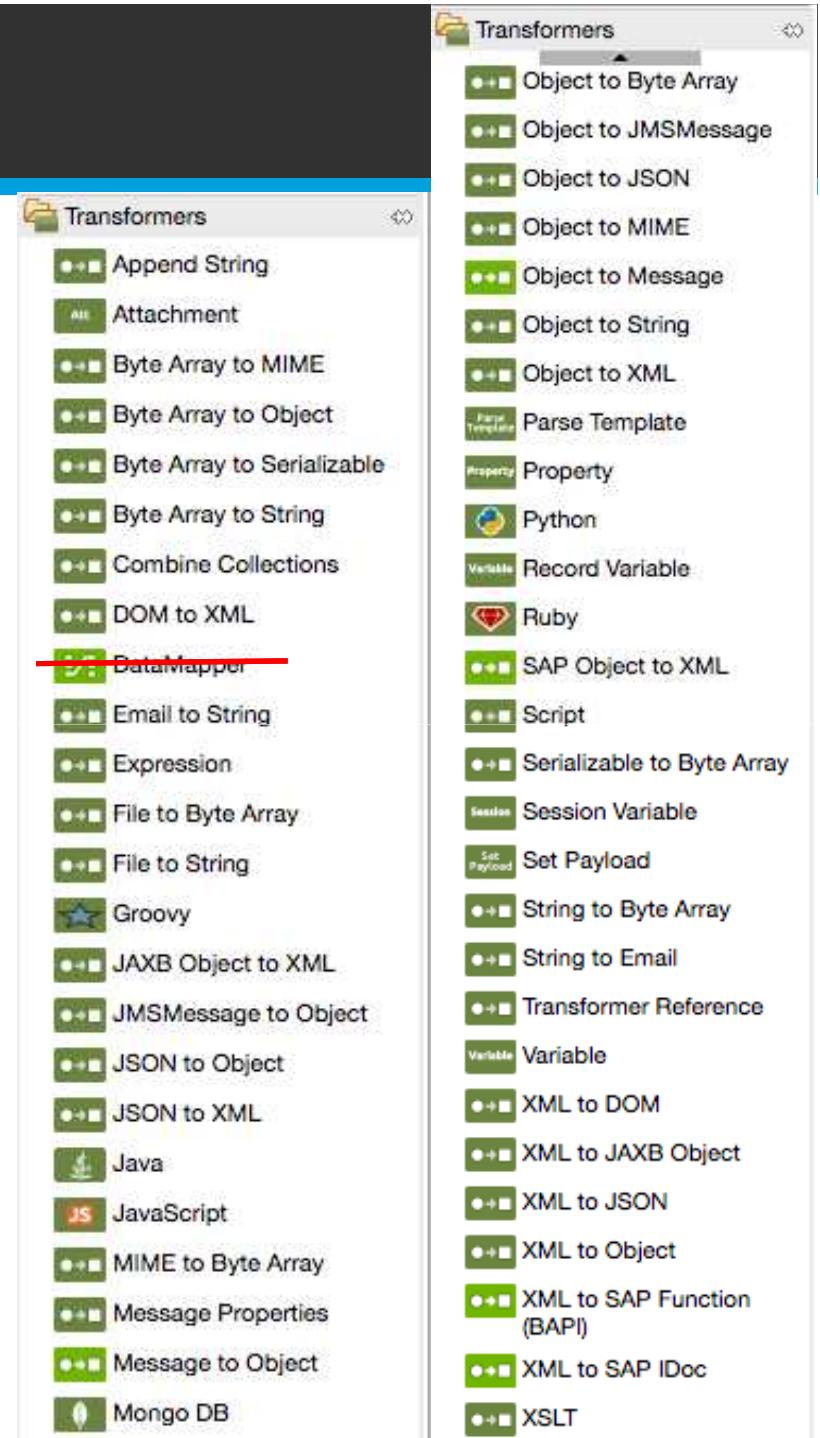
Walkthrough 5-1: Load external content into a message

- Create a flow that receives GET requests at <http://localhost:8081/flights>
- Use the Parse Template transformer to load the content of an HTML file into a flow
- Examine the HTML code and determine what data it sends where



Java object transformers

- Transform a Java object into another Java object or some other data type or vice versa
- Some can simply be dropped in (like the ones we used so far)
- Others require configuration using 3rd party libraries
 - JAXB
 - Jackson
 - org.w3c.dom



Complex transformations

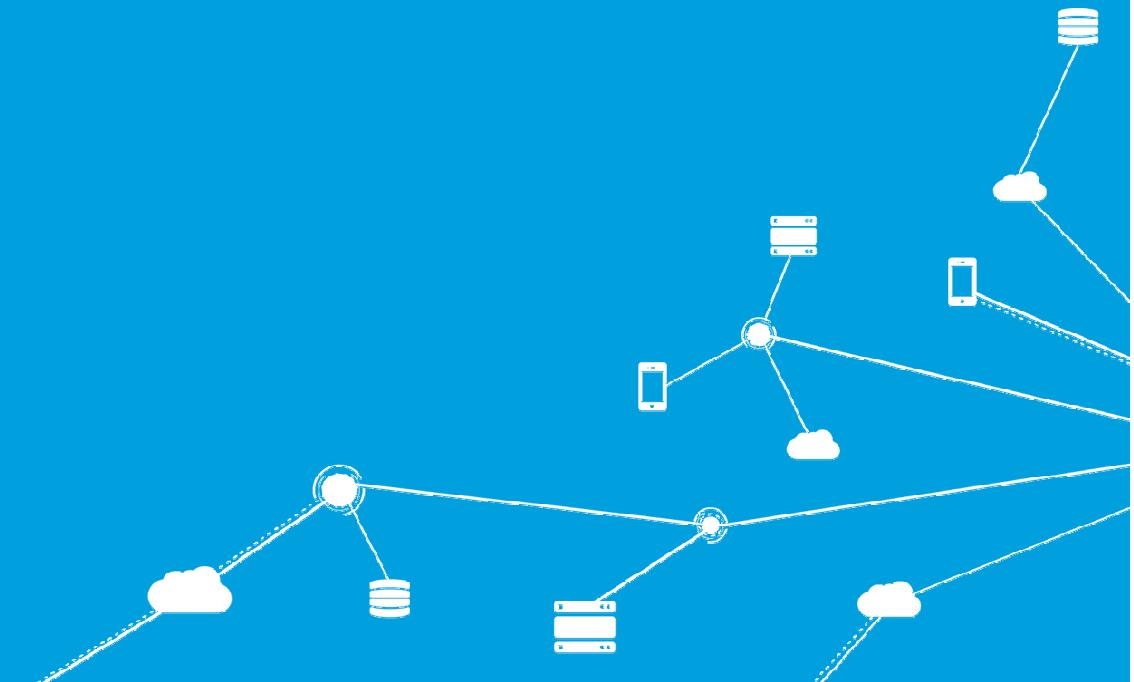
- Up to Mule 3.6, if a transformer did not exist for your specific needs you would
 - Chain transformers
 - Create a custom Java transformer
 - Use a Script transformer to write the transformation in Groovy, JavaScript, Python, or Ruby

Using DataWeave for all transformations



- For Mule 3.7, you can use the new DataWeave framework for all your transformations
 - From simple to complex
 - No longer need to use most other transformers unless you want to use specific Java frameworks
 - Like JAXB, Jackson, org.w3c.dom
 - To integrate with existing code bases or leverage existing skill sets

Introducing DataWeave



Introducing DataWeave (again)



- DataWeave is a full-featured and fully native framework for querying and transforming data on Anypoint Platform
- Powered by the DataWeave data transformation language
 - A JSON-like language that's built just for data transformation use cases
- Powered by the core Mule runtime
 - Provides 5x performance vs previous approaches
- Fully integrated with Anypoint Studio and DataSense
 - Graphical interface with payload-aware development



- A universal, simple, JSON-like language for transforming and querying data
- Easy to write, easy to maintain, and capable of supporting simple to complex mappings for any data type
 - Supports XML, JSON, Java, CSV, EDI out of the box
 - Extensible for new formats via an API
 - Excel support coming later this year
- More elegant and re-usable than custom code
 - Data transformations can be stored in external DWL files and used across applications



- DataWeave was purposefully built to make it easy to write simple to complex transformations
 - Simple 1-to-1 mappings
 - Transforming hierarchical data models
 - De-duplication of data
 - Filtering data
 - Grouping and partitioning data
 - Joining data across multiple data sources
 - Streaming inbound and outbound data

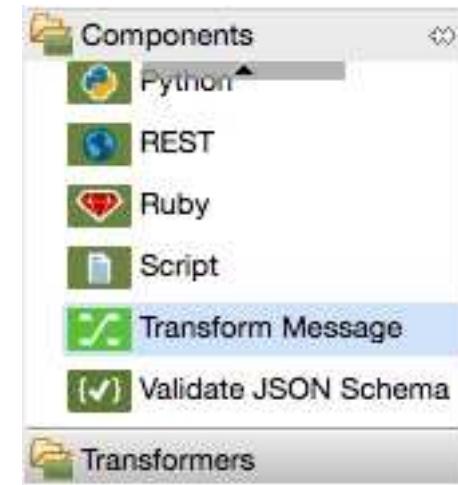


- Underneath, DataWeave includes a connectivity layer and engine that is fundamentally different from other transformation technologies
- It contains a data access layer that indexes content and accesses the binary directly, without costly conversions
 - Enables larger than memory payloads
 - Random access to input documents
 - Very high performance

DataWeave integration with Anypoint Studio



- The DataWeave Transform Message component provides an Anypoint Studio interface
- The Properties view has Input, Transform, and Output sections



The screenshot shows the Anypoint Studio Properties view for a 'Transform Message' component. The interface is divided into three main sections: Input, Transform, and Output.

Input: Displays the incoming message payload: `{"firstname": "Max", "lastname": "Mule"}`.

Transform: Shows the DataWeave script being executed:

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

Output: Displays the transformed message structure:

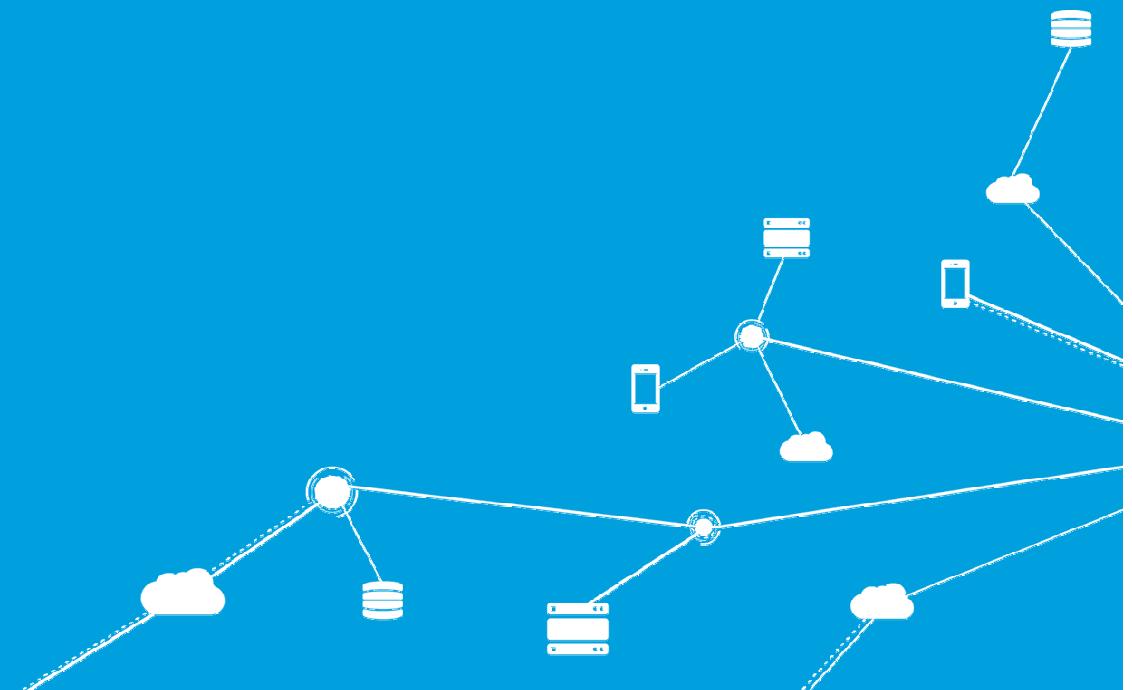
Name	Value
root : LinkedHashMap	
firstname : String	Max
lastname : String	Mule

Below the main sections are tabs for Context, Payload, Preview, and Structure.



- DataWeave is fully integrated with DataSense allowing payload-aware development
- Metadata from connectors, schemas and sample documents can be used to more easily build transformations
 - In the Input section of the Properties view
 - Any metadata of incoming message displayed
 - In the Transform section
 - Auto-completion of code
 - Auto-scaffolding of transforms
 - In the Output section
 - Any metadata for outbound message displayed
 - Live transformation previews

Writing DataWeave expressions



Example DataWeave expression

The **header** contains directives – high level info about the transformation

Input	Transform	Output
{ "firstname": "Max", "lastname": "Mule" }	%dw 1.0 %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>

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

DataWeave expressions

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

The output directive

- Sets the output type of the transformation
- Specified using content/type
 - application/json, text/json
 - application/xml, text/xml
 - application/java, text/java
 - application/csv, text/csv
 - application/dw
- The structure of the output is defined in the DataWeave body

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

Previewing transformations in Anypoint Studio

- As you write a DataWeave expression, a live preview of output will be shown
 - You will not see actual values unless you add sample data

The screenshot shows the Anypoint Studio interface with two main windows:

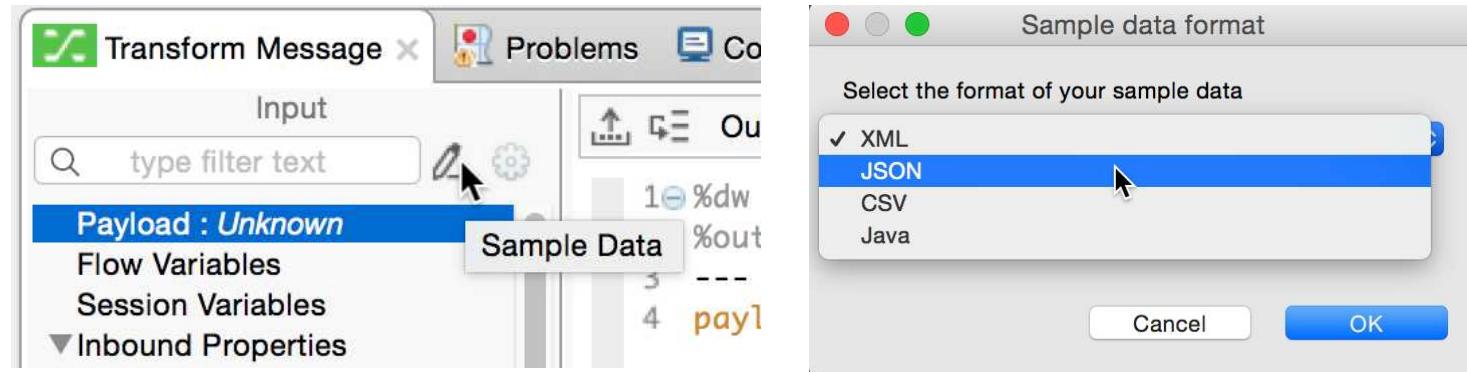
- Message Flow:** A top-level window titled "dataweave-demos" containing a "json2xml" flow. The flow consists of an "HTTP" connector, a "Set Payload" component, a "Transform Message" component, and a "Logger" component. The "Transform Message" component is highlighted.
- Transform Message Component Preview:** A detailed view of the "Transform Message" component. It shows the "General" tab with a success message ("There are no errors."), a "Display Name" of "Set Payload", and a "Value" field containing the JSON object `{"firstname": "Max", "lastname": "Mule"}`.
- Transform Message Editor:** A bottom window titled "Transform Message" showing the DataWeave code and its XML output preview. The DataWeave code is:

```
1 %dw 1.0
2 %output application/xml
3 ---
4 {
5     user: {
6         lName: payload.lastname,
7         fname: payload.firstname
8     }
9 }
```

The XML output preview is:

```
<?xml version='1.0' encoding='UTF-8'?>
<user>
    <lName xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true"/>
    <fname xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true"/>
</user>
```

Adding sample data to get live transformation previews



The screenshot shows the Mule Studio interface with the 'Transform Message' editor open. The 'Input' pane on the left displays a JSON payload:

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

. The 'Output' pane in the center shows the resulting XML output:

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

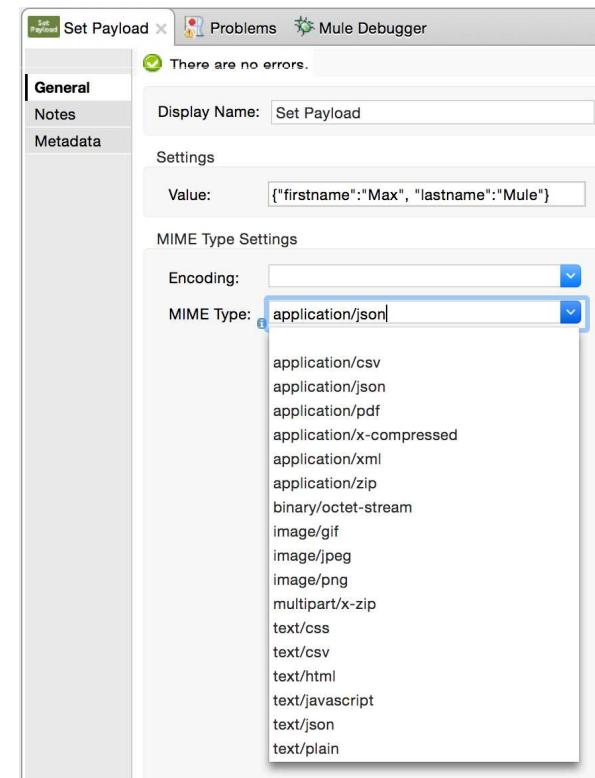
. The 'Payload' pane at the bottom shows the transformation logic:

```
1 %dw 1.0
2 %output application/xml
3 ---
4 {
5   user: {
6     lName: payload.lastname,
7     fname: payload.firstname
8   }
9 }
```

. At the bottom right, there are tabs for 'Preview' and 'Structure', with 'Preview' currently selected. The status bar at the bottom shows 'Context payload' and 'Payload'.

Setting input data MIME types

- When you run your application, you may get an error unless the MIME type for the input data has been set
 - It may be set automatically if the data is posted to the flow
 - See inbound properties content-type
 - Otherwise, you can set it
 - The transformers have a mimeType property

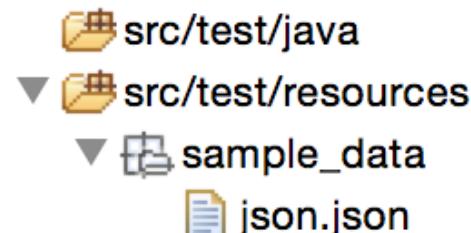


Where is the code?

- By default, it is placed inline

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

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



Walkthrough 5–2: Write your first DataWeave transformation

- Create a flow that receives POST requests at <http://localhost:8081/flights>
- Use the DataWeave Transform Message component
- Add sample data and use live preview
- Transform the form data from JSON to a Java object

The screenshot shows the Mule Studio interface with two main components:

- Mule United Airport Flight**: A browser window showing a search interface for flights from SFO to United Airlines.
- getFlightsFlow**: A Mule flow diagram with three components: an **HTTP** connector, a **Transform Message** component, and a **Logger**.

The **Transform Message** component is expanded to show its configuration:

- Input**: A JSON payload: `{"destination": "SFO", "airline": "united"}`.
- Output**: Set to `Payload`. The DataWeave script is:

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

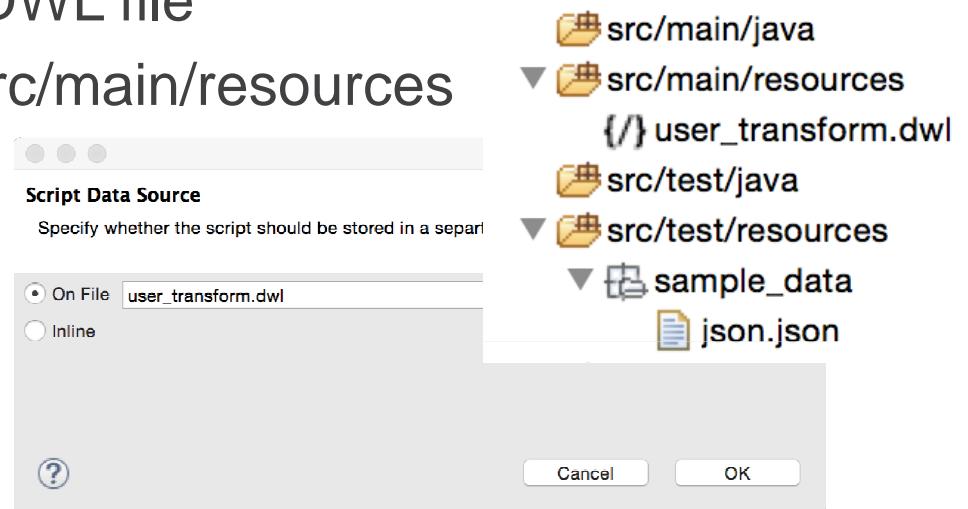
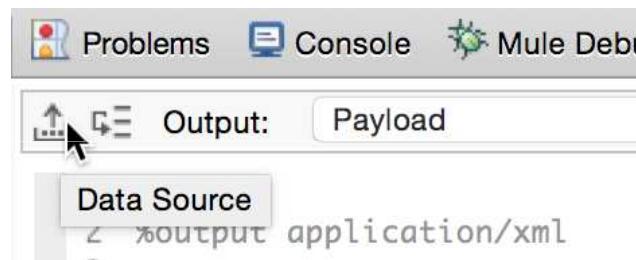
- Output** table:

Name	Value
root : LinkedHashMap	
destination : String	SFO
airline : String	united

At the bottom right of the interface, there is a **MuleSoft** logo.

Reusing transformations

- To place it an external file, click the Data Source button
 - Transform is saved in a DWL file
 - DWL files are stored in src/main/resources



- To reference an existing DWL, specify it in code

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

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

```
Transform Message x Problems Mule Debugger
Input
{
  "firstname": "Max",
  "lastname": "Mule"
}
Output: Payload
{
  "fname": payload.firstname,
  "lName": payload.lastname
}
Can not update preview. Validate your mappings.
```

- This will work

```
Transform Message x Problems Mule Debugger
Input
{
  "firstname": "Max",
  "lastname": "Mule"
}
Output: Payload
{
  "dw": 1.0
  "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>
```

Specifying attributes for XML output



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

The screenshot shows the 'Transform Message' editor in Mule Studio. The 'Input' tab displays a JSON payload: {"firstname": "Max", "lastname": "Mule"}. The 'Output' tab is set to 'Payload'. The 'Payload' tab contains the following Mule configuration:

```
1 %dw 1.0
2 %output application/xml
3 ---
4 {
5     user @(fname:payload.firstname): {
6         lName: payload.lastname
7     }
8 }
```

The resulting 'Output' XML is displayed in the 'Output' tab:

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

Writing expressions for XML input

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

Input	Transform	JSON Output
<user firstname="Max"> <lastname>Mule</lastname> </user>	payload	{ "user": { "lastname": "Mule" } }
	payload.user	{"lastname": "Mule" }
	{ fname: payload.user.@firstname, lname: payload.user.lastname }	{ "fname": "Max", "lname": "Mule" }

- Note: Be sure to update to Anypoint Studio 5.2.1 or later
 - Live preview for XML sample input does not work in 5.2.0

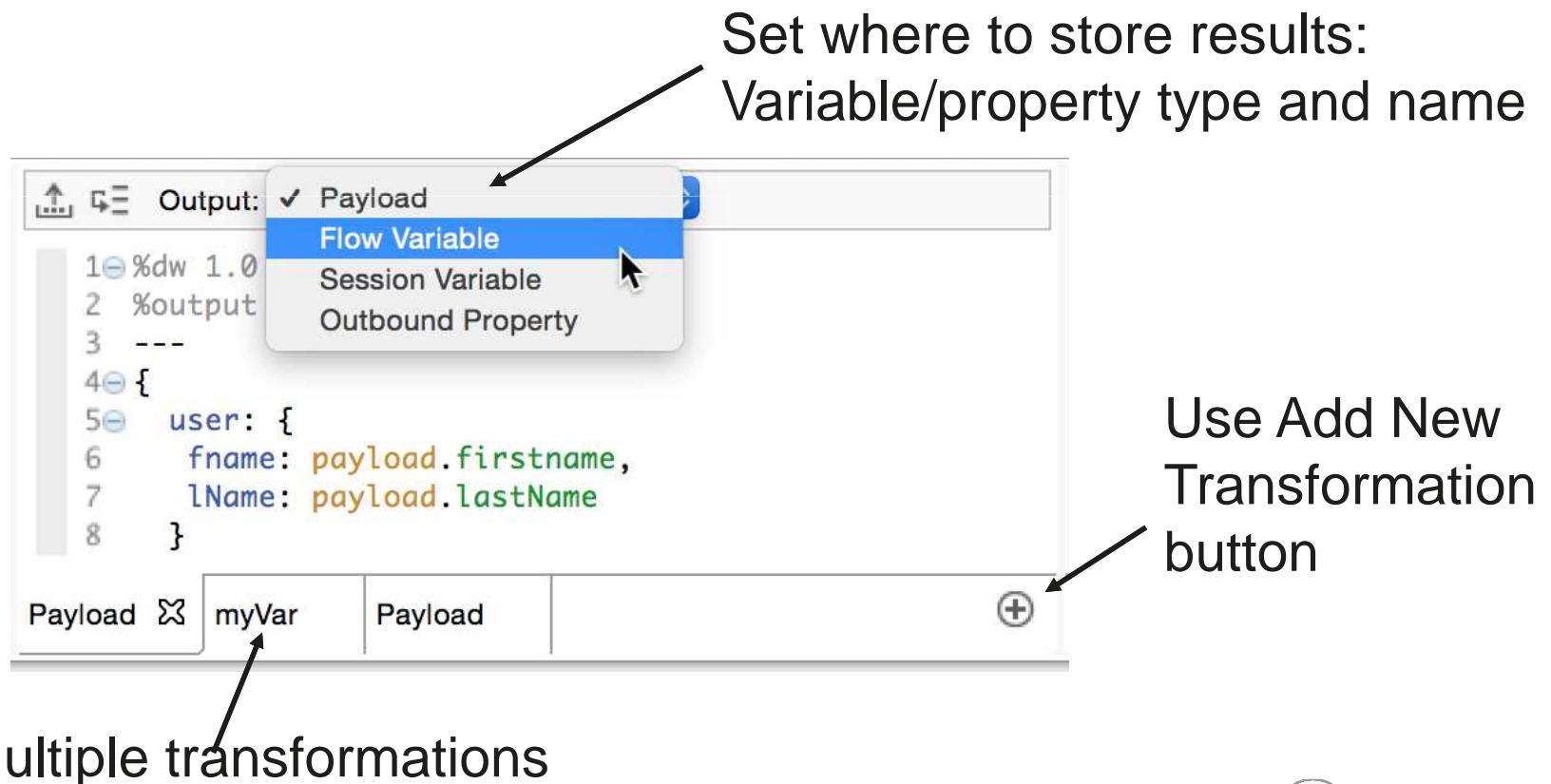


- So far, we referenced payload
- You can also reference
 - flowVars
 - inboundProperties
 - outboundProperties
- This is not MEL!
 - Do not preface these values by “message.” or use #[]

```
%dw 1.0
%output application/xml
---
{
    a: flowVars.userName
}
```

Creating multiple transformations

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



Walkthrough 5–3: Transform basic Java, JSON, and XML data structures

- Write transformations to store data in multiple outputs as different types of data
- Create a 2nd transformation to store destination in a flow variable
- Create a 3rd transformation to output data as JSON
- Create a 4th transformation to output data as XML

The screenshot shows the 'Transform Message' editor in Mule Studio. The 'Input' tab displays a JSON payload: `{"destination": "SFO", "airline": "united"}`. The 'Output' tab is set to 'Flow Variable' with the name 'xml'. The 'Script' pane contains a Java script (DWL) for transforming the input into XML:

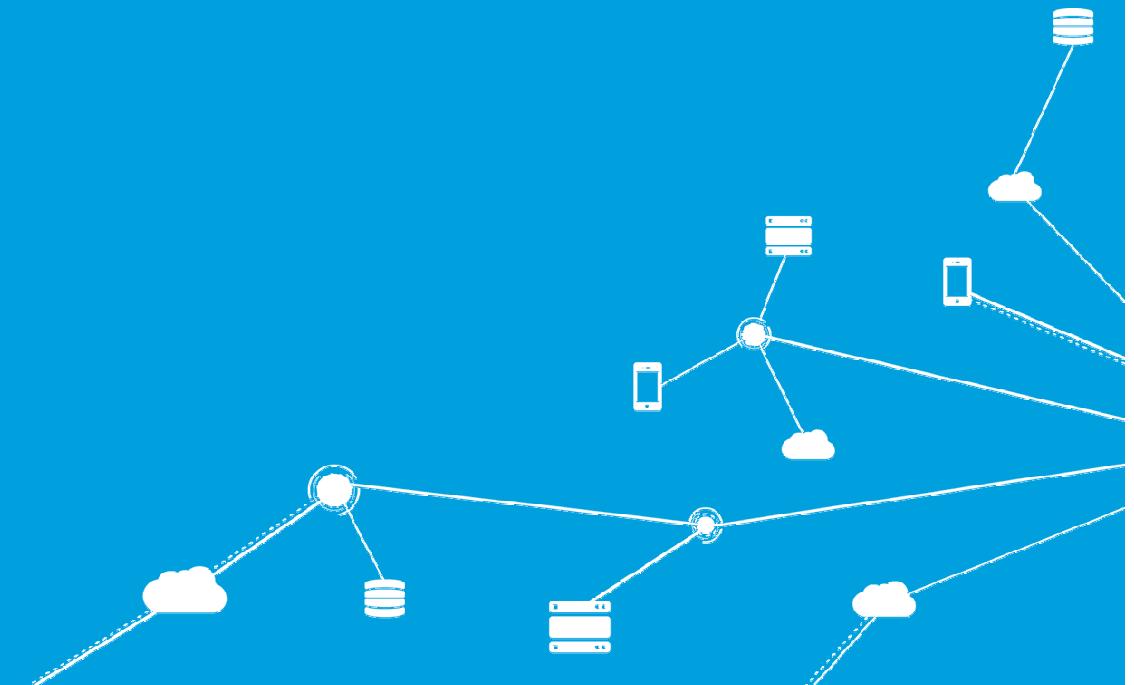
```
1 %dw 1.0
2 %output application/xml
3 ---
4 data: {
5   hub: "MUA",
6   flight @{airline: payload.airline}: {
7     code: payload.destination
8   }
9 }
```

The 'Output' pane shows the resulting XML output:

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

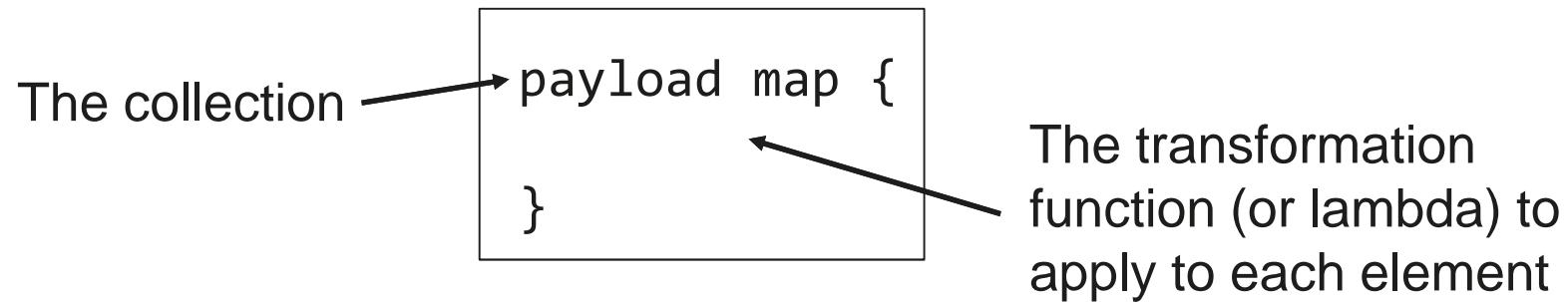
Below the editor, tabs for 'Context', 'payload', 'Payload', 'destination', 'json', 'xml', 'Preview', and 'Structure' are visible.

Transforming complex data structures with DataWeave





- Use the **map** operator to apply a transformation to each element in a collection
 - A collection can be JSON or Java arrays or XML repeated elements



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

The transformation function

- 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 1.0 %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 1.0 %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>

Using the index as a key in a transformation function

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

Input	Transform	Output
[{"firstname": "Max", "lastname": "Mule"}, {"firstname": "Molly", "lastname": "Mule"}]	%dw 1.0 %output application/json --- payload map { num: \$\$, (\$\$): \$ } payload map { num: \$\$, '\$\$': \$ }	[{ "num": 0, "0": { "firstname": "Max", "lastname": "Mule" } }, { "num": 1, "1": { "firstname": "Molly", "lastname": "Mule" } }]
	payload map { 'num\$\$': \$ }	[{ "num0": { "firstname": "Max", "lastname": "Mule" } }, { "num1": { "firstname": "Molly", "lastname": "Mule" } }]

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
[{"firstname": "Max", "lastname": "Mule"}, {"firstname": "Molly", "lastname": "Mule"}]	%dw 1.0 %output application/xml --- users: payload map { fname: \$.firstname, lname: \$.lastname }	Cannot coerce an array to an object (starting with 3.7.1)
	users: {(payload map { fname: \$.firstname, lname: \$.lastname })}	<users> <fname>Max</fname> <lname>Mule</lname> <fname>Molly</fname> <lname>Mule</lname> </users>

Writing expressions for XML output (cont)

Input	Transform	Output
[{"firstname":"Max", "last Name":"Mule"}, {"firstname":"Molly", "lastName":"Mule"}]	users: {{payload map { fname: \$.firstname, lname: \$.lastname }}}	<users> <fname>Max</fname> <lname>Mule</lname> <fname>Molly</fname> <lname>Mule</lname> </users>
	users: {{ payload map { user: { fname: \$.firstname, lname: \$.lastname } }}}	<?xml version='1.0' encoding='UTF-8'?> <users> <user> <fname>Max</fname> <lname>Mule</lname> </user> <user> <fname>Molly</fname> <lname>Mule</lname> </user> </users>

Writing expressions for XML input

- Use * to reference repeated elements

Input	Transform	JSON Output
<pre><users> <user firstname="Max"> <lastname>Mule</lastname> </user> <user firstname="Molly"> <lastname>Mule</lastname> </user> </users></pre>	payload	{ "users": { "user": { "lastname": "Mule" } } }
	payload.users	{ "user": { "lastname": "Mule" } }
	payload.users.user	{ "lastName": "Mule" }
	payload.users.*user	[{ "lastName": "Mule" }, { "lastName": "Mule" }]
	payload.users.*user map { fname: \$.@firstname, lname: \$.lastName }	[{ "fname": "Max", "lname": "Mule" }, { "fname": "Molly", "lname": "Mule" }]

Walkthrough 5-4: Transform complex data structures

- Create a new flow that receives GET requests at <http://localhost:8081/static>
- Transform a JSON array of objects to Java, JSON, and XML
- Explicitly set the MIME type of the data to be transformed
- Transform XML with repeated elements to XML and JSON

Note: You will work with CSV data in the Processing Records module

The screenshot shows the Mule Studio interface with a 'Set Payload' component selected. The 'General' tab is active, displaying the following configuration:

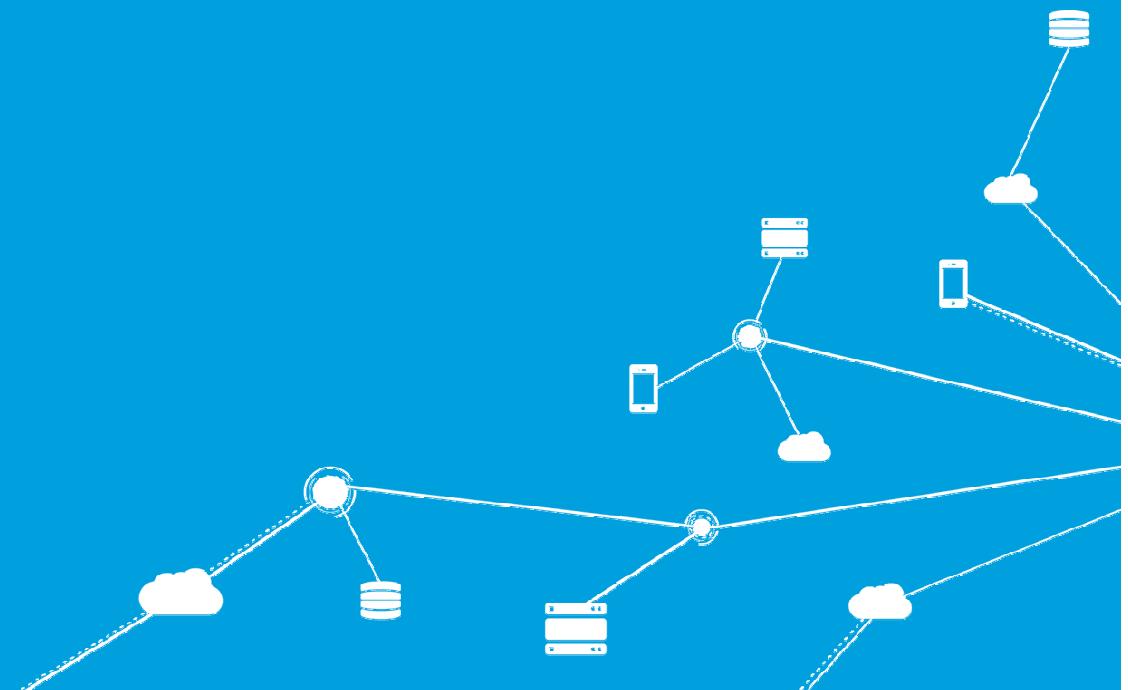
- Display Name:** Set Payload
- Value:** `[{"airlineName": "United", "price": 400, "departureDate": "2015/03/20", "planeType": "Boeing 737"}, {"airlineName": "United", "price": 945, "departureDate": "2015/09/11", "planeType": "Boeing 757"}]`
- MIME Type Settings**:
 - Encoding:** `%dw 1.0`
 - MIME Type:** application/json

On the right side, the 'Output' pane shows the generated XML output:

```
<?xml version='1.0' encoding='UTF-8'?>
<flights>
  <flight0>
    <airlineName>United</airlineName>
    <price>400</price>
    <departureDate>2015/03/20</departureDate>
    <planeType>Boeing 737</planeType>
    <originination>MUA</originination>
    <flightCode>ER38sd</flightCode>
    <availableSeats>0</availableSeats>
    <destination>SFO</destination>
  </flight0>
  <flight1>
    <airlineName>United</airlineName>
    <price>945</price>
    <departureDate>2015/09/11</departureDate>
    <planeType>Boeing 757</planeType>
    <originination>MIA</originination>
  </flight1>
</flights>
```

The bottom navigation bar includes tabs for 'Payload' (selected), 'Preview', and 'Structure'.

Using DataWeave operators



DataWeave reference documentation

- <https://developer.mulesoft.com/docs/dataweave>

From an Incoming Mule
Message

4. Operators
 4.1. Map
 4.2. Map Object
 4.3. Pluck
 4.4. Filter
 4.5. Remove
 4.6. Default
 4.7. When / Otherwise
 4.8. Unless / Otherwise
 4.9. AND
 4.10. OR
 4.11. Concat
 4.12. AS (Type Coercion)
 4.13. Flatten
 4.14. Size Of
 4.15. Push
 4.16. Reduce
 4.17. Join By
 4.18. Split By
 4.19. Order By
 4.20. Group By
 4.21. Distinct By

Weave Documentation

MuleSoft Inc.

[IMPORTANT] You're viewing documentation written using our new documentation platform. The following content is official MuleSoft documentation. If you have any feedback, please contact documentation@mulesoft.com.

1. Introduction

The DataWeave Language is a powerful template engine that allows you to transform data to and from any kind of format (XML, CSV, JSON, Pojos, Maps, etc).

1.1. Getting started:

In order to show the power of DataWeave, here are a few examples to get started.

Formatting operators



Input	Transform	Output
	%dw 1.0 %output application/xml ---	
{ "name": "max_mule" }	n: upper payload.firstname	<n>MAX_MULE</n>
	n: lower payload.firstname	<n>Max_mule</n>
	n: camelize payload.name	<n>maxMule</n>
	n: capitalize payload.name	<n>Max Mule</n>
	n: dasherize payload.name	<n>max-mule</n>
	n: pluralize payload.name	<n>max-mules</n>
	n: upper (dasherize payload.name)	<n>MAX-MULE</n>
{ "name": "max mules" }	n: singularize payload.name	<n>max mule</n>
	n: underscore payload.name	<n>max_mules</n>
{"place": 2}	n: ordinalize payload.place	<n>2nd</n>

Using the as operator for type coercion

```
price: payload.price as :number
```

- Defined types include
 - :string
 - :number
 - :boolean
 - :object
 - :array
 - :date, :time, :timezone, :datetime, :localdatetime, :period
 - :regex

Specifying custom data types

- Specify inline

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

- Assign a custom name with the type directive
 - Name has to be all lowercase letters
 - No special characters, uppercase letters, or numbers

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

Using format patterns

- Use metadata **format** key to format numbers and dates
- Inline

```
tax: (tax * 100) as :number {format: "##.#" } ++ "%"  
someDate as :datetime {format: "yyyyMMddHHmm"}
```

For pattern letters, see Java `DateTimeFormatter` class API

<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

- With custom type

```
%type currency = :number {format: "##"}  
price: $.price as :currency
```

Conditional logic operators



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

Input	Transform	Output
	%dw 1.0 %output application/xml ---	
{"firstname":"Max", "lastname":"Mule"}	n: payload.nickname default payload.firstname	<n>Max</n>
{ "firstname": "Max", "lastname": "Mule", "nickname": "" }	n: payload.nickname when payload.nickname != "" otherwise payload.firstname	<n>Max</n>
	n: payload.firstname unless payload.nickname != "" otherwise payload.nickname	<n>Max</n>
	n: payload.firstname unless payload.nickname != "" or payload.firstname != "" otherwise payload.nickname	<n></n>
	n: payload.lastname unless payload.nickname != "" and payload.firstname != "" otherwise payload.nickname	<n>Max</n>



- +
 - -
 - *
 - /
-
- max: returns the highest number in an array or object
 - min: returns the lowest number in an array or object
 - sizeOf: returns number of elements in an array



- concat
`n: payload.firstname ++ " " ++ payload.lastname`
- orderBy
- distinctBy
- groupBy
- replace
- matches
- regex
- More...

Walkthrough 5-5: Use DataWeave operators

- Format strings, dates, and numbers
- Convert data types
- Replace data values using pattern matching
- Order data, filter data, and remove duplicate data
- Define and use custom data types
- Transform objects to POJOs

The screenshot shows the Mule Studio interface with the 'Transform Message' tab selected. The 'Input' pane displays an XML payload:

```
<?xml version='1.0' encoding='UTF-8'?>
<ns2:listAllFlightsResponse
  xmlns:ns2="http://soap.training.mulesoft.com/"><return
  airlineName="United"><code>A1B2C3</code><departureDate>2015/10/20</departureDate><destination>SF0</destination><emptySeats>40</emptySeats><origin>MUA</origin><planeType>Boeing 737</planeType><price>400.0</price></return
  airlineName="Delta"><code>A1B2C4</code><departureDate>2015/10/21</departureDate><destination>LAX</destination>
```

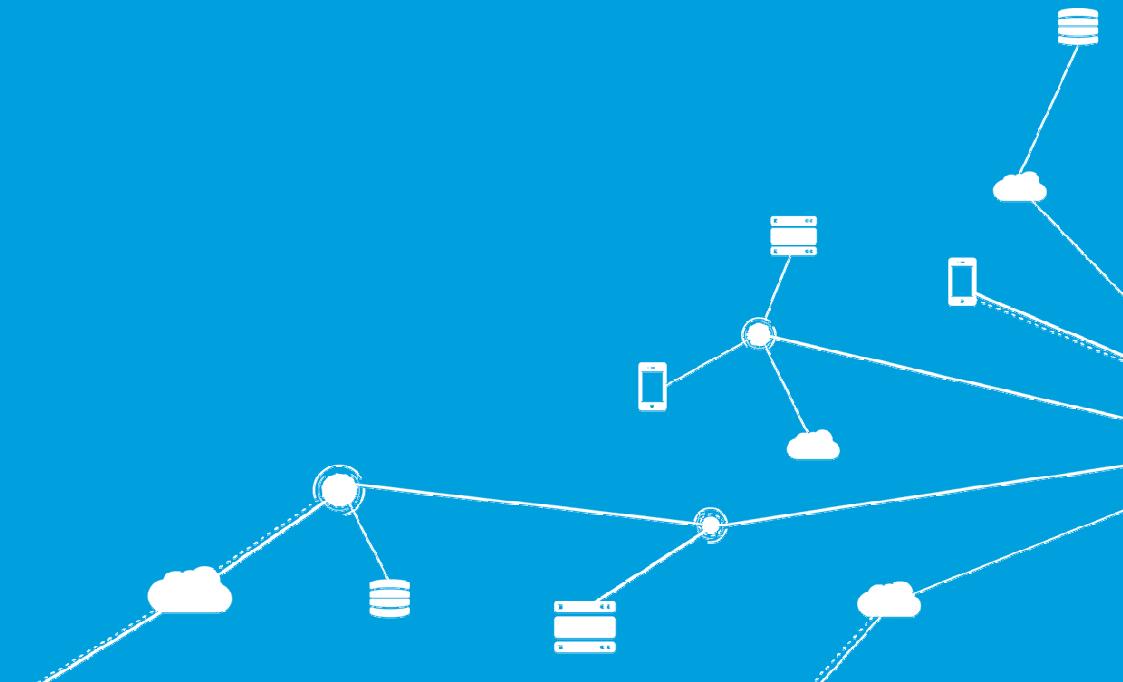
The 'Output' pane shows the DataWeave script:

```
%dw 1.0
%output application/java
%type currency = :number {format: "###"}
%type flight = :object {class: "com.mulesoft.training.Flight"}
---
payload.listAllFlightsResponse.*return map {
    destination: $.destination,
    price: $.price,
    planeType: upper $.planeType replace /(Boing)/ with "Boeing",
    departureDate: $.departureDate as :date {format: "yyyy/MM/dd"} as :string {format: "MMM dd, yyyy"},
    availableSeats: $.emptySeats as :number
} as :flight orderBy $.departureDate orderBy $.price distinctBy $.filter ($.availableSeats != 0)
```

The 'Output' pane also displays the resulting Java object structure and its values:

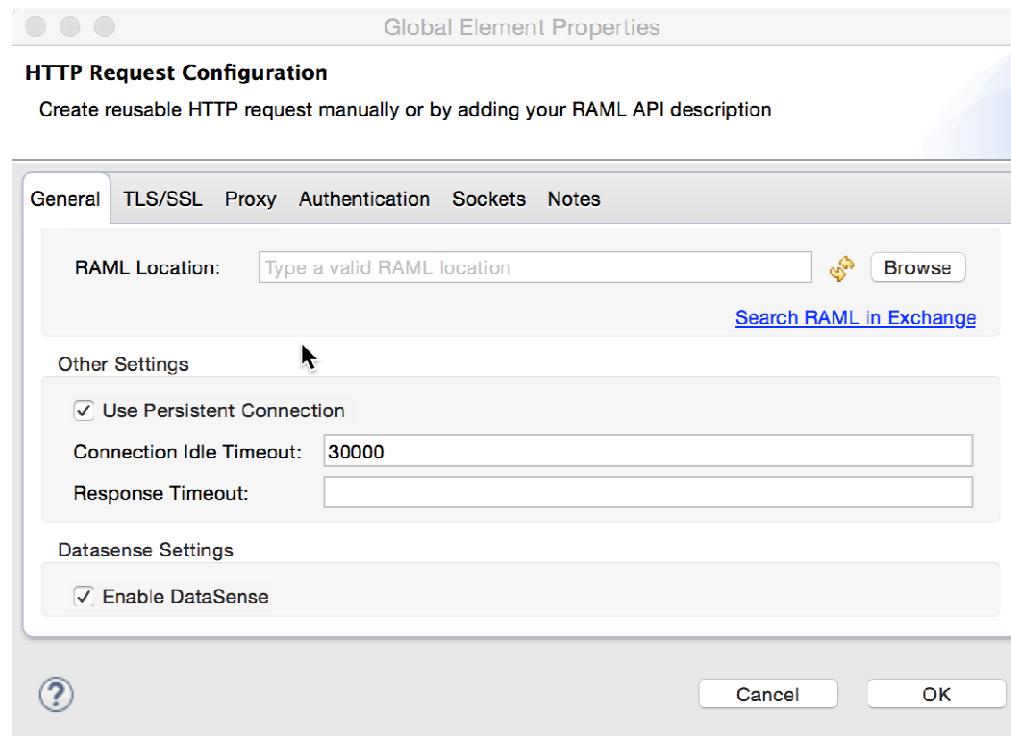
Name	Value
root : ArrayList	
[0] : Flight	airlineName : String availableSeats : Integer 10 departureDate : String Oct 21, 2015 destination : String LAX flightCode : String origination : String planeType : String BOEING 737 price : Double 199.99
[1] : Flight	airlineName : String availableSeats : Integer 23 departureDate : String Oct 20, 2015

Using DataWeave to transform data structures that have associated metadata



DataSense and metadata

- Many connectors can be DataSense enabled

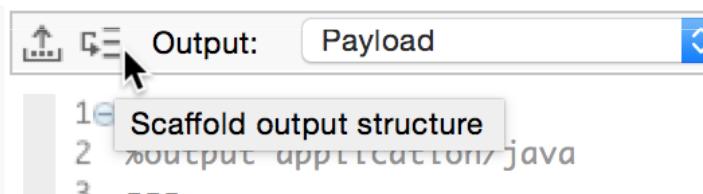


Transformations and metadata

- If message input and/or output has metadata, Anypoint Studio will build an initial scaffolding for the transformation based on it
 - Based on the processors on either side of the transformer
 - The scaffolding is just the starting transformation code automatically written based on metadata
 - You may need to modify this a little or a lot depending upon what the metadata is and what you want to accomplish
- For this reason, it is best to add processors first and then the Transform Message component

Updating the scaffolding

- If you add new processors upstream or downstream or add metadata to existing ones, you can update the scaffolding
 - Refresh metadata and/or recreate scaffolding

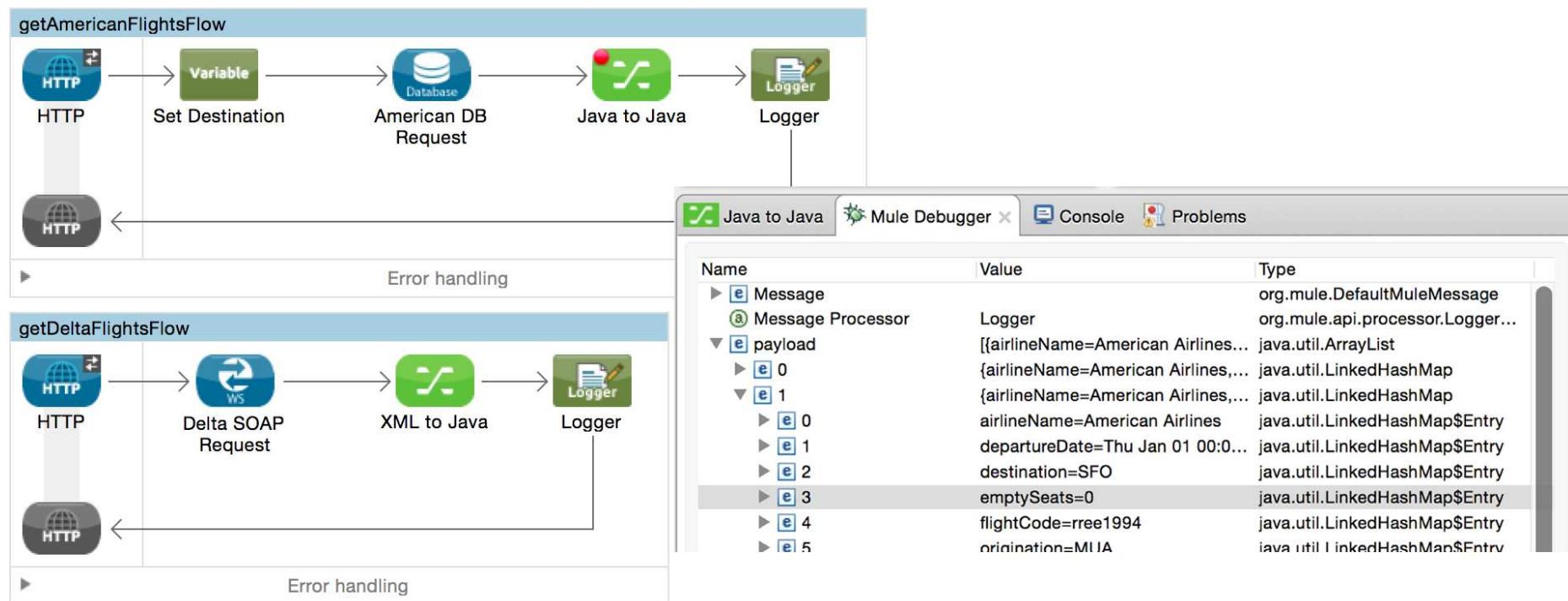


- To recreate scaffolding from metadata, click Scaffold output structure button
 - Deletes existing DataWeave code and re-scaffolds output

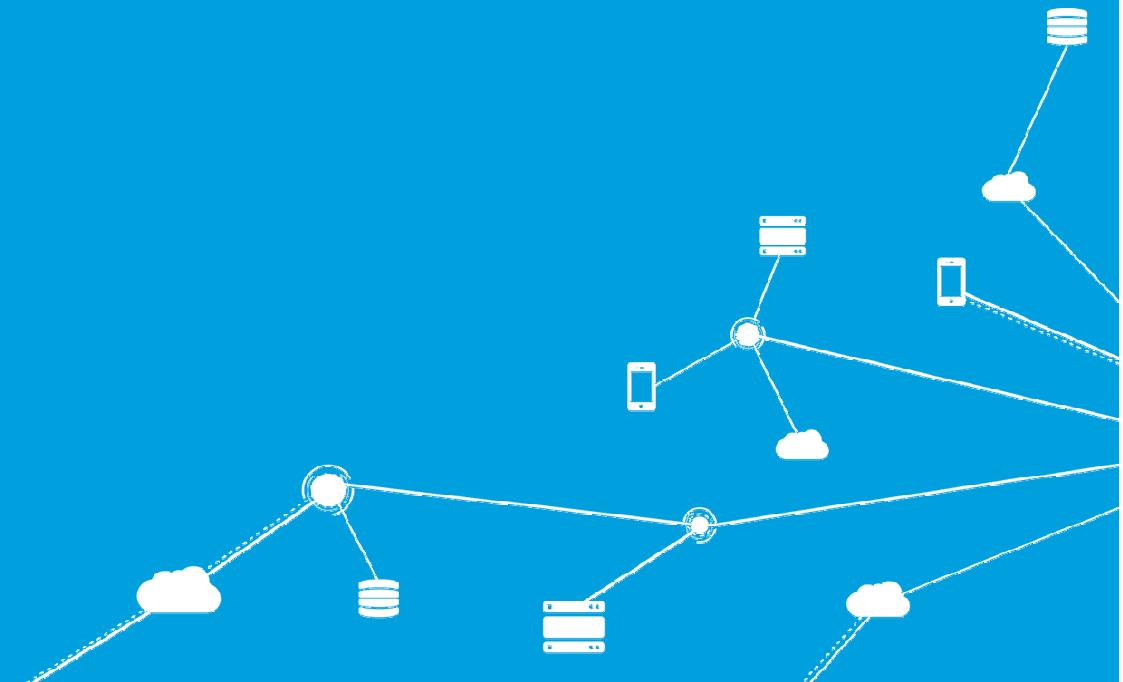
Walkthrough 5–6: Transform data structures that have associated metadata



- Use DataWeave to transform American flight data from a collection of Java objects to one with a different data structure
- Use DataWeave to transform the Delta flight data from XML to a collection of Java objects



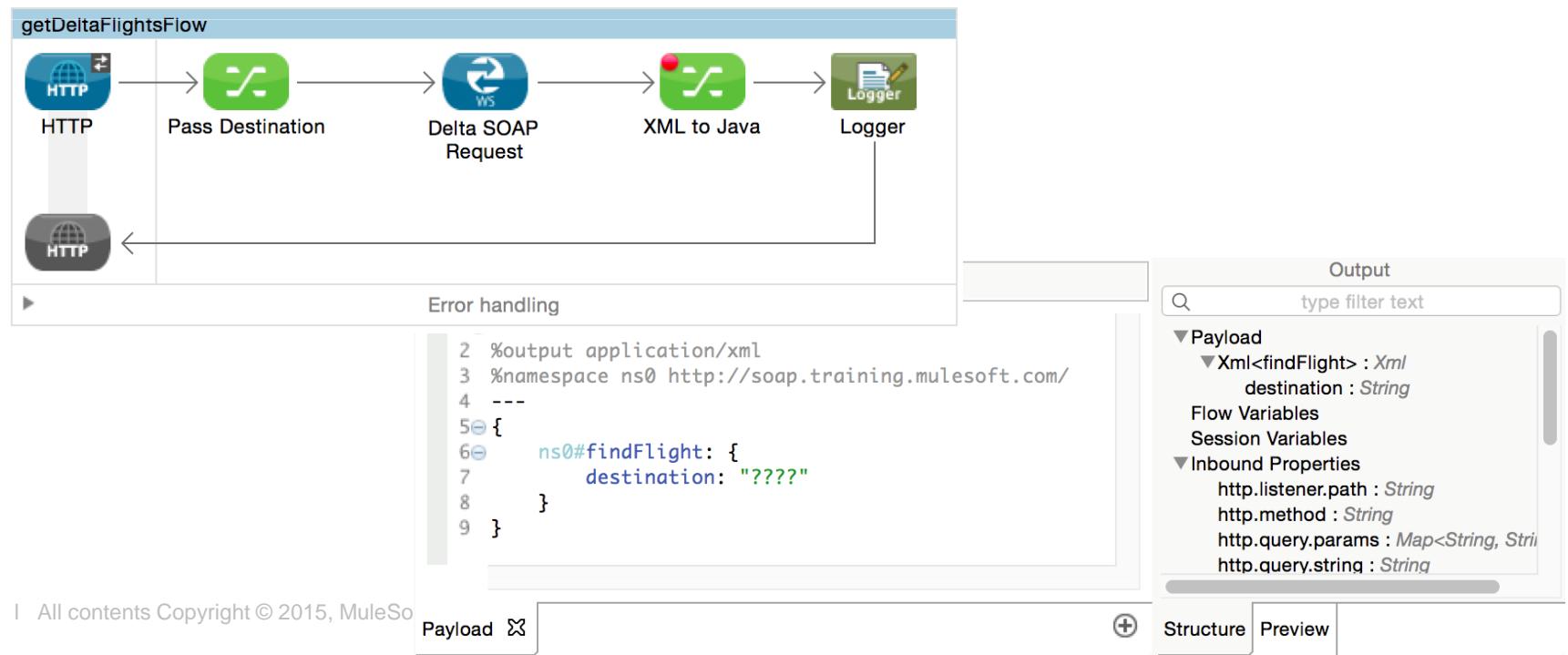
Using DataWeave to pass arguments to SOAP web services



Using DataWeave to specify SOAP web service input arguments



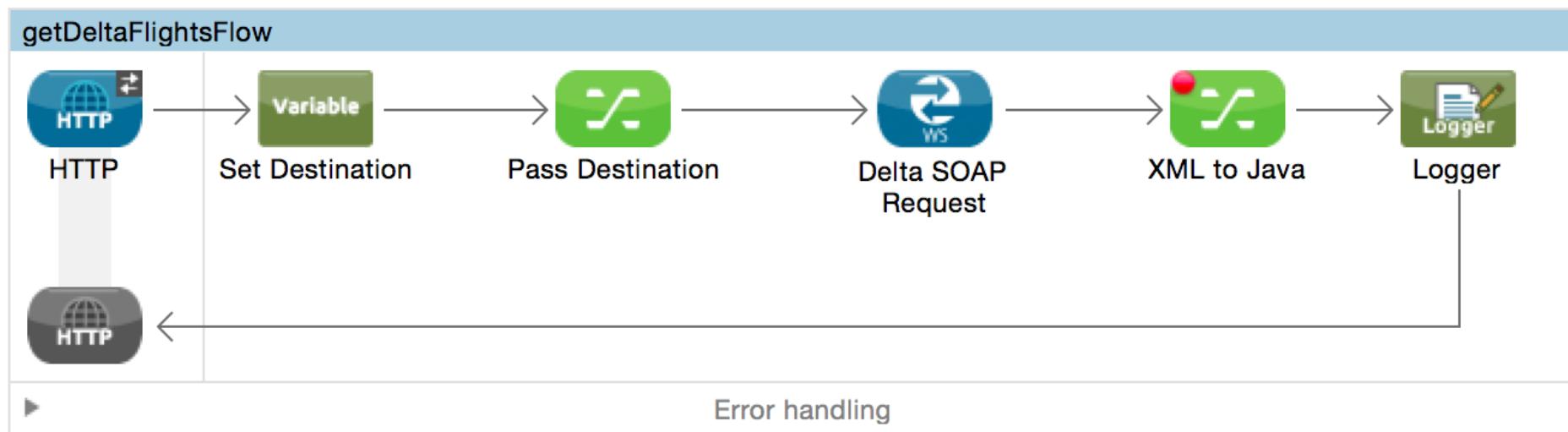
- When you drag-and-drop a DataWeave message transformer before a Web Service Consumer that has input arguments, a scaffold for populating the arguments will be created automatically



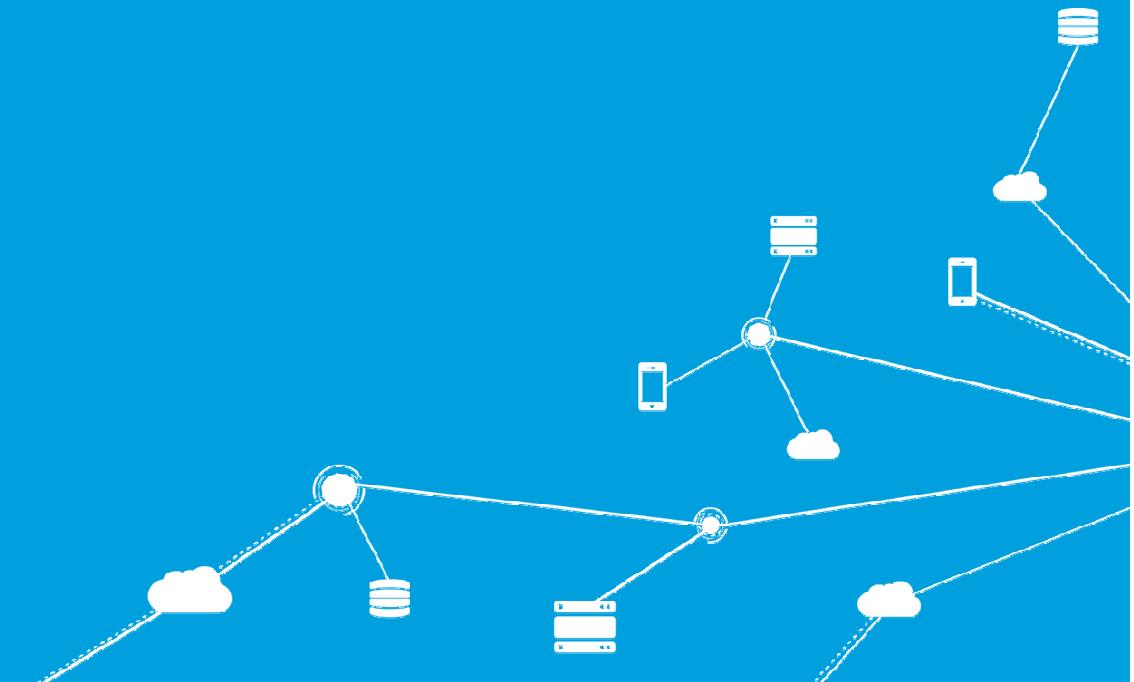
Walkthrough 5-7: Pass arguments to a SOAP web service



- Return the flights for a specific destination instead of all the flights
- Change the web service operation invoked to one that requires a destination as an input argument
- Use DataWeave to pass an argument to a web service operation
- Create a variable to set the destination to a dynamic query parameter value

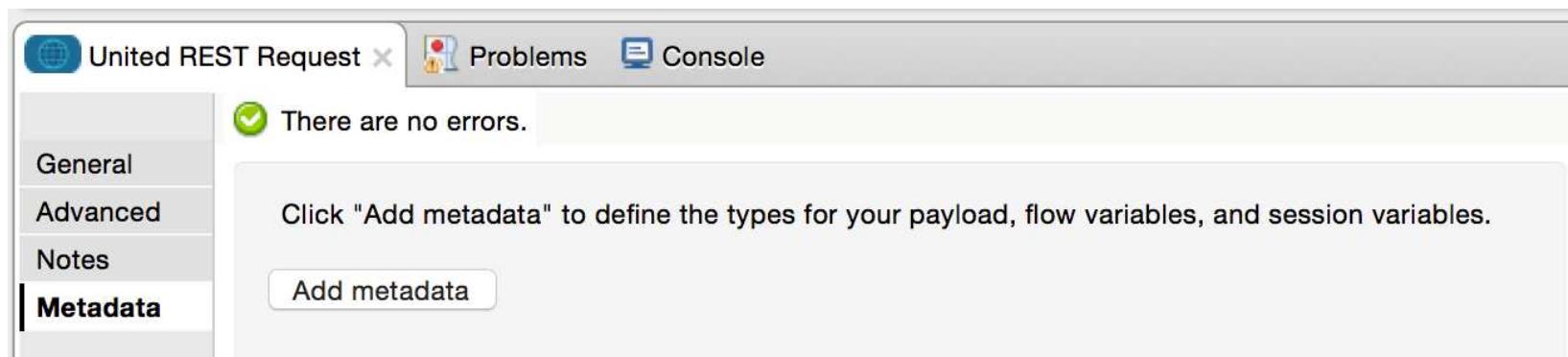


Using DataWeave to transform data structures that need custom metadata



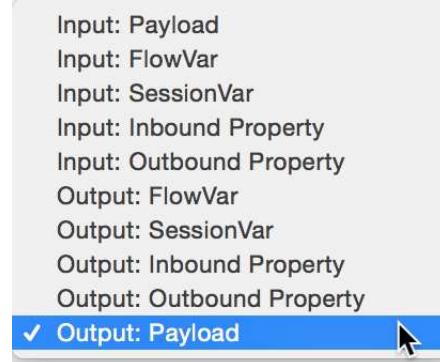
A more metadata aware Anypoint Studio and Mule runtime

- New in 3.7, Mule now tracks the payload type internally so that metadata can be used during transformations
- In Anypoint Studio, you can now provide design time metadata to message processors and declare the type of the payload
 - Provides content-assist capabilities
 - Gives you visibility into your payload everywhere

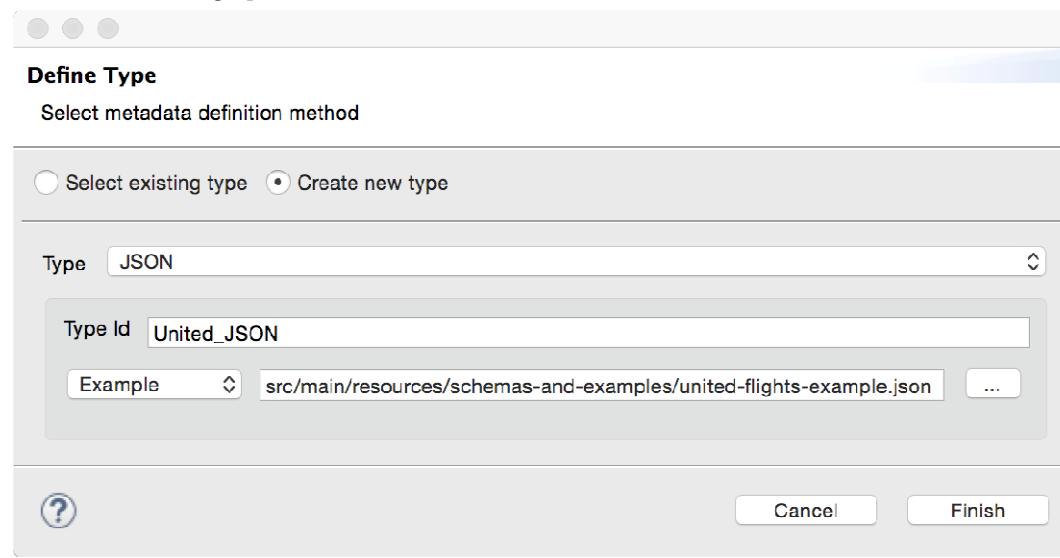


Adding metadata

- Specify what the metadata is for

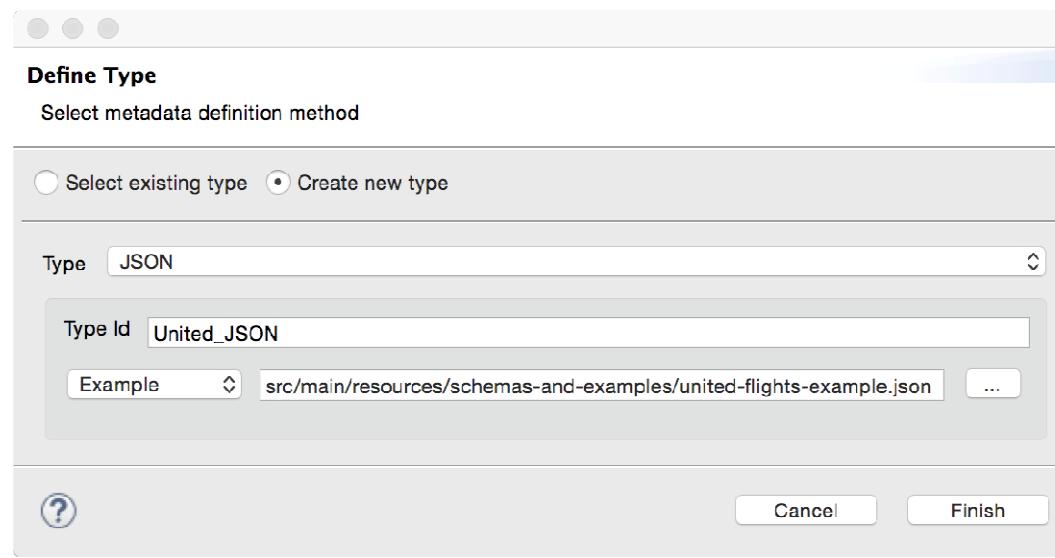


- Specify the data type: JSON, XML, CSV, JAVA



Adding metadata

- For XML and JSON
 - Provide schema or example file
- For CSV
- Specify the data type: JSON, XML, CSV, JAVA



Walkthrough 5–8: Transform a data source to which you add custom metadata



- Add custom metadata to an HTTP Request endpoint
- Use DataWeave to transform the United flight data from JSON to a collection of Java objects

The screenshot shows the Mule Studio interface with the following details:

- Project Bar:** United REST Request
- Toolbars:** Problems, Console, Mule Debugger
- Metadata Panel:** Shows "There are no errors." and an "Add metadata" button.
- Flow Editor:** A flow named "getUnitedFlightsFlow" is displayed.
 - Starts with an **HTTP** connector.
 - An arrow points to a **Variable** component labeled "Set Destination".
 - From the Variable component, an arrow points to a second **HTTP** connector labeled "United REST Request".
 - From the United REST Request connector, an arrow points to a **JSON to Java** component.
 - From the **JSON to Java** component, an arrow points to a **Logger** component.
 - A feedback loop arrow originates from the **Logger** component and points back to the **HTTP** connector at the start of the flow.
- Properties Panel:** Shows the **Input** tab with the payload type set to "User Defined/United_JSON". The output section shows the following structure:
 - Payload:** Unknown : Unknown
 - Flow Variables:** destination : Map<String, String>
 - Session Variables:**
 - Inbound Properties:**
 - http.listener.path : String
 - http.method : String
 - http.query.params : Map<String, String>
 - http.query.string : String
 - http.remote.address : String
 - http.request.path : String
 - http.request.uri : String
- Buttons:** Refresh Metadata

Using DataWeave with CSV data

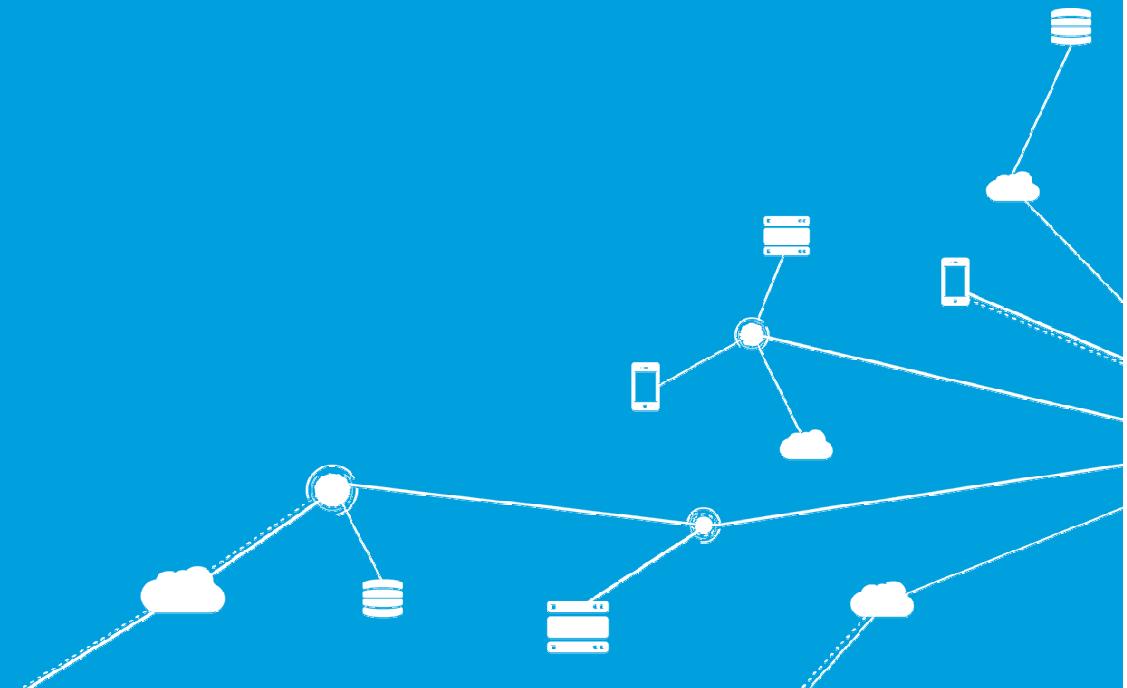
- You will apply this technique to transform CSV data in a later module, Processing Records
 - Add metadata to File endpoint using example CSV file

Using dw() to query data



- The DataWeave universal language for data access can not only be used for transformation, but also for querying data throughout your flow
- Using the dw() function, you can quickly query data and use it to log information from payloads, route data, or extract it for message enrichment

Summary



Summary

- In this module, you learned about the different types of transformers and the DataWeave framework
- There are Java object, message and variable, content, and script transformers
- The Parse Template transformer loads the content of an external file (that can have MEL expressions)
- The DataWeave Transform Message component can be used in place of most other transformers
- DataWeave a full-featured and fully native framework for querying and transforming data on Anypoint Platform
- DataWeave is new in Mule 3.7



- For the DataWeave component, you set the output type and a transformation expression using the DataWeave data transformation language
 - A JSON-like language built just for data transformation use cases
- DataWeave transformations are fast and reusable
- DataWeave is fully integrated with Anypoint Studio and DataSense
 - There is a graphical interface that is aware of associated metadata for input and output structures
 - Easy to use with data sources that have associated metadata