



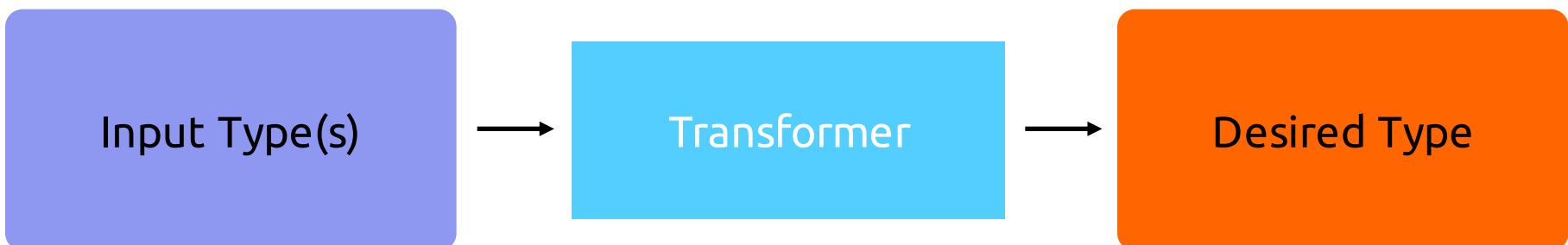
MuleSoft®

# Module 5: Transforming Data



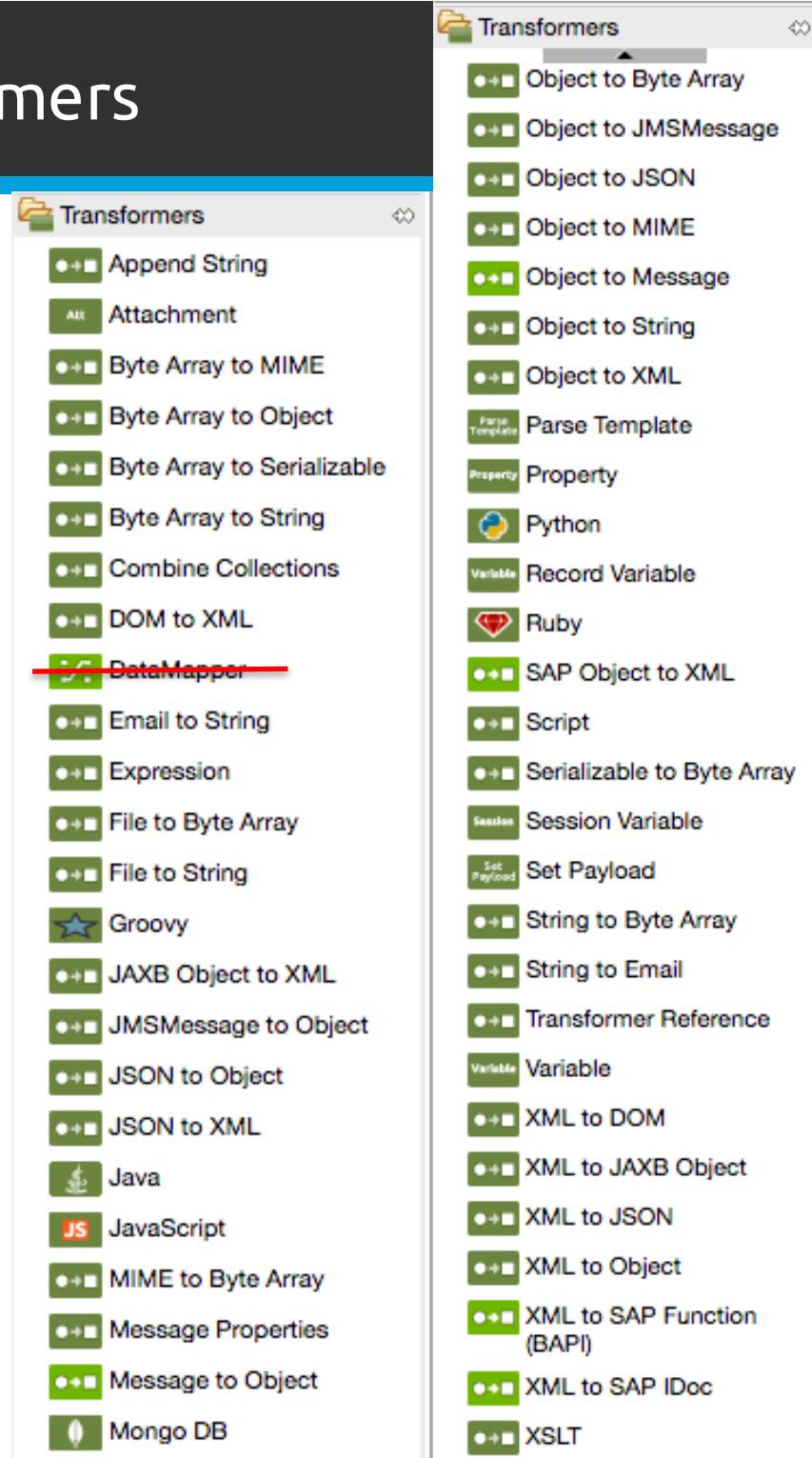
# Transforming data using transformers

- 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



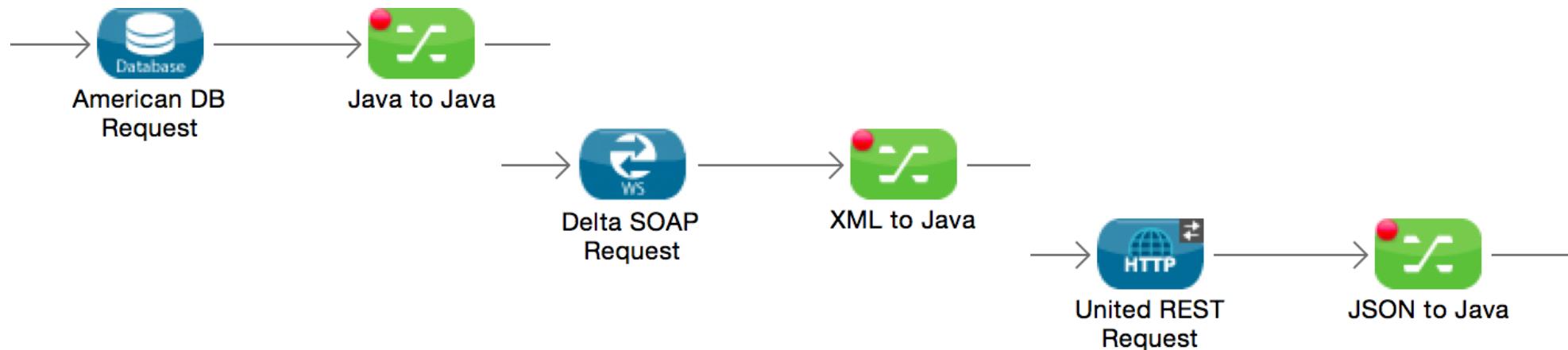
- Introduced in Mule 3.3 (2012)
- Provided a way for users to graphically manipulate data and work with many different data formats
- Customers loved the direction, but wanted much more in terms of capabilities, performance, and ease of use
- DataWeave is the new solution for the future
- DataMapper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

# Transforming data using DataWeave (new in 3.7!)



- 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 XML to Java component configuration.

**XML to Java Component Configuration:**

- Input:** type filter text
- Output:** Payload
- Script (MEL):**

```
1 %dw 1.0
2 %output application/java
3 ---
4 payload.findFlightResponse.*return map {
5     airlineName: $.airlineName,
6     departureDate: $.departureDate,
7     destination: $.destination,
8     emptySeats: $.emptySeats as :number,
9     flightCode: $.code,
10    origination: $.origin,
11    planeType: $.planeType replace /(Boeing)/ with "Boeing"
12    price: $.price as :number {format: "###.##"}
13 }
```

- Output:** (Table)

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

# Objectives

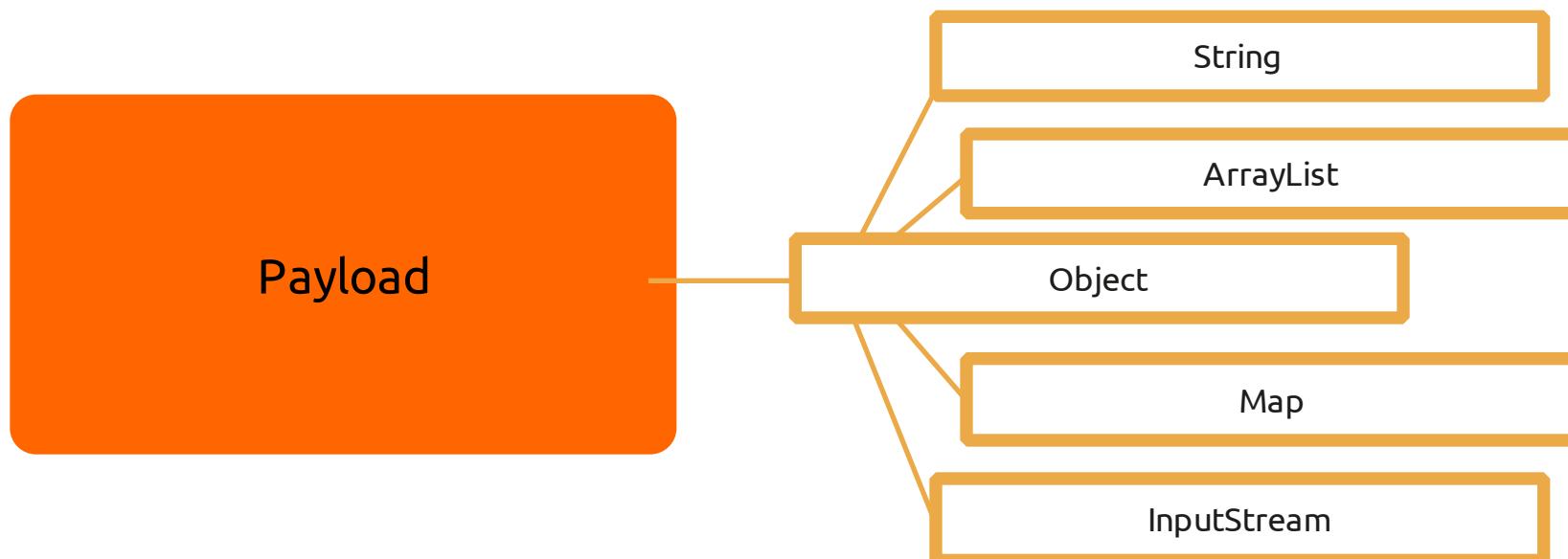
- In this module, you will learn:
  - About the different types of transformers
  - To use the DataWeave Transform Message component
  - To write DataWeave expressions for basic and complex XML, JSON, and Java transformations
  - To use DataWeave with data sources that have associated metadata
  - To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

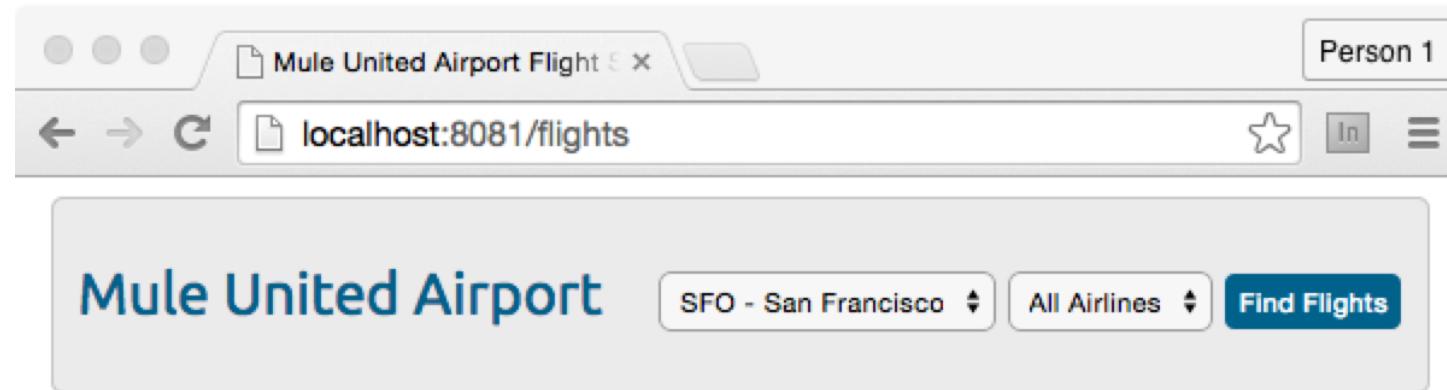
# Parse Template transformer



- 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

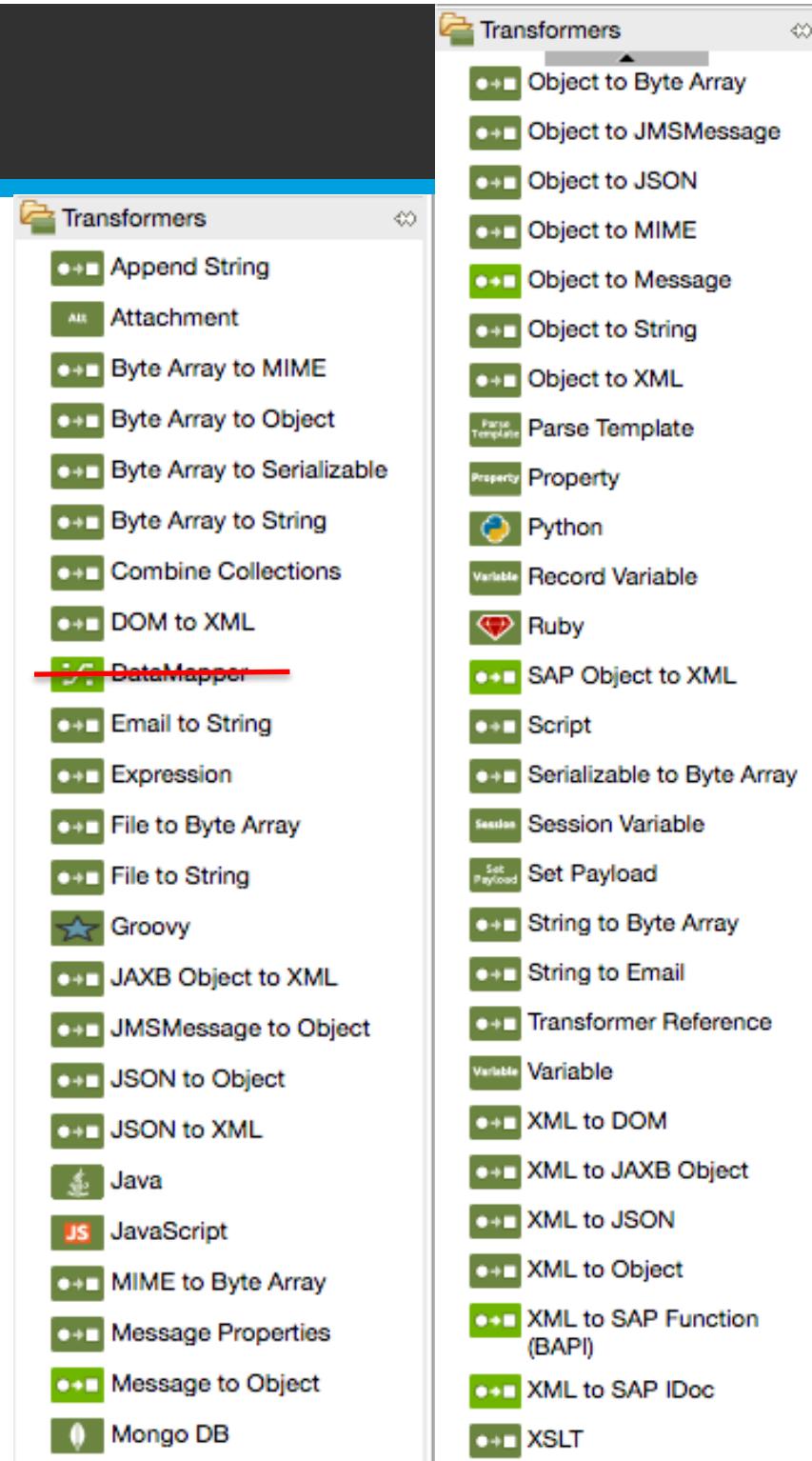
# Walkthrough 5-1: Load external content into a message

- 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 3<sup>rd</sup> 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 data transformation use cases



- 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

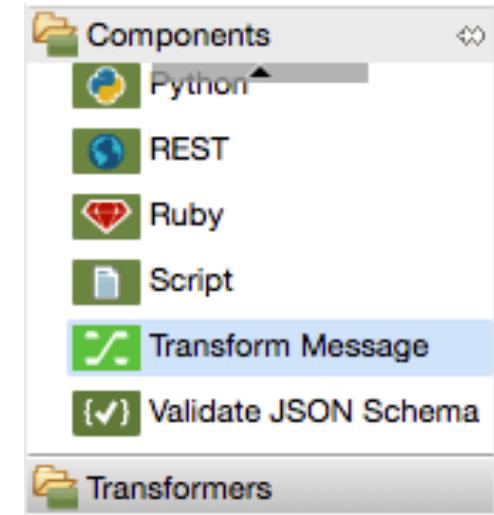


- Underneath, DataWeave includes a connectivity layer and engine that is fundamentally different from other transformation technologies
- It contains a data access layer that indexes content and accesses the binary directly, without costly conversions
  - Enables larger than memory payloads
  - Random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

A screenshot of the Anypoint Studio interface showing the Transform Message properties view. The 'Input' tab shows a JSON payload: {"firstname": "Max", "lastname": "Mule"}. The 'Output' tab shows a DataWeave script:

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

The 'Output' section on the right shows the transformed output:

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

Below the tabs are buttons 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 three main windows:

- Message Flow:** A diagram of a data pipeline. It starts with an "HTTP" icon, followed by a "Set Payload" step, a "Transform Message" step, and a "Logger" step. The "Transform Message" step is highlighted.
- Set Payload:** A configuration dialog for the "Set Payload" step. It shows a green checkmark and the message "There are no errors." It includes fields for "Display Name" (Set Payload) and "Value" ({"firstname":"Max", "lastname":"Mule"}).
- Transform Message:** A configuration dialog for the "Transform Message" step. It shows the input payload (a DataWeave script) and its output XML representation. The input payload is:

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

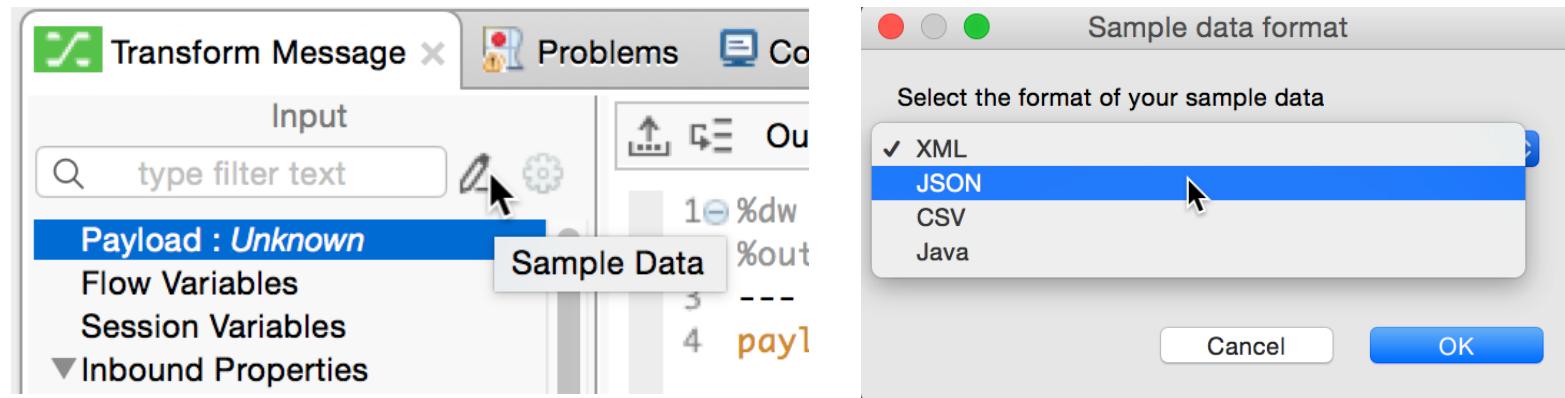
The output XML is:

```
<?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>
```



MuleSoft®

# Adding sample data to get live transformation previews



The screenshot shows the 'Transform Message' editor with sample data added. The 'Input' section contains the JSON payload: 

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

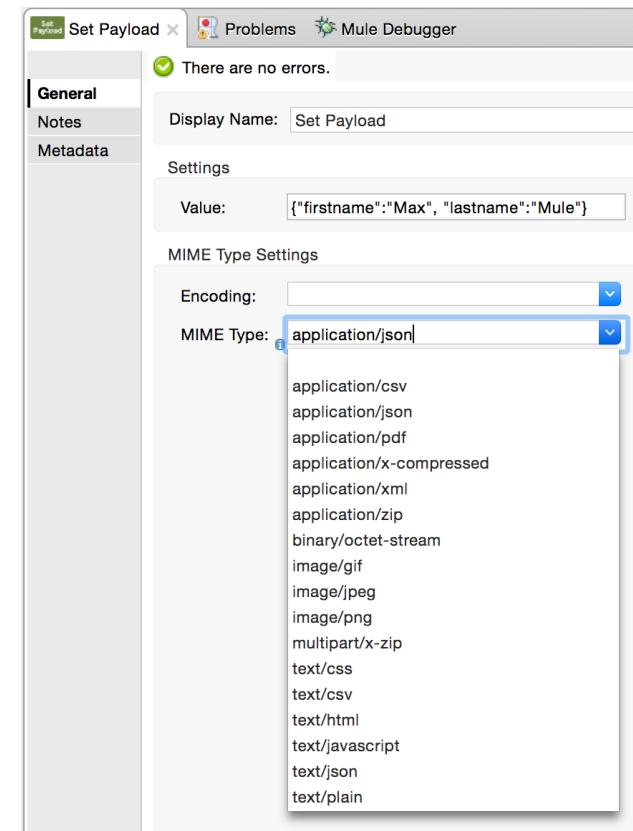
 The 'Output' section shows the resulting XML: 

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

 At the bottom, there are tabs for 'Context' (set to 'payload'), 'Payload' (set to 'payload'), 'Preview', and 'Structure'.

# 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

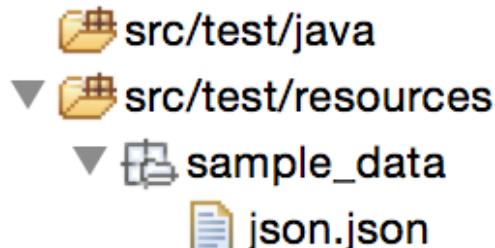


# Where is the code?

- By default, it is placed inline

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

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



# 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 flight search interface with fields for destination (SFO - San Francisco) and airline (All Airlines), and a **Find Flights** button.
- getFlightsFlow**: A Mule flow diagram with three components: **HTTP**, **Transform Message**, and **Logger**. The flow starts with an **HTTP** connector, followed by a **Transform Message** component, and ends with a **Logger** component.
- Transform Message**: A detailed view of the **Transform Message** component. It shows the **Input** JSON: `{"destination": "SFO", "airline": "united"}`. The **Output** section shows the DataWeave script:

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

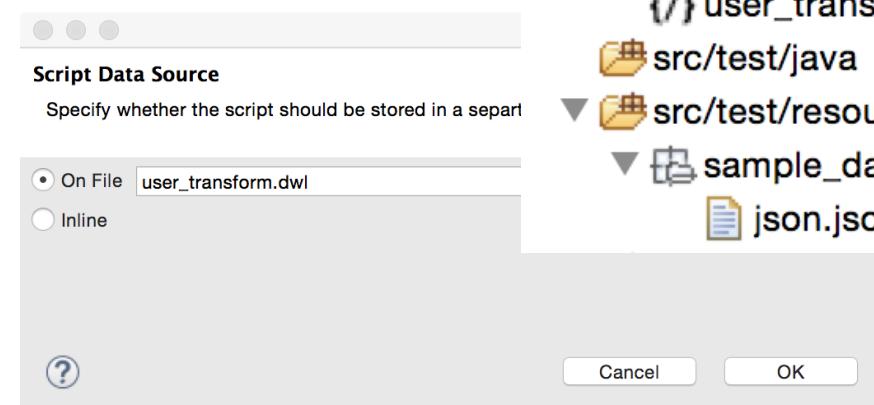
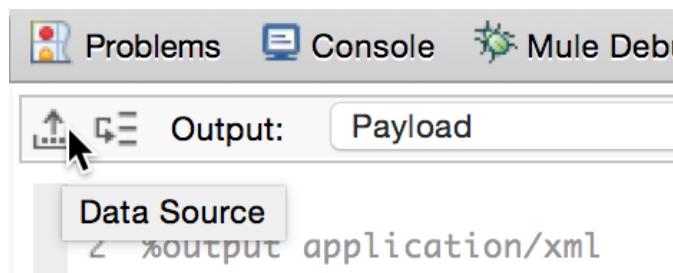
The **Output** table shows the transformed Java object:

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

At the bottom left, the number **31** is visible. On the right, the **MuleSoft** logo is present.

# 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" allowedMethods="GET,POST" />
    <set-payload value="{"firstname";"Max";, "lastname":"Smith"}" type="JSON" />
    <dw:transform-message doc:name="Transform Message">
        <dw:input-payload doc:sample="json.json"/>
        <dw:set-payload resource="classpath:user_transform.dwl"/>
    </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

The screenshot shows the 'Transform Message' editor in Mule Studio. The 'Input' pane contains the JSON payload: `{"firstname": "Max", "lastname": "Mule"}`. The 'Output' pane shows the MEL expression: `%output application/xml`, followed by a blank line, then `{`, then  `fname: payload.firstname,`, then  `lName: payload.lastname`, and finally `}`. A red error message at the bottom states: `Can not update preview. Validate your mappings.`. The 'Preview' tab is selected.

- This will work

The screenshot shows the 'Transform Message' editor in Mule Studio. The 'Input' pane contains the same JSON payload: `{"firstname": "Max", "lastname": "Mule"}`. The 'Output' pane shows the MEL expression: `<%dw 1.0`, then `%output application/xml`, then `---`, then `{`, then  `user: {`, then  `fname: payload.firstname,`, then  `lName: payload.lastname`, and finally `}`. The 'Output' pane displays the resulting XML: `<?xml version='1.0' encoding='UTF-8'?>`, then `<user>`, then `<fname>Max</fname>`, then `<lName>Mule</lName>`, and finally `</user>`. The 'Preview' tab is selected.

# Specifying attributes for XML output



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

The screenshot shows the Mule Studio interface with the 'Transform Message' tab selected. The 'Input' pane contains a JSON payload: `{"firstname": "Max", "lastname": "Mule"}`. The 'Output' pane is set to 'Payload' and displays the resulting XML code:

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

The transformation logic is as follows:

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

The 'Preview' and 'Structure' tabs are also visible at the bottom of the editor.

# 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
<pre>&lt;user firstname="Max"&gt;   &lt;lastname&gt;Mule&lt;/lastname&gt; &lt;/user&gt;</pre>	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

# Referencing message variables

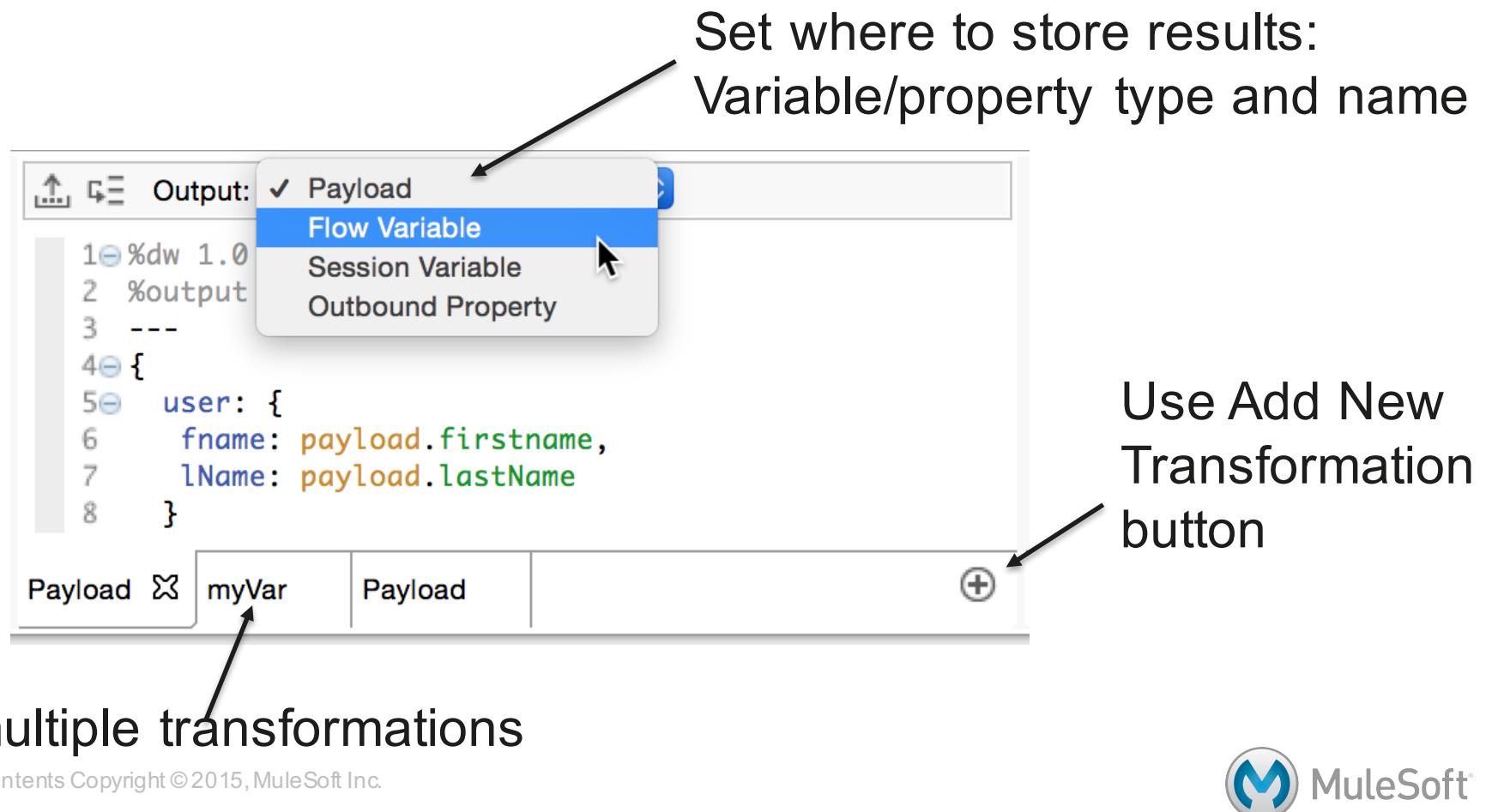


- So far, we referenced payload
- You can also reference
  - flowVars
  - inboundProperties
  - outboundProperties
- This is not MEL!
  - Do not preface these values by “message.” or use #[]

```
%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 2<sup>nd</sup> transformation to store destination in a flow variable
- Create a 3<sup>rd</sup> transformation to output data as JSON
- Create a 4<sup>th</sup> transformation to output data as XML

The screenshot shows the 'Transform Message' editor in Mule Studio. The interface includes tabs for 'Input', 'Output', and 'Output' (selected), and buttons for 'Context', 'Payload', 'destination', 'json', 'xml', 'Preview', and 'Structure'.

**Input:**

```
{"destination": "SFO", "airline": "united"}
```

**Output:** Flow Variable

**MVEL Script (Output):**

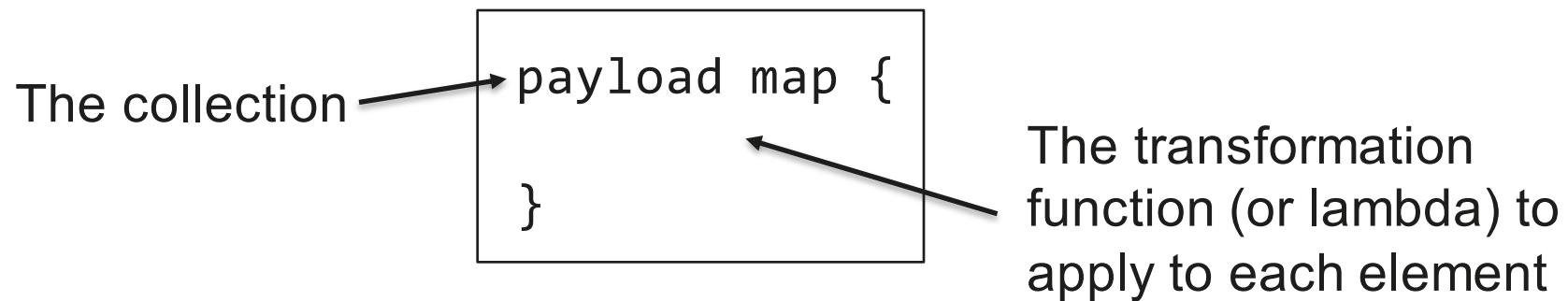
```
1 %dw 1.0
2 %output application/xml
3 ---
4 data: {
5   hub: "MUA",
6   flight @(airline: payload.airline): {
7     code: payload.destination
8   }
9 }
```

**Output:**

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

# 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>

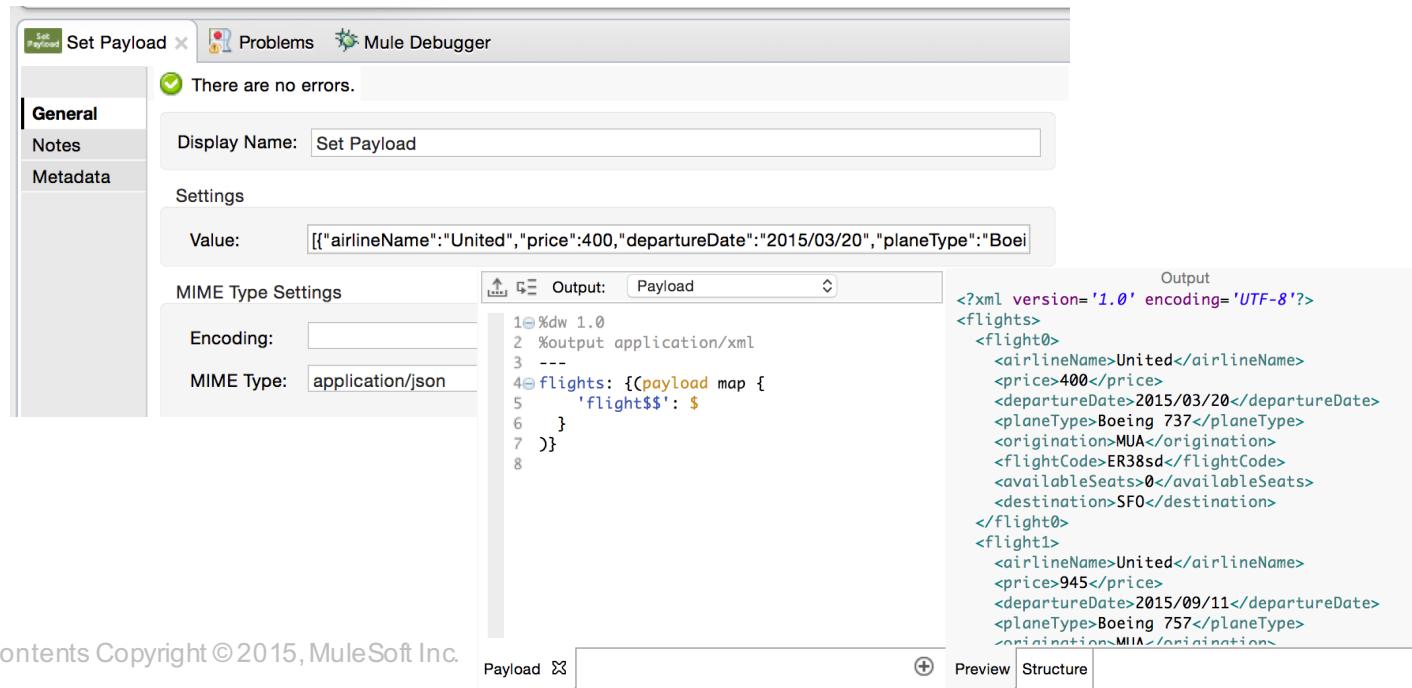
# Writing expressions for XML input

- Use \* to reference repeated elements

Input	Transform	JSON Output
<users> <user firstname="Max"> <lastname>Mule</lastname> </user> <user firstname="Molly"> <lastname>Mule</lastname> </user> </users>	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*



# 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](mailto: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

- upper
- lower
- camelize
- capitalize
- dasherize
- underscore
- singularize
- pluralize

Input	Transform	Output
	%dw 1.0 %output application/xml ---	
{"name": "max_mule"}	n: upper (dasherize payload.name)	<n>MAX-MULE</n>

# Formatting operators



Input	Transform	Output
	%dw 1.0 %output application/xml ---	
{ "name": "max_mule" }	n: upper payload.name	<n>MAX_MULE</n>
	n: lower payload.name	<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
```



- +
- -
- \*
- /
  
- 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

# Conditional logic operators



- **default**
- **when**
- **unless**
- **otherwise**
- In expressions, you can use
  - `==`
  - `!=`
  - `~==` (equal regardless of type)

Input	Transform	Output
	<pre>%dw 1.0 %output application/xml ---</pre>	
<pre>{"firstname":"Max", "lastname":"Mule"}</pre>	<pre>n: payload.nickname <b>default</b> payload.firstname</pre>	<pre>&lt;n&gt;Max&lt;/n&gt;</pre>
<pre>{"firstname":"Max", "lastname":"Mule", "nickname":""}</pre>	<pre>n: payload.nickname <b>when</b> payload.nickname != "" <b>otherwise</b> payload.firstname</pre>	<pre>&lt;n&gt;Max&lt;/n&gt;</pre>

# 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 <b>default</b> payload.firstname	<n>Max</n>
{"firstname":"Max", "lastname":"Mule", "nickname":""}	n: payload.nickname <b>when</b> payload.nickname != "" <b>otherwise</b> payload.firstname	<n>Max</n>
	n: payload.firstname <b>unless</b> payload.nickname != "" <b>otherwise</b> payload.nickname	<n>Max</n>
	n: payload.firstname unless payload.nickname != "" <b>or</b> payload.firstname != "" otherwise payload.nickname	<n></n>
	n: payload.lastname unless payload.nickname != "" <b>and</b> payload.firstname != "" otherwise payload.nickname	<n>Mule</n>

# Additional operators



- concat
- 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 active. The 'Input' pane displays an XML message structure. The 'Payload' pane contains DataWeave code. The 'Output' pane shows the resulting Java object structure.

**Input:**

```
<?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>Boing 737</
planeType><price>400.0</price></
return><return
airlineName="Delta"><code>A1B2C4</
code><departureDate>2015/10/21</
departureDate><destination>LAX</
destination>
```

**Payload:**

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

**Output:**

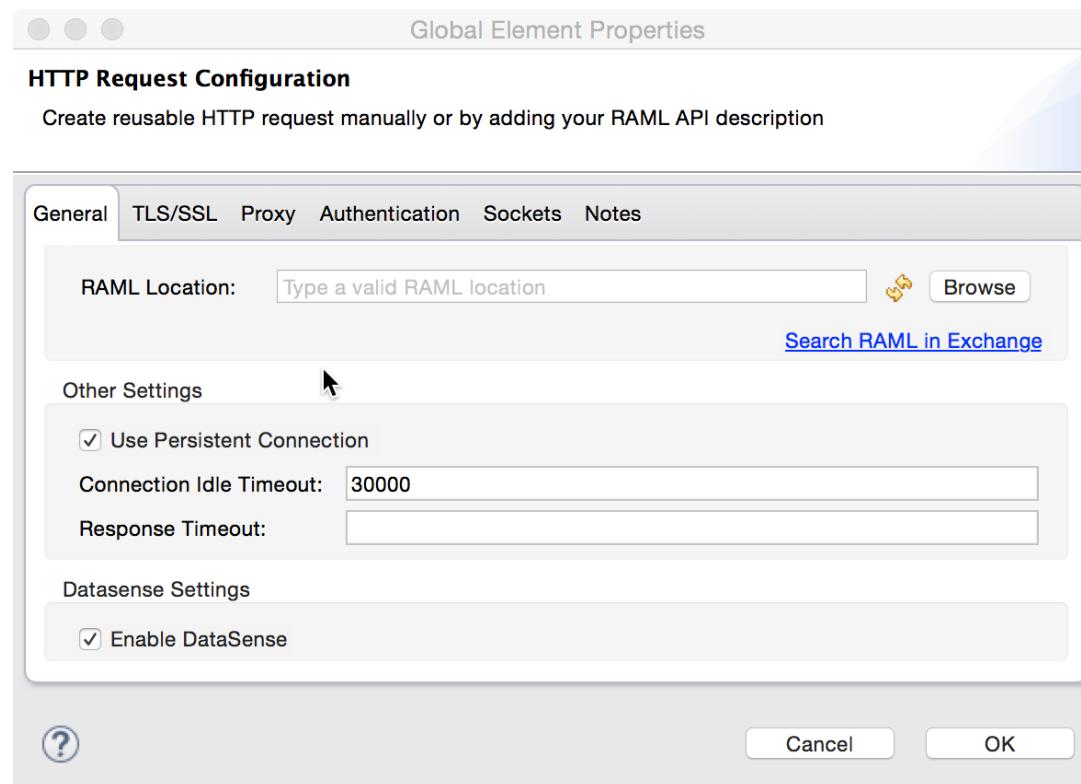
Name	Value
root : ArrayList	
[0] : Flight	
airlineName : String	
availableSeats : Integer	10
departureDate : String	Oct 21, 2015
destination : String	LAX
flightCode : String	
origination : String	
planeType : String	BOEING 737
price : Double	199.99
[1] : Flight	
airlineName : String	
availableSeats :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

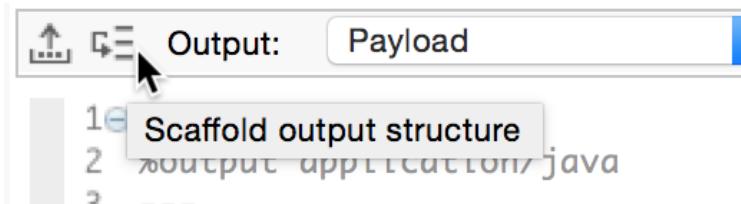


# Transformations and metadata

- 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

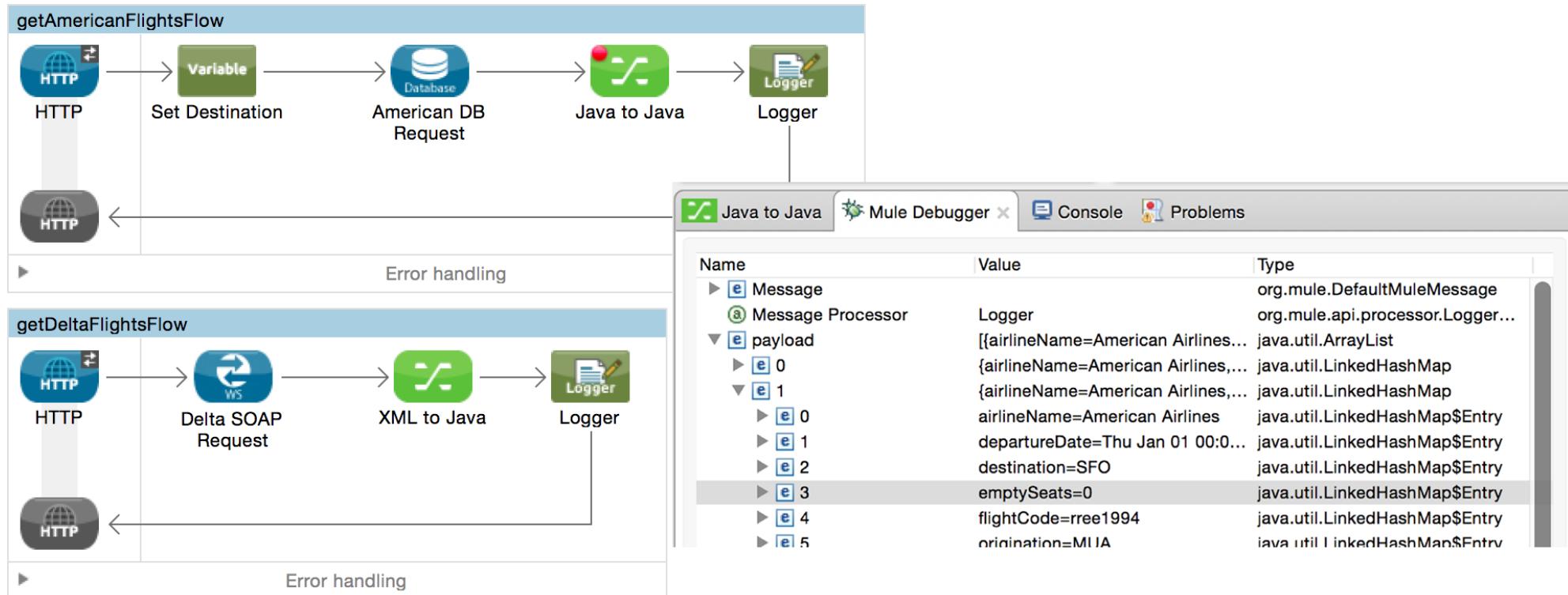


- To recreate scaffolding from metadata, click Scaffold output structure button
  - Deletes existing DataWeave code and re-scaffolds output

# 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

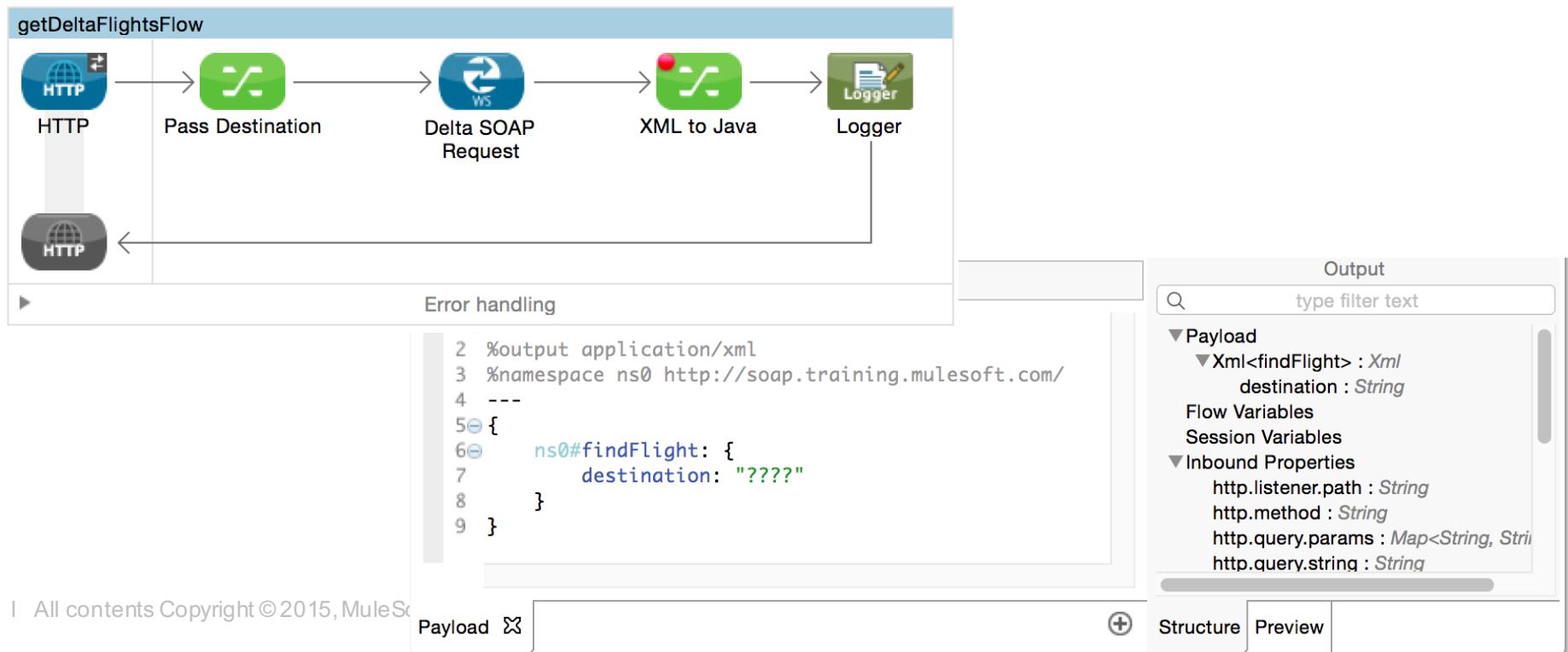


# Using DataWeave to pass arguments to SOAP web services

# Using DataWeave to specify SOAP web service input arguments



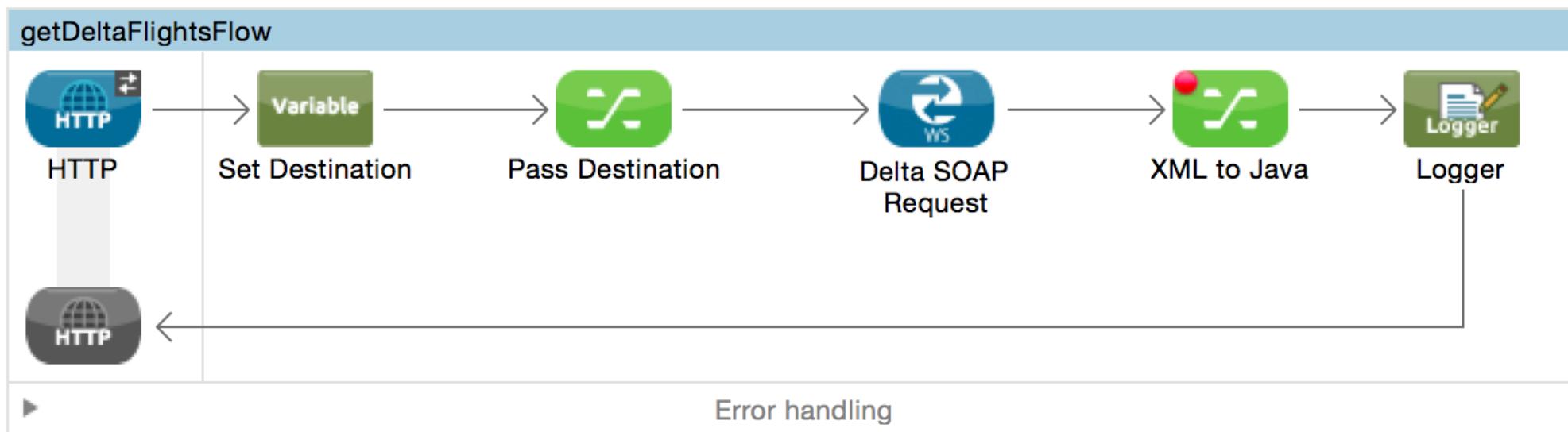
- When you drag-and-drop a DataWeave message transformer before a Web Service Consumer that has input arguments, a scaffold for populating the arguments will be created automatically



# 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

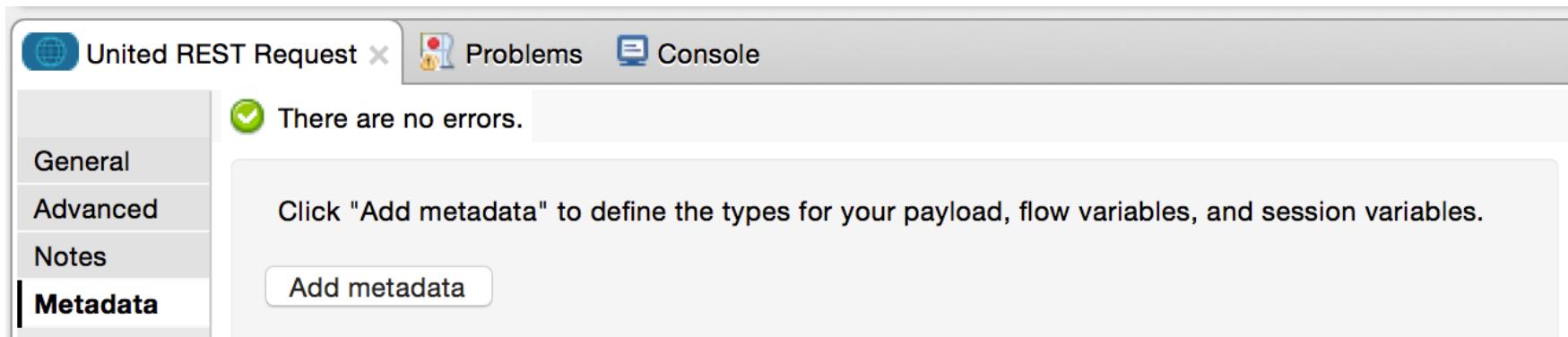


# Using DataWeave to transform data structures that need custom metadata



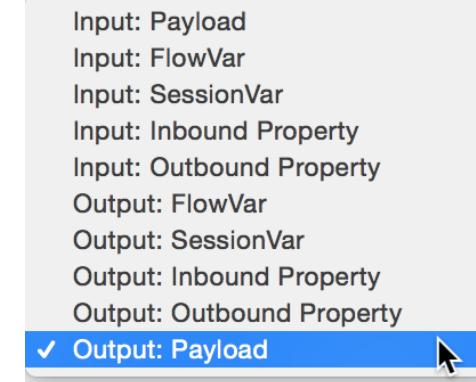
# A more metadata aware Anypoint Studio and Mule runtime

- 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

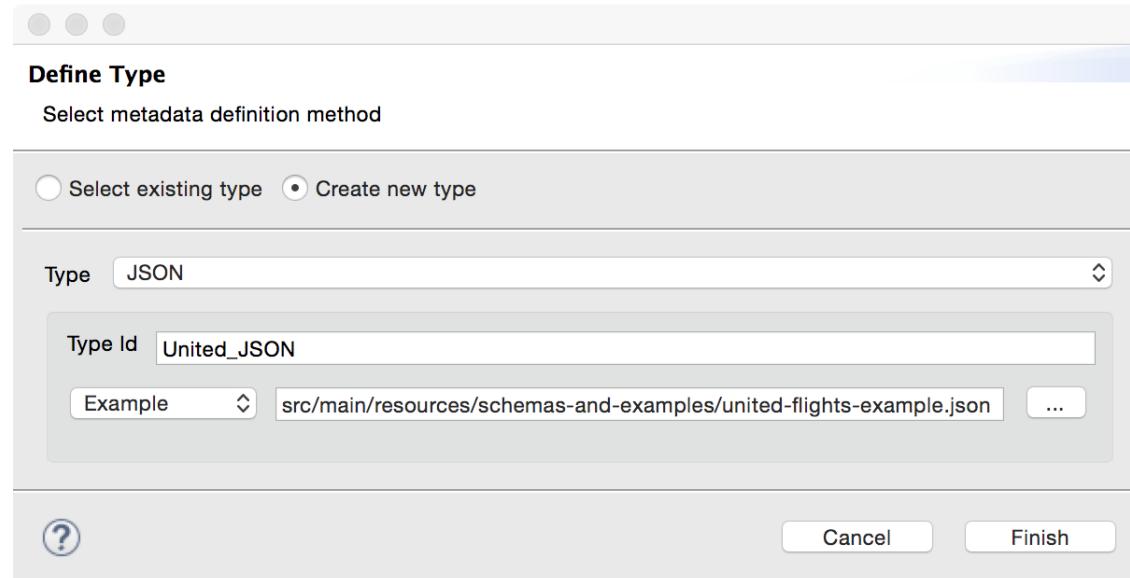


# Adding metadata

- Specify what the metadata is for

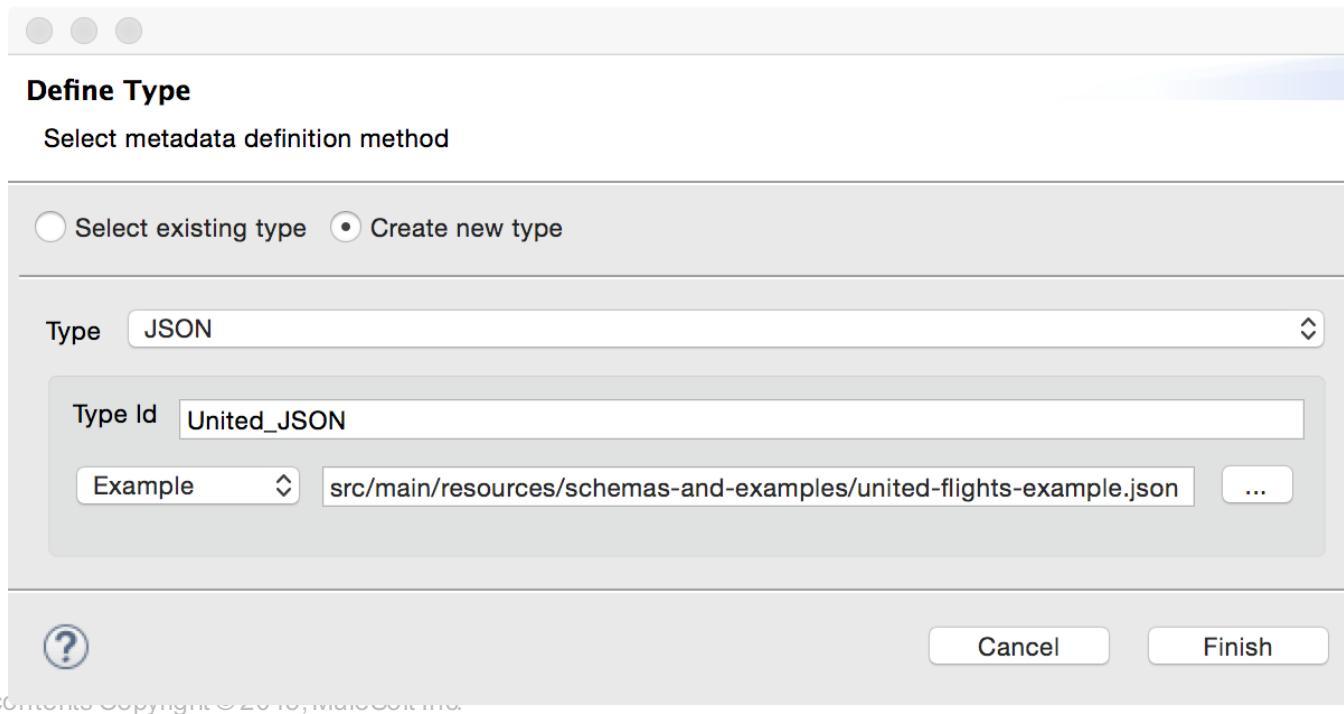


- Specify the data type: JSON, XML, CSV, JAVA



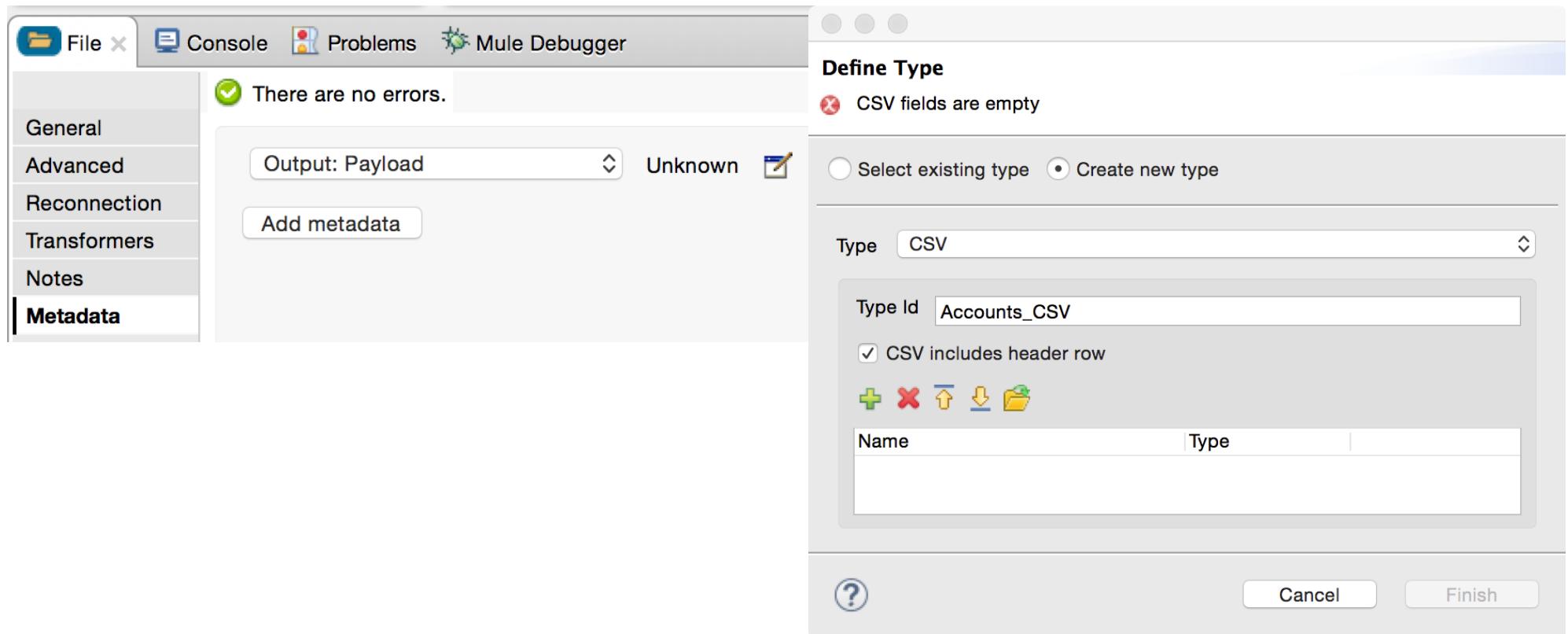
# Adding metadata

- For XML and JSON
  - Provide schema or example file
- For CSV
  - Provide example file or add fields manually



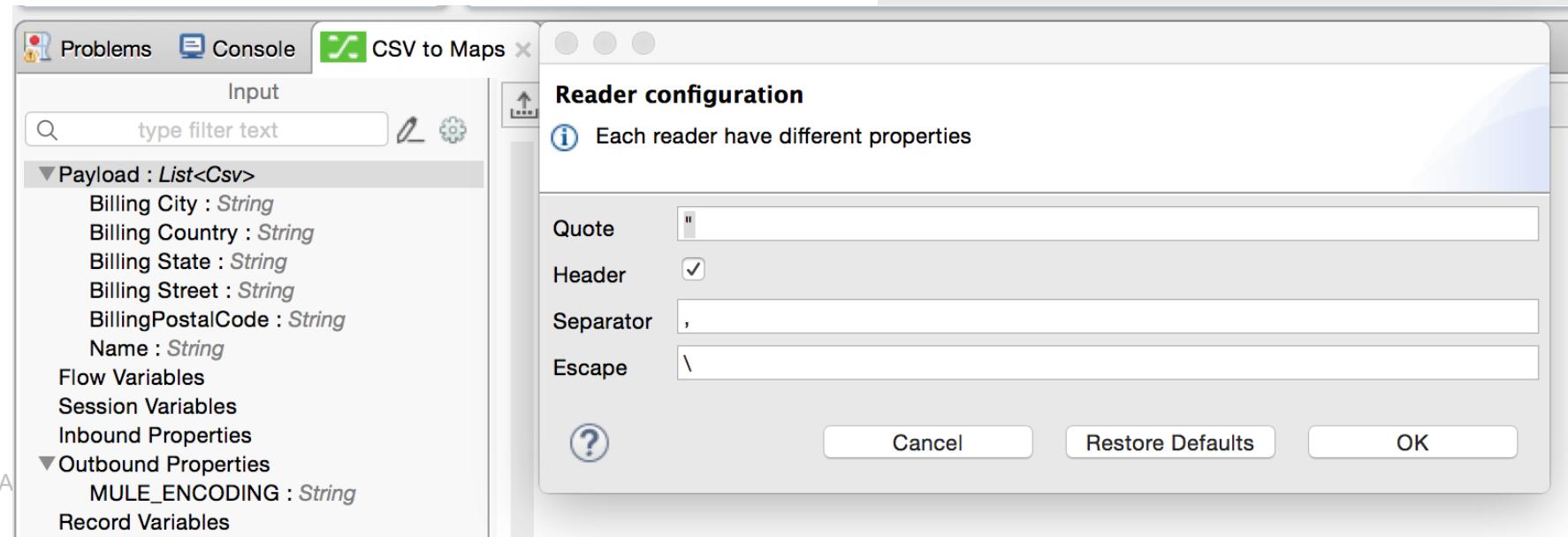
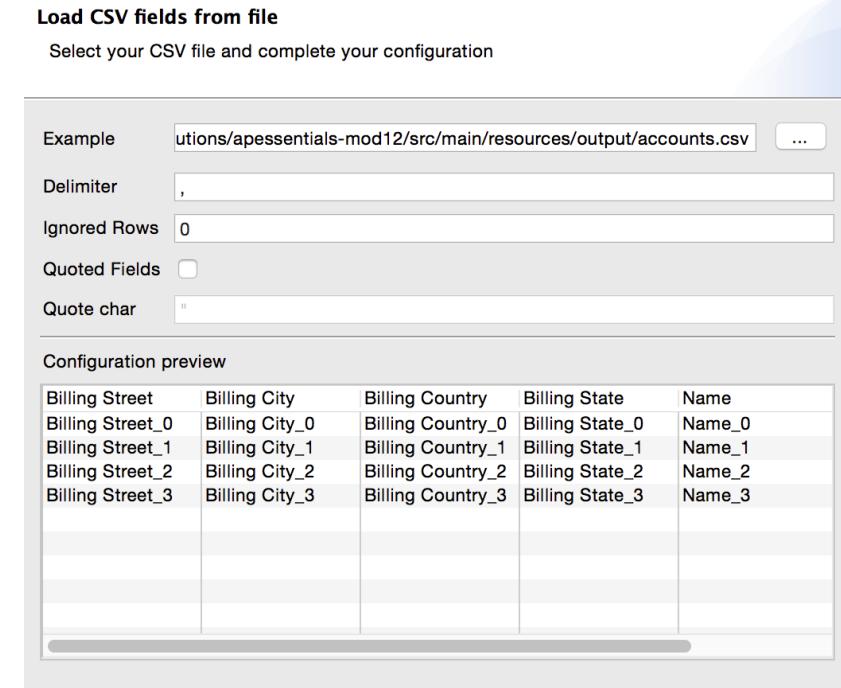
# Add metadata to the File endpoint

- Add fields manually or load fields from an example CSV file



# Specify delimiter and other file properties

- In the metadata if using an example CSV file
- Or using the Reader Configuration button in the Input section of the Properties View



# Use DataWeave with the endpoint that has metadata

The screenshot shows the Mule Studio interface with two main windows open:

- Define Type Dialog:** A modal window titled "Define Type" with the sub-instruction "Select metadata definition method". It contains two radio buttons: "Select existing type" (unchecked) and "Create new type" (checked). Below this, a dropdown menu shows "Type CSV". The "Type Id" field is set to "Accounts\_CSVD". A checked checkbox "CSV includes header row" is present. Below these settings is a table with columns "Name" and "Type", listing fields: Billing Street (String), Billing City (String), Billing Country (String), Billing State (String), Name (String), and BillingPostalCode (String).
- CSV to Maps Component Editor:** This is the main workspace. At the top, tabs include "CSV to Maps", "Mule Debugger", and "api APIkit Consoles (apessentials)". The "Output" tab is selected, showing "Payload" as the output type. The "Input" tab shows a "Payload : List<Csv>" structure with fields: Billing City, Billing Country, Billing State, Billing Street, BillingPostalCode, and Name. The "Payload" tab displays the following DataWeave code:

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

The "Output" tab shows the resulting structure:

Name	Value
root	: ArrayList
[0]	: LinkedHashMap
Billing Street	: String ?????
Billing City	: String ?????
Billing Country	: String ?????
Billing State	: String ?????
Name	: String ?????
BillingPostalCode	: ?????

# 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

# 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

United REST Request X Problems Console Mule Debugger

General Advanced Notes Metadata

Output: Payload User Defined/United\_JSON

Add metadata

getUnitedFlightsFlow

```
graph LR; A[HTTP] --> B[Variable]; B --> C[HTTP]; C --> D[JSON to Java]; D --> E[Logger]; E --> F[HTTP]
```

Set Destination      United REST Request      JSON to Java

Error handling

Input Output

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*

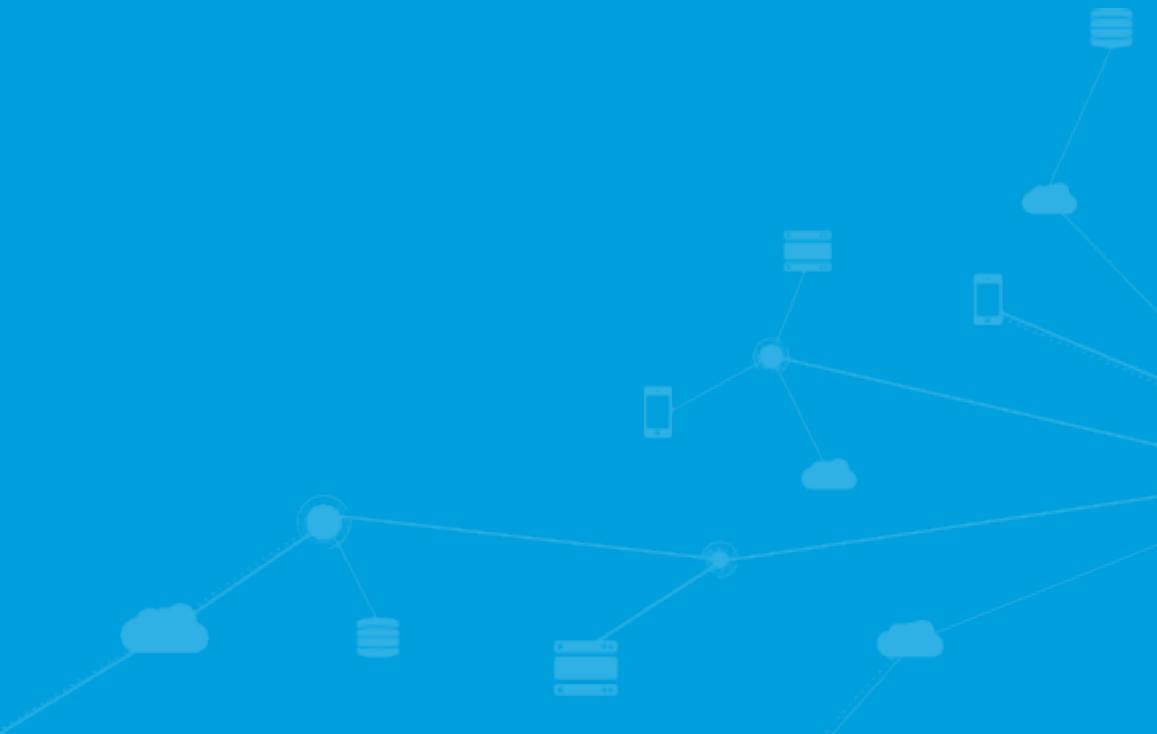
[Refresh Metadata](#)

# 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