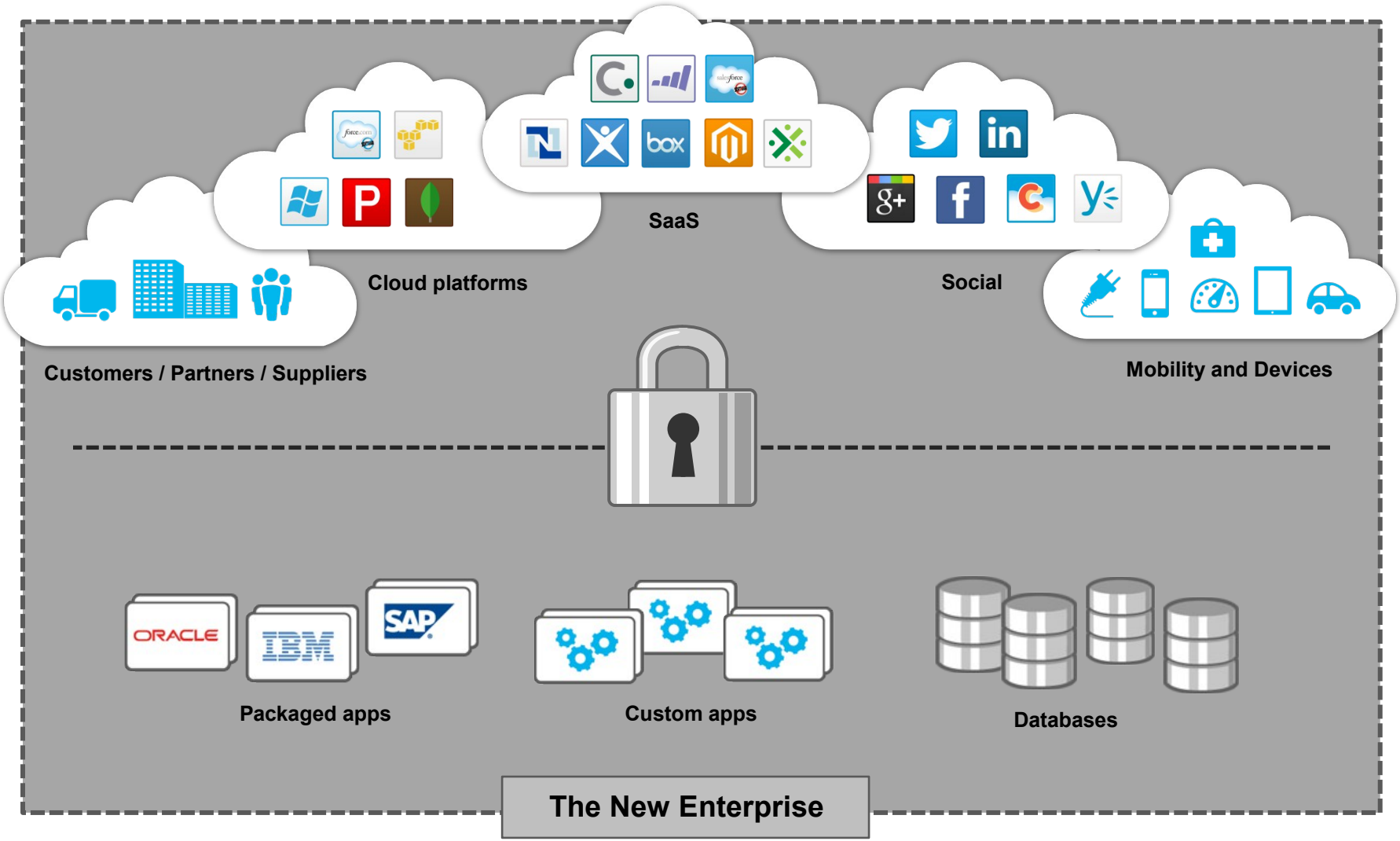
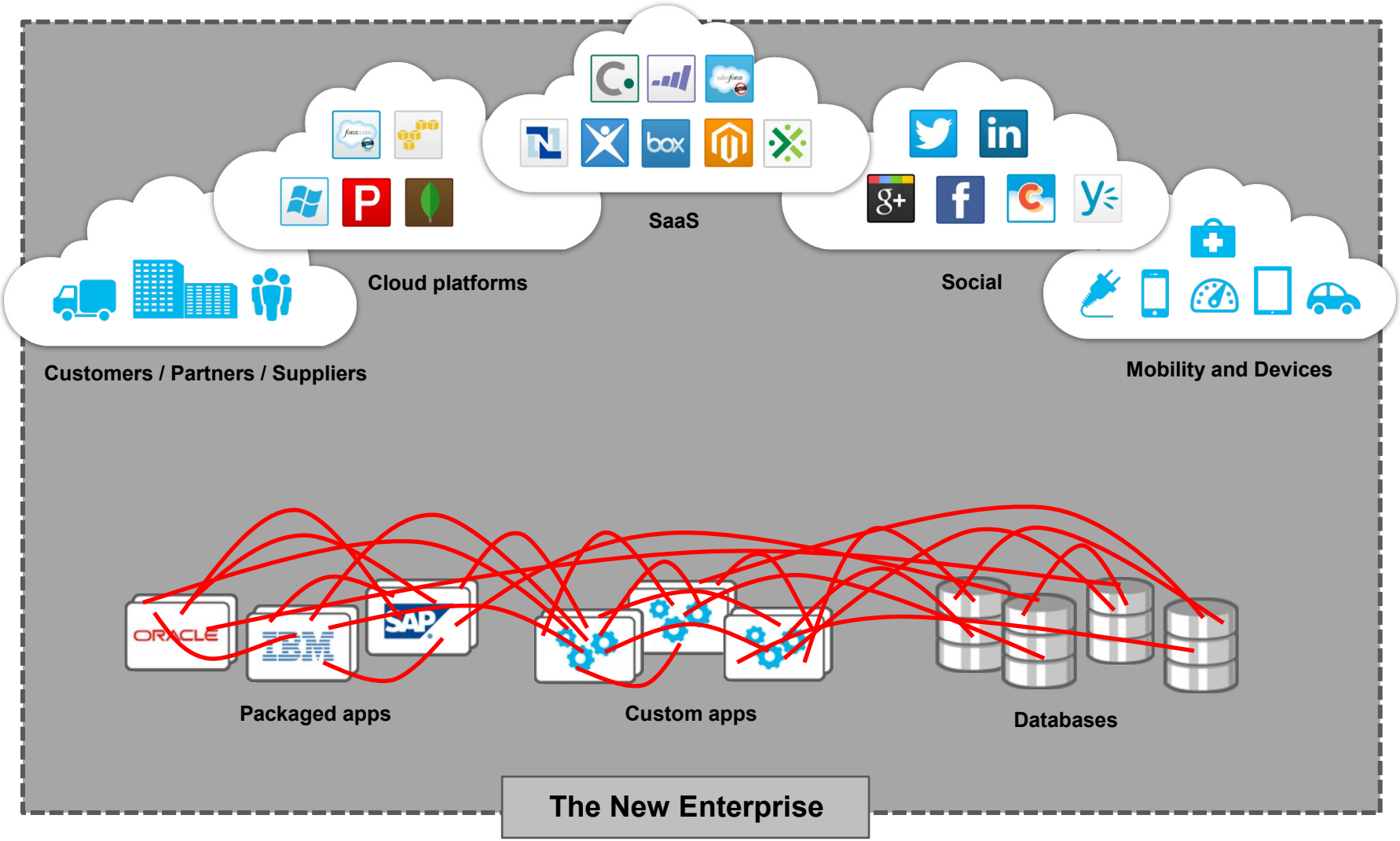
Anypoint Platform Essentials

What is Mule?

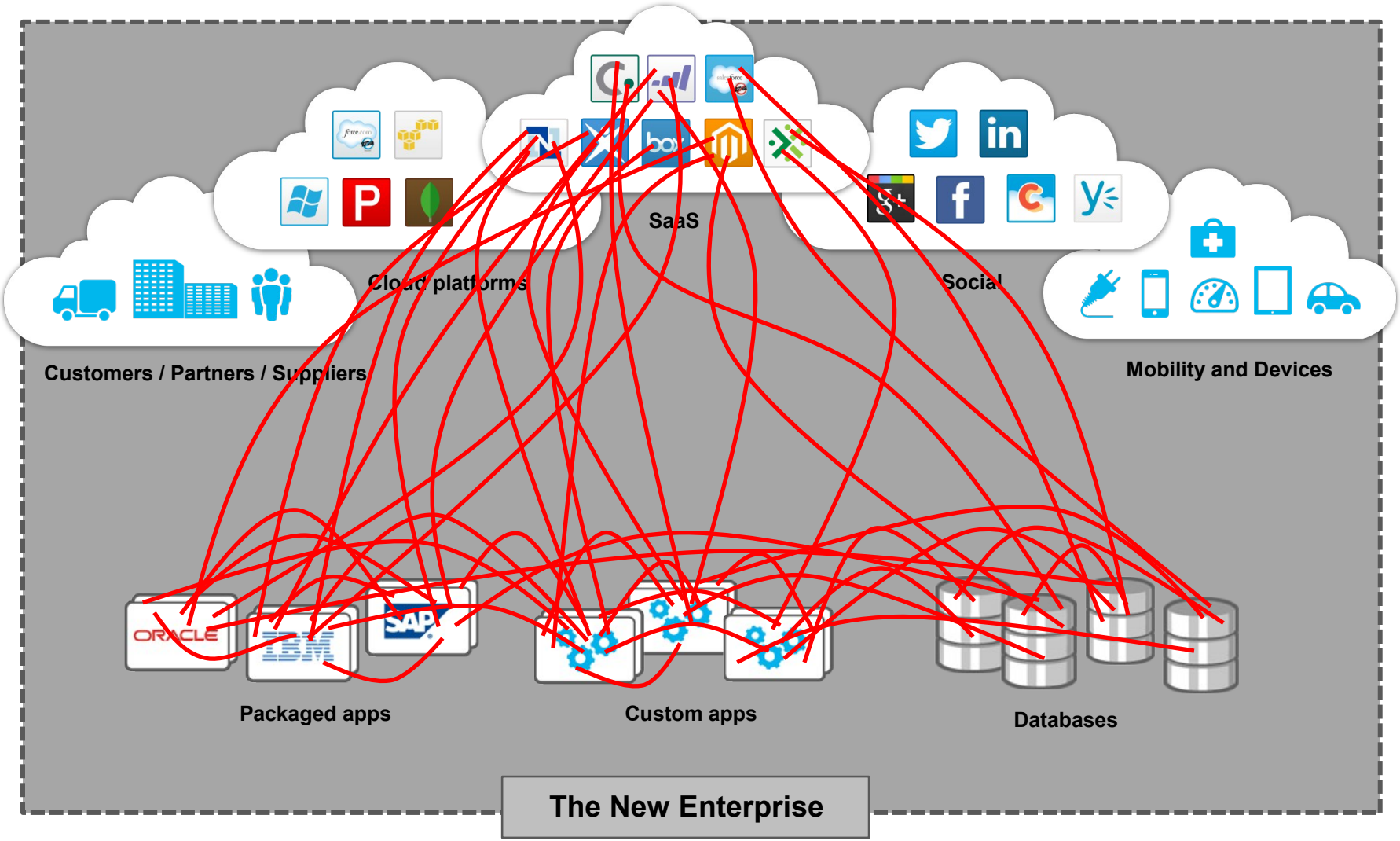
Welcome to the New Enterprise



Connect backend



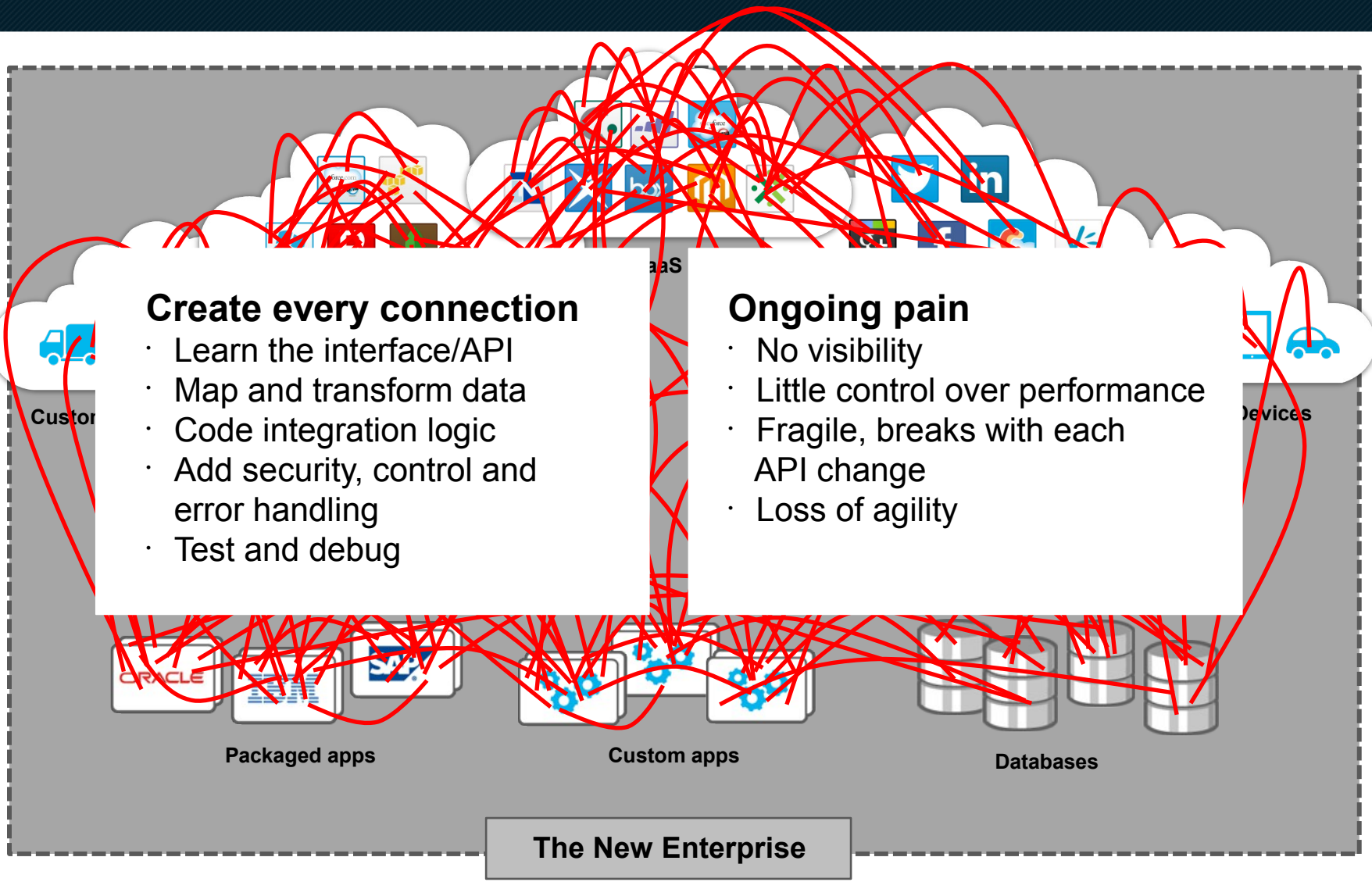
Backend to SaaS



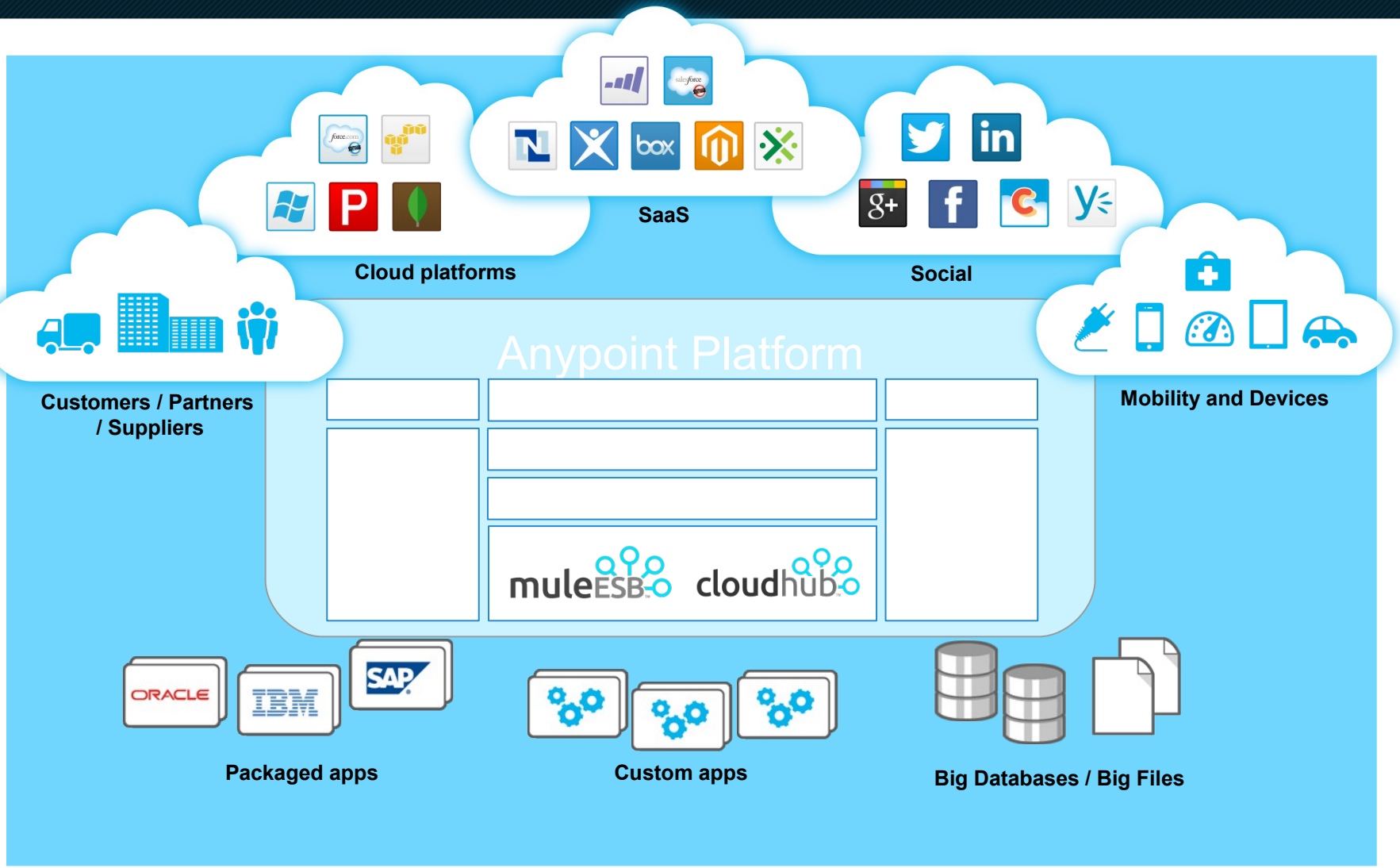
Integration = pain

Integration = pain

Integration = pain



MuleSoft connects the New Enterprise



Leading on-premise and in the cloud

MuleSoft

3,500+ on-premise enterprise deployments

25,000+ cloud deployments

35% of the Global 500

HQ in San Francisco with offices in New York, Atlanta, London, Sydney, Singapore, Hong Kong and Buenos Aires

Enabling many paths to The New Enterprise

MuleSoft

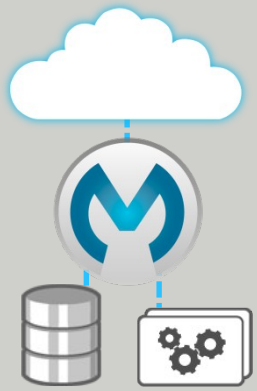
SOA

Application integration

Legacy modernization

SaaS Integration

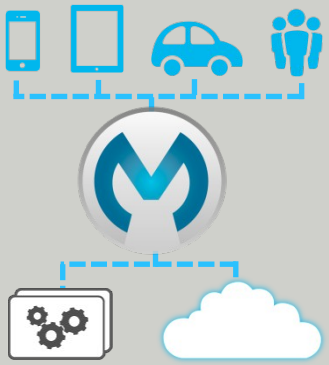
Cloud to Cloud



Cloud to ground

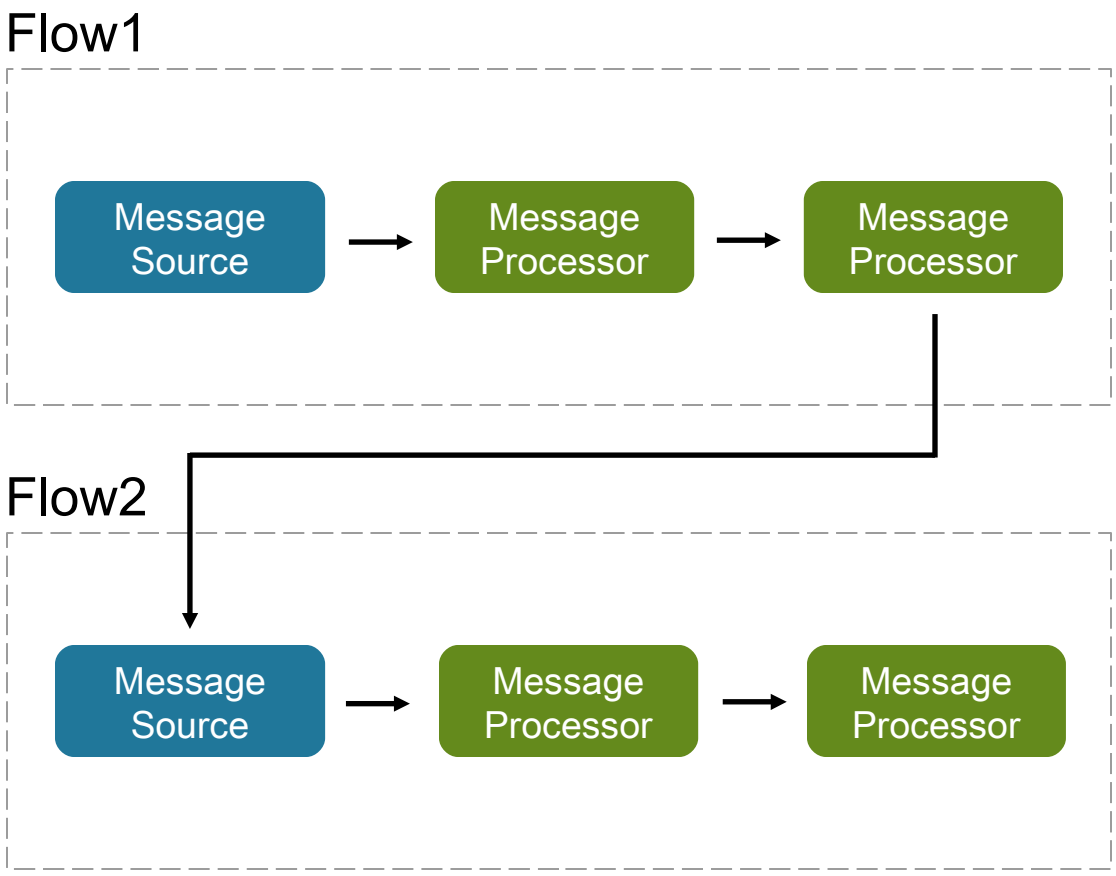
Enterprise APIs

SOA governance



API Enablement

What is a Flow ?

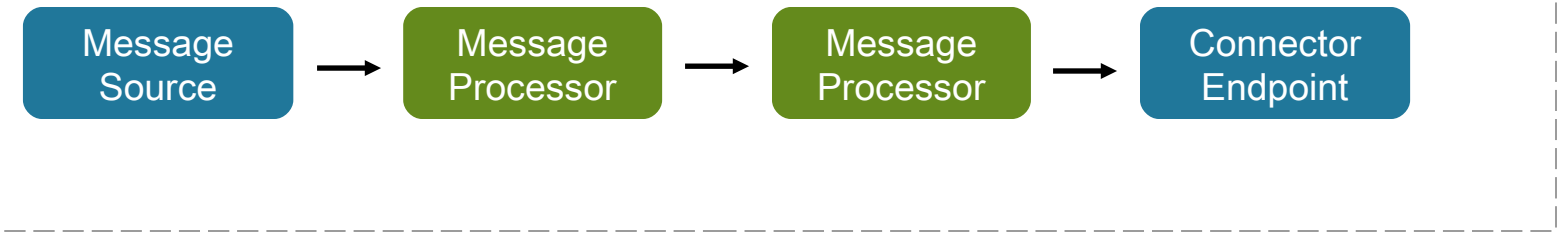


- Mule Applications are composed of flows
- Flows are initiated by a Message Source
- Flows contain message processors
- Flows can send messages off to other flows and eventually external resources

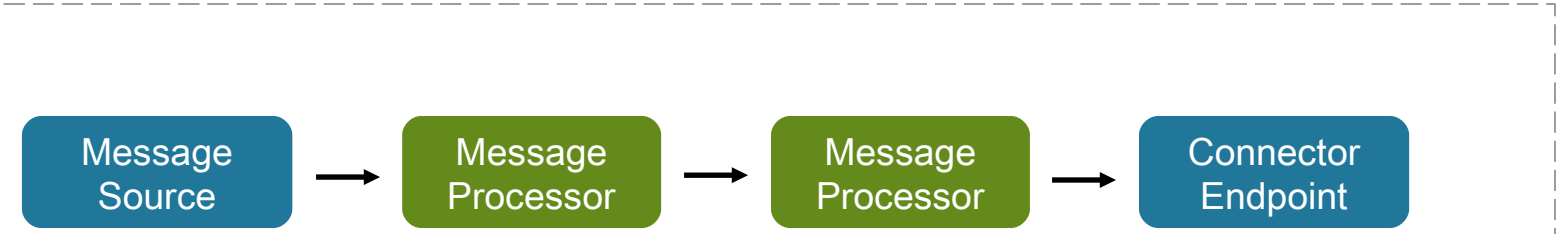
Mule Applications

MuleSoft

Flow1



Flow 2



What is a Message Processor ?

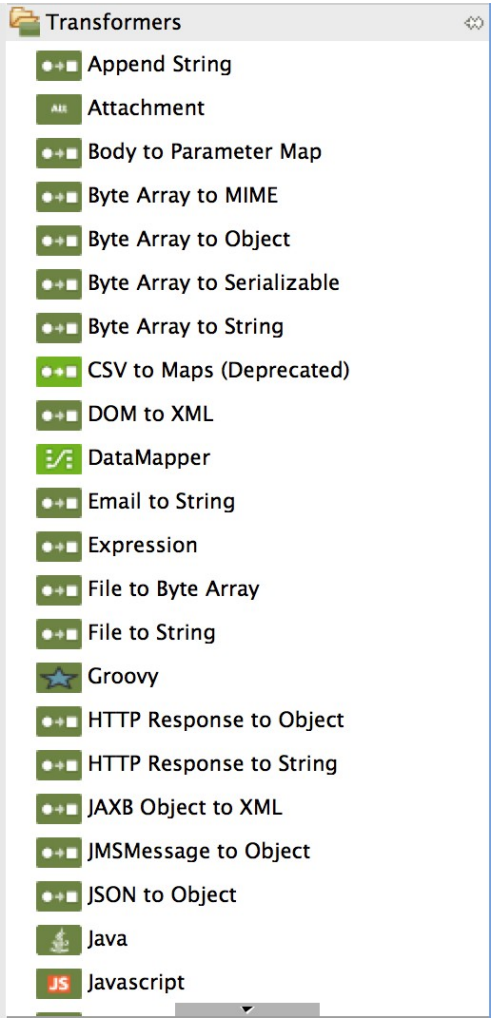
- Message processors are responsible for processing the received message.
- These message processors are categorized by function:
  - Components: perform business logic & are typically application specific
  - Transformers: transform the message
  - Filters: accept/reject messages
  - Routers: control the message flow
  - Endpoints: send/receive messages over a transport

Message Processors

MuleSoft

Transformers

- Transformers prepare the Mule Message for further processing
- Introduce new properties or variables to the message
- Move the payload between types (Class)
- Custom transformers can be configured in Java and JSR-223 Scripting Languages



Message Processors

MuleSoft

Components

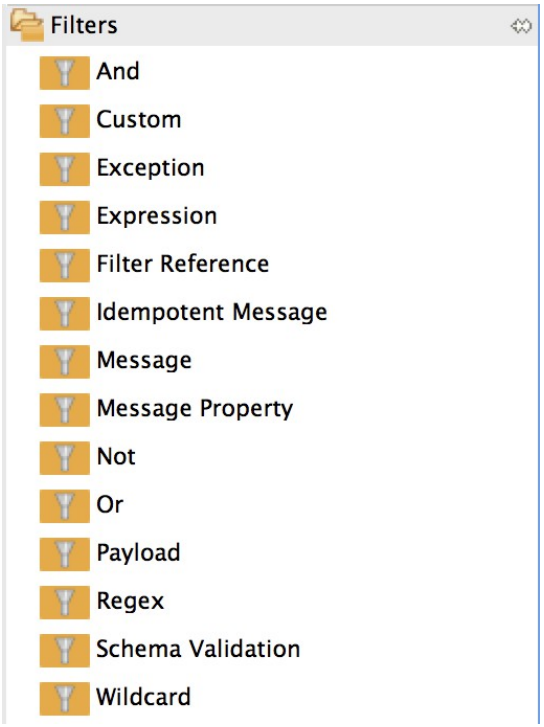
- Introduces business logic into our flows
- Support for JSR-223 scripting languages
- Web service oriented components include REST and CXF Soap

Message Processors

MuleSoft

Filters

- Ensures messages which do not meet a criteria persist within a Mule Flow
- Drops ‘filtered’ messages
- Filters can be nested with more advanced logic
- Can be configured to throw exceptions when filtering out messages

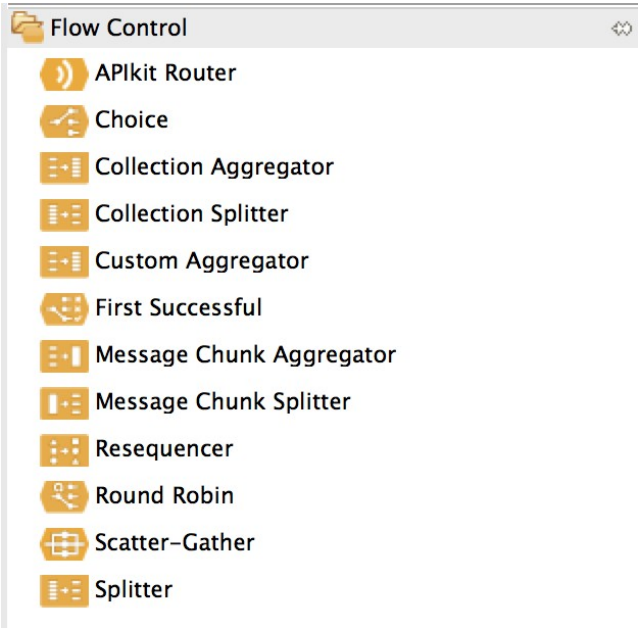


Message Processors

MuleSoft

Flow Control

- Performs multicasts of a Mule Message
- Splits payloads into multiple Mule Messages
- Aggregates Multiple Mule Messages into one
- Changes the processing path based on developer defined logic

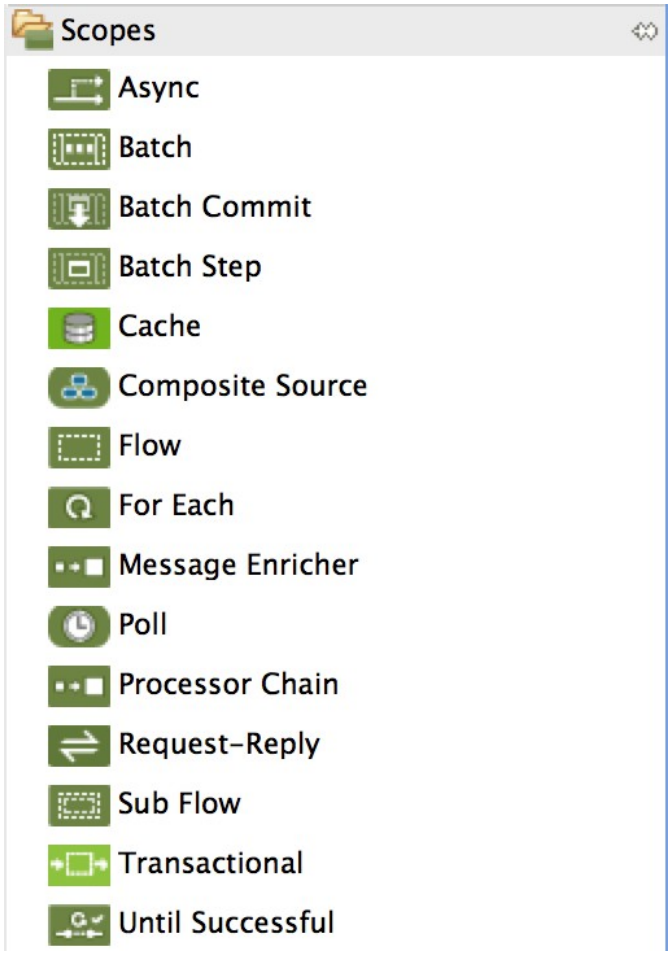


Message Processors

MuleSoft

Scopes

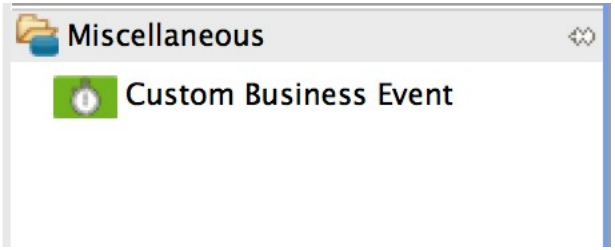
- Surrounds one or many message processors
- Changes the intrinsic behavior of the ‘wrapped’ processors



Message Processors

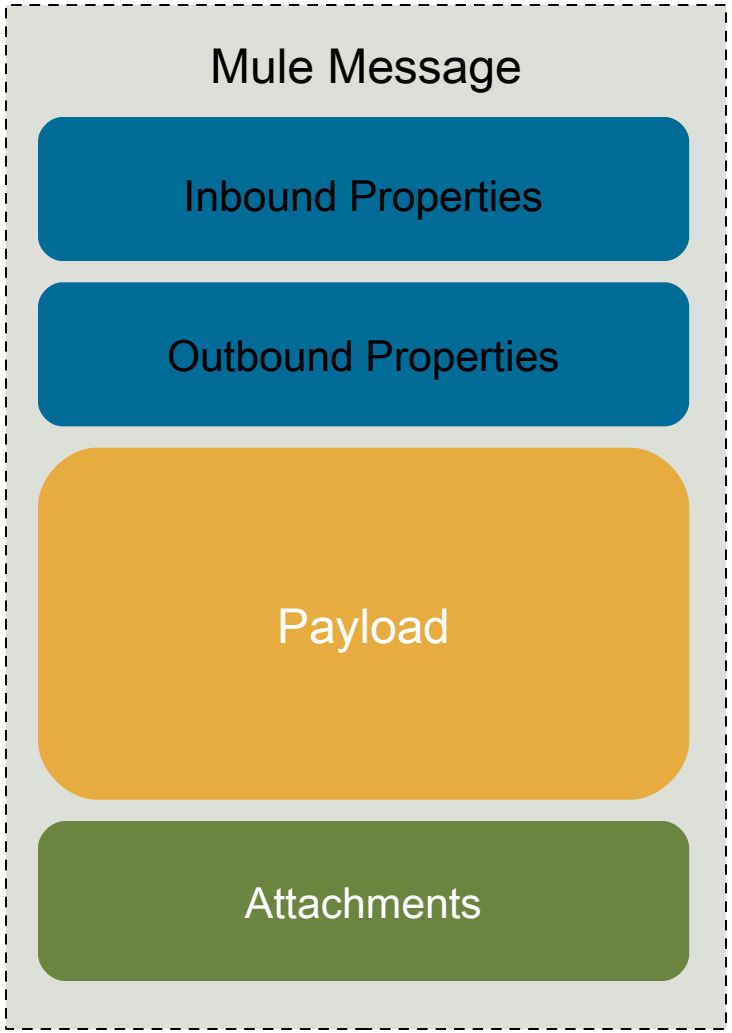
MuleSoft

Business Events



- Capture information about the current processing within your Mule flow
- Configured with KPI (Key Performance Indicators)
- Monitorable in both Mule Management Console (ESB) and CloudHub Insight

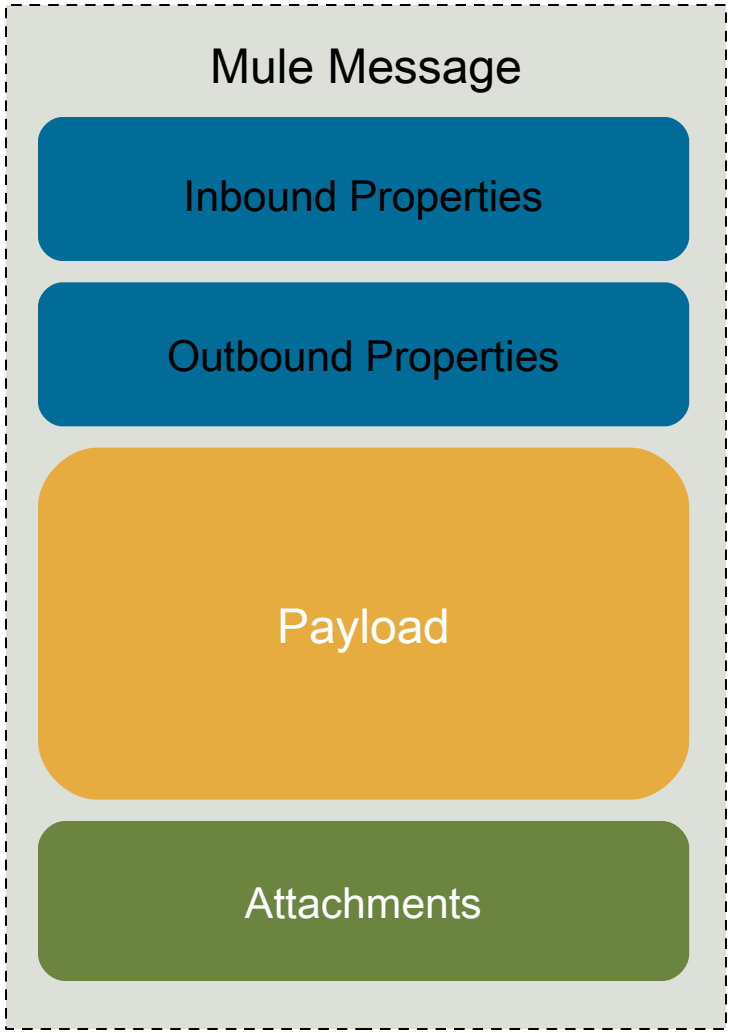
The Mule Message



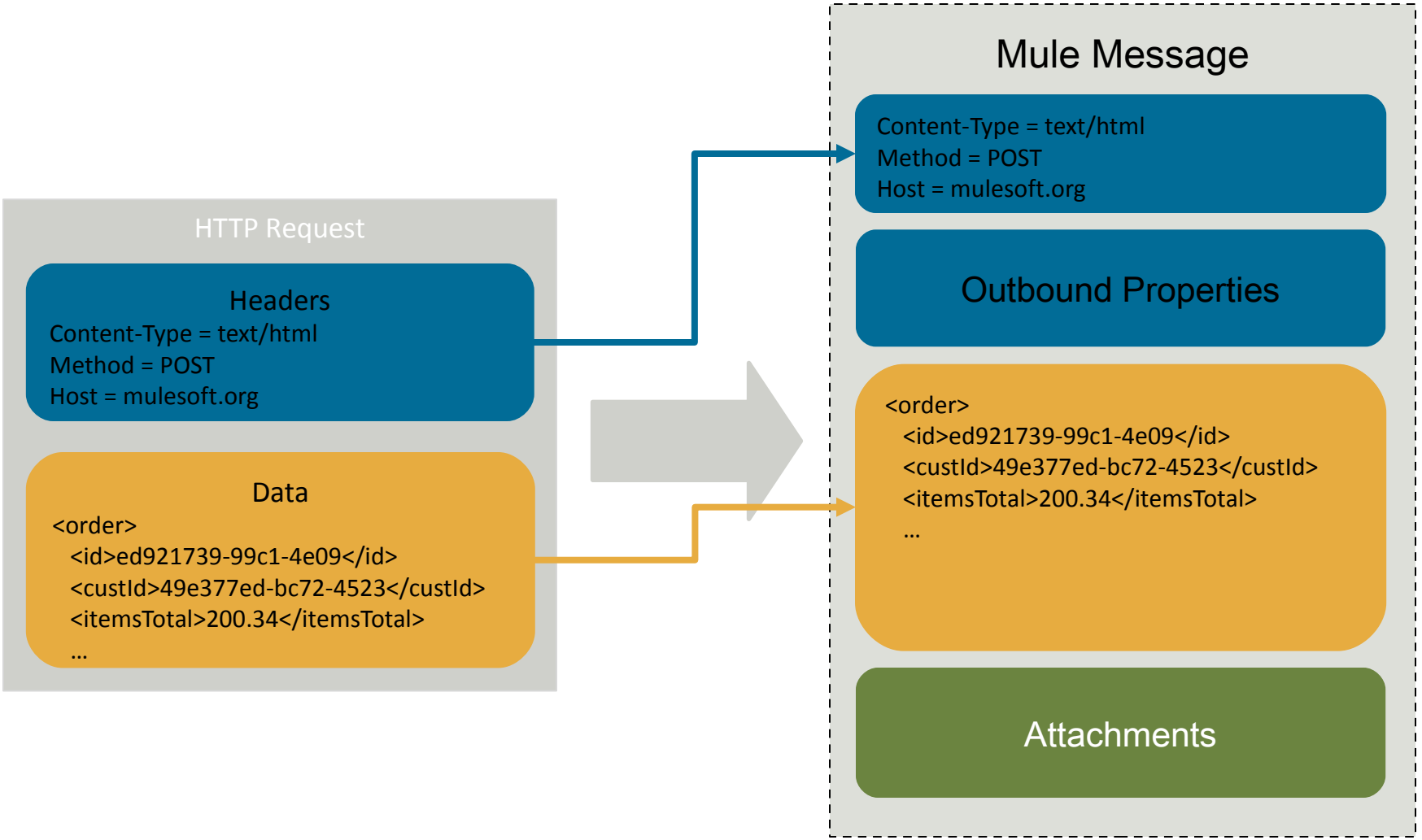
- Immutable data from the source of a message
- Mutable data which will be carried out of its current *flow*
- The core of the message
- Contains primary information to be processed by Mule
- Contains a Java Object
- Ancillary information to the message
- Similar to an email attachment

Message Properties

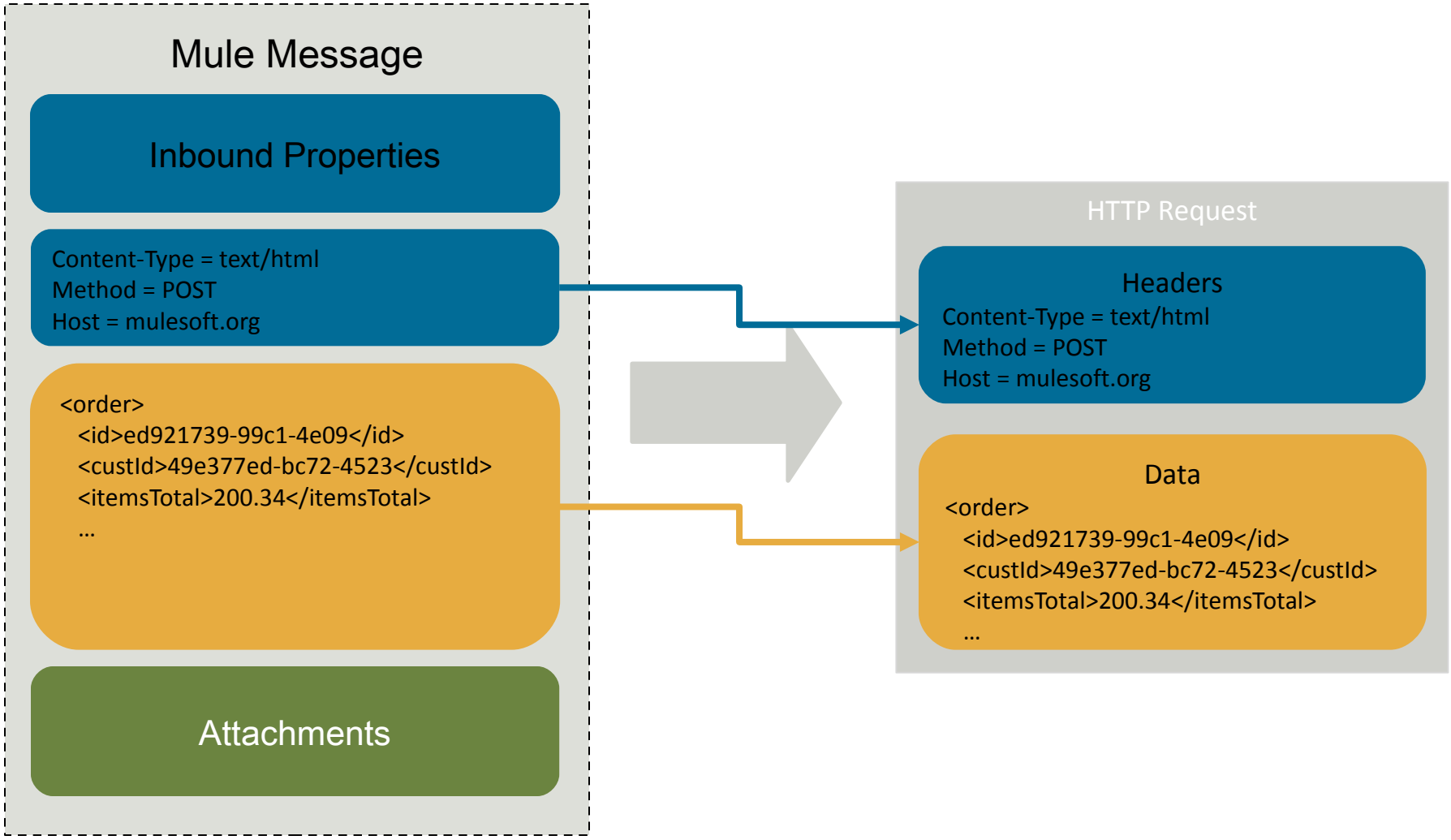
- Inbound Properties:
  - Read-only Access
  - Set from the Message Source
  - Persists throughout the flow
- Outbound Properties
  - Read/Write Access
  - Added by Property MP
  - Ability to:
    - Write
    - Delete
    - Copy



Message Properties – Inbound



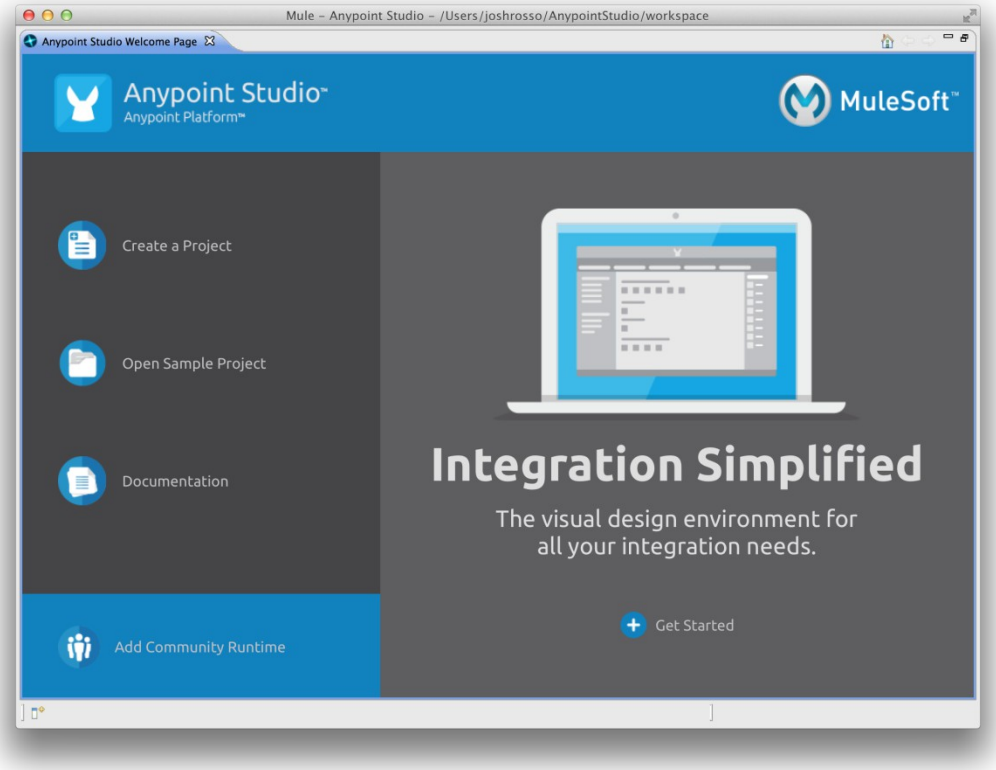
Message Properties – Outbound



Anypoint Studio

Anypoint Studio Tour

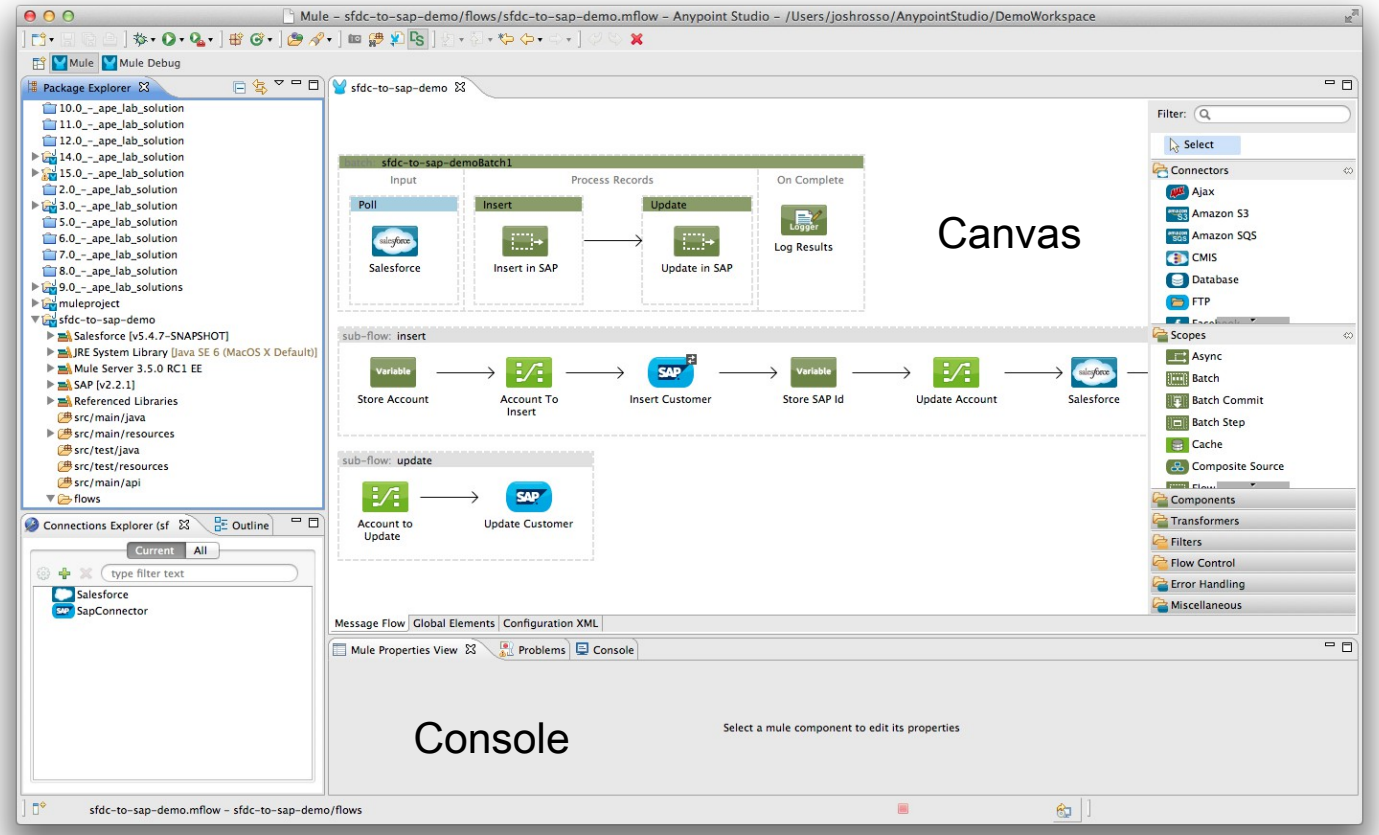
- Ensure Anypoint Studio is: *
  - Downloaded
  - Updated
  - Opened to the welcome screen



Anypoint Studio Tour

Package Explorer

Connection Explorer



Connection Explorer

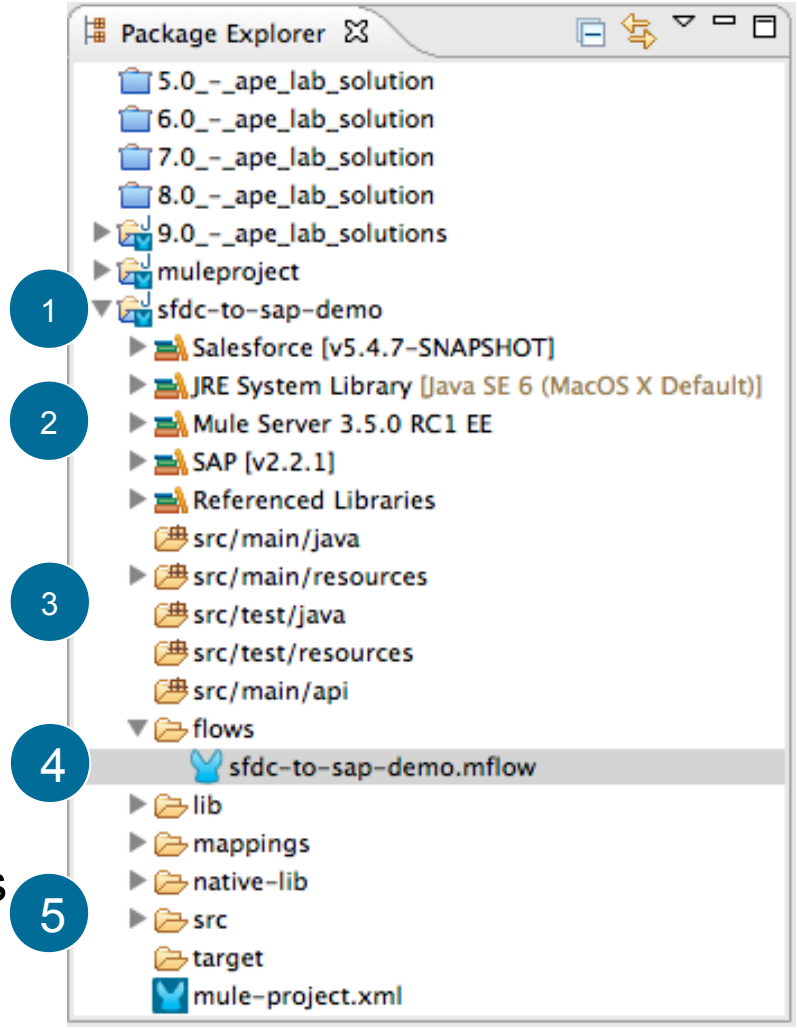
Palette

Canvas

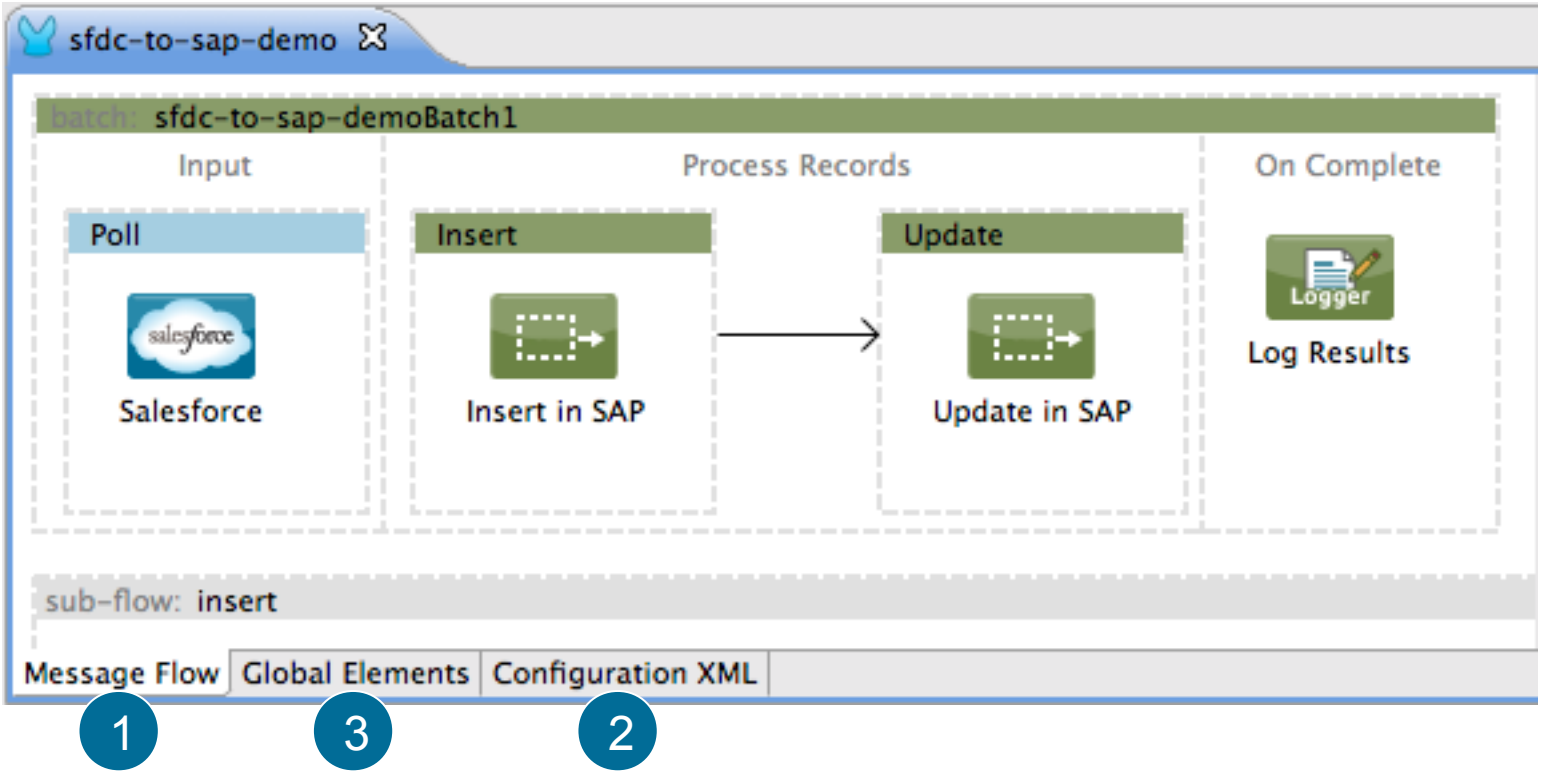
Console

Package Explorer

1. Access to all projects in our workspace
2. Libraries and runtimes used in project
3. Java files, test files, and resources used in project
4. mflow file – studio specific file used for graphical application representation
5. Application XML and properties files

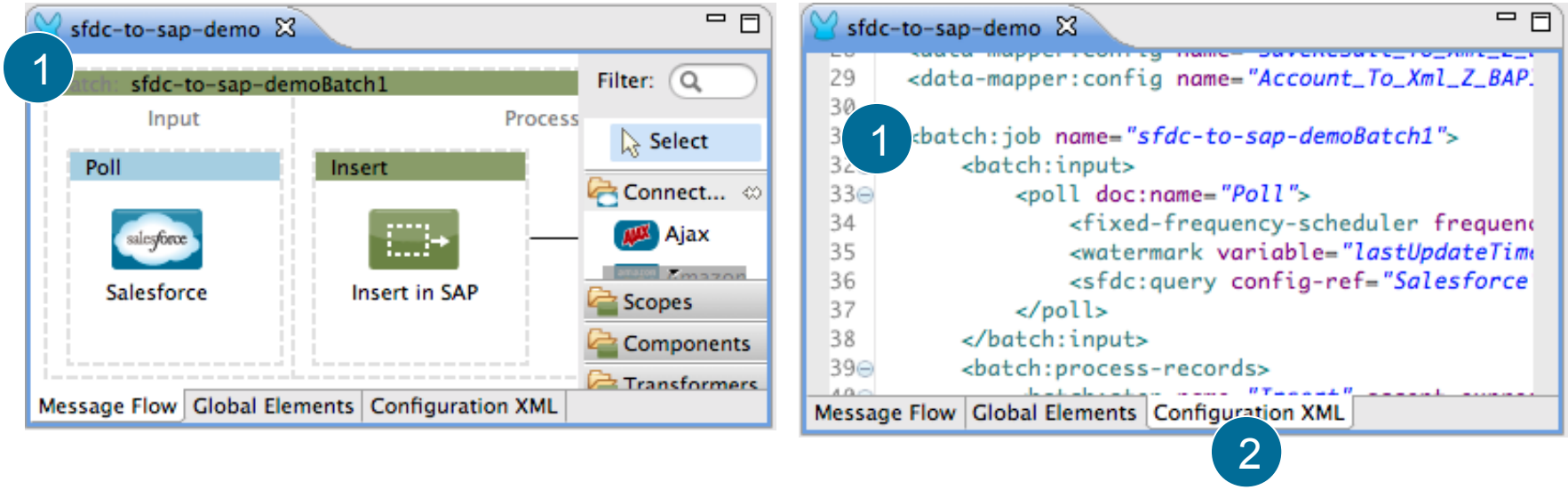


Canvas



1. Message Flow – Graphical (drag and drop) application representation
2. XML – XML representation of application
3. Global Elements – Elements defined once and referenced in the app

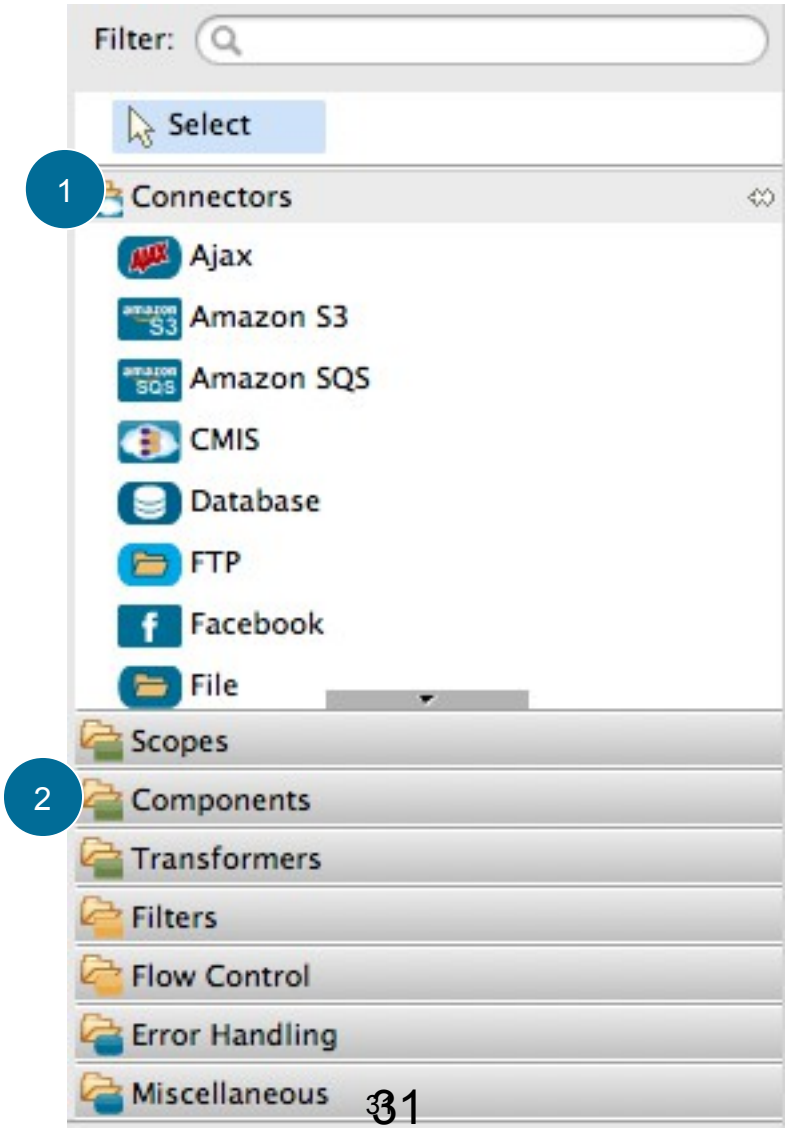
Message Flow & XML



1. XML and Graphical interfaces are tied together
   - Regardless of where we add to our application, they will update each other
2. XML supports auto complete based on available namespaces
   - Ctrl + Space

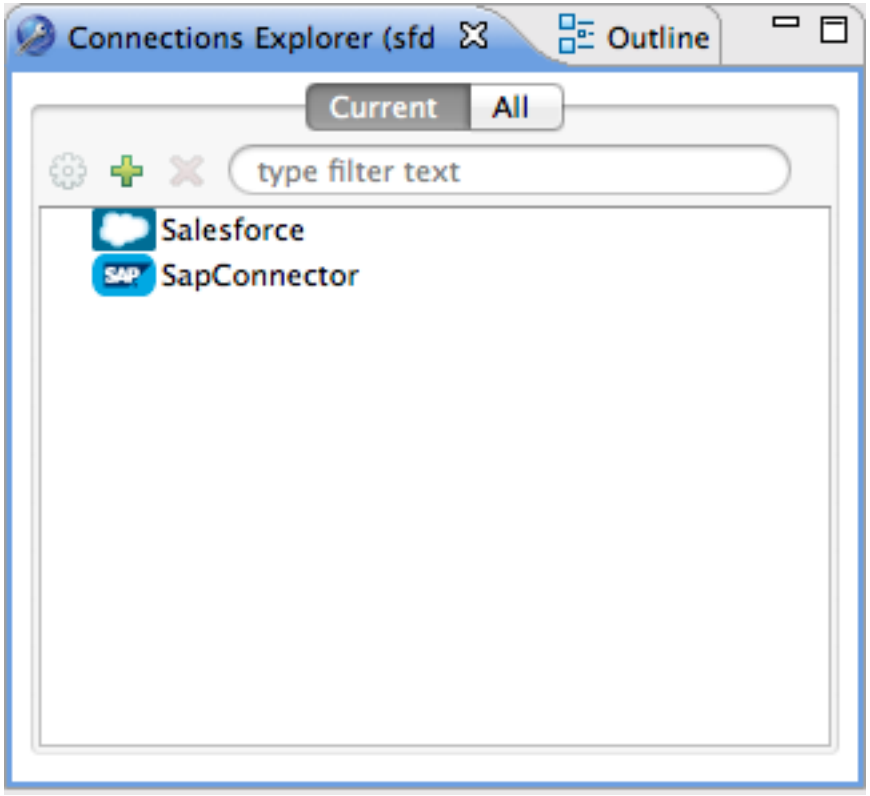
Palette

1. Elements which can be added (or dragged into our application
2. Categories include:
    1. Connectors
    2. Scopes
    3. Components
    4. Transformers
    5. Filters
    6. Flow Control
    7. Error Handling
    8. Miscellaneous (Business Events)

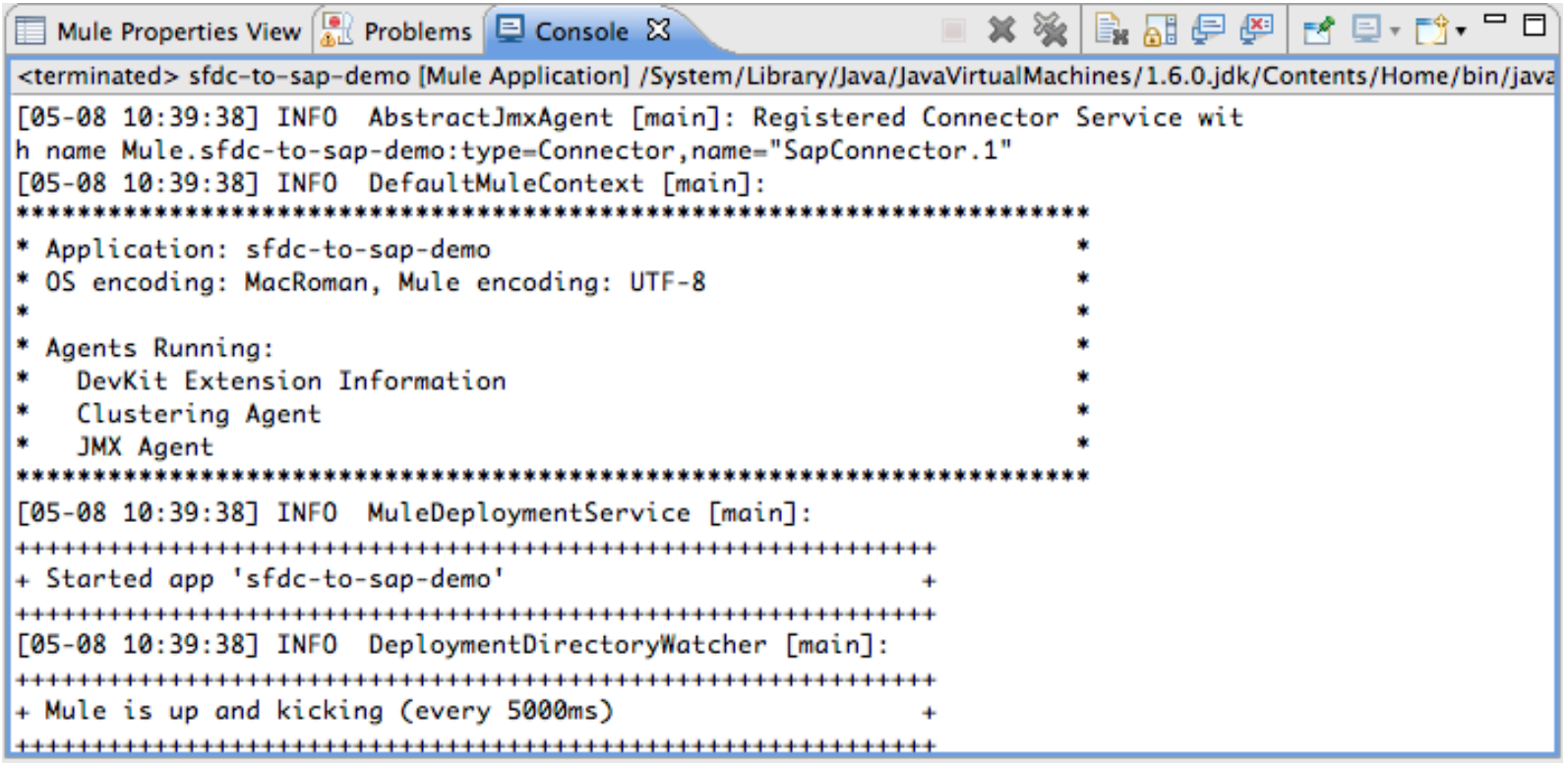


Connections Explorer

1. All Global Elements which are ‘Connectors’.
2. Gives a quick view of connections available for our application to reference

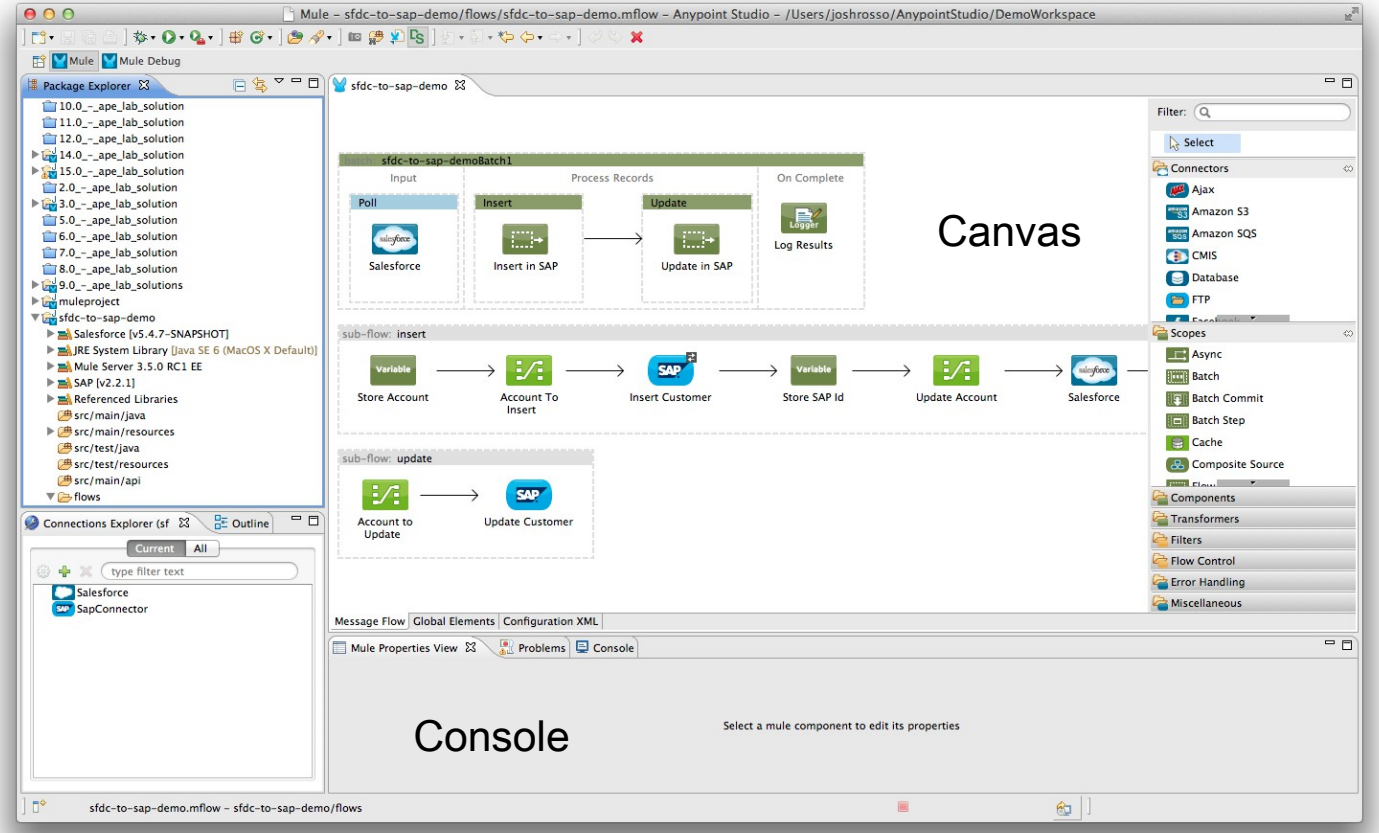


Console



1. Anypoint Studio comes complete with an embedded Mule runtime to test applications without leaving studio
2. Console outputs our application logs and information

Anypoint Studio Review



LAB

Data and Messages

Data Formats

XML JSON Streams Binary

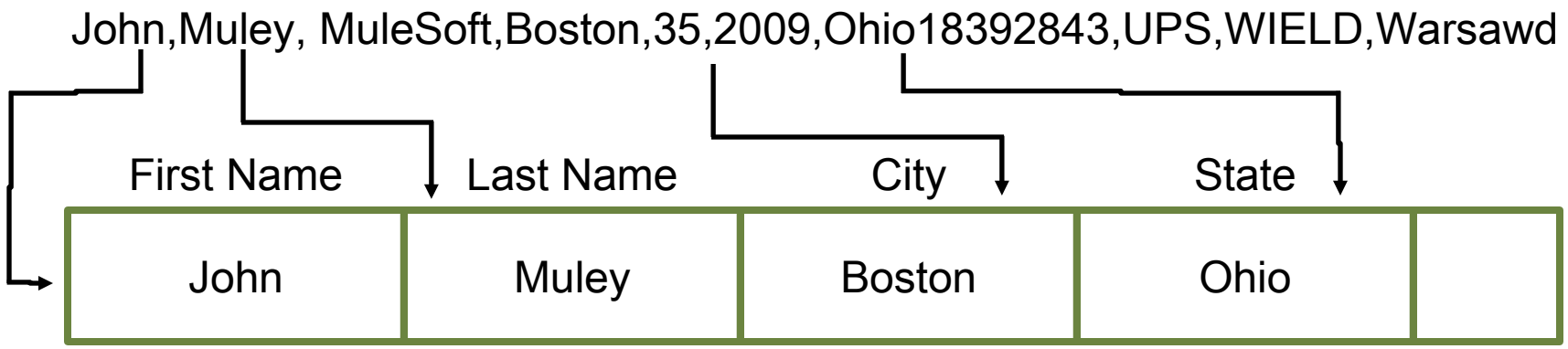
CSV 'Objects' Compressed

- Applications work with various data
- This data can contain anything from People records to Sales numbers
- This data often exists in many different formats

Raw Data

- Raw data is often optimal for performance during transmission of messages
- Accessing specific data often requires parsing

Structured Data



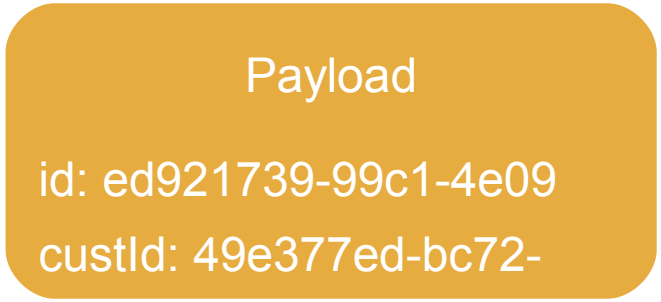
- In memory representation of data
- Origin is often parsed raw data or data from a specific datasource
- Structured data allows for easier accessing of specific data within a message

Payload Representation in Mule

Raw Data

- Raw data is often represented by
  - String
  - InputStream
  - Byte[] (Byte array)

Structured Data



- Structured data often of type
  - Map
  - Structured Java Object
    - (Order, Account, etc)
  - XML DOM
  - JSON Node

MEL Basics

Syntax

MuleSoft

#[] Encapsulates all Mule Expressions

#[message] Holds a context object

#[message.payload]

Dot notation to access fields or methods

Accessing Properties

- message.inboundProperties and message.outboundProperties are maps using which we can access inbound and outbound properties

- We can also use the following syntax for accessing any inbound property : **#[header:inbound:inboundpropertyname]**

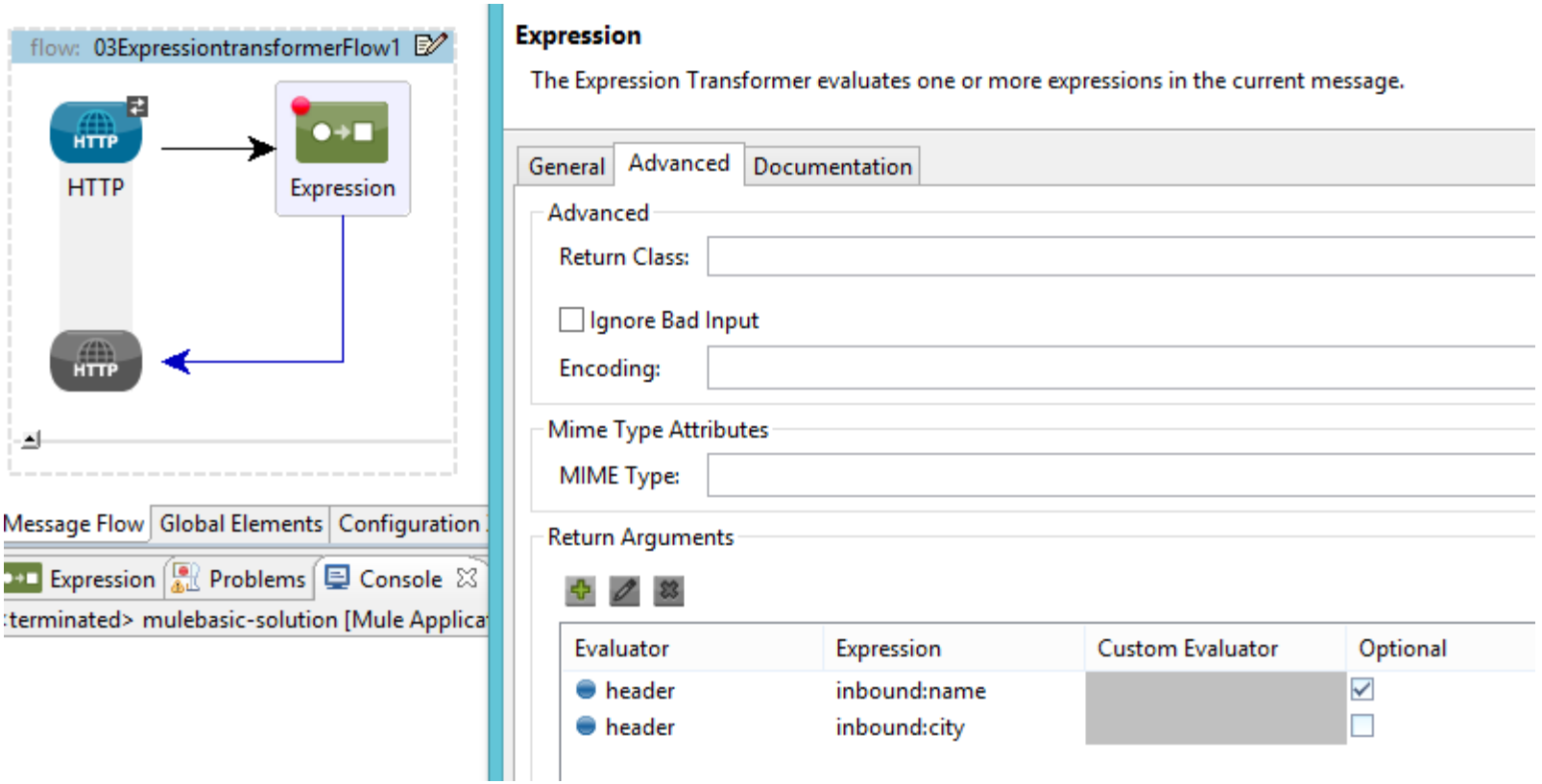
Expression Transformer

- Expression transformer can be used to transform the payload

```
<flow name="03ExpressiontransformerFlow1" doc:name="03ExpressiontransformerFlow1">
    <http:inbound-endpoint exchange-pattern="request-response" host="localhost"
                                                    port="8083" doc:name="HTTP"/>

    <expression-transformer expression="#[payload.substring('2')]"
                                                    doc:name="Expression"/>
</flow>
```

Expression Transformer returning array

- If u want expression transformer to return an array of arguments, u could do as below :



Lab – Using Expression transformer

Entry Point Resolvers

Method Entry point resolvers

- What if two or more conflicting methods are present in same class ?

- Solution :

```
<component class="com.mulesoft.training.PriceComponentWithTwoMethods" doc:name="Java">
    <method-entry-point-resolver>
        <include-entry-point method="getRate"/>
    </method-entry-point-resolver>
</component>
```

Reflection Entry point resolver

MuleSoft

- What if a method has to be invoked on your Java component based on method argument types and number of arguments based on reflection ?

- Consider the following java class :

```
public class ReflectionEntryPointResolverExample {

    public String multipleArgumentsMethod(String arg1, String arg2) {
        return "Request resolved to multipleArgumentsMethod ("
                                        + arg1 + ", " + arg2 + ")";
    }
    public String singleArgumentMethod(String arg1) {
        return "Request resolved to singleArgumentsMethod(" + arg1 + ")";
    }
    public String noArgumentsMethod() {
        return "Request resolved to noArgumentsMethod";
    }
}
```

- Solution : Use reflection-entry-point-resolver as below :

```
<flow name="03ReflectionEntryPointResolverFlow1" doc:name="03ReflectionEntryPointResolverFlow1">
    <http:inbound-endpoint exchange-pattern="request-response" host="localhost" port="8085" doc:name="HTTP"/>
    <expression-transformer doc:name="Expression"   >
        <return-argument expression="null" optional="true" />
        <!-- <return-argument evaluator="header" expression="inbound:name" optional="true"/> -->
       <!--  <return-argument evaluator="header" expression="inbound:dept" optional="true"/> -->
    </expression-transformer>
    <logger  message="#[message.payload]" doc:name="Logger" level="INFO"/>

     <component class="com.mulesoft.training.ReflectionEntryPointResolverExample" doc:name="Java" >
        <reflection-entry-point-resolver/>
    </component>
</flow>
```

Entry point resolver set

- What if u want to use more than one entry point resolvers ?

```
<component class="com.mulesoft.training.PriceComponentWithTwoMethods" doc:name="Java" >
    <entry-point-resolver-set>
        <reflection-entry-point-resolver />
        <method-entry-point-resolver>
            <include-entry-point method="getRate"/>
        </method-entry-point-resolver>
    </entry-point-resolver-set>
</component>
```

Property entry point resolver

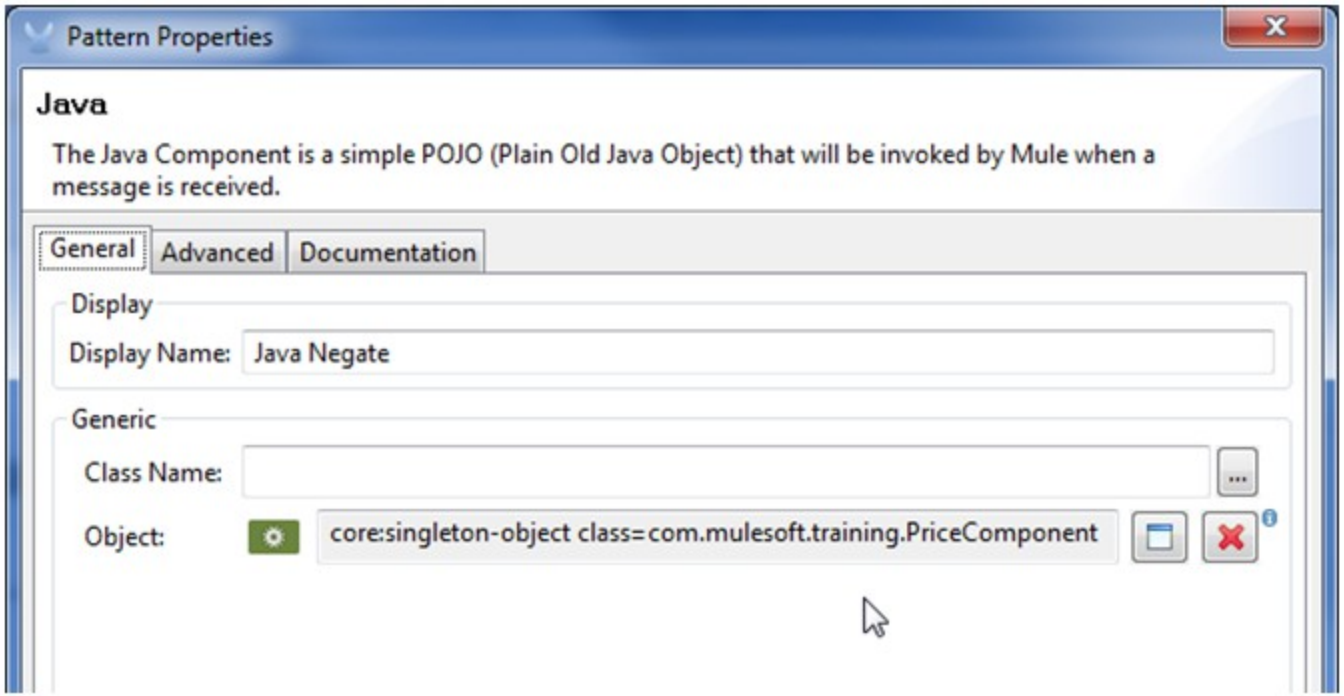
- What if u want to resolve the method to be invoked based on a property in message ?

- Solution : Use **<property-entry-point-resolver />**

```
<property-entry-point-resolver property="methodName"/>
```

LAB – Using Entry point Resolvers

Defining a Singleton component




```
<component doc:name="Java Negate">
     <singleton-object class="com.mulesoft.training.PriceComponent">
          <property key="mykey" value="myvalue"/>
     </singleton-object>
</component>
```

Annotations Example

MuleSoft

```
public class AnnotatedComponent {

    @Lookup
    private PriceComponent priceComponent;


    public String processMessage(@Payload String name,
            @Mule(value="message.id",optional=true)String id,
            @InboundHeaders("Host")String hostName){

        System.out.println("Price Comp : "+priceComponent);
        System.out.println("Payload : "+name);
        System.out.println("Message Id : "+id );
        System.out.println("Host Name : "+hostName);
        return "Hello "+name;
    }
}
```

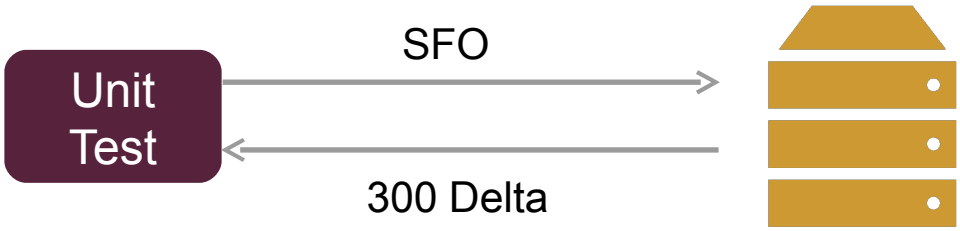
Available Annotations

- @Lookup (field/parameter):
  - Inject object lookup from registry
- @Payload (parameter)
- @InboundHeaders (parameter):
  - Supports Map, List, single header & wildcards
- @OutboundHeaders (parameter):
  - Use Map to add outbound headers
- @Xpath (parameter)
- @Groovy (parameter)
- @Mule (parameter)
- @Function (parameter)
- @Schedule (method)

LAB : Using Annotations

Functional TESTING

"Black box" testing



- Uses **JUnit**
- Can test **all** or **part** (a "**unit**") of an application
- Invokes code to be tested, runs **assertions**
- Can be automated using **Maven**

A Mule JUnit test case

```
package com.mulesoft.mule.training.examples;

import org.mule.tck.junit4.FunctionalTestCase;
import …

public class ApplicationTestCase extends FunctionalTestCase {

    @Test
    public void testFirstCondition() throws Exception{
     // TODO: Fill in code …
    }

    protected String getConfigFile() {
        return "src/main/app/resources.xml";
    }
}
```

Mule class for JUnit test case

Java annotation for JUnit test method

implement to get resources

- **Note** that getConfigFiles is an alternative method which can be used to feed multiple configuration files into one test class. getConfigFiles returns a String[].

Communicating with the Mule Server

- Get an instance with a MuleClient interface
- **import** org.mule.api.client.**MuleClient**; // client interface
- MuleClient **client** = muleContext.**getClient**();

- Send / Receive events *via* MuleClient synchronously:

MuleMessage send(String url, **MuleMessage** msg); // Send and Receive - blocking

- Send / Receive events *via* MuleClient asynchronously:

void **dispatch**(String url, **MuleMessage** msg); //Put message on endpoint
MuleMessage request(String url, long timeout); //Wait for return message

Asserting results

- Property assertions take a single argument:

static void **assertNull**(…);
static void **assertNotNull**(…);
static void **assertTrue**(…);
static void **assertFalse**(…);

- Relational assertions take two arguments:

static void **assertEquals**(<expected>,<actual>);

- Asserting a String Return Value

static void **assertEquals**("500 United",myPayloadAsString);

Adding a unit test class

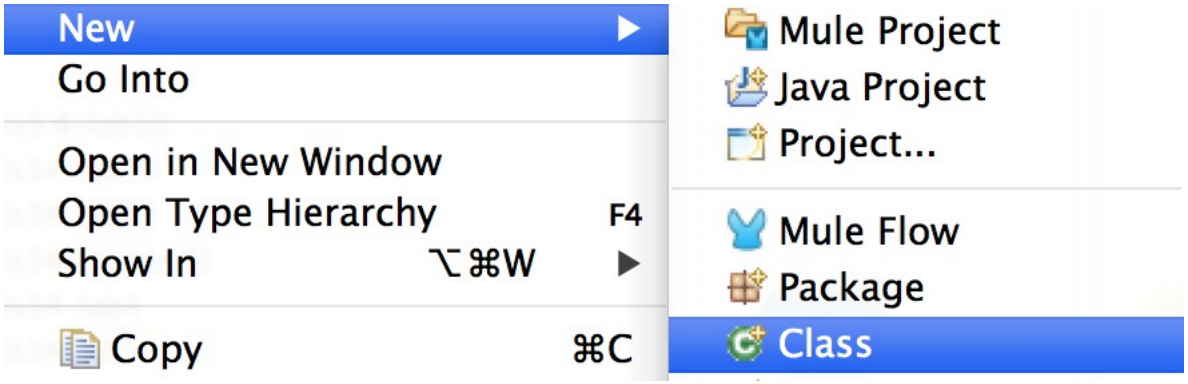
MuleSoft

- Walkthrough Steps

1. **Create** and **extend** the Mule class for test cases
2. **Import** (static) assertion methods from JUnit
3. **Annotate** the test method
4. **Implement** the method to set configuration resources

Adding a unit test class

1. **Create** and **extend** the Mule class for test cases
   - In Mule Studio, right click on **src/test/java**
   - Choose **New** > **Class**



   - In the **New Java Class** dialog, for **Superclass**, enter the name of the **Mule test case base class**:

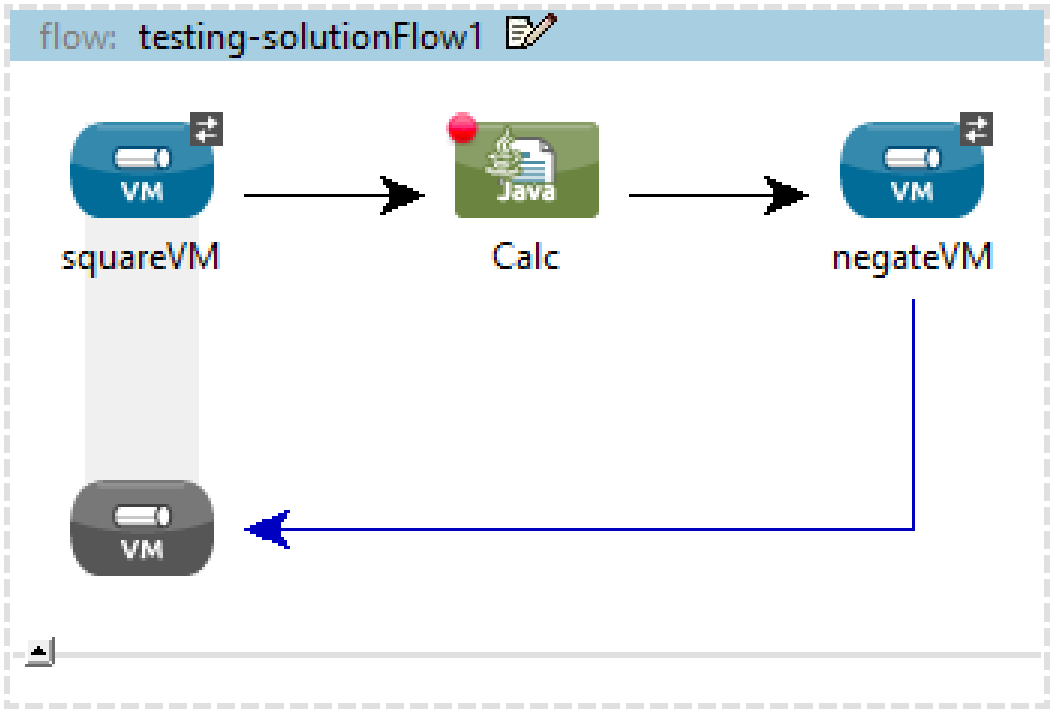
     org.mule.tck.junit4.FunctionalTestCase

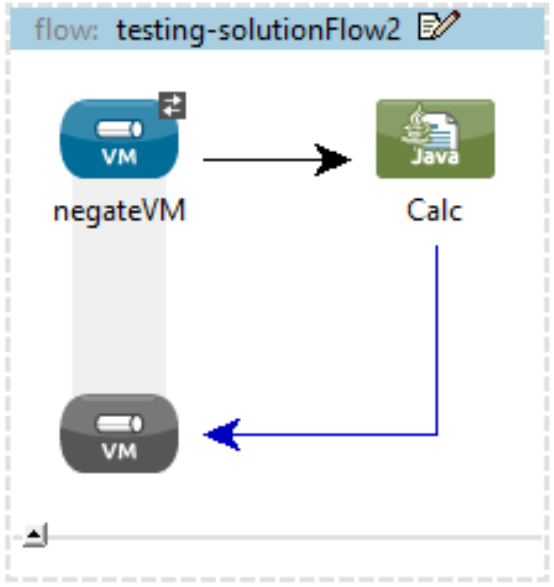
Example of test case

MuleSoft

```
public class ApplicationTestCase extends FunctionalTestCase {

    @Test
    public void testFirstCondition() throws Exception{
        // TODO: Fill in code ...
    }

    protected String getConfigResources() {
            return "/src/main/app/flows1.xml";
    }
}
```

Sample Unit Test





```
public class SquareNegateTest extends FunctionalTestCase {

@Test
public void testSquareNegate1() throws MuleException {
        MuleClient client = muleContext.getClient();
        Integer inte = new Integer(5);
        MuleMessage reply = client.send ("vm://math", inte, null, 5000);
        assertNotNull(reply);
        assertNotNull(reply.getPayload());
        assertTrue(reply.getPayload() instanceof Integer);
        Integer result = (Integer)reply.getPayload();
        assertEquals(result.intValue(),-25);

}
@Override
protected String getConfigResources() {
    return "src/main/app/math.xml";

}
}
```

MEL in Detail

Context Objects

MuleSoft

server	Operating system which message processor is running
mule	Mule instance which the application is running
application	User application which current flow is deployed
message	The Mule Message which the message processer is processing

Context Variables

flowVars

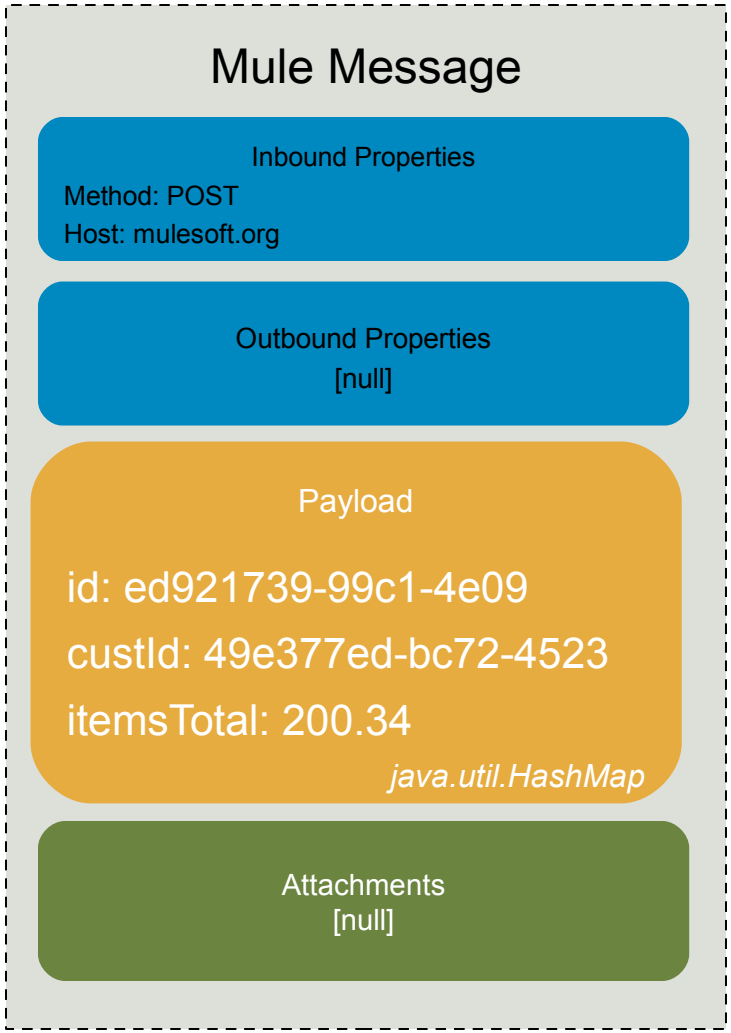
sessionVars

recordVars

- Allows for the accessing of various variable types
- Example of accessing the flow variable 'ticketNum'
- More on variables in a future module

#[flowVars['ticketNum']]

Accessing Map Data



#[message.payload['id']]

ed921739-99c1-4e09

#[message.payload.custId]

49e377ed-bc72-4523

#[message.payload['itemsTotal']]

200.34

#[message.inboundProperties['host']]

mulesoft.org

Accessing 'Relational' Map Data

MuleSoft

First Name	Last Name	City	State	
John	Muley	Boston	Ohio	
Mark	Dailer	Cleveland	Ohio	
Bill	Muley	Avon	Ohio	

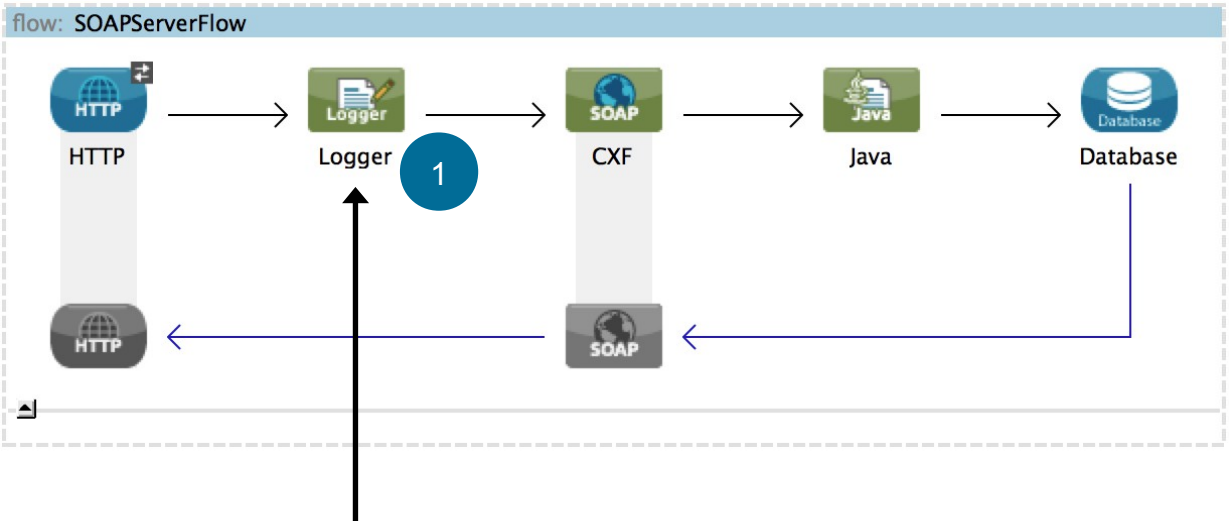
#[message.payload[1]['LastName']]

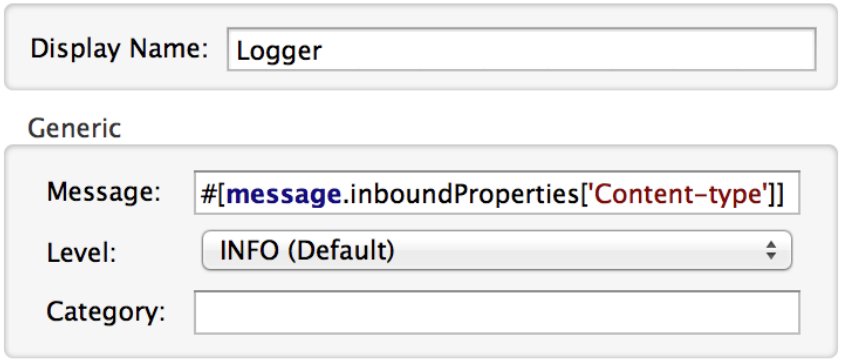
Dailer

#[message.payload[0]['City']]

Boston

MEL Example

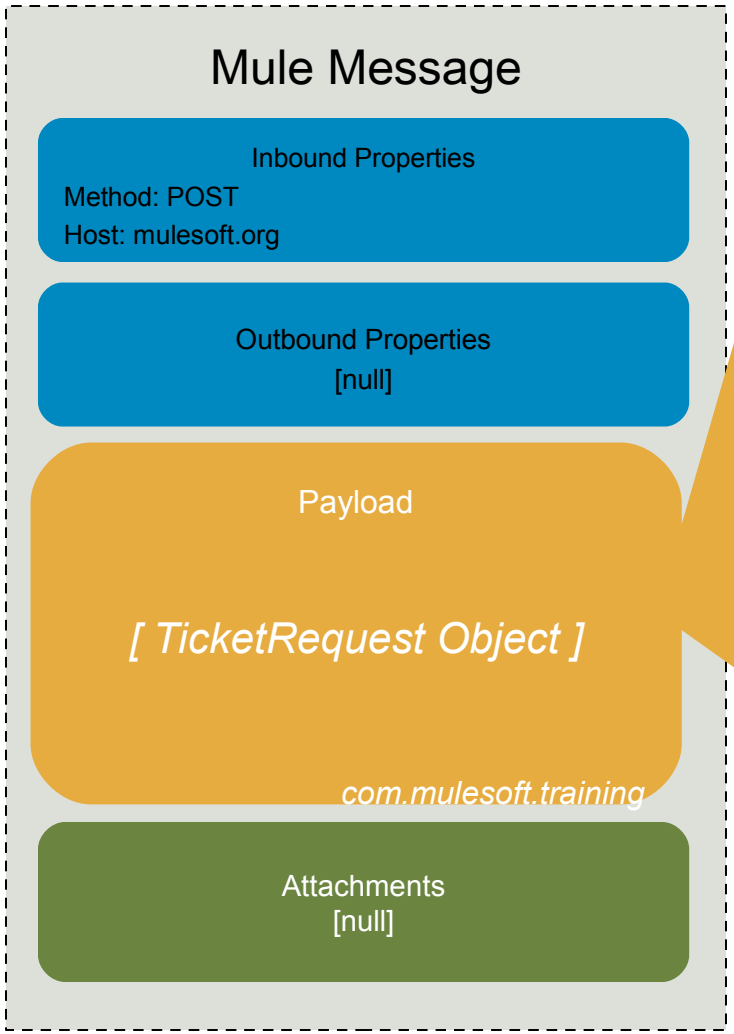




Logs: 'text/xml'

- MEL calls the inbound property from the Mule Message for the value of 'Content-type'

Accessing an Object's Method(s)



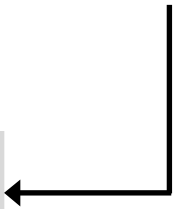

```
package com.mulesoft.training;

public class TicketRequest  {
    String airline;
    double price;
    double fees
    String airportCode;

    // Constructor(s)

1   public String getTotal() {
        String currentTotal =
        "$" +
        String.valueOf(price + fees);
        return currentTotal;
    }
```

1. How can we trigger this method with MEL?

Accessing an Object's Method(s)

[Context Object] [Type = TicketRequest] [Method in POJO]

```
#[message.payload.getTotal()]
```



```
public String getTotal() {
        String currentTotal = "$" +
            String.valueOf(price + fees);
        return currentTotal;
}
```

- Also can be helpful when looking to use methods defined in Java's 'Object' class.
  - #[message.payload.toString()];
  - #[message.payload.getClass()];

Accessing Properties and Variables

MuleSoft

< Key,Value >

By placing the key we wish to retrieve into an MEL expression, we'll receive the object stored within the property / variable

['JMScorrelationId'] produces 0173234

Accessing the key **JMSCorrelationId** produces the value 0173234. This value is of type 'String'

MuleSoft

Accessing Properties

```
#[message.inboundProperty['http.method']]
```

POST

```
#[message.outboundProperty['Content-Type']]
```

application/xml

MuleSoft

Accessing Variables

```
#[flowVars['fileName']]
```

```
03042013StandardReport.txt
```

```
#[sessionVars['flagCount']]
```

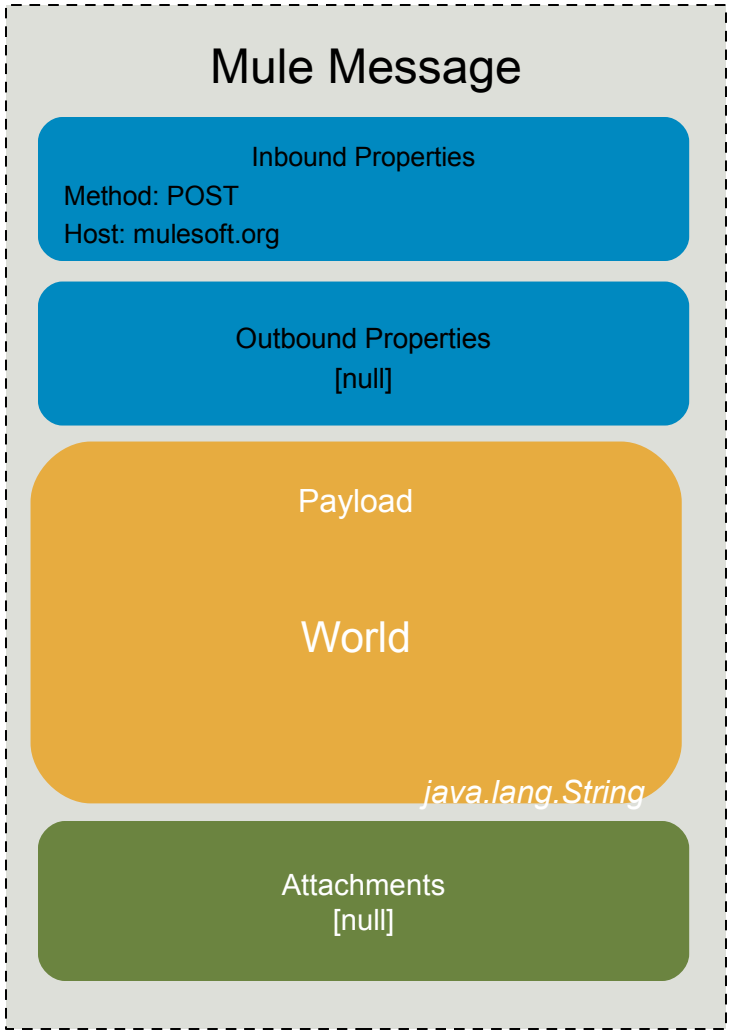
```
15
```

Operators - Arithmetic

MuleSoft

+	Plus – additions for numbers, concatenation for Strings
-	Minus
/	Over
*	Multiply
%	Modulo – Remainder operator

Operators - Arithmetic




```
#['Hello ' + message.payload]
```

↓

Hello World

Operators - Evaluation

MuleSoft

==, !=, >, <, >=, =<

Standard comparison operators

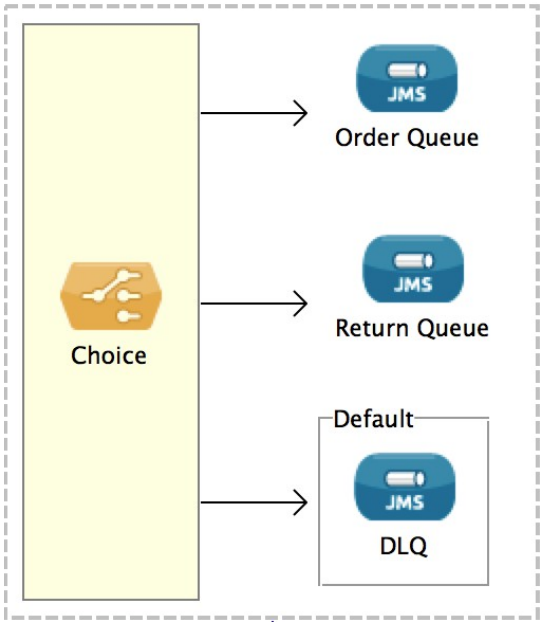
contains

Similar to 'like' operator

is

Checks the type of an object

Operators - Evaluation



	When	Route Message to
1	#[message.inboundProperties['requestType'] == 'order']	Order Queue
2	#[message.inboundProperties['requestType'] == 'return']	Return Queue
	Default	DLQ

```
<choice doc:name="Choice">
    <when expression="#[message.inboundProperties['requestType'] == 'order']">
        <jms:outbound-endpoint queue="order.processing" doc:name="Order Queue" connector-
            ref="Active_MQ" />
    </when>
    <when expression="#[message.inboundProperties['requestType'] == 'return']">
        <jms:outbound-endpoint queue="return.processing" doc:name="Return Queue" connector-
            ref="Active_MQ" />
    </when>
    <otherwise>
        <jms:outbound-endpoint queue="dlq.processing" doc:name="DLQ" connector-ref="Active_MQ" />
    </otherwise>
</choice>
```

Data Extraction

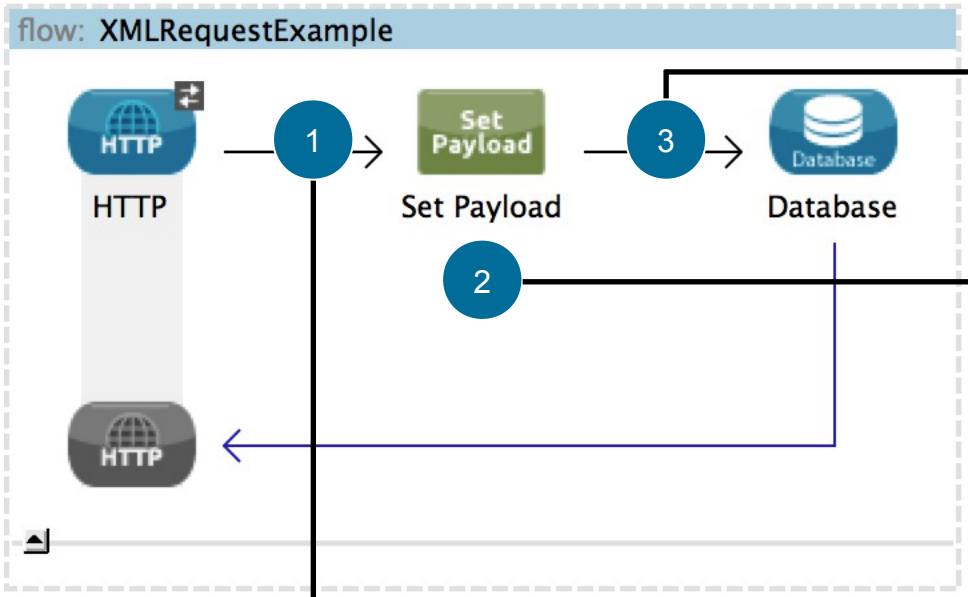
MuleSoft

#[xpath('expression')]

#[regex('expression')]

Both xpath and regular expression are supported with MEL

xpath Extraction



Payload:

```
Computer
```

Payload:

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
    <item>Computer</item>
    <requestor>Dave Muley</requestor>
    <requestDate>12/25/2012</requestDate>
</request>
```

Display Name: Set Payload

Settings

Value: #[xpath('/request/item').text]

LIST AND MAPS IN MEL

- Creating Map in MEL :

```
['id':'5','name':'Siva','city':'bang']
```

- Creating Array in MEL :

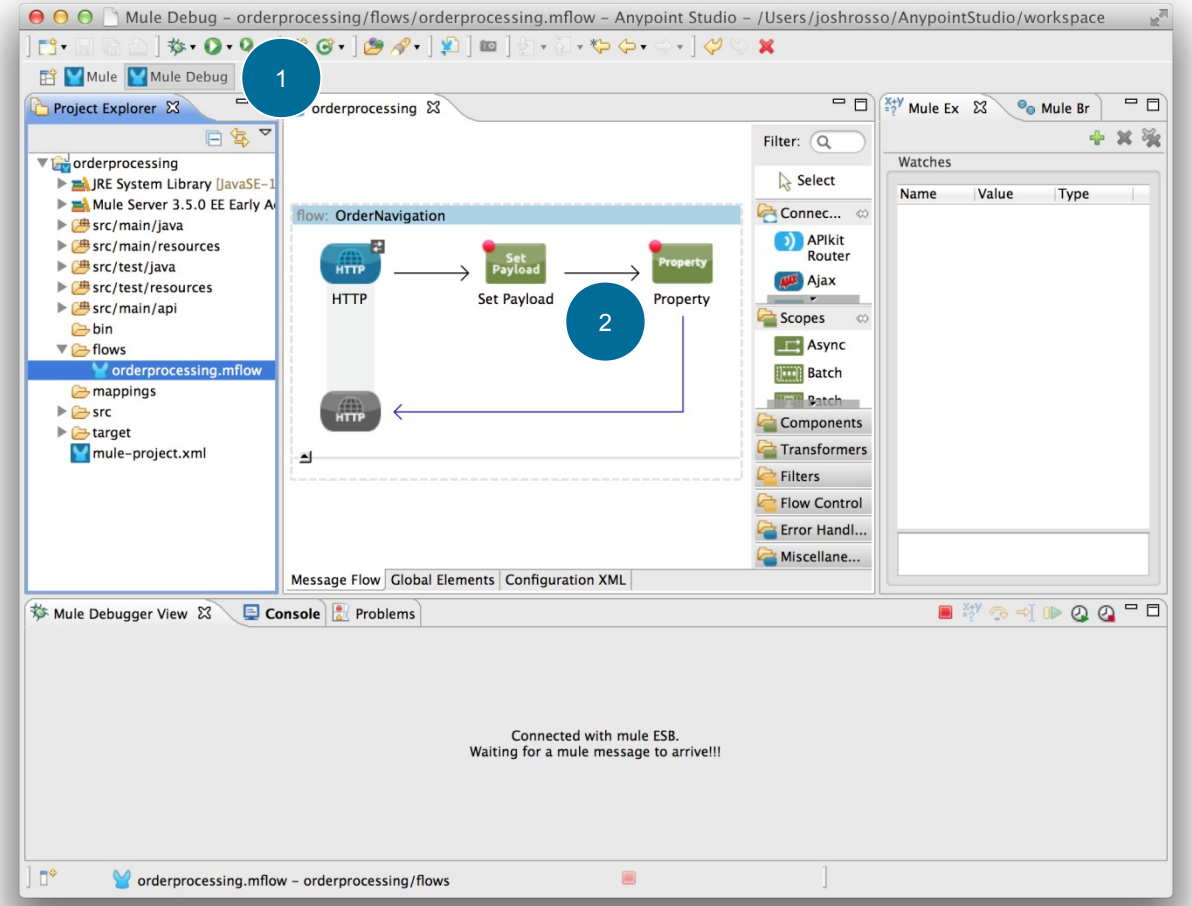
```
[
    ['id':'5','name':'Siva','city':'bang'],
    ['id':6,'name':'b','city':'bang'],
    ['id':'7','name':'c','city':'bang']
]
```

Creating a List in MEL

```
{
    ['id':'5','name':'Siva','city':'bang'],

    ['id':6,'name':'b','city':'bang'],

    ['id':'7','name':'c','city':'bang']
}
```

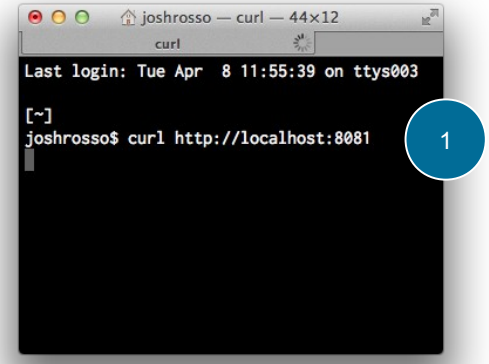
Tools

Debug Perspective

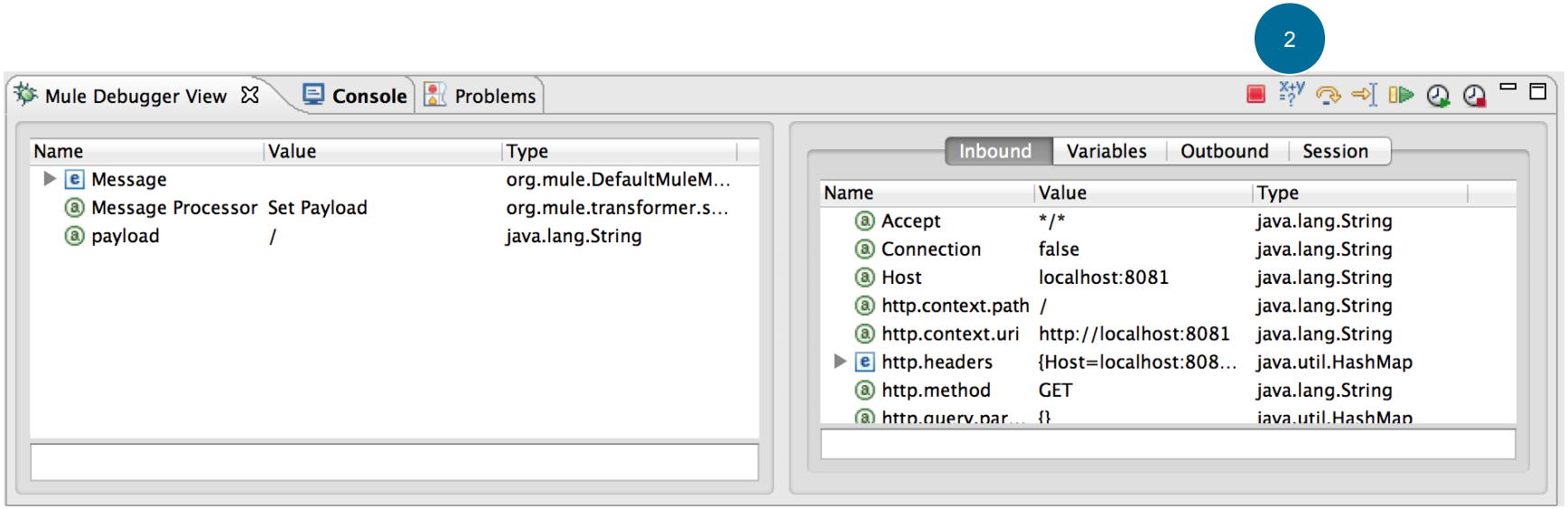


1. Run the application in Mule Debug perspective
2. Toggle breakpoints on each message processor

Evaluate Mule Expression



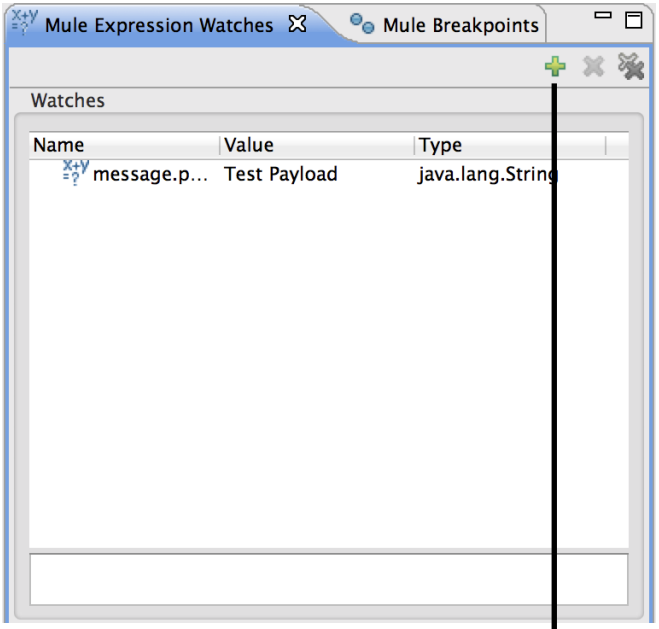
1. Trigger a message that will initiate the flow.
2. Click Evaluate Mule Expression within the Mule Debugger view

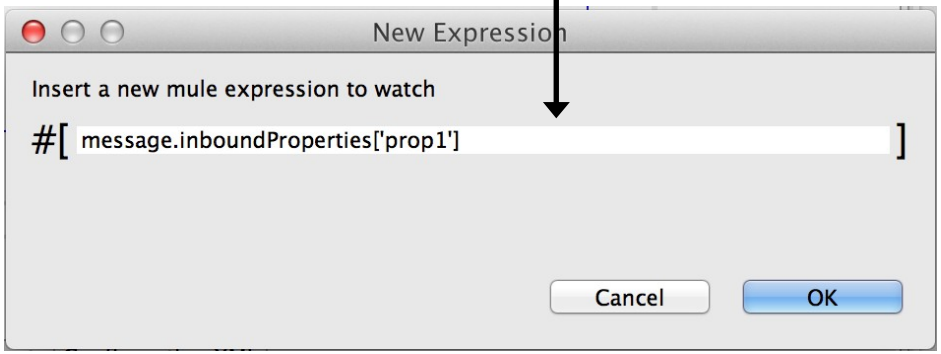


Evaluate Mule Expression

- As we step through the debugger we can use this window to test out MEL and see what the value is that comes back
- This is volatile and will reset when changing steps

Mule Expression Watches

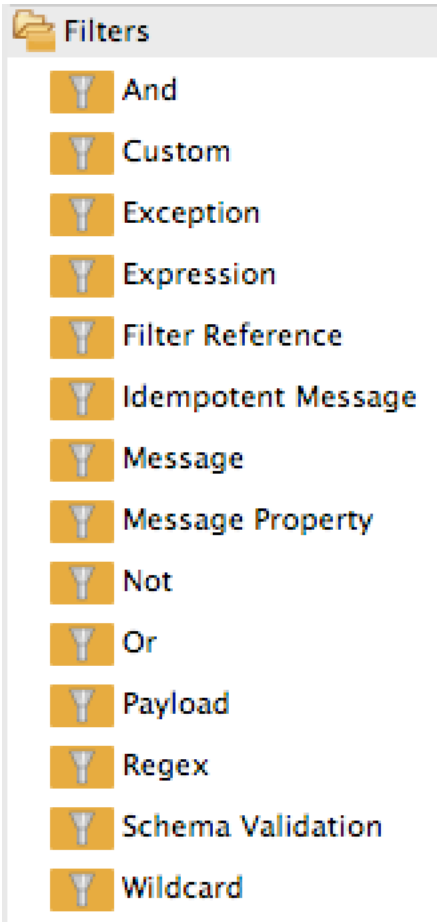




- Mule Expression watches allow us to see MEL be evaluated at each step
- These expressions will persist throughout steps and tests
- We can add properties with the 'add' button

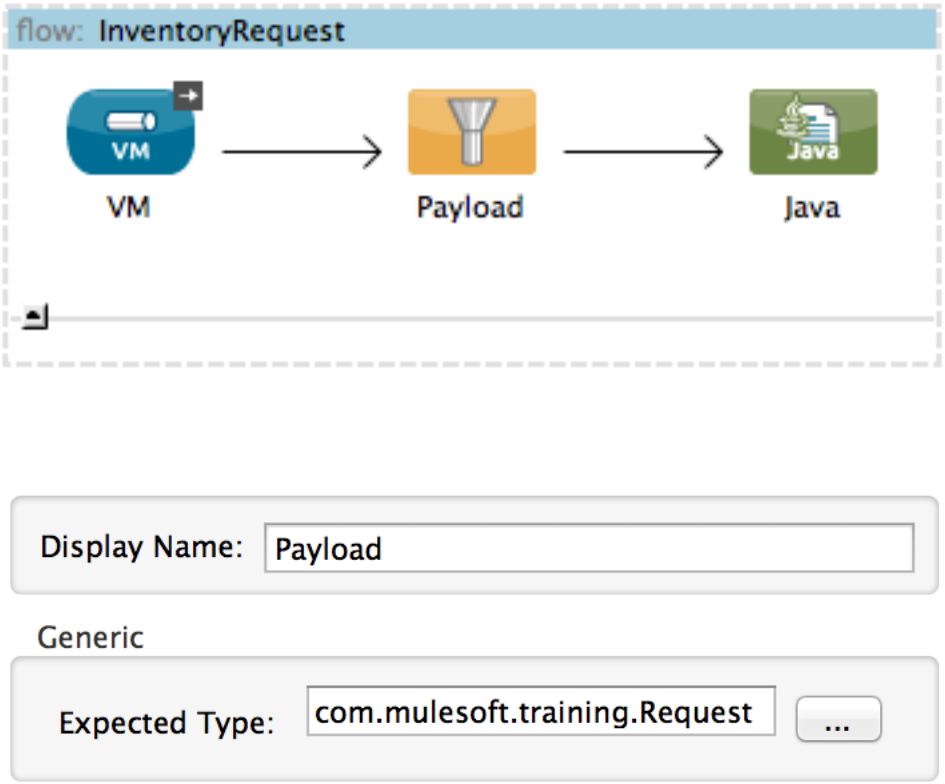
Filters

Filtering Messages



- **Purpose**
  - Subsequent processors in flow can **avoid** receiving **irrelevant** or **incomprehensible messages**
  - Separate, single **encapsulation** of filtering
- **Behavior**
  - **Evaluates** message based on filter subtype, often using Mule Expression Language (MEL)
  - **Accepts** message or not
  - **Passes** message only when accepted
  - Can throw by **Throw On Unaccepted** boolean property of message filter, which nests other filters
- **Subtypes**
  - **And**, **Or**, **Not** apply boolean logic to multiple filters
  - **Message** filter nests other filters and throws
  - **Exception** matches expected exception and passes

Payload Type Filter



- Checks the type of a message's payload
- exceptedType attribute allows for the specification of a Java class
- All messages with payload types other than Request are dropped

Expression Filter

- **Using XPath Expressions**

```
<expression-filter evaluator="xpath" expression="(msg/header/resultcode)='success'"/>
```

```
<flow name="04XpathExpressionFilterFlow1" doc:name="04XpathExpressionFilterFlow1">
    <vm:inbound-endpoint exchange-pattern="request-response" path="testxpathq" doc:name="VM"/>

    <logger message="#[xpath:/shiporder/shipto/city]" level="INFO" doc:name="Logger"/>

    <expression-filter evaluator="xpath" expression="/shiporder/shipto/city='hyderabad'"
                doc:name="Expression"/>

    <logger message="payload" level="INFO" doc:name="Logger"/>
</flow>
```

RegEx Filter

- Applies a regular expression pattern to the **message payload.**

```
<regex-filter pattern="the quick brown (.*)"/>
```

Wildcard Filter

- Applies a wildcard pattern to the message payload.
- The filter applies toString() to the payload

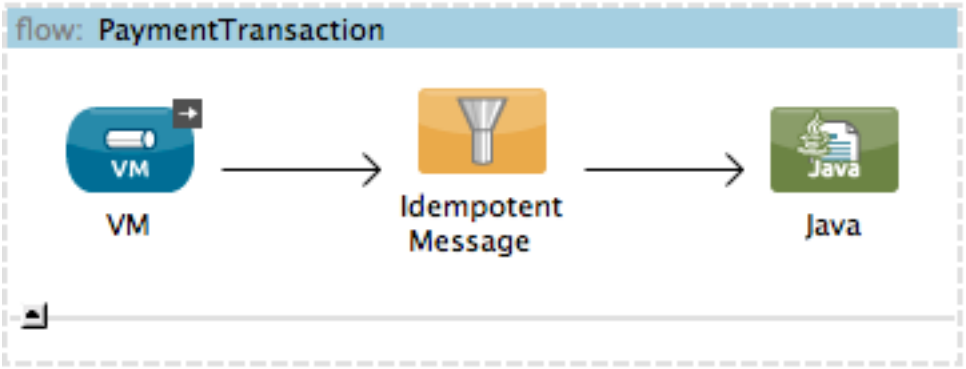
```
<wildcard-filter pattern="the quick brown *"/>
```

Exception Type Filter

- A filter that matches an exception type.

```
<exception-type-filter expectedType="java.lang.RuntimeException"/>
```

Idempotent Filter



- Ensures a message is not delivered more than once
- Threshold set for how long transactions are stored
- Allows for storage in both objectstores and simple text files

```
 <idempotent-message-filter doc:name="Idempotent Message"
idExpression="#[message.payload['transID']]" >
            <simple-text-file-store name="TransactionStore"
expirationInterval="60000"/>
        </idempotent-message-filter>
```

Message Property Filter

- This filter **allows you add logic to your routers** based on the value of one or more properties of a message.
- This filter can be very powerful because the message properties are exposed, allowing you to reference any transport-specific or user-defined property

```
<message-property-filter pattern="Content-Type=text/xml" caseSensitive="false"/>
```

```
<message-property-filter pattern="JMSCorrelationID=1234567890"/>
<message-property-filter pattern="JMSReplyTo=null"/>
```

MuleSoft

- **And Filter**

```
<and-filter>
    <payload-type-filter expectedType="java.lang.String"/>
    <regex-filter pattern="the quick brown (.*)"/>
</and-filter>
```

- **OR Filter**

```
<or-filter>
    <payload-type-filter expectedType="java.lang.String"/>
    <payload-type-filter expectedType="java.lang.StringBuffer"/>
</or-filter>
```

Message Filter

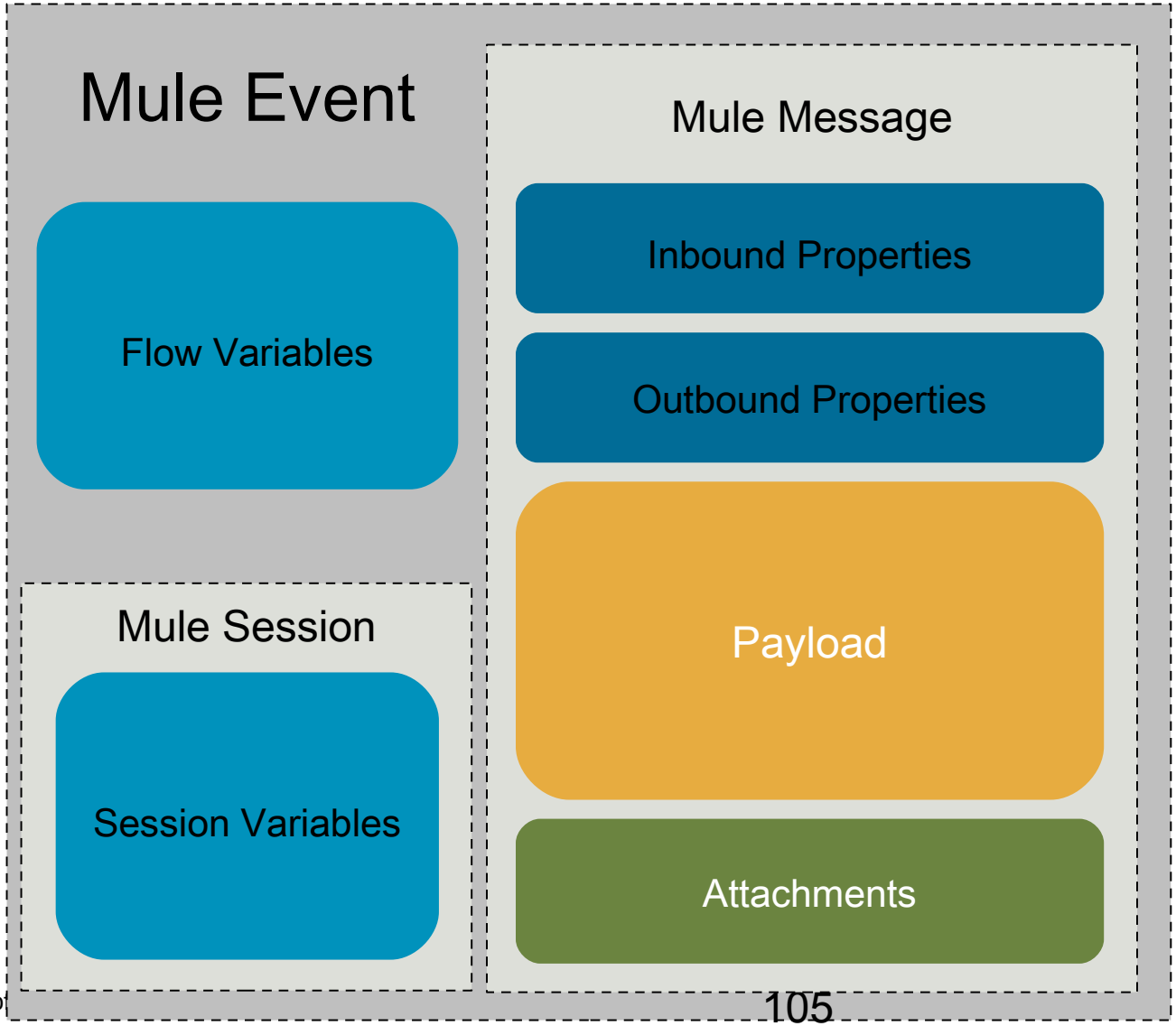
- Message Filter defines
  - A single filter which consists of multiple filters that the message should match for it to continue progressing through a flow
  - What happens when message doesn’t meet the filter?
    - We can configure whether exception should be thrown or it should be redirected to a specific channel

```
<flow name="MessageFilterFlow">
 <vm:inbound-endpoint path="MessageFilterRequest"/>

 <message-filter throwOnUnaccepted="false">
   <wildcard-filter pattern="*OK*"/>
 </message-filter>
 <echo-component/>

 <vm:outbound-endpoint path="MessageFilterResult"/>
</flow>
```

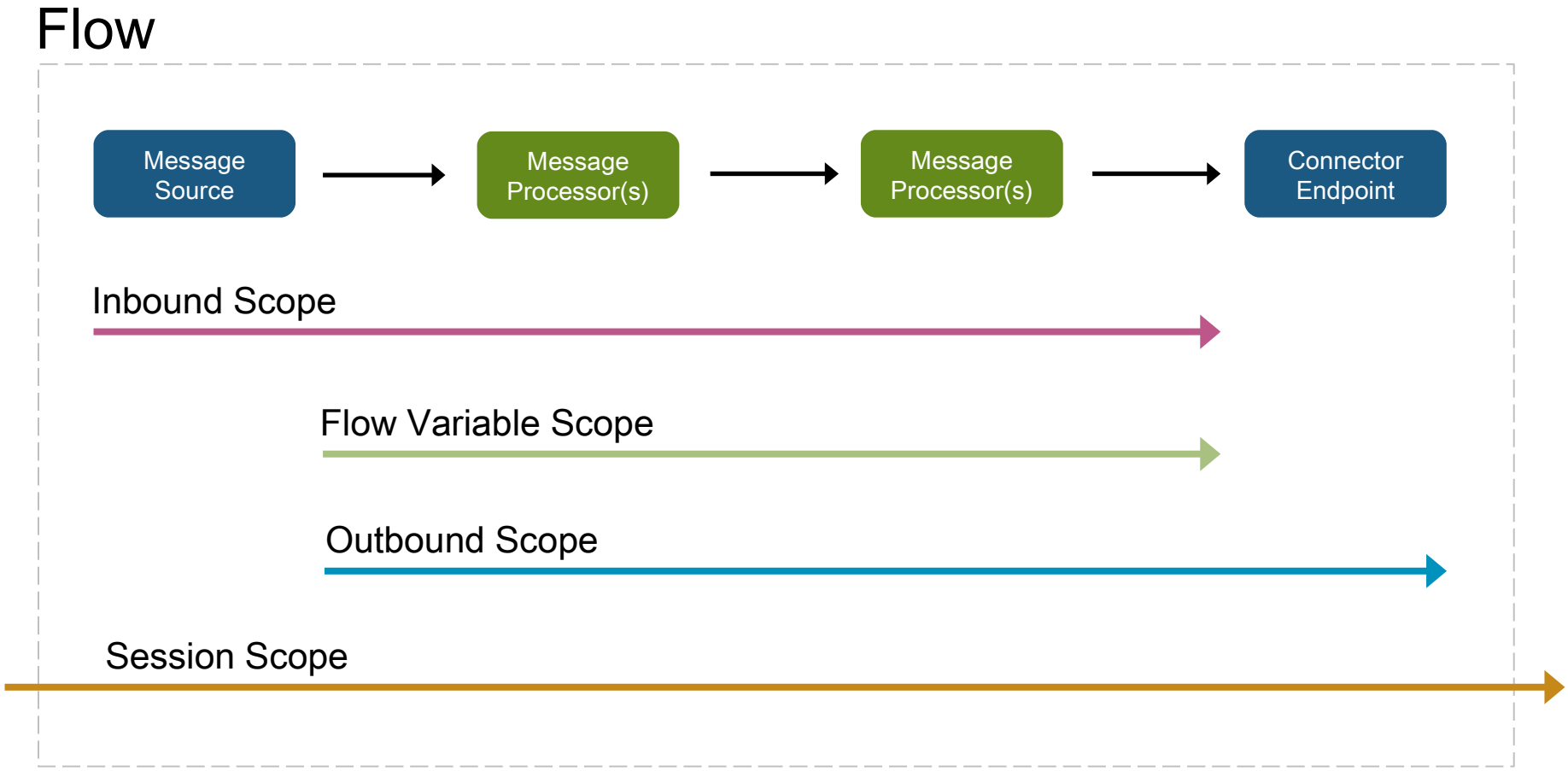
LAB – Using Filters

Mule Variables

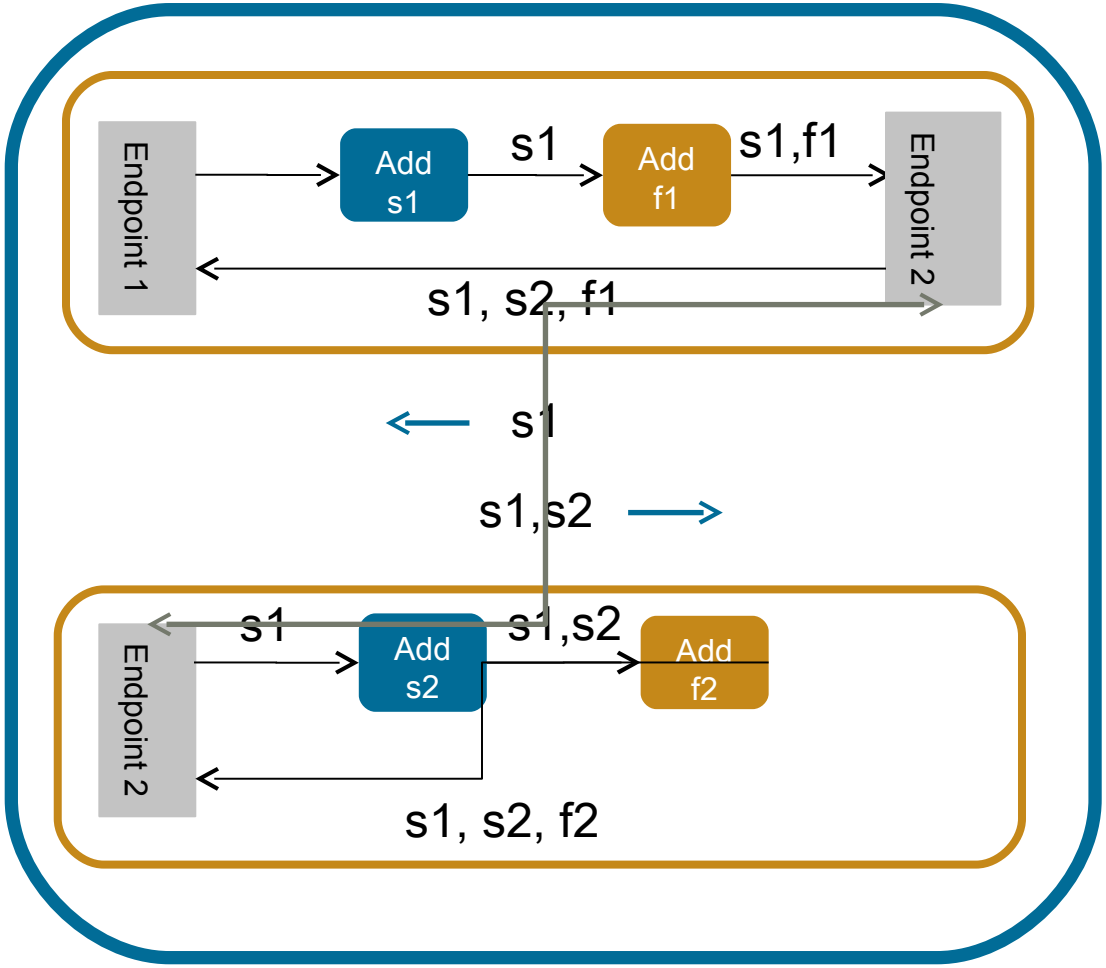
- Flow Variables
- Session Variables



Variable Persistence



Variable Persistence

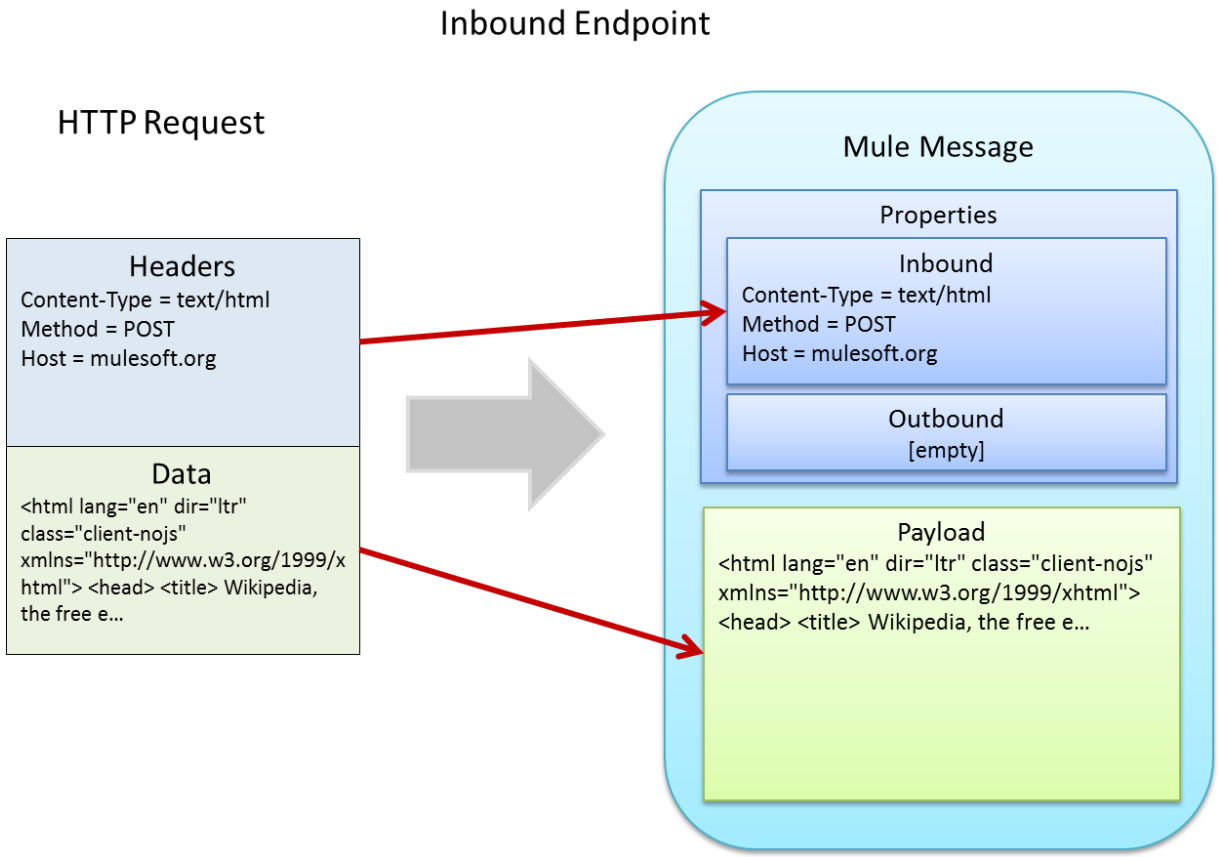


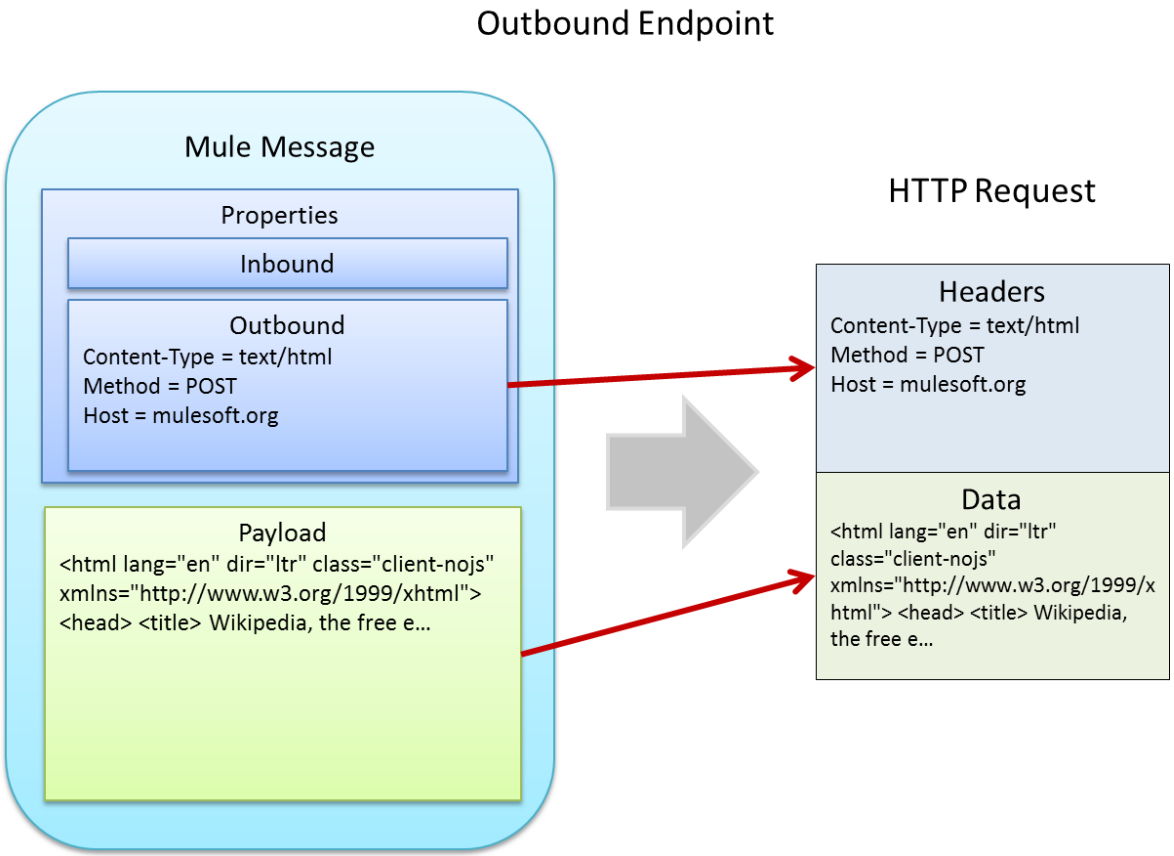
- **Flow 1**
  - **session** variable s1
  - **flow** variable f1
- **Flow 2**
  - **session** variable s2
  - **flow** variable f2

How are inbound Properties populated?

MuleSoft

- A transport reads from a Mule message's properties to populate an outgoing message's header fields (e.g., HTTP)
- A transport extracts an incoming message's header fields and casts them into properties, which are added to the Mule message.





110

Variable/Property Message Processors

MuleSoft

Variable

Sets or Removes Flow Variables

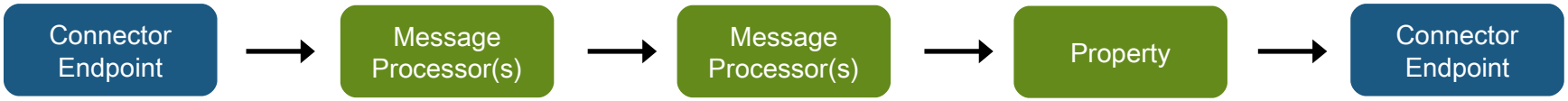
Session Variable

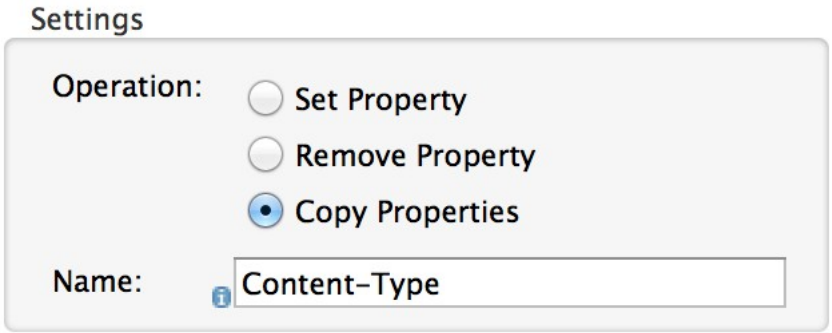
Sets or Removes Session Variables

Property

Sets, Removes, or Copies Properties

'Copying' Properties





- Copies properties from inbound to outbound
- Great when you wish the final endpoint to assess data in inbound properties

MuleSoft

Accessing Variables

```
#[flowVars['fileName']]
```

```
03042013StandardReport.txt
```

```
#[sessionVars['flagCount']]
```

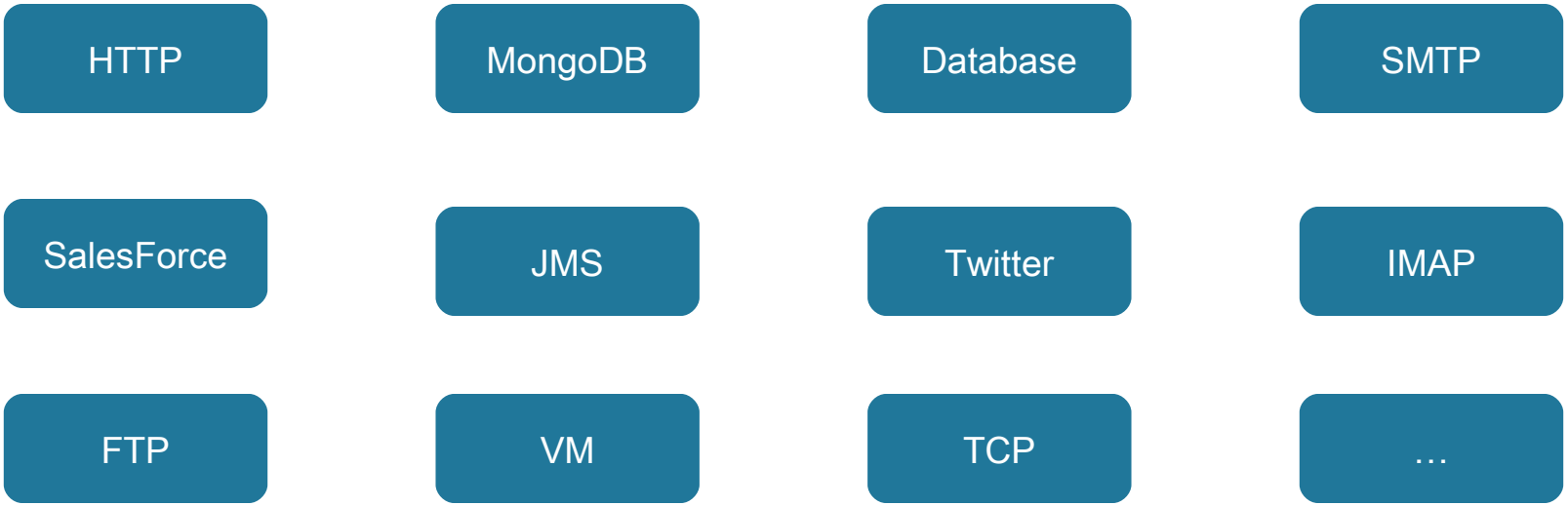
```
15
```

LAB -using property transformer to copy inbound property to outbound property

LAB - Making the Flight Form Browser Friendly Using ParseTemplate

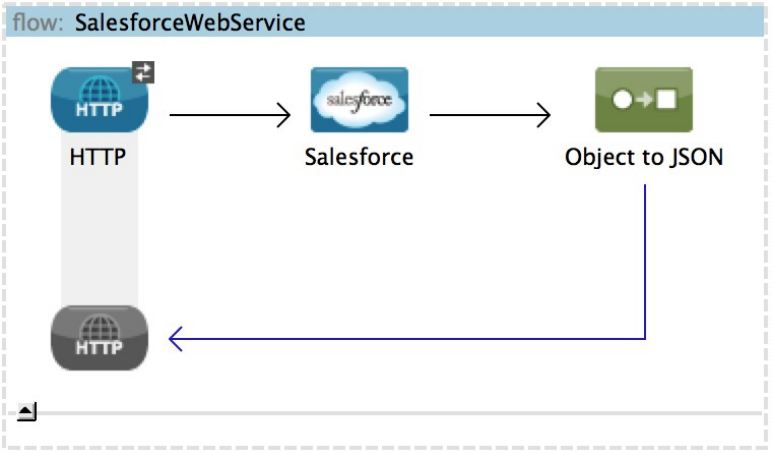
Connector Endpoints

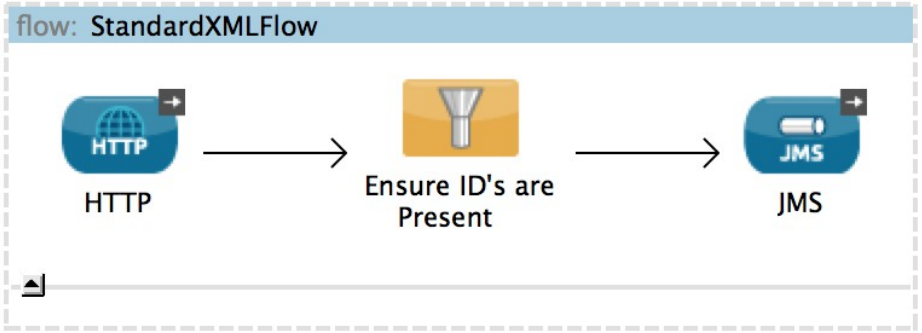
Connector Endpoints



- Allows for the receiving of messages into a flow
  - Inbound Endpoint
- Allows for the sending of messages to another resource or flow
  - Outbound Endpoint

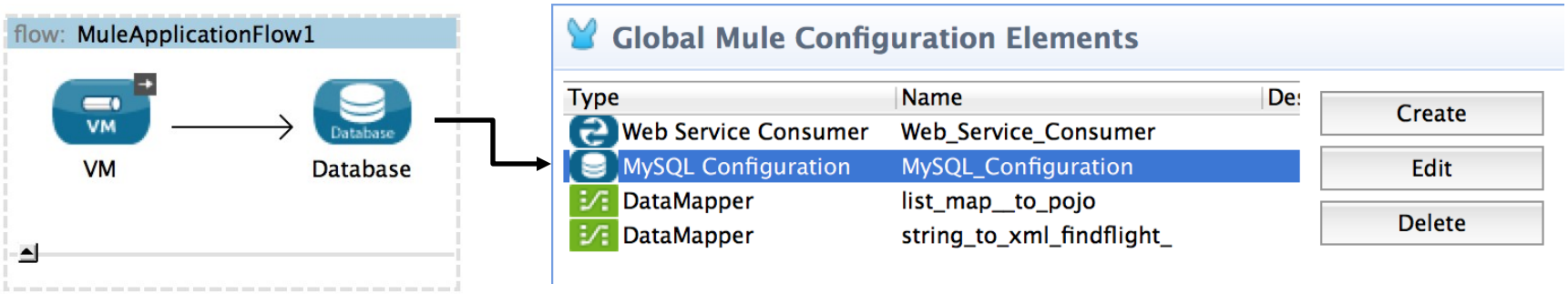
Connector Endpoints





- Ability to interact with SAAS Applications through APIs
- Ability to leverage various transport technologies
- Mule determines inbound or outbound based on the placement of the Connector Endpoints.

Connector Endpoints

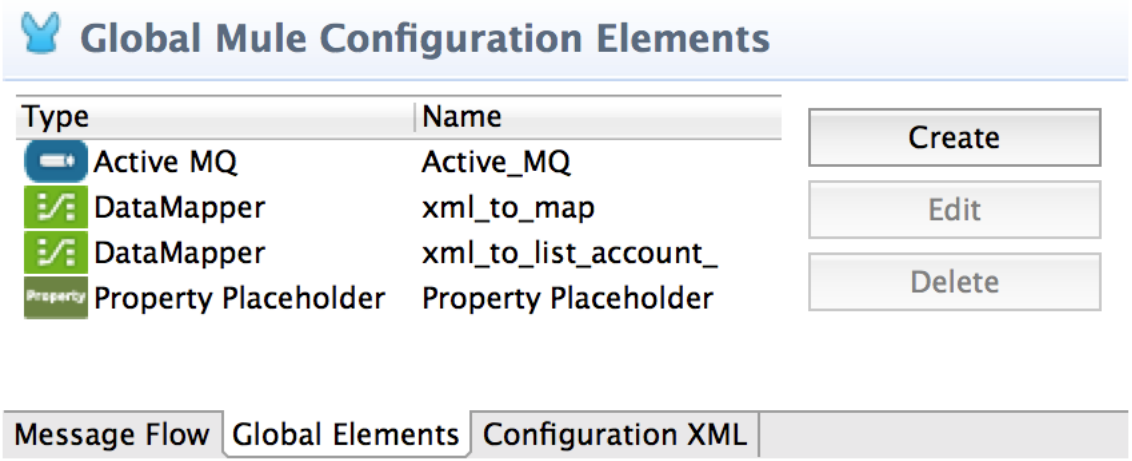


- Connector endpoints often have an associated global element.
- This ‘Global Connector’ encapsulates connection information and other global properties

Integrating with Database

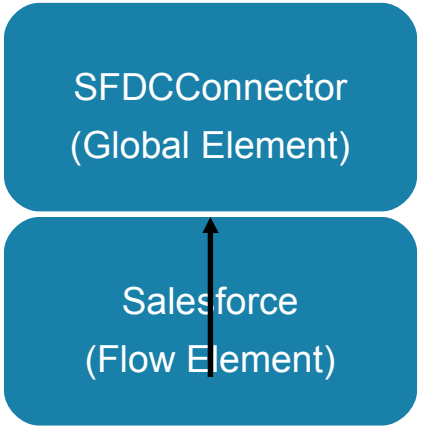
Global Elements

Global Elements



- Accessible under the 'Global Elements' tab
- Configure once, reference later
- Promoting Reusability
- Fostering flexibility

Global Connectors

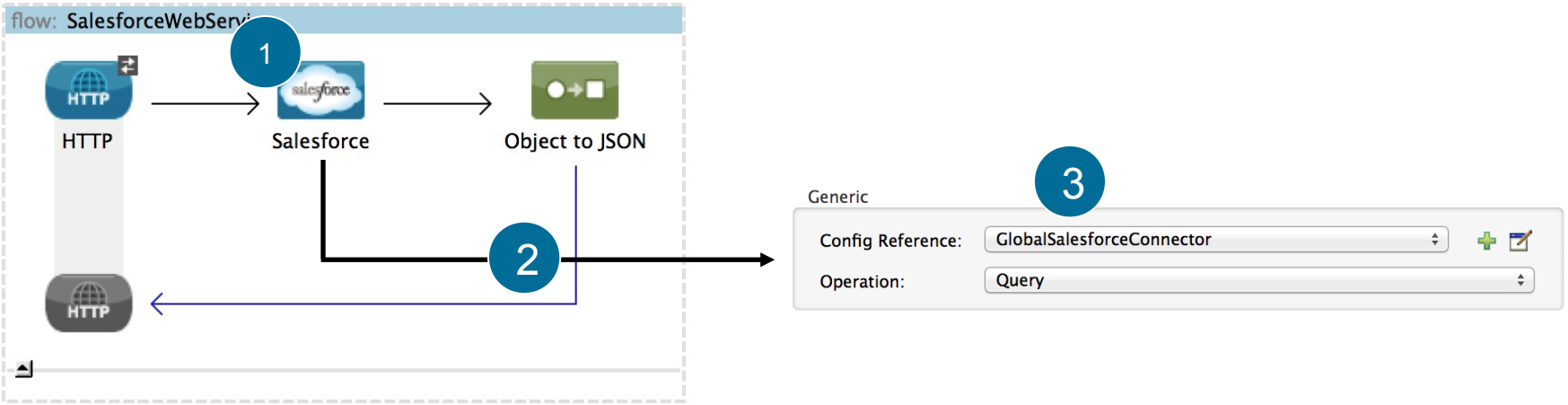


Username, Password, Token, Timeout

Operation and Reference to SFDCConnector

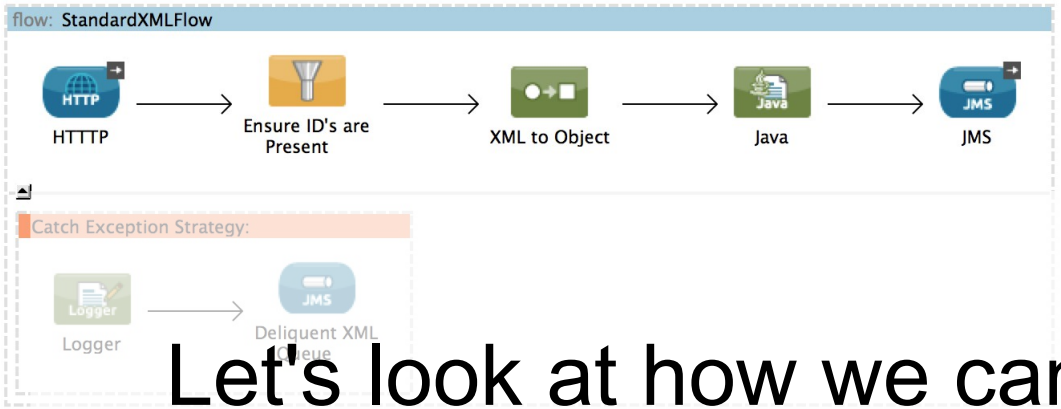
- Common global element is a global connector
- Often encapsulates connection information
- Can hold operations and advanced settings for the local endpoints

Global Connectors Example



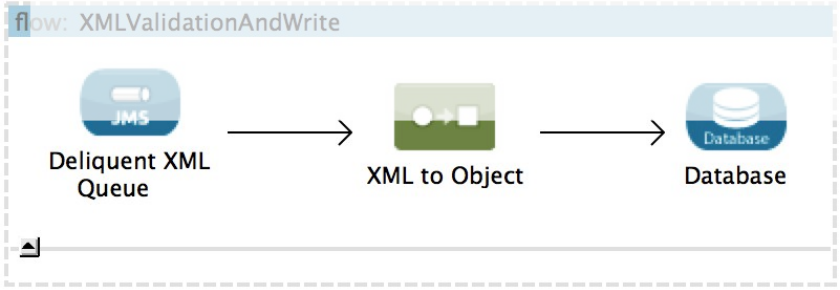

```
3 <sfdc:config name="GlobalSalesforceConnector"
username="traininguser@mulesoft.com" password="pass1" securityToken="ebc234"
doc:name="Salesforce" />

<flow name="SalesforceWebService" doc:name="SalesforceWebService">
        <http:inbound-endpoint exchange-pattern="request-response"
host="localhost" port="8081"      doc:name="HTTP" path="sfaccounts"/>
 1 <sfdc:query con    -ref="GlobalSalesforceConnector" query="<QueryHere>"
doc:name="sfdc"/>  2
     <json:object-to-json-transformer doc:name="Object to JSON/>
</flow>
```

Global Elements



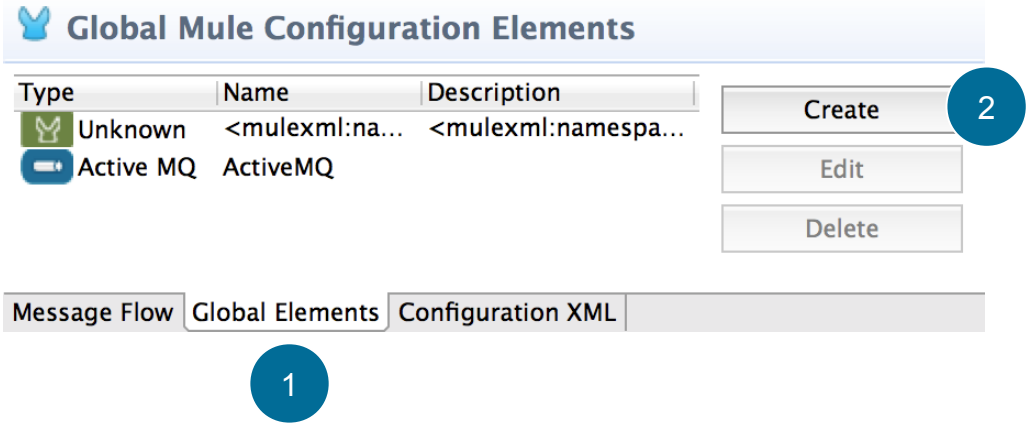
Flow 1 Implementing XML to Object

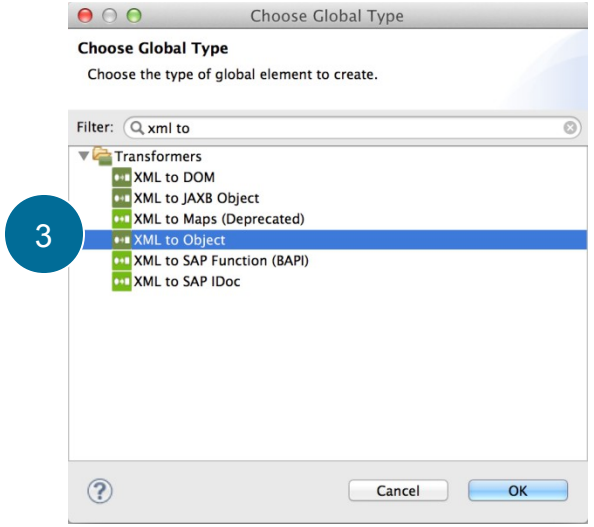
Let's look at how we can configure this transformer just once.



Flow 2 Implementing XML to Object

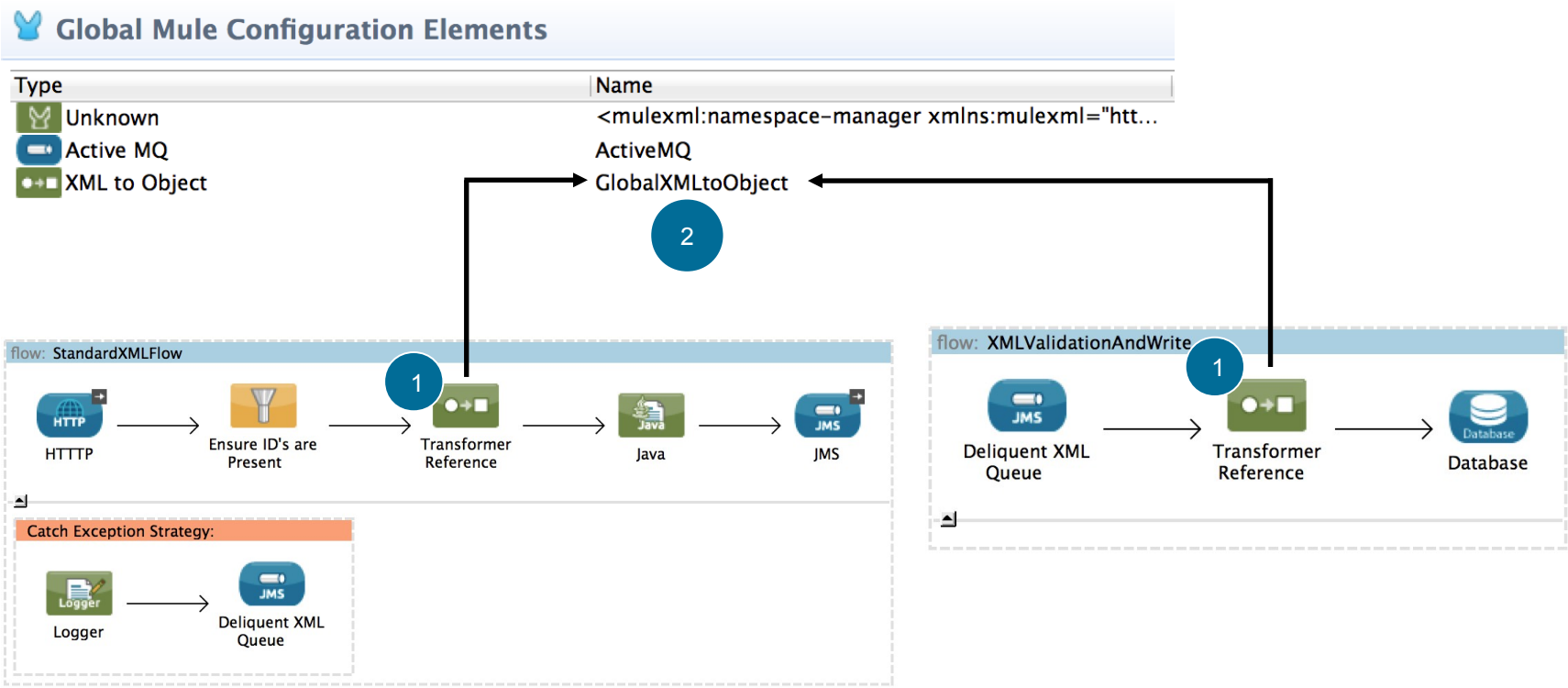
Global Elements





1. Open 'Global Elements' within the canvas
2. Create a new Global Element
3. Create the transformer type

Global Elements



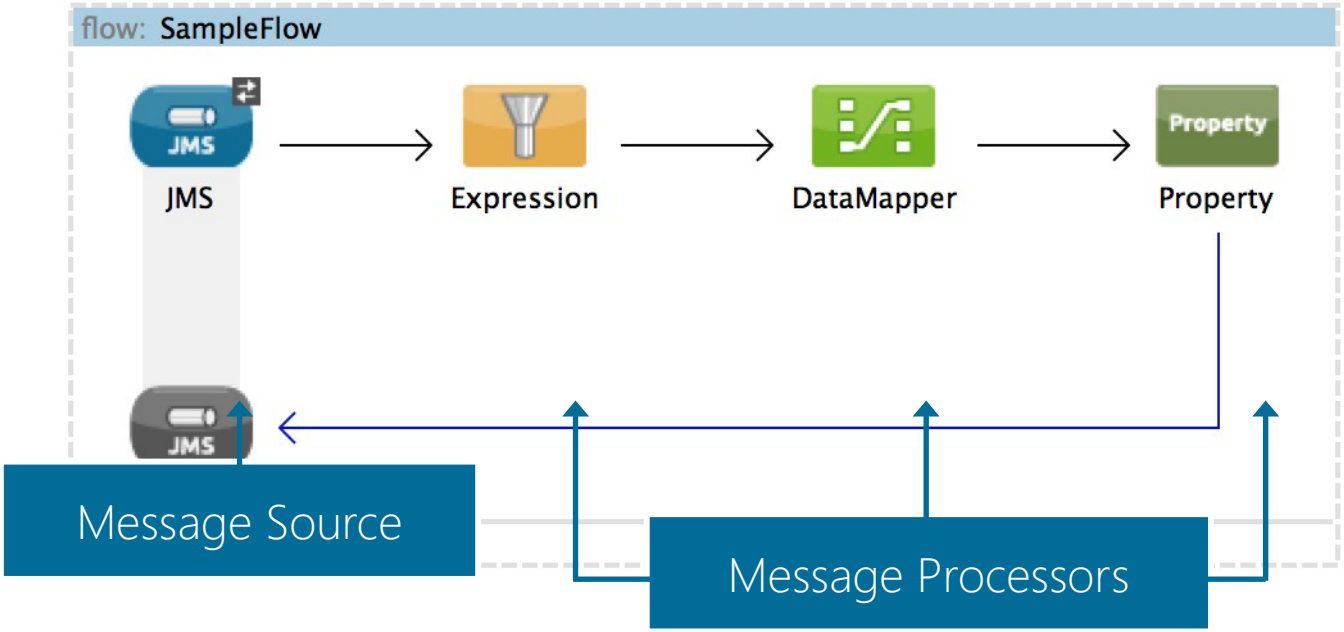
1. Implement 'Transformer References' in the flow
2. Set config-ref attribute to "GlobalXMLtoObject"

Lab – Consuming Soap Webservice and basic Datamapper

Flows, Private Flows and Sub Flows

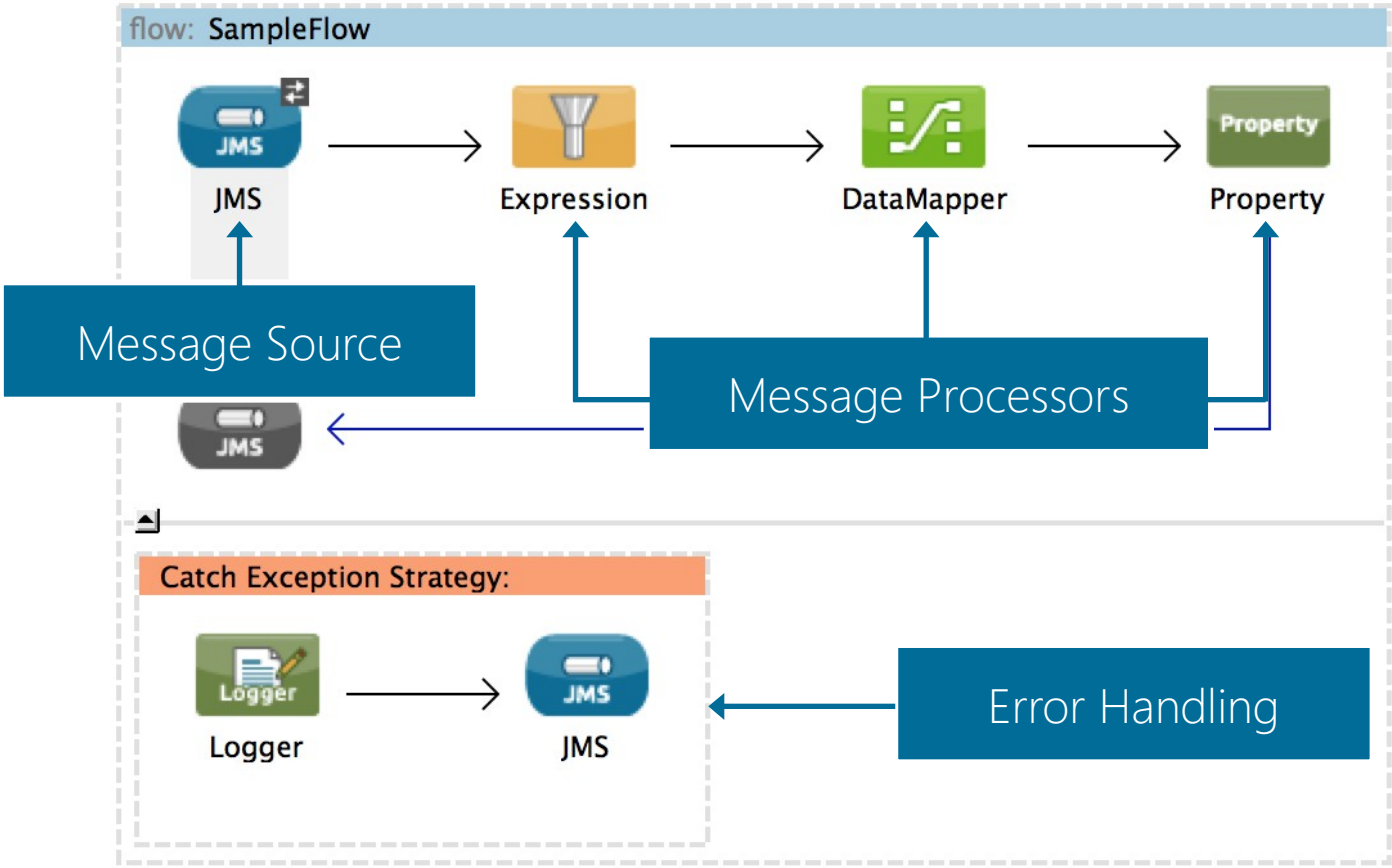
Anatomy of a Flow

MuleSoft




```
<flow name="SampleFlow">
    <jms:inbound-endpoint doc:name="JMS" connector-ref="GlobalJMS"
        queue="queue1" exchange-pattern="request-response" />
    <expression-filter expression="#[message.inboundProperties" />
    <data-mapper:transform doc:name="DataMapper" />
    <set-property name="ProcessedByJMS" value="true"/>
</flow>
```

Anatomy of a Flow



* Each flow operates on its own thread pool

Anatomy of a Flow

```
<flow name="SampleFlow" doc:name="SampleFlow">
  1 <jms:inbound-endpoint doc:name="JMS" connector-ref="GlobalJMS"
        queue="queue1" exchange-pattern="request-response" />
    <expression-filter expression="#[message.inboundProperties]"
        doc:name="Expression "/>
  2 <data-mapper:transform doc:name="DataMapper" />
    <set-property name="ProcessedByJMS" value="true" doc:name="Property"
        propertyName="escalated"/>
  3 <catch-exception-strategy doc:name="Catch Exception Strategy">
        <logger level="INFO" doc:name="Logger"/>
        <jms:outbound-endpoint queue="dlq" connector-ref="glConn"
           doc:name="JMS"/>
    </catch-exception-strategy>
</flow>
```

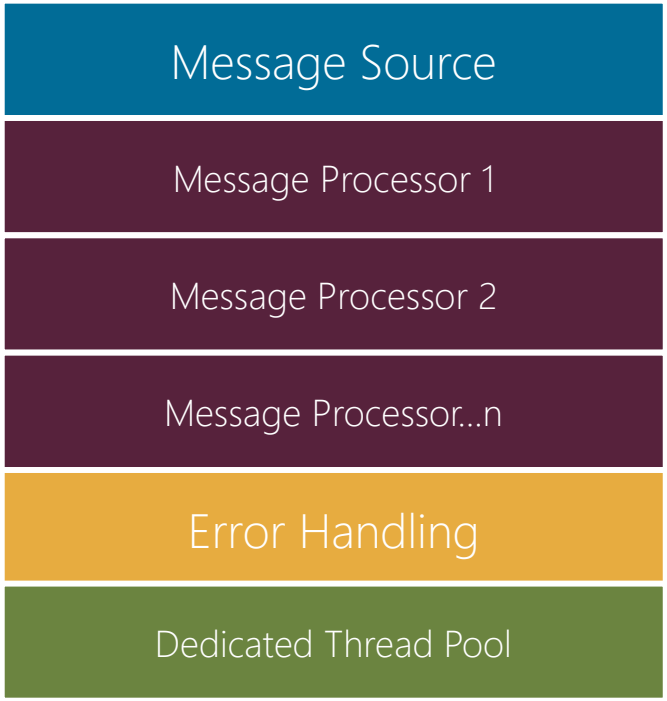
1. Message Source (JMS waiting on Queue)
2. Message Processors (Filter, DataMapper, Property Transformer)
3. Exception Strategy (Optional, but available)

Anatomy of a Flow

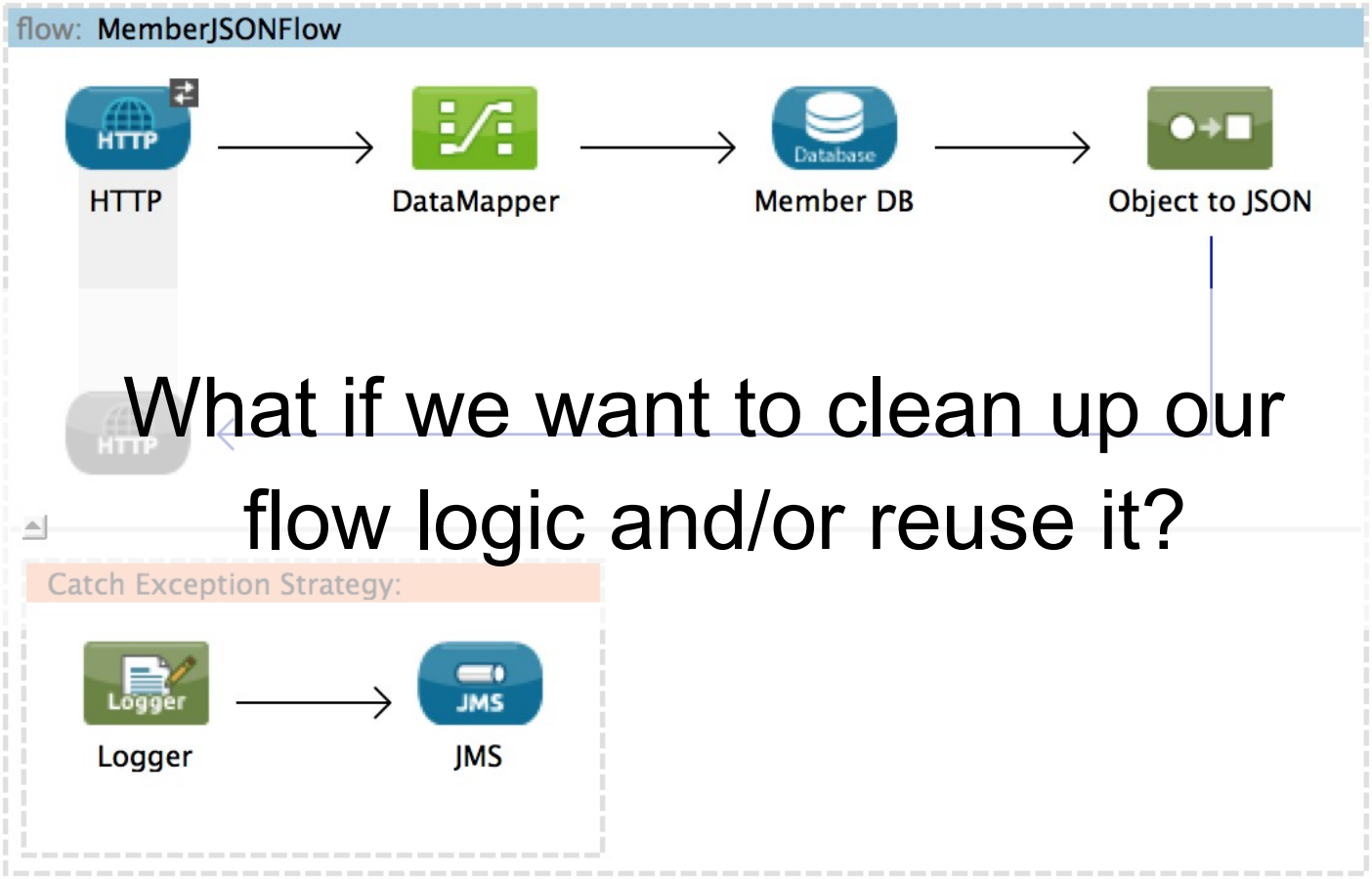
MuleSoft

Recap:

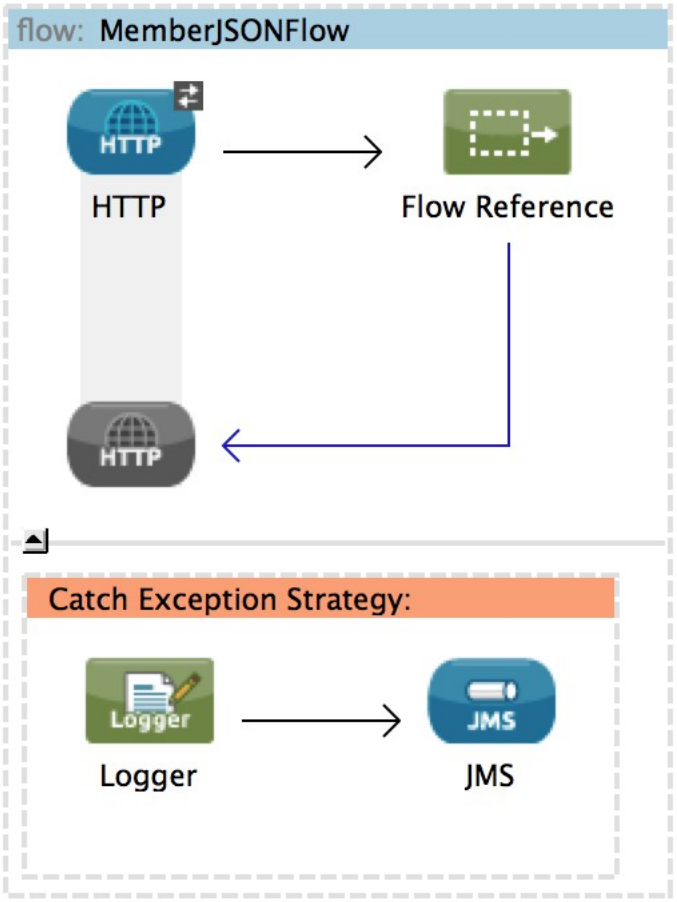
- Message Source (1)
- Message Processors (n)
- Error Handling Strategy (1)
- Dedicated thread pool
- Can be called by:
  - Message Source
  - Flow Reference



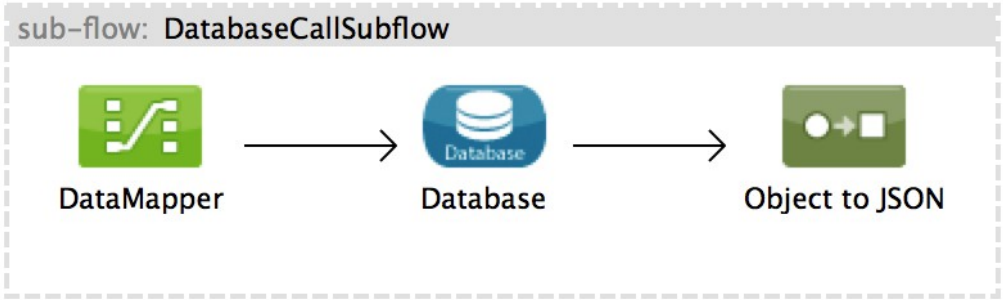
Subflows and Private Flows



Subflows and Private Flows

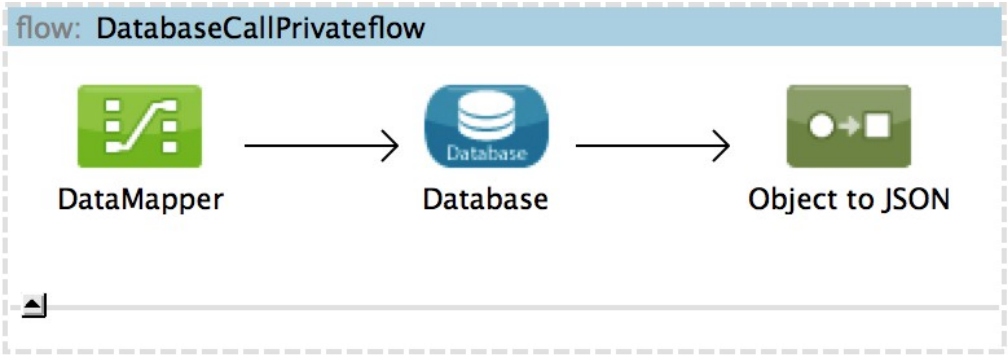


Subflow

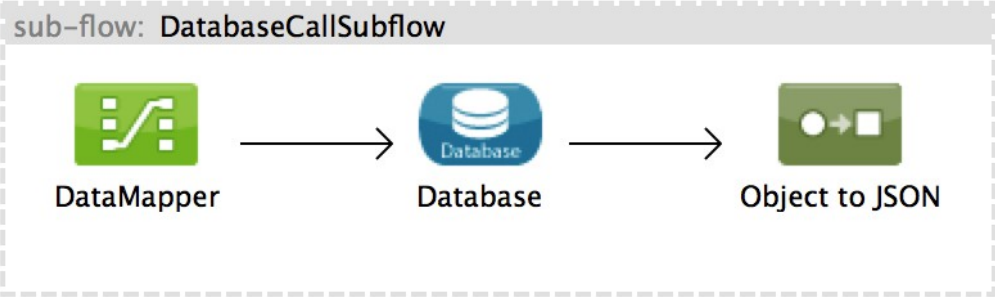


- or -

Private Flow

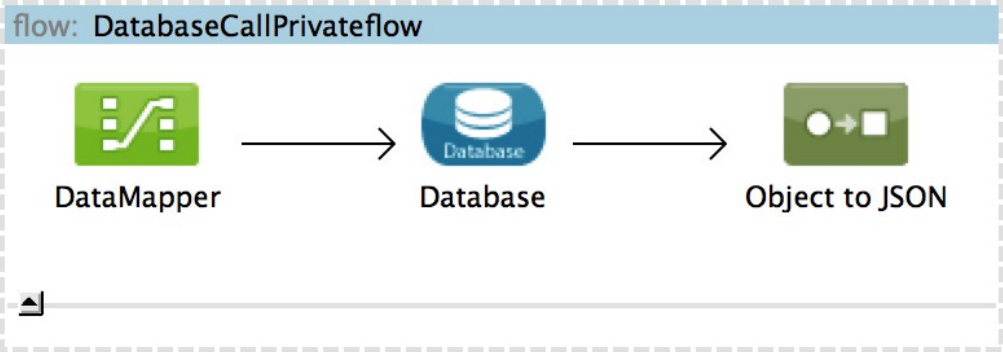


Subflow



- Does ***not*** have an **inbound endpoint** at the beginning of the flow
- Does ***not*** have its **own thread pool**
- **Callable** only by using a **flow reference**
- Cannot have their own **error handling**; errors caught in parent flow

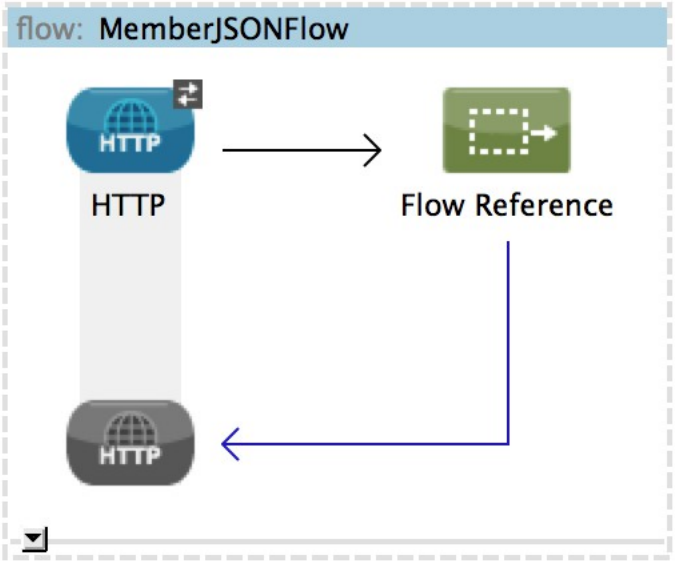
Private Flow

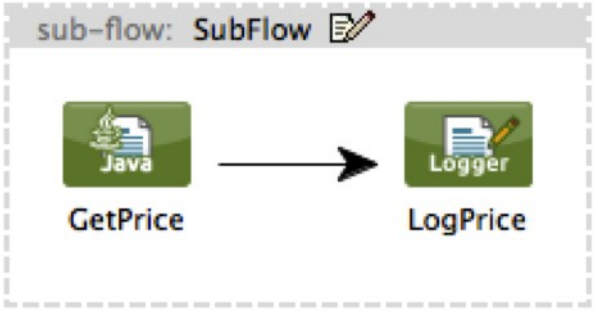


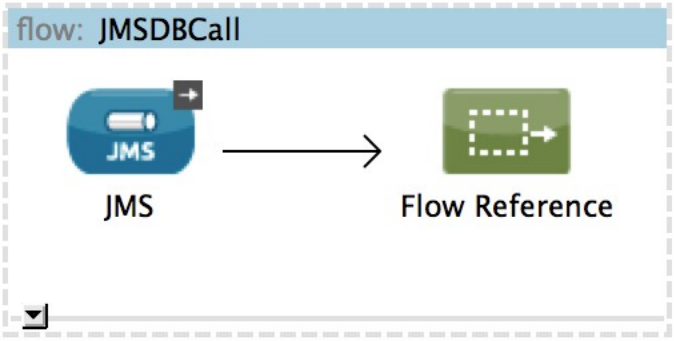
- Does not have an inbound endpoint at the beginning of the flow
- Callable only by using a flow reference
- Operates on its own thread
- Can contain its own error handling

Flow Reference

Flow Reference

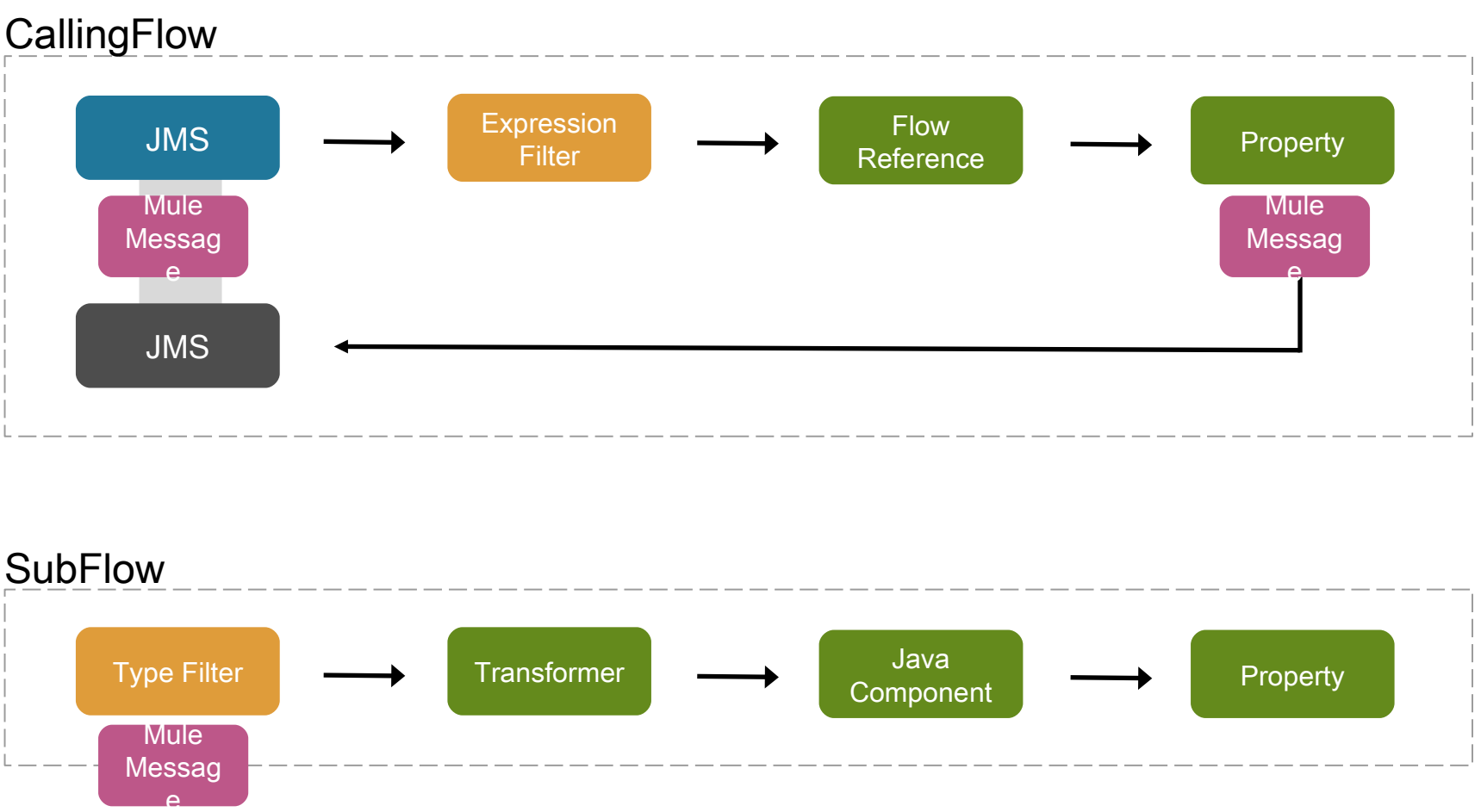




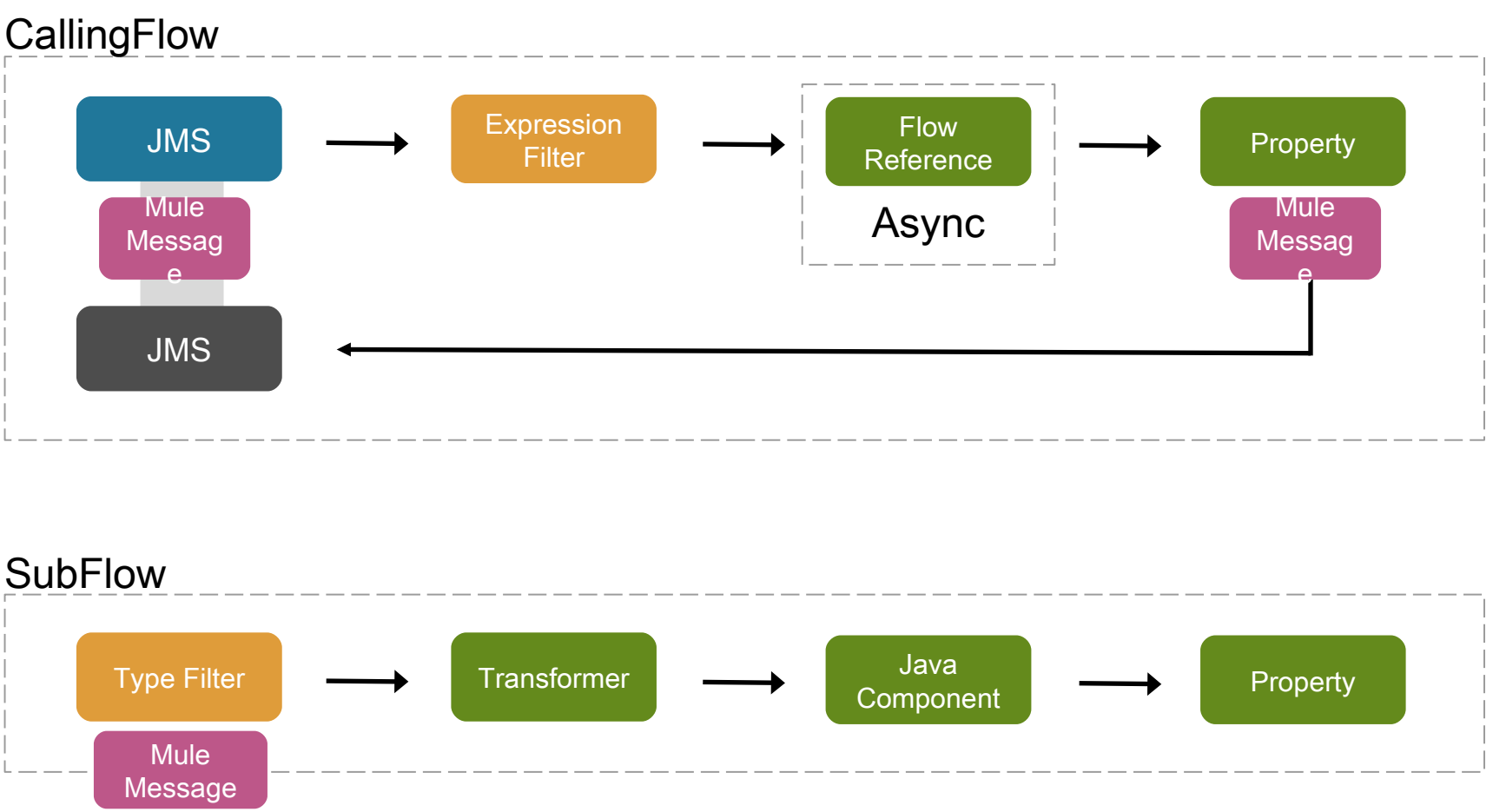


- Calls subflows or private flows
- Can be called by multiple flows
  - Within the same application
- Mule Messages are sent in their exact state
- Only flows with inbound endpoints are accessible across applications

Flow Reference



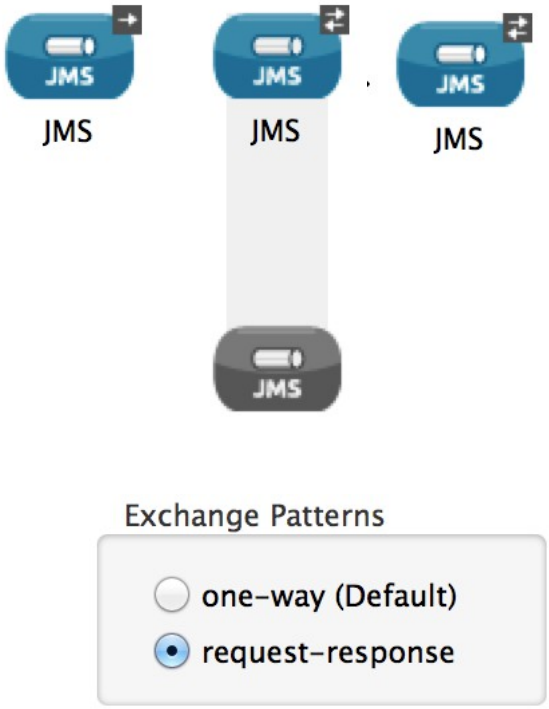
Asynchronous Flow Reference



Exchange Patterns

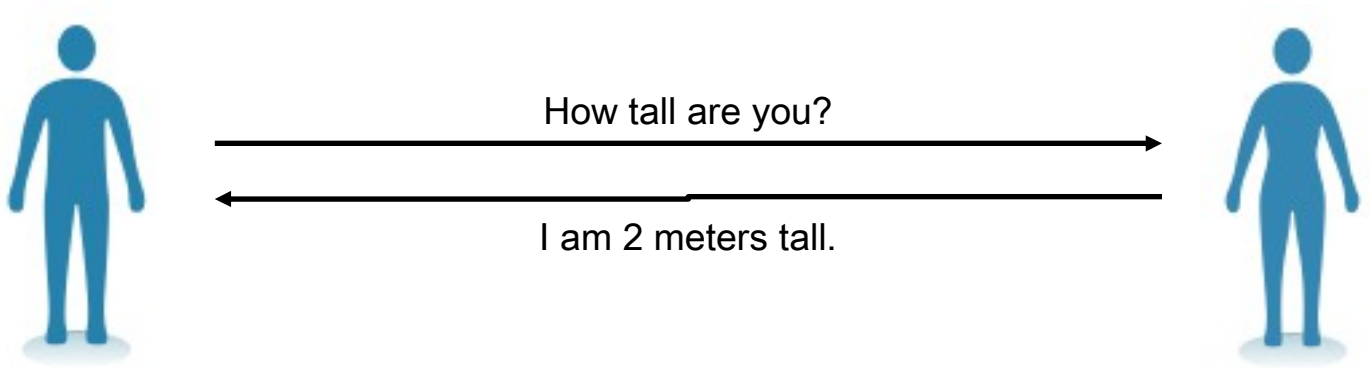
Exchange Patterns

- Request Response and One-Way
- Choice for Connector Endpoints
  - Some support both patterns
- Set by exchange-pattern attribute



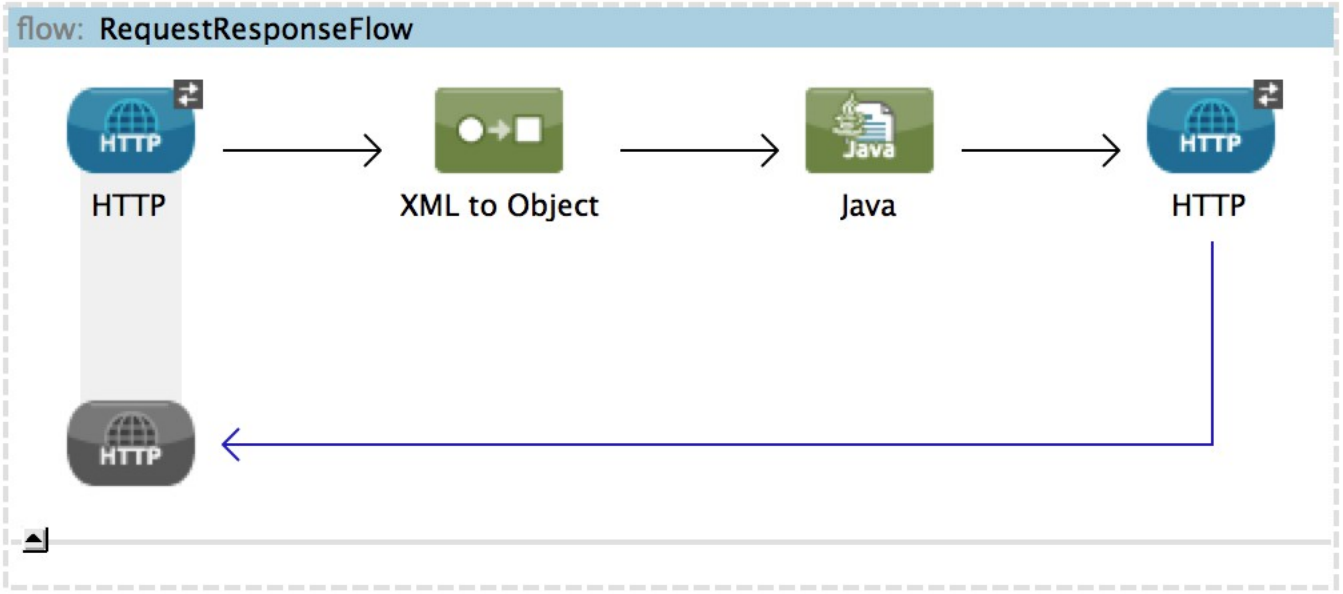

```
<jms:inbound-endpoint exchange-pattern="request-response"  doc:name="JMS"/>
```

Request Response



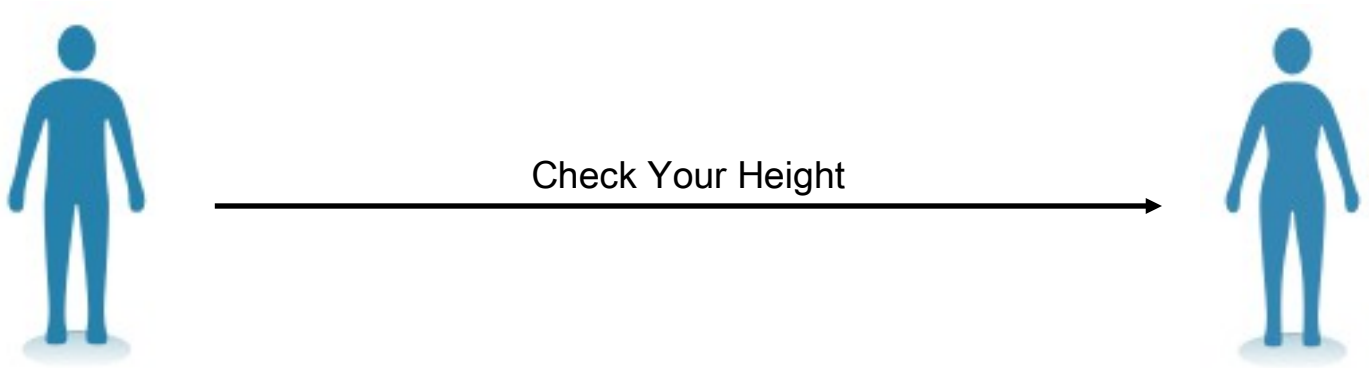
- Sends signal (request)
- Blocks (wait)
- Receives signal (response)

Studio Representation



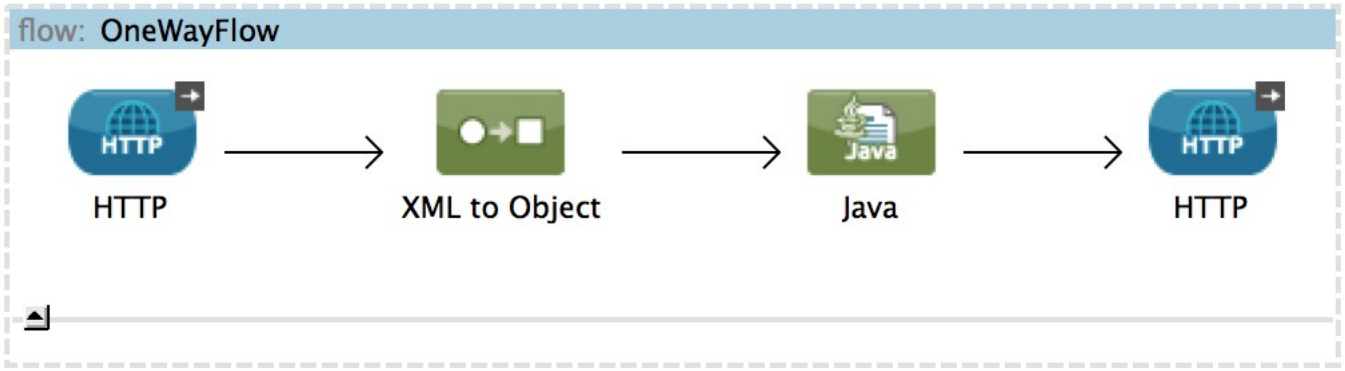
- Connector endpoints show ⇄ arrows for request-response
- Inbound Endpoints (message source) shows a return arrow coming back into the Connector Endpoint

One Way



- Sends signal (request)
- Continues processing (no wait)

Studio Representation

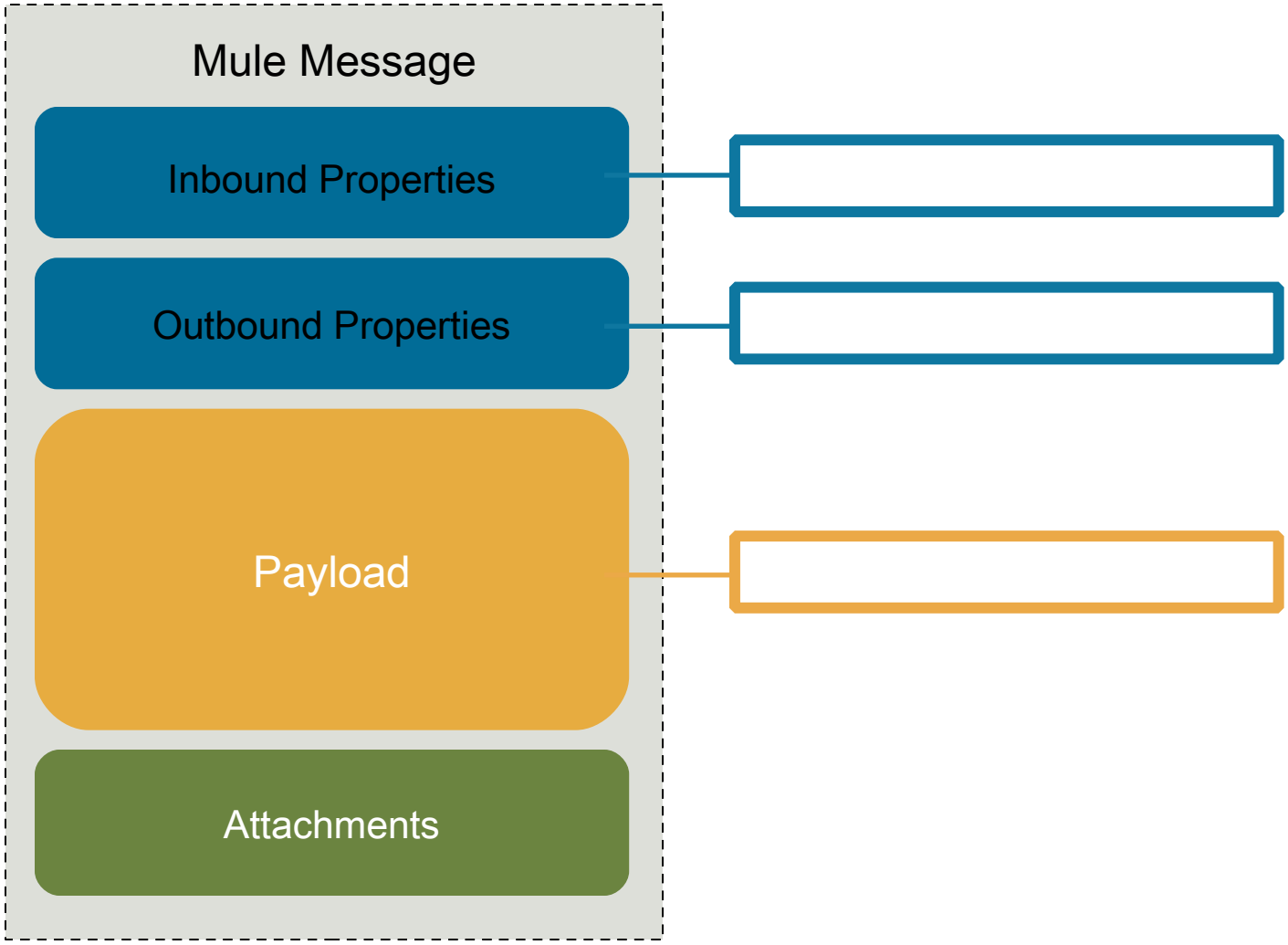


- Connector endpoints show arrow for one-way
- Does not include return arrow for inbound

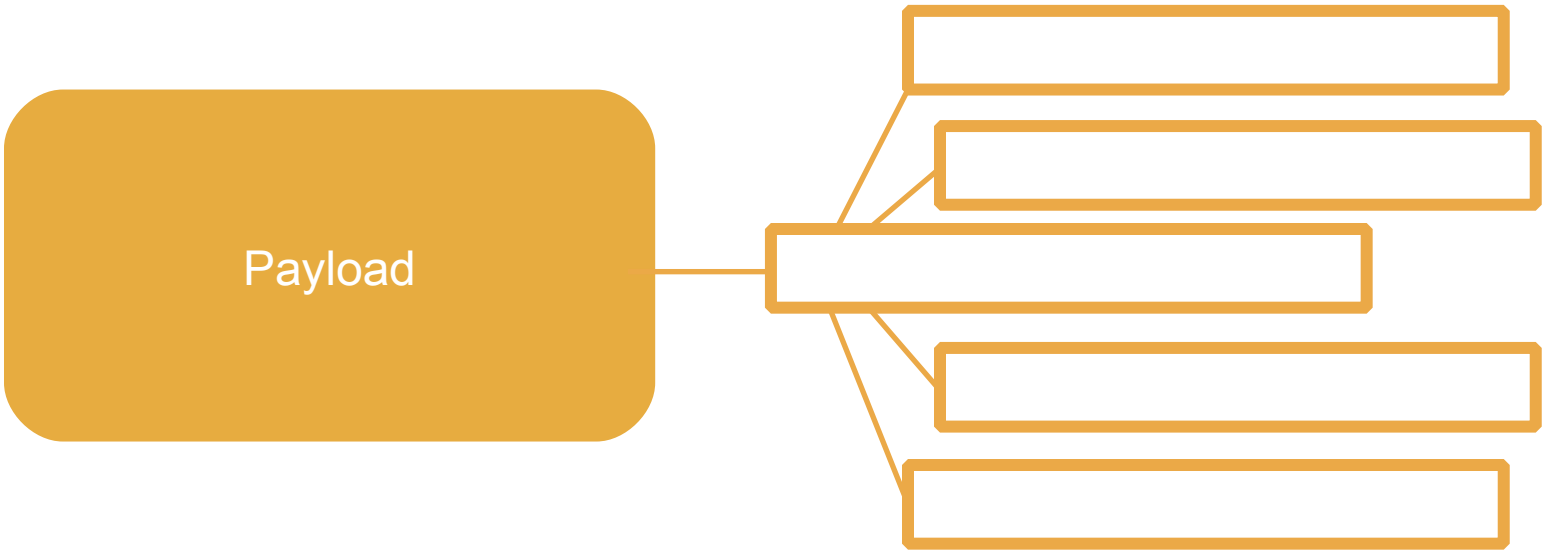
Lab – Integrating with Rest Service

Transforming Data

The Mule Message



The Mule Message



- Payloads are Java Objects
- This means, a Payload can be any Java Type
- The focus of this module will be how we change between types and prepare a message for delivery / future processing

The Mule Message

- Goal:
  - Determine desired type
  - Determine our input type(s)
  - Implement the correct transformer

Core Transformers

Common Transformation Attributes

MuleSoft

returnClass

- Optionally a specify return type
- Useful for more specific types than the default
  - Example: an extension class

ignoreBadInput

- Turns off the processing of messages with bad input
- Message is returned as it was before engaging the transformer

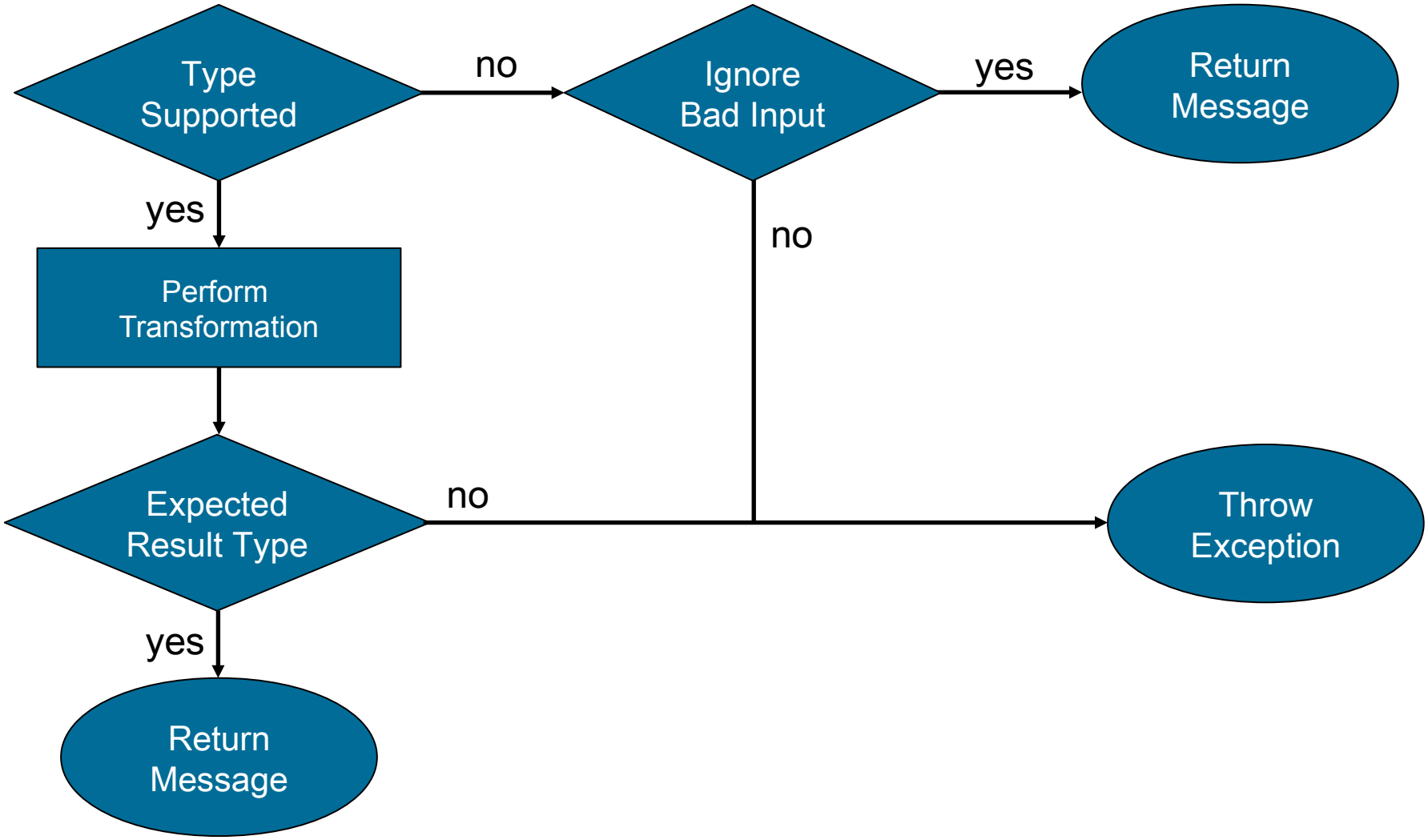
encoding

- Specify encoding

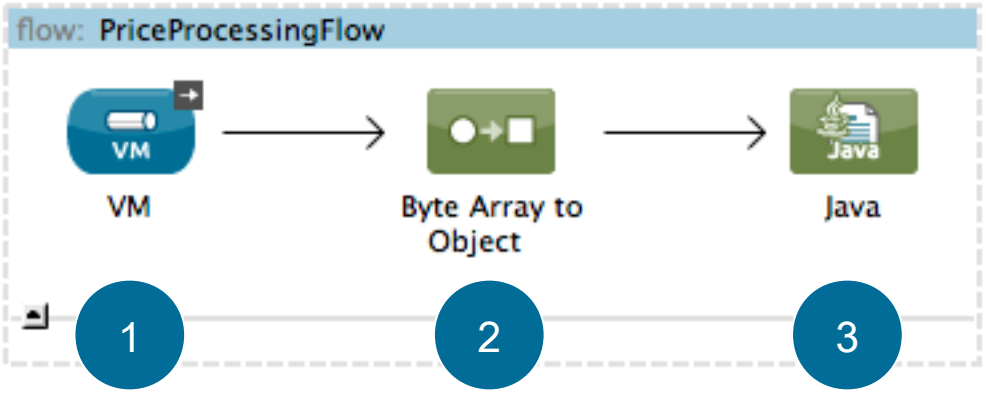
mimeType

- Explicitly set internet media type
  - text/html, text/csv, text/xml

Transformer Flow



Transformer Example

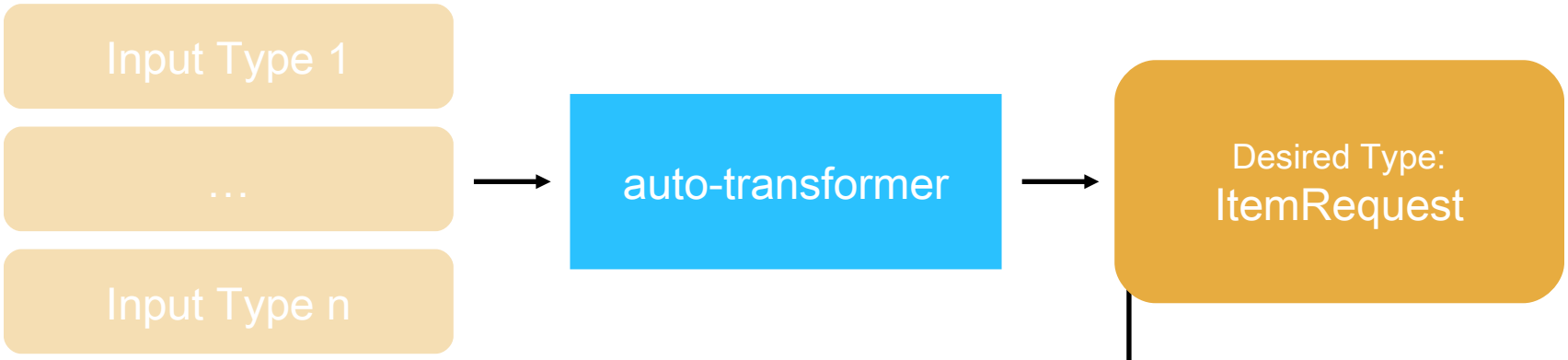


1. Mule Message comes in on a VM Endpoint
2. Core transformer turns the byte array into an object (based on the return class attribute via deserialization
3. Message is passed to a Java Component implementing business logic

Multiple Input Types

- There exists multiple ways of handling > 1 input type
  - Routers, Filters, etc
- Limiting ourselves to just one transformer, how could we dynamically transform messages coming to a desired type?

Automatic Transformer




```
<auto-transformer
returnClass="com.mulesoft.training.ItemRequest" />
```

- Searches for a transformer per the returnClass
- Transformers implementing *com.mule.api.transformer.DiscoverableTransformer* are searched

XML and JSON Transformation

JSON Primer

- **J**ava**S**cript **O**bject **N**otation
  - a lightweight data-interchange format
- Human-Readable Syntax
- Supports
  1. Collections
  2. Maps

1

```
[
    element1, element2, element3, ...
]
```

2

```
{
        "name1" : "value",
        "name2" : "value"
}
```

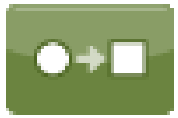
JSON Primer

- Supports the combining:
  1. Collections
     - employees[0]
     - employees[1]
  2. Maps
- Supporting more complex data

```
{
  "employees": [ 1
      {
        "firstName":"John",
        "lastName":"Doe",  2
        "age": 30
      },
      {                    1
        "firstName":"Anna",
        "lastName":"Smith", 2
        "age":25
      }
  ]
}
```

JSON transformers

- **Object** to **JSON**
  - receives an object and returns JSON syntax
  - uses reflection (no configuration necessary)



- **JSON** to **Object**
  - populates a Java object from a JSON structure
  - requires at least the name of the class

JSON Example

MuleSoft

Input:

```
{   "orderId":"ABDj325sfg",
    "item":"Dog Battery",
    "requestor":"Eno",
    "price":100.00
}
```

Output:

```
{   "orderId":"ABDj325sfg",
    "item":"Dog Battery",
    "requestor":"Eno",
    "price":105.00
}
```

1. JSON input enters with an order request
2. Order business logic exists in a java component
3. After processing, a response comes back with the tax included

JSON Example

MuleSoft

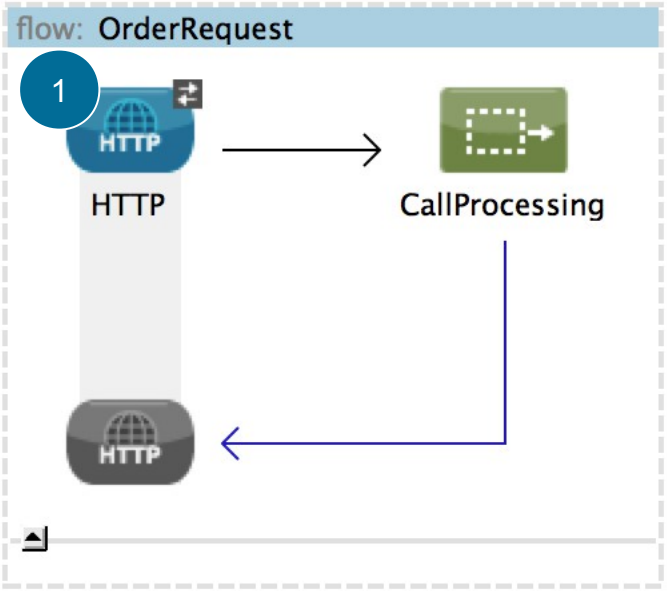
Java Class:

```
package com.mulesoft.training;

public class Order {
    String orderId;
    String item;
    String requestor;
    double price;

    /* Getters and Setters */
}
```

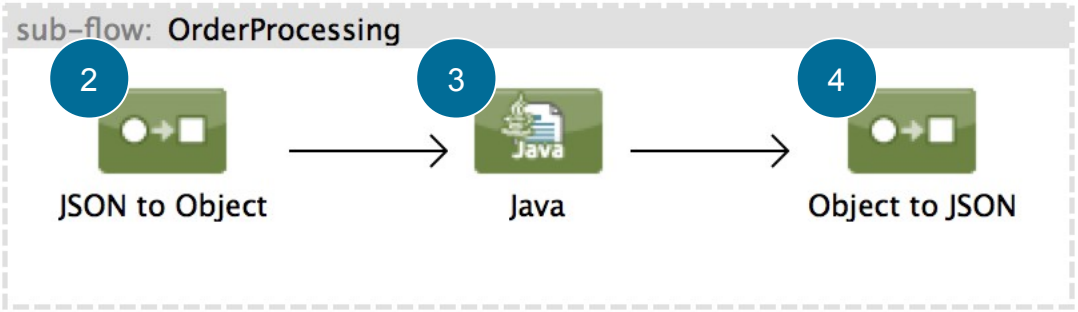
1. Java object representation of an order
2. Can be read by business logic
3. Fields match that of JSON request.

Mule uses the Jackson framework which has powerful mapping annotations

JSON Example



1. Request enters
2. JSON is transformed to Order
3. Business logic is executed
4. Processed order sent back as JSON
   - Example of 'Round-Trip' Transformation

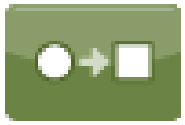


JSON Example

```
<flow name="OrderRequest" doc:name="OrderRequest">
    <http:inbound-endpoint exchange-pattern="request-response"
address="http://localhost:8084" doc:name="HTTP"/>
    <flow-ref name="OrderProcessing" doc:name="CallProcessing"/>
</flow>
<sub-flow name="OrderProcessing" doc:name="OrderProcessing">
1 <json:json-to-object-transformer
returnClass="com.mulesoft.training.Order" doc:name="JSON to Object"/>
    <component class="com.mulesoft.training.OrderProcessor"
doc:name="Java"/>
2 <json:object-to-json-transformer doc:name="Object to JSON"/>
</sub-flow>
```

1. JSON is transformed to Order
   - returnClass specified
2. Order transformed back to JSON

XML Transformation



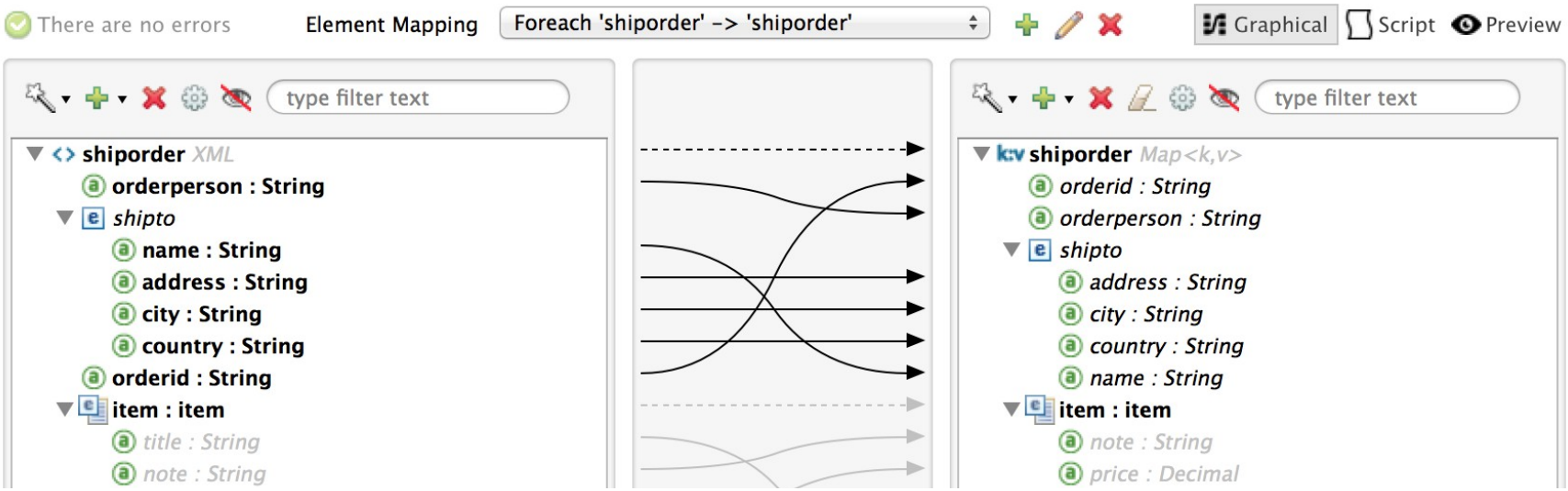
- **XML** to **JAXB**
  - Uses the JAXB binding framework to serialize to xml or in reverse to an object
  - Counterpart exists at **JAXB** to **XML**

- **XML** to **Object**
  - Utilizes Xstream to convert data to and from XML and Java Objects
  - Counterpart exists as **Object** to **XML**

DataMapper

DataMapper

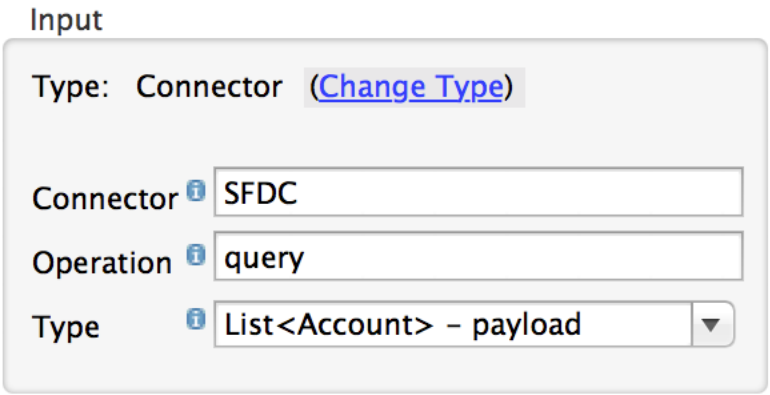
- Complex data mapping using drag a and drop interface
- Scripting support (MEL, XPath, and CTL2)
- Live, design-time previews for transformation results
- DataSense support for automatic metadata discovery



DataMapper Data Formats

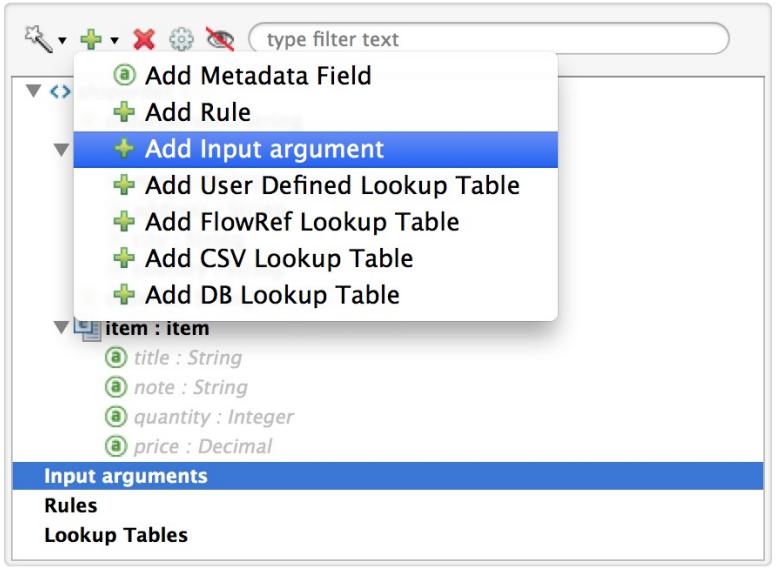
- Support for
  - XML and JSON
  - CSVs, MS Excel, Fixed Width
  - Pojo, Map<k,v>
  - Connector

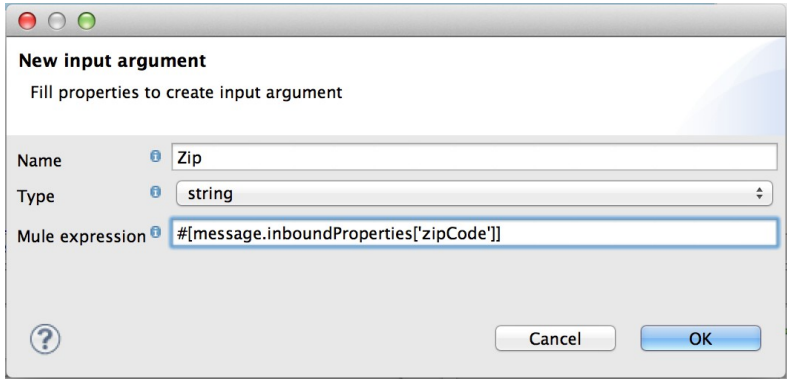
- DataSense enabled
  - Connector attributes are picked up automatically
  - When DataSense is enabled



Input Arguments

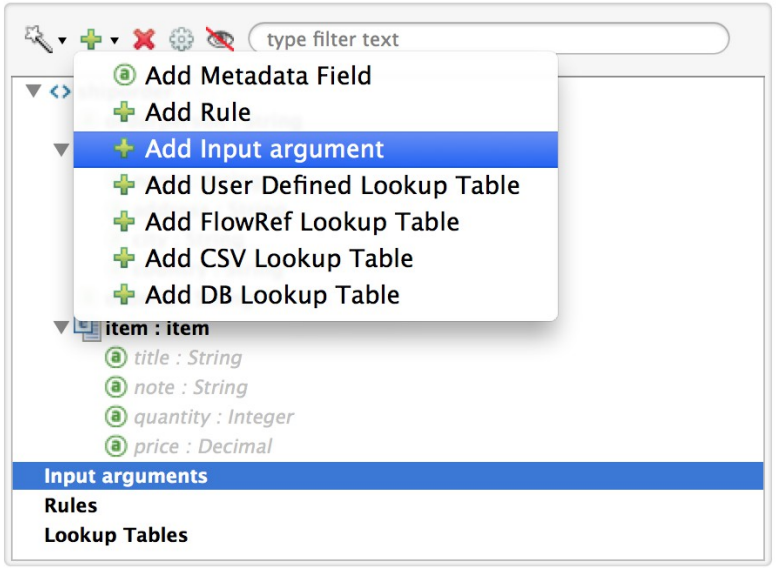
- Allows us to evaluate mule expressions and map those to metadata or output arguments
- Great for accessing properties and variables during mapping

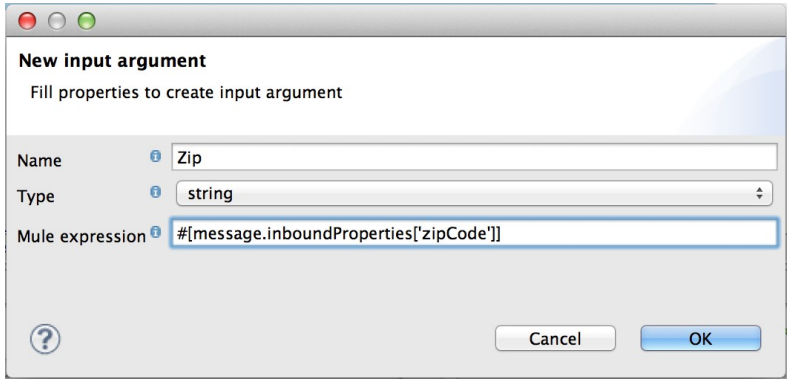




Lookup Tables

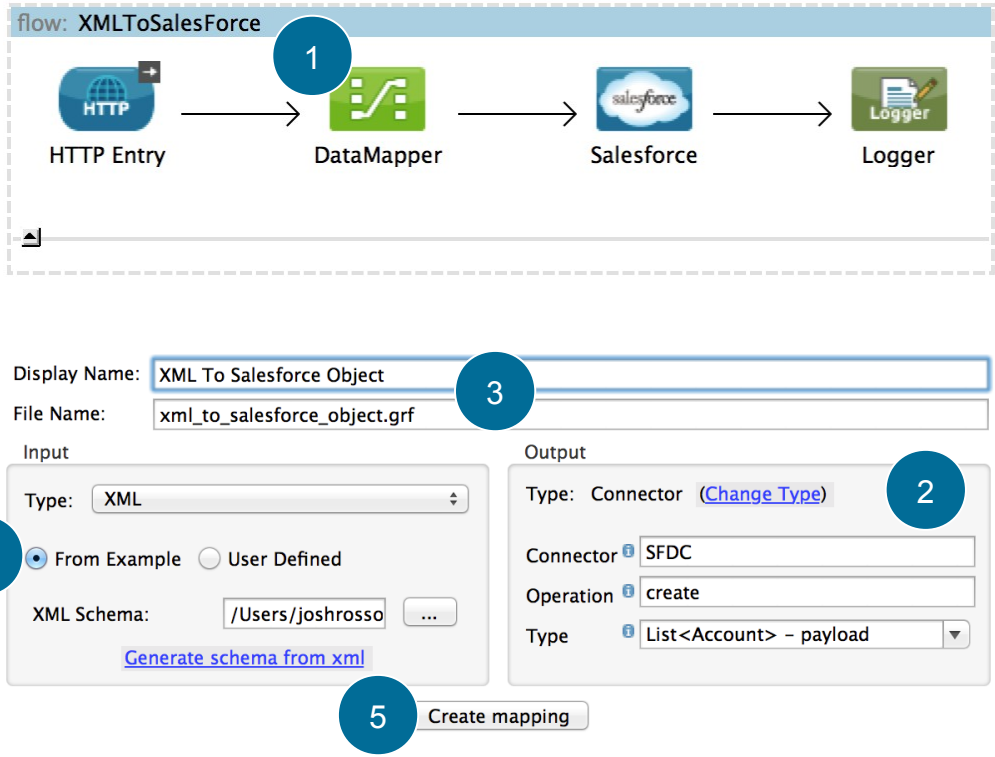
- Lookup tables can tap into data from:
  - Flows
  - CSVs
  - DBs





DataMapper Implementation

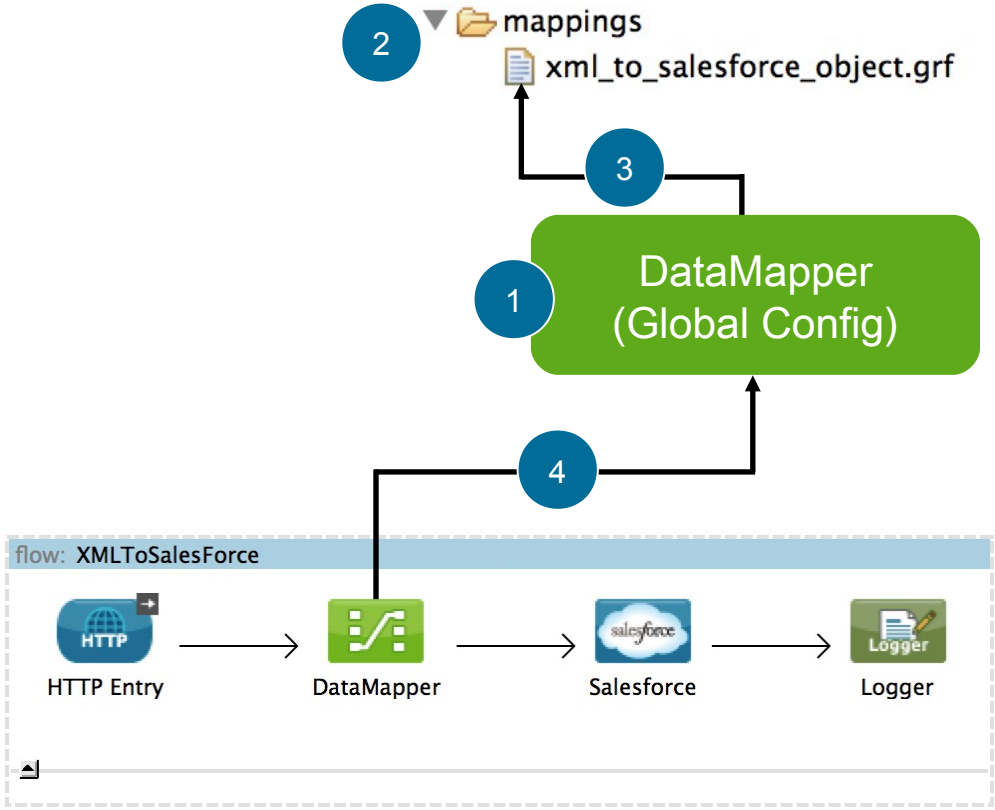
1. DataMapper is dragged into a flow
2. If Connector with DataSense exists, type and fields are recognized
3. .grf – Mapping file name specified here
4. Choose to create types 'From Example' or by manual definition
5. Create mapping



DataMapper Implementation

DataMapper Has:

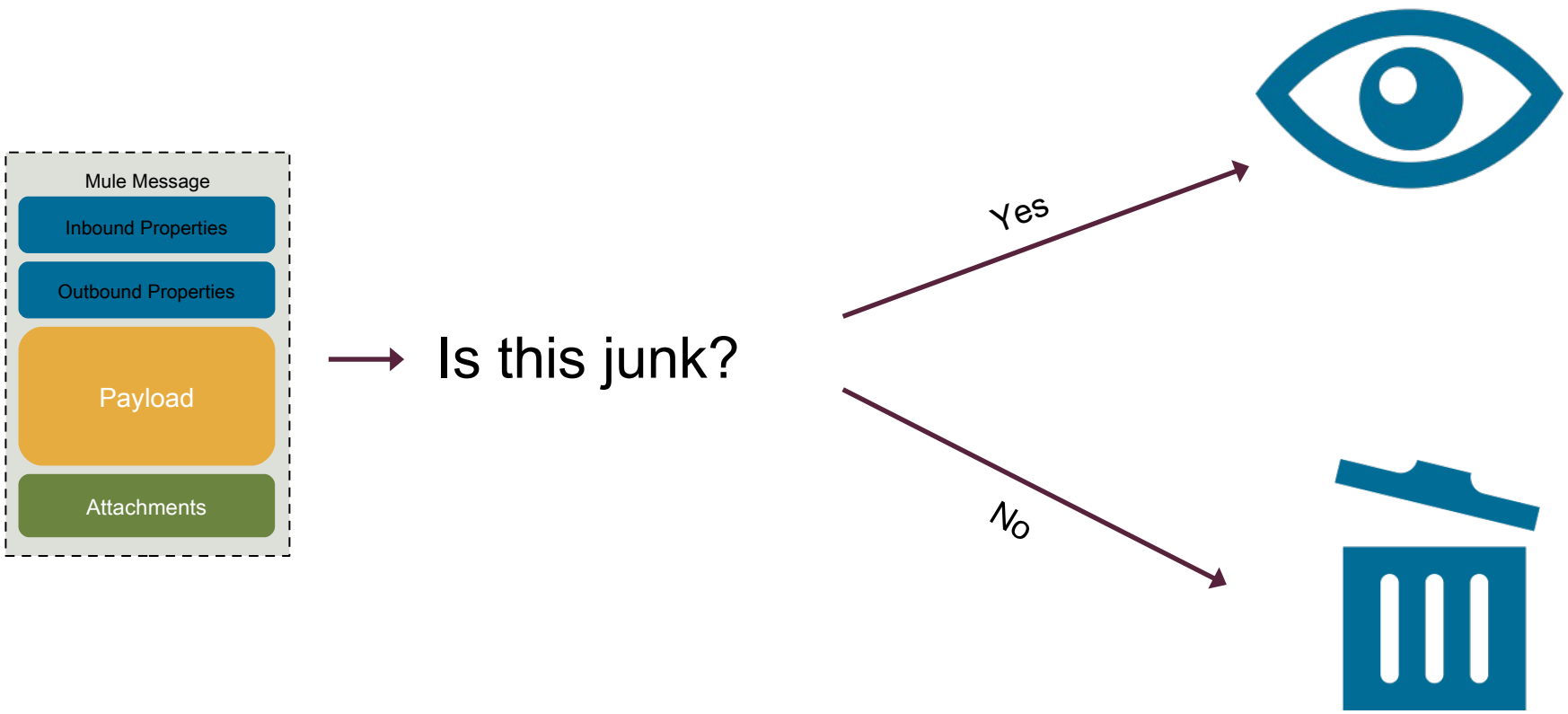
1. Created a Global DataMapper Element
2. Generated a .grf – Mapping file
3. Referenced the .grf from the global DataMapper element
4. Referenced the global DataMapper element from the local DataMapper instance



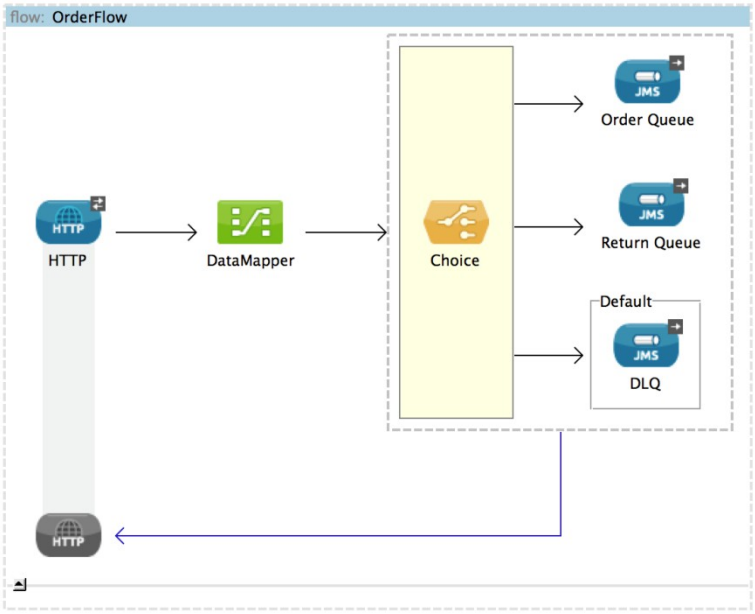
LAB - Transformation to a Canonical Type Using Data Mapper

Routing Messages

Routing Messages



Choice Router



- Sends message to specific MPs based on conditions
- Conditions are evaluated with MEL
- Each path can include multiple message processors

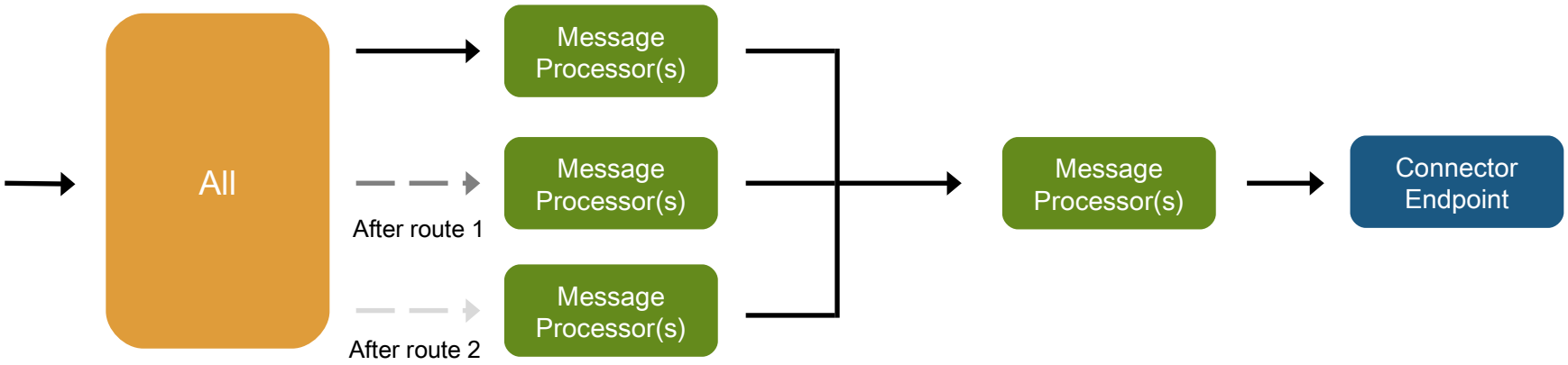
When	Route Message to
#[message.inboundProperties['requestType'] == 'order']	Order Queue
#[message.inboundProperties['requestType'] == 'return']	Return Queue
Default	DLQ

Choice Router

```
<choice doc:name="Choice">
  <when expression="#[message.inboundProperties['requestType'] == 'order']">
      <jms:outbound-endpoint queue="order.processing" doc:name="Order Queue"
connector-ref="Active_MQ" />
  </when>
  <when expression="#[message.inboundProperties['requestType'] == 'return']">
      <jms:outbound-endpoint queue="return.processing" doc:name="Return Queue"
connector-ref="Active_MQ" />
  </when>
  <otherwise>
      <jms:outbound-endpoint queue="dlq.processing" doc:name="DLQ" connector-
ref="Active_MQ" />
   </otherwise>
</choice>
```

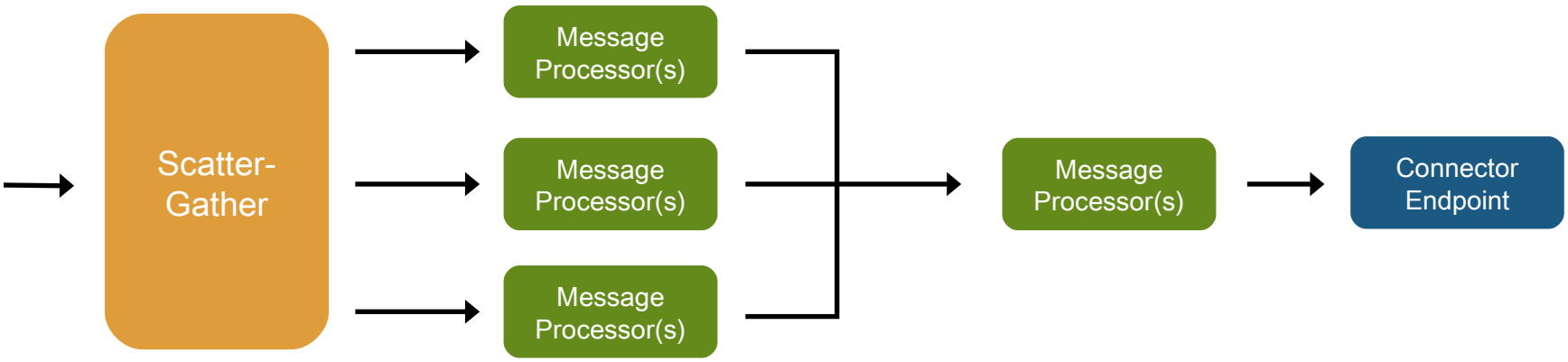
1. Expressions used for conditions to determine which route is chosen
2. Otherwise route is required

All Router



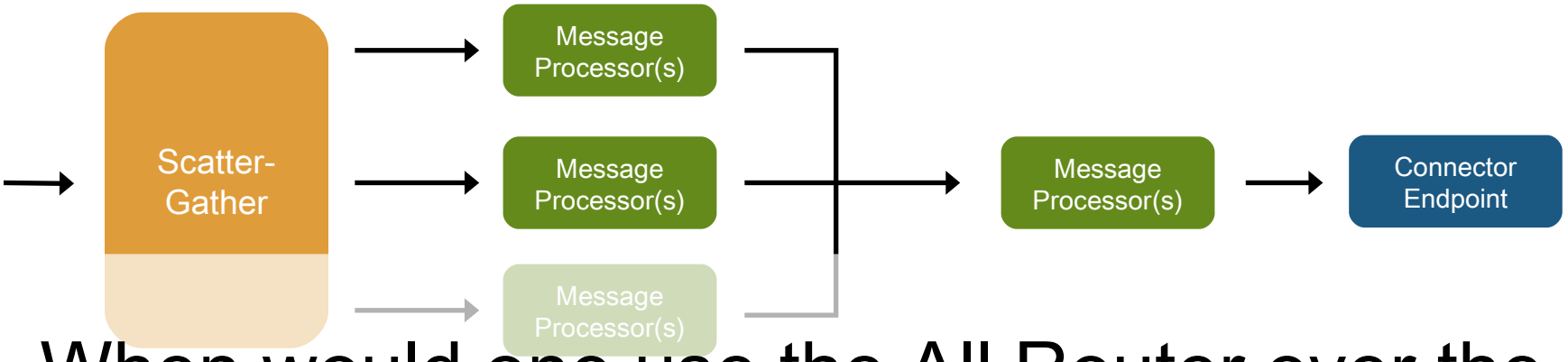
- Creates a copy of the Mule Message for each route
- Synchronously performs a multicast to each route
- Returns each route's resulting message in a MuleMessageCollection

Scatter-Gather

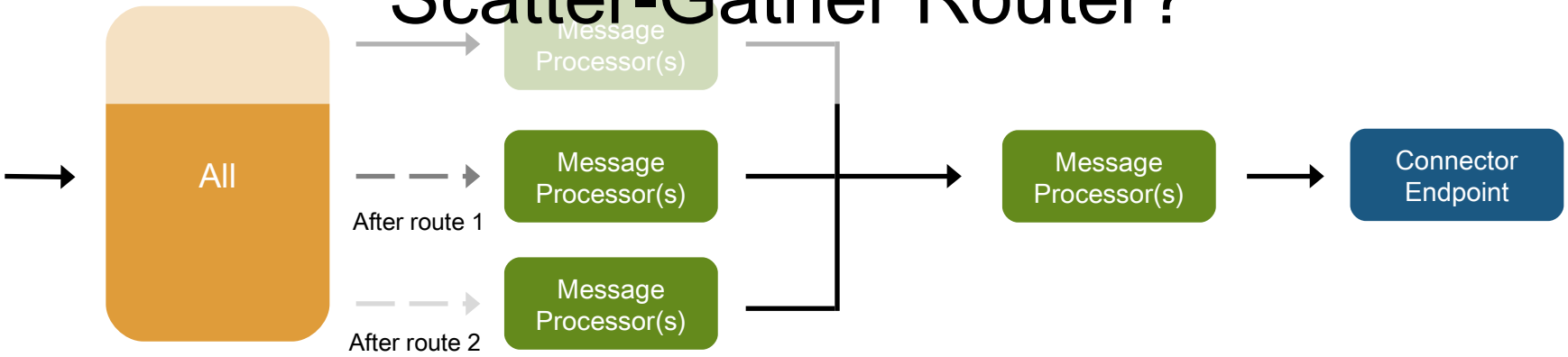


- Creates a copy of the Mule Message for each route
- Performs a multicast to several routes concurrently
- Gathers messages in a MuleMessageCollection
- Ensures all routes are executed, even with exceptions

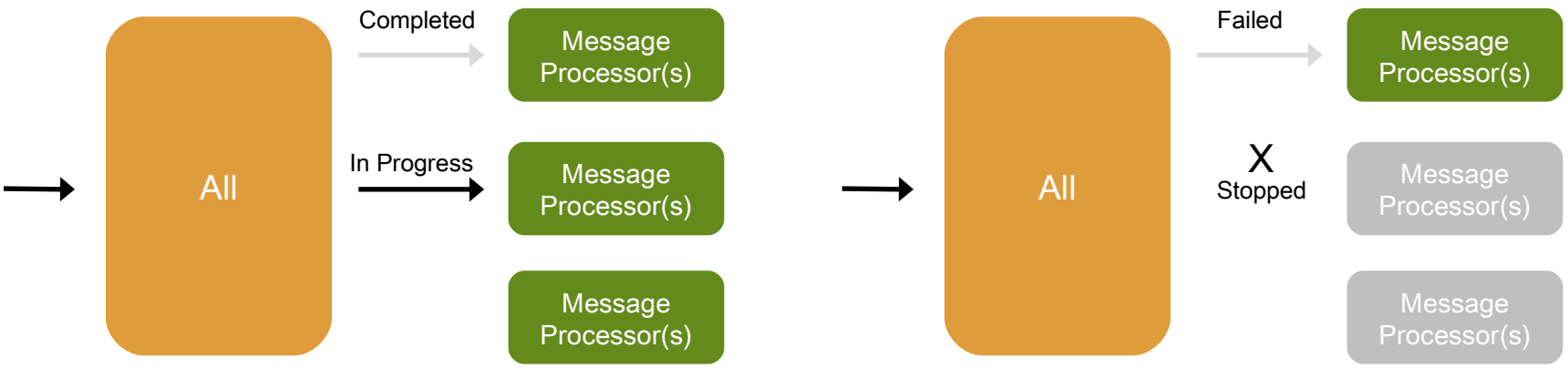
All vs Scatter-Gather



When would one use the All Router over the Scatter-Gather Router?



Choosing the All Router



- When route (n) depends on route (n-1)
- When an exception at route (n) should prevent the message from being sent to route (n+1)
- In other cases, Scatter-Gather is likely more efficient

Additional Routers

First Successful

Round Robin

http://www.mulesoft.org/documentation/display/current/Routing+Message+Processors

Lab – Using Choice Router, all router and Scatter-Gather

Splitting and Aggregating Messages

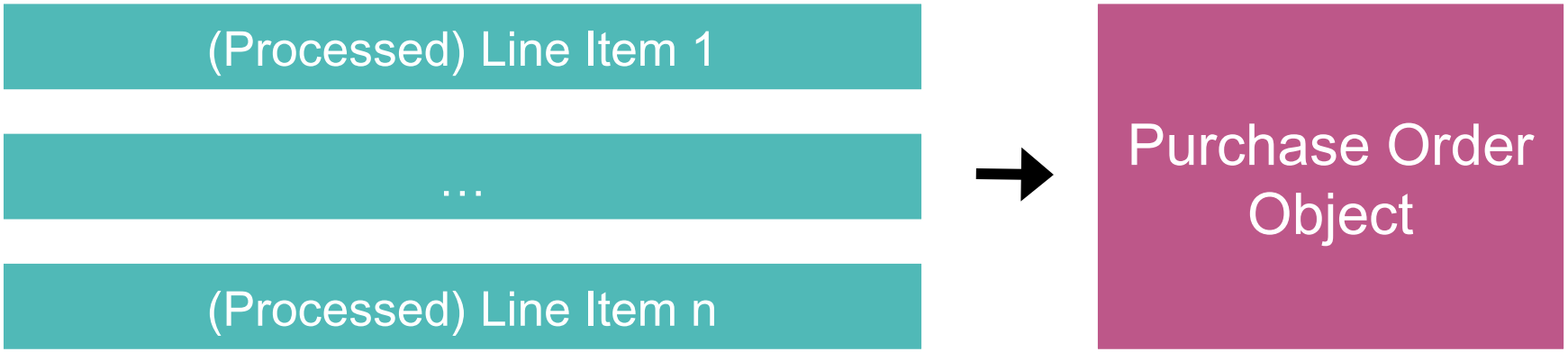
Splitting and Aggregating

- Purchase Order object comes into your system
- Inside the Purchase Order are several line items

Splitting and Aggregating

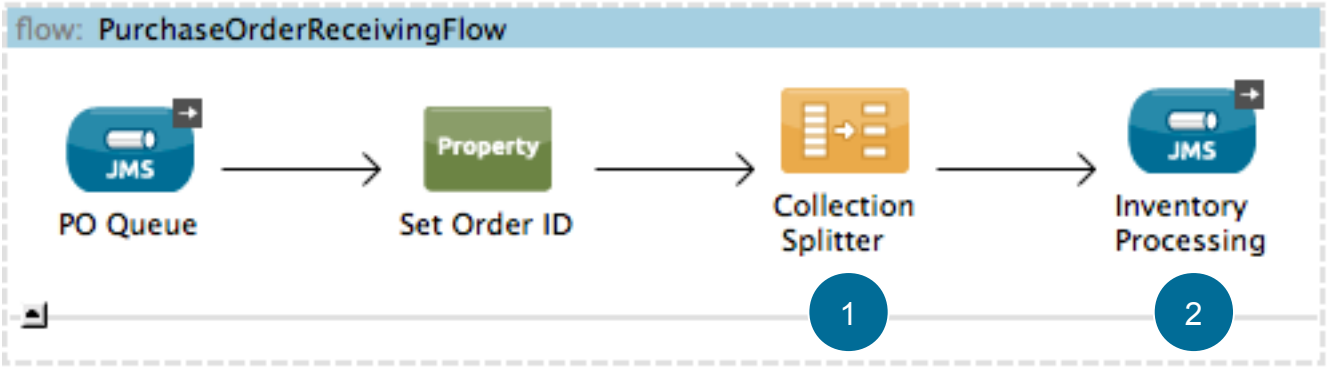
- Line Items will be broken out into their own 'object'
- Each line item needs to move through several processes

Splitting and Aggregating



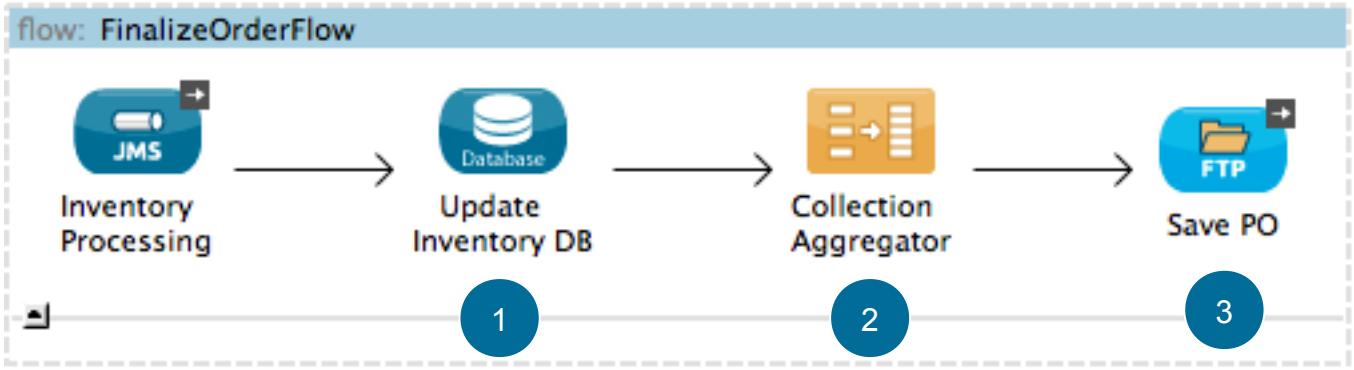
- Once processing is completed, the line items are repackaged into a Purchase Order Object
- *Let's take a look at how a similar process looks in Mule*

Splitting Messages



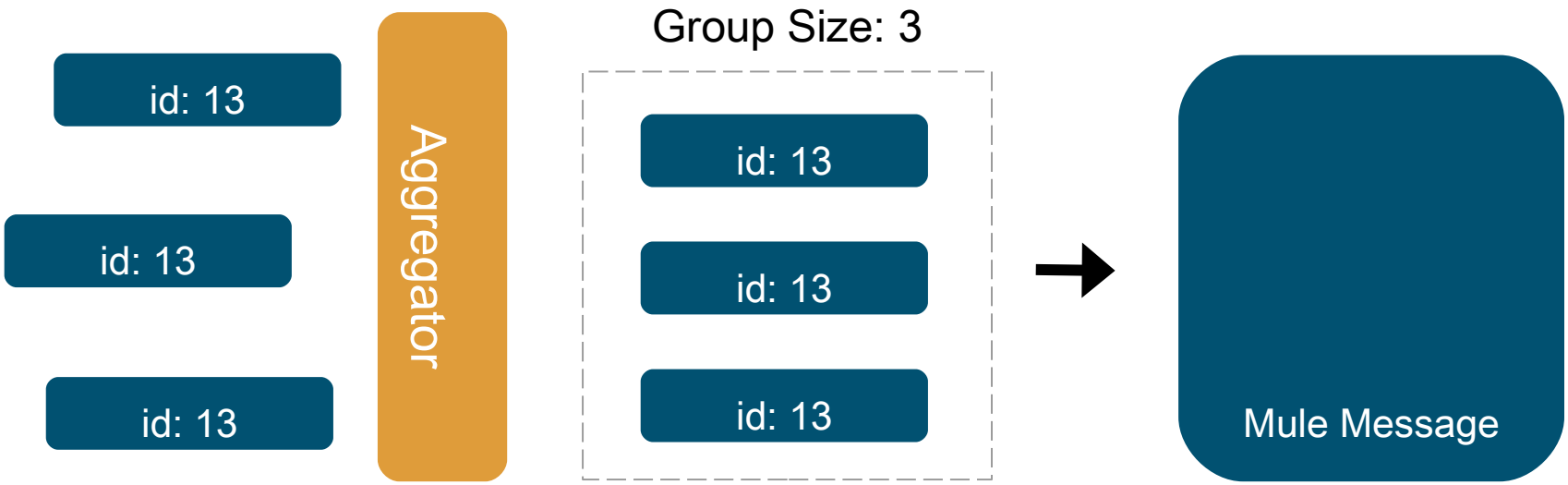
1. PO object's payload (collection) is split into separate Mule Messages
   - Properties set:
     - MULE_CORRELATION_ID
     - MULE_CORRELATION_GROUP_SIZE
2. Each line item is sent to the inventory.processing queue

Aggregating Messages



1. Each line item updates the inventory database
2. All the line items are aggregated back into on Mule Message
3. The finalized message is written to a file and stored in an FTP

Aggregation Phase



- Split messages are regrouped by MULE_CORRELATION_ID
- **failOnTimeout** attribute determines whether message continues if specified timeout is reached
- During aggregation, completion is validated by MULE_CORRELATION_GROUP_SIZE

- What is the result message type after Collection Aggregator is executed ?
- Is it DefaultMuleMessage ?
- No . It is DefaultMesageCollection.
- DefaultMesageCollection is a collection of DefaultMuleMessage objects.
- How to convert DefaultMesageCollection to DefaultMuleMessage with payload as List of payloads of individual messages ?
- Use

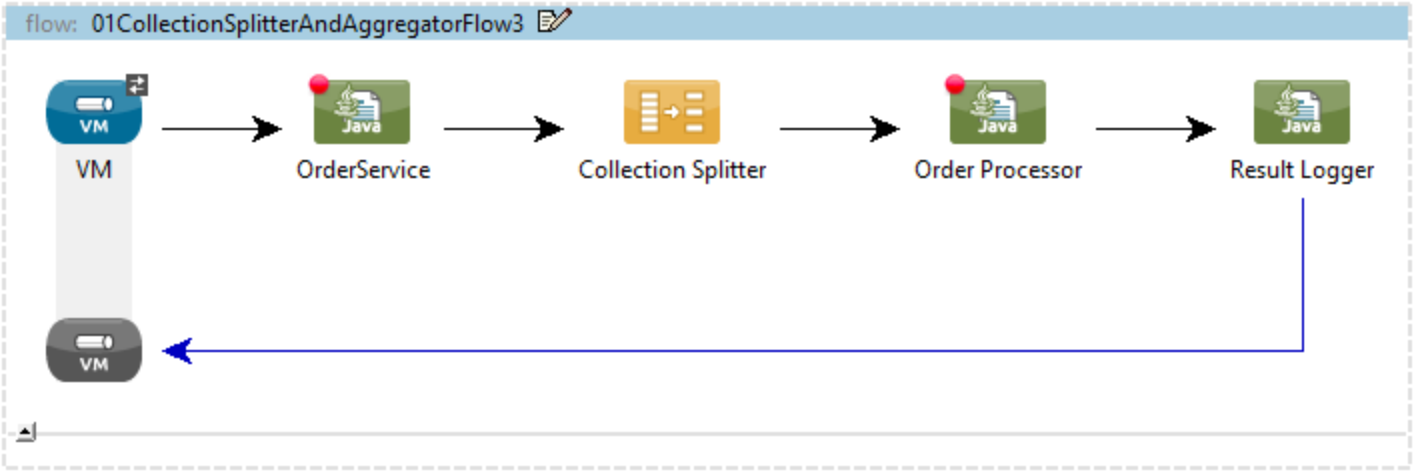
```
<combine-collections-transformer/>
```

MuleSoft

```
public class OrderService {

    public List<Order> getOrders(String inout){
        List<Order> orders= new ArrayList<Order>();
        orders.add(new Order("1", "electronic"));
        orders.add(new Order("2", "sport"));
        orders.add(new Order("3", "kitchen"));

        return orders;
    }

}
```

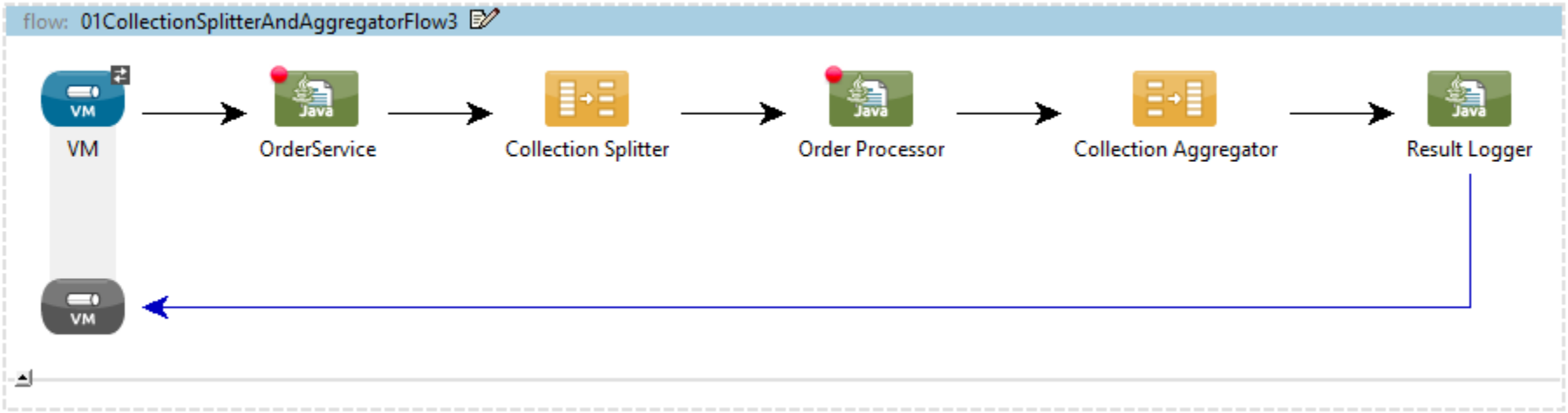
```
public class OrderProcessor {

    public String processOrder(Order order){
        System.out.println("OrderProcessor.processOrder()");
        return order.getType()+"";
    }
}
```

Collection Splitter Example




```
<flow name="01CollectionSplitterAndAggregatorFlow3" doc:name="01CollectionSplitterAndAggregatorFlow3">
    <vm:inbound-endpoint exchange-pattern="request-response" path="collectionsplitterVM" doc:name="VM"/>
    <component class="com.mulesoft.OrderService"   doc:name="OrderService"/>
    <collection-splitter doc:name="Collection Splitter"/>
    <component class="com.mulesoft.OrderProcessor" doc:name="Order Processor"/>
    <component class="com.mulesoft.ResultLogger" doc:name="Result Logger"/>
</flow>
```

Example Using Collection Aggregator




```
<flow name="01CollectionSplitterAndAggregatorFlow3" doc:name="01CollectionSplitterAndAggregatorFlow3">
    <vm:inbound-endpoint exchange-pattern="request-response" path="collectionsplitterVM" doc:name="VM"/>
    <component class="com.mulesoft.OrderService"   doc:name="OrderService"/>
    <collection-splitter doc:name="Collection Splitter"/>
    <component class="com.mulesoft.OrderProcessor" doc:name="Order Processor"/>
    <collection-aggregator failOnTimeout="true" doc:name="Collection Aggregator"/>
    <component class="com.mulesoft.ResultLogger" doc:name="Result Logger"/>
</flow>
```

Splitter Using Expression

```
<mulexml:namespace-manager xmlns:mulexml="http://www.mulesoft.org/schema/mule/xml">
    <mulexml:namespace uri="http://musicbrainz.org/ns/mmd-2.0#" prefix="mb"/>
</mulexml:namespace-manager>

<flow name="04SplitterAggregatorFlow1" doc:name="04SplitterAggregatorFlow1">

    <vm:inbound-endpoint exchange-pattern="request-response" path="splitterVM"
     doc:name="VM"/>
    <http:outbound-endpoint exchange-pattern="request-response" method="GET"
     address="http://musicbrainz.org/ws/2/artist?query=killer" mimeType="text/xml"
     doc:name="HTTP"/>

    <splitter expression="xpath:/mb:metadata/mb:artist-list/mb:artist/mb:name"
    doc:name="Splitter"/>

       <component class="com.mulesoft.ResultLogger"   doc:name="Java"/>
    <logger message="#[payload]" level="INFO" doc:name="Logger"/>
</flow>
```

Additional Splitters and Aggregators

Splitter – Splits messages by expression

Message Chunk Splitter

Message Chunk Aggregator

Resequencer

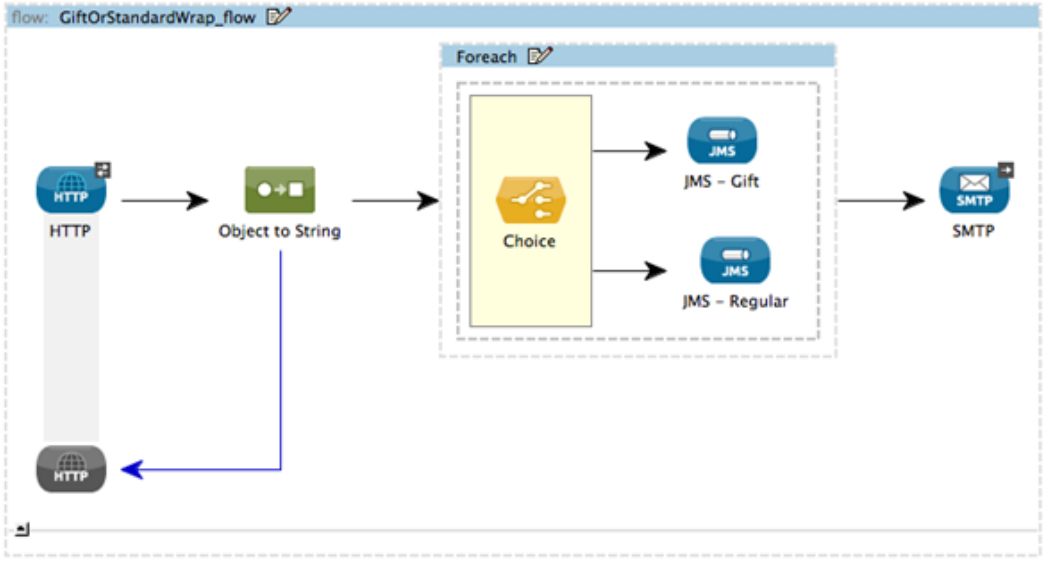
List message splitter router

```
<list-message-splitter-router>
    <jms:outbound-endpoint queue="order.queue">
        <payload-type-filter expectedType="com.foo.Order"/>
    </jms:outbound-endpoint>
    <jms:outbound-endpoint queue="item.queue">
        <payload-type-filter expectedType="com.foo.Item"/>
    </jms:outbound-endpoint>
    <jms:outbound-endpoint queue="customer.queue">
        <payload-type-filter expectedType="com.foo.Customer"/>
    </jms:outbound-endpoint>
    <payload-type-filter expectedType="java.util.List"/>
</list-message-splitter-router>
```

Expression Splitter router

```
<expression-splitter-router evaluator="xpath" expression="/mule:mule/mule:model/mule:service"
                    disableRoundRobin="true" failIfNoMatch="false">
    <outbound-endpoint ref="service1">
      <expression-filter evaluator="xpath" expression="/mule:service/@name = 'service splitter'"/>
    </outbound-endpoint>
    <outbound-endpoint ref="service2">
      <expression-filter evaluator="xpath" expression="/mule:service/@name = 'round robin deterministic'"/>
    </outbound-endpoint>
 </expression-splitter-router>
```

For Each



- Simplifies the iteration over a collection of elements present in the current message.
- Allows performing operations using each item in the selected collection
- Then, continues processing the original message.

For Each Example

```
<mulexml:namespace-manager >
    <mulexml:namespace uri="http://musicbrainz.org/ns/mmd-2.0#" prefix="mb"/>
</mulexml:namespace-manager>


<flow name="06foreachFlow1" doc:name="06foreachFlow1">
    <http:inbound-endpoint exchange-pattern="request-response" host="localhost" port="8055"
                           path="artist" doc:name="HTTP"/>

    <http:outbound-endpoint exchange-pattern="request-response" method="GET"
        address="http://musicbrainz.org/ws/2/artist?query=#[header:inbound:q]"
         mimeType="text/xml" doc:name="HTTP"/>

    <foreach collection="#[xpath:/mb:metadata/mb:artist-list/mb:artist/mb:name]"
                                          doc:name="For Each">
        <logger message="#[payload.getTextContent()]" level="INFO" doc:name="Logger"/>
    </foreach>
</flow>
```

Using For each to simulate splitter

MuleSoft

```
<flow name="07forEachToAggregateFlow1" doc:name="07forEachToAggregateFlow1">
    <http:inbound-endpoint exchange-pattern="request-response" host="localhost"
                           port="8086" doc:name="HTTP"/>
    <component class="com.mulesoft.OrderService" doc:name="OrderService"/>
    <set-variable variableName="resultList" value="#[new java.util.ArrayList()]"
                  doc:name="Variable"/>
    <foreach doc:name="For Each">
        <component class="com.mulesoft.OrderProcessor" doc:name="Java"/>
        <expression-component doc:name="Expression">
            <![CDATA[flowVars['resultList'].add(payload)]]>
        </expression-component>
    </foreach>
    <set-payload value="flowVars['resultList']" doc:name="Set Payload"/>
    <component class="com.mulesoft.ResultLogger" doc:name="Java"/>
    <object-to-string-transformer mimeType="text/plain" doc:name="Object to String"/>
</flow>
```

Lab – Splitter and Aggregator

SOAP Web Services

Introduction to SOAP

MuleSoft

- **S**imple **O**bject **A**ccess **P**rotocol (SOAP)
  - web services protocol
  - based on XML (SOAP envelope uses XML)
- **Transport-independent**
  - mostly over HTTP
  - JMS, SMTP and even TCP are possible
- **Extensible**
  - WS-Security
  - WS-Addressing
- **Self-descriptive**
  - WSDL (**W**eb **S**ervice **D**escription **L**anguage)

Introduction to SOAP

- SOAP message example

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=
"http://www.w3.org/2003/05/soap-envelope">
    <soap:Header>
    </soap:Header>
        <soap:Body>
            <m:GetStockPrice
                xmlns:m="http://www.fooCo.com/stockPrice">
                <m:StockName>FOOC</m:StockName>
            </m:GetStockPrice>
        </soap:Body>
```

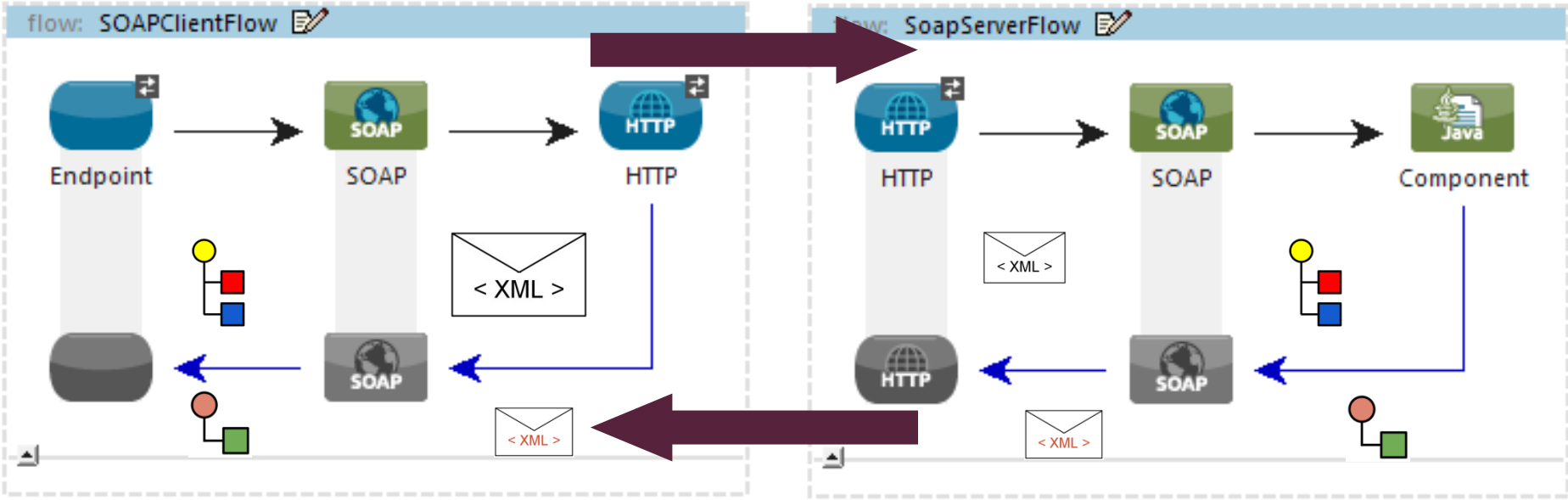
SOAP services and CXF in Mule

MuleSoft

- Set of **SOAP message processors**
  - implement SOAP services
  - consume SOAP services
- Operates over **multiple transports**:
  - normally HTTP
  - also JMS
- Mule's SOAP support is based on Apache **CXF**
  - web services framework in Java for SOAP messaging
  - handles all **serialization** and **deserialization**
  - handles all **SOAP envelope** and **namespace** processing
  - developer sees only POJOs, etc. - not SOAP XML

SOAP services and CXF in Mule

- Mule's SOAP message handling in a nutshell



- Mule's SOAP endpoints transform between Java (etc.) and XML
  - **wrap** or **unwrap** SOAP envelope
  - **marshall** or **unmarshall** objects
  - **insert** to or **extract** from XML, **convert** from strings to data types

Building a JAX-WS CXF web service

- Steps

1. Define the **service interface**:
   - defines the **operations** that the service can perform
2. Create the **component**:
   - contains the **implementation** of each service operation
3. **Configure** the service:
   - declare the service
   - use a CXF message processor

- **Note**: **JAX-WS** service uses **annotations**

Building a JAX-WS CXF web service

- 1. Define the service interface

```
import javax.jws.WebService;
import javax.jws. WebResult;
import javax.jws.WebParam;

@WebService
public interface PriceOfferService
{
        @WebResult(name = "response",  partName = "response")
        public PriceOffer getOffer(
                        @WebParam(partName = "request", name = "request")
        PriceRequest request
             ) throws Exception;
}
```

Building a JAX-WS CXF web service

MuleSoft

- 2. Create the component implementation

```
package com.mulesoft.training.pricing;
import com.mulesoft.training.studentcreated.PriceOfferService;

public abstract class PriceOfferServiceImpl implements PriceOfferService{
        protected String airline;
        // Get offer
        public Offer getOffer(PriceRequest request) throws Exception {
                // Create offer for this airline, but without price
                PriceOffer offer = new PriceOffer(airline);
                // Set the offer price
                offer.setPrice(this.getPrice(request.getDestination(),

request.getOrigin()));
                // Return offer
                return offer;
        }
    // Get price (implemented in subclasses)
        protected abstract int getPrice(String destination, String origin)                throws
Exception
}
```

Building a JAX-WS CXF web service

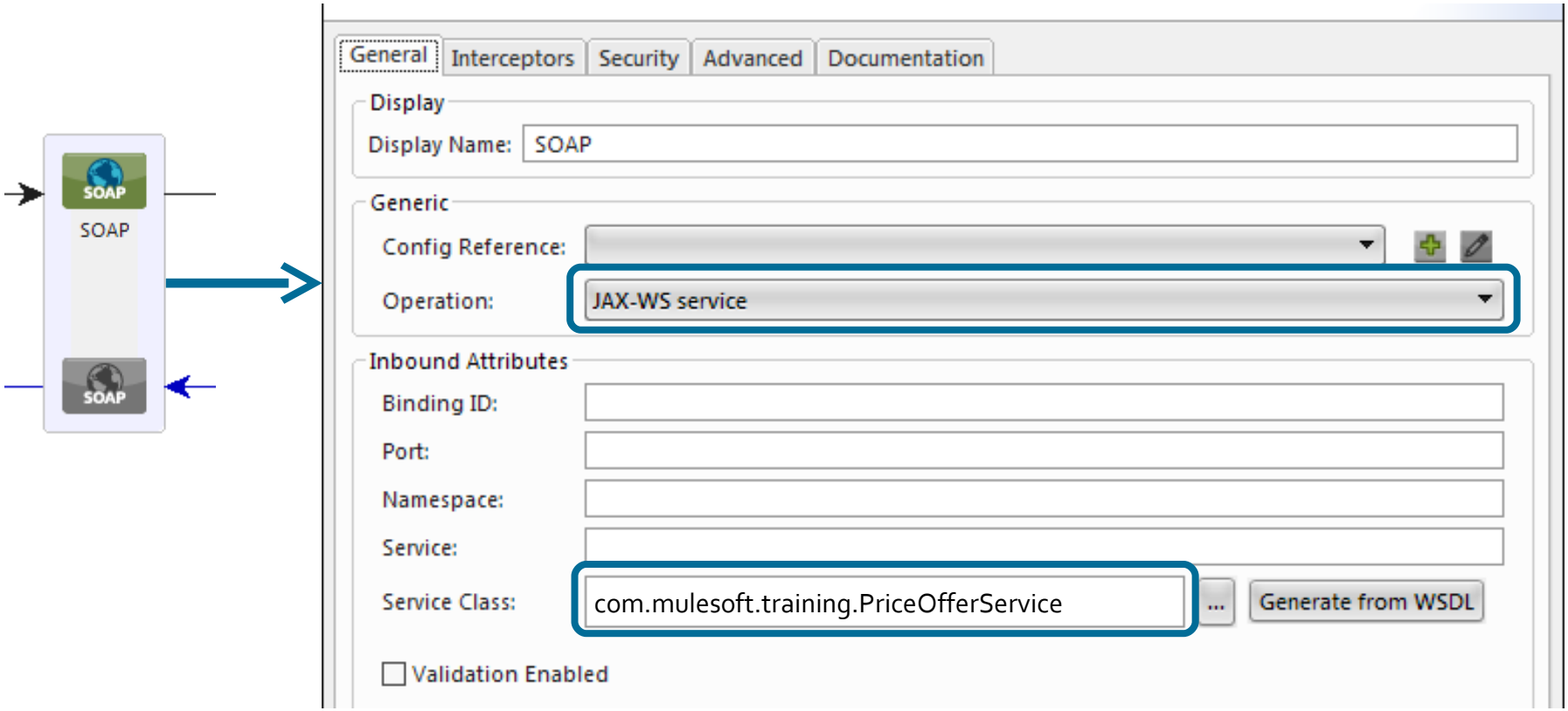
MuleSoft

- 2. Create the component implementation

```
package com.mulesoft.training.pricing;
import com.mulesoft.training.studentcreated.PriceOfferService;

public abstract class PriceOffering implements PriceOfferService{
        protected String airline;
        // Get offer (implement in airline subclasses)
        public PriceOffer getOffer(PriceRequest request) throws Exception {
                // Create priceoffer for this airline, but without price
                …
                // Set the priceoffer price
                …
          // Return offer
                return priceOffer;
        }
    // Get price (implemented in airline subclasses)
        protected abstract int getPrice(String destination, String origin)                    throws
Exception
}
```

Building a JAX-WS CXF web service

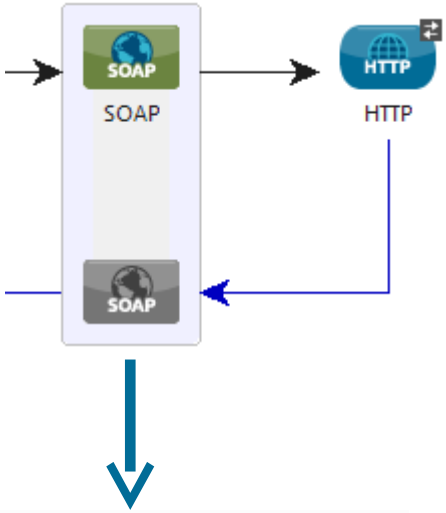
- 3. Configure the service

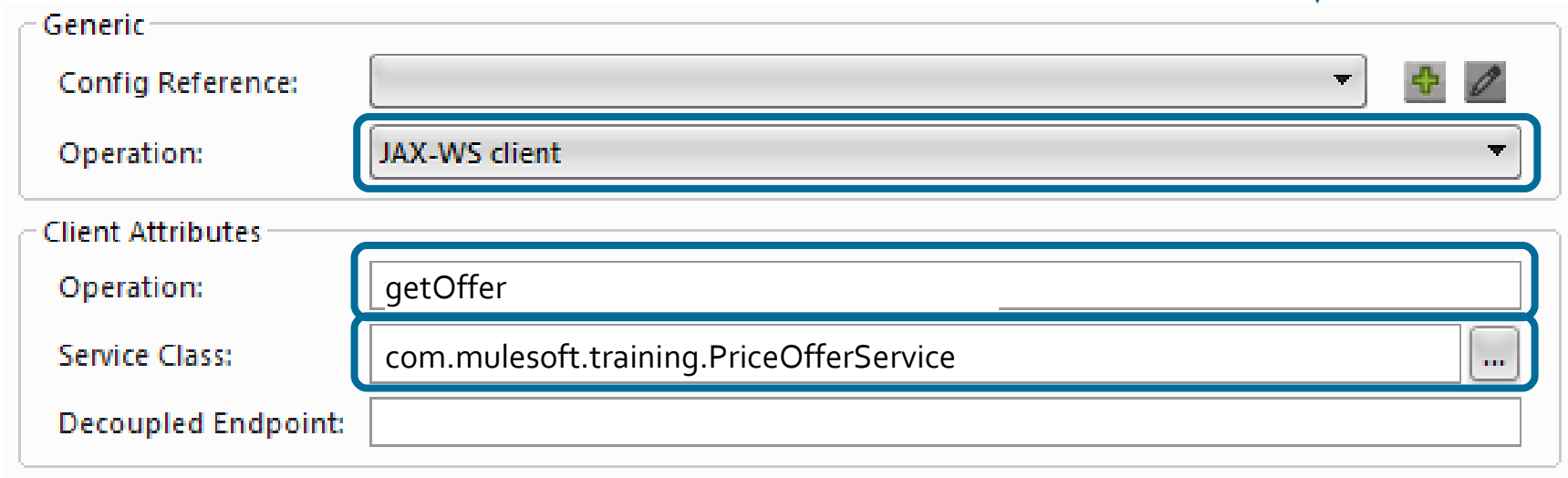


- The **service class** is the **interface** not the implementation
- Double-click on the SOAP component to **configure** it

Configuring a web service client

- Configure the SOAP **client** component:
  - set the generic operation to **JAX-WS Client**
  - set the **interface**
    - use the same as before
  - set the **operation** (method) name





Using WSDL first

MuleSoft

- **WSDL available** before classes created (**contract-first**)
- Generate a **web service interface** from WSDL:
  - use WSDL-to-Java tool
  - implement this interface in a class
  - use class as in previous example
- Refer to the WSDL in **CXF**:
  - through **annotation**
  - through **CXF message processor** (in simple-service)
- WSDL **file** can be located in:
  - classpath
  - file system
  - web app

Using WSDL first

MuleSoft

Example

- **WSDL location** can be file, classpath, or web URL

```
wsdlLocation="./Price.WSDL"
```

- For a **JAX-WS** service:

```
@WebService(endpointInterface =
                    "com.mulesoft.training.PriceOfferService",
    serviceName = "PriceOfferService",
    wsdlLocation="./Price.WSDL")
```

- For a **simple service**:

```
<cxf:simple-service
    serviceClass="com.mulesoft.PriceOfferService"
    wsdlLocation="./Price.WSDL"
/>
```

Summary

MuleSoft

- Introduction to **SOAP**
- SOAP services and CXF in **Mule**
  - message processors, support for multiple transports, based on CXF
  - SOAP components transform between Java and XML
  - three types of hosting on CXF
- Building a **JAX-WS CXF web service**
- Configuring a web service **client**
- Using **WSDL first**

Resources

- Using Mule with Web Services documentation

http://www.mulesoft.org/documentation/display/current/Using+Mule+with+Web+Services

- Mule **SOAP** component documentation

http://www.mulesoft.org/documentation/display/current/SOAP+Component+Reference

- Example of a XML-only **SOAP application**

http://www.mulesoft.org/documentation/display/current/XML-only+SOAP+Web+Service+Example

- **CXF** Module Reference documentation

http://www.mulesoft.org/documentation/display/current/CXF+Module+Reference

- Example of a **WS-Security** application

http://www.mulesoft.org/documentation/display/current/SOAP+Web+Service+Security+Example

REST Web Services

Overview

- **Creating** a **JAX-RS** resource
- **Deploying** a **JAX-RS** resource
- **Consuming** a **REST** resource
- **JSON capabilities** in Mule

What is REST?

MuleSoft

- **RE**presentational **S**tate **T**ransfer (**REST)**
- **Architectural style** for a well-designed (**RESTful**) hypertext application
- Design of the **Web** supports the REST architectural style
- REST applies to **client–server** architecture:
  - resources are found on a **server**
  - **client** requests a "**representation**" of a resource's current **state**
- Every resource has a **URI** (similar to web pages)
- **REST methods** are **conceptual** and general, but use same names as methods in HTTP:
  - **GET | PUT | POST | DELETE**

JAX-RS

MuleSoft

- **Annotation-based API** for creating RESTful services
- Maps a **POJO** to a **resource**
- Part of **J2EE**
- There are a number of **implementations**:
  - Jersey
  - Restlet
  - RESTEasy (JBoss project)
- **Jersey** is Oracle's reference implementation:
  - used by Mule

JAX-RS Annotations

MuleSoft

- Specify **path** of resource (the resource's URI):
  - **@Path** placed on class or method
- Specify **REST method** (request type):
  - **@GET|@PUT|@POST|@DELETE** placed on method
- Specify return **MIME type** (e.g. text/html or application/json):
  - **@Produces** placed on method
- Specify accepted **request types**:
  - **@Consumes** placed on method
- Specify **parameters**:
  - **@PathParam** gets parameter from URI
  - **@QueryParam** gets parameter from query string (?x=1&y=2)
  - **@HeaderParam** gets parameter from HTTP header

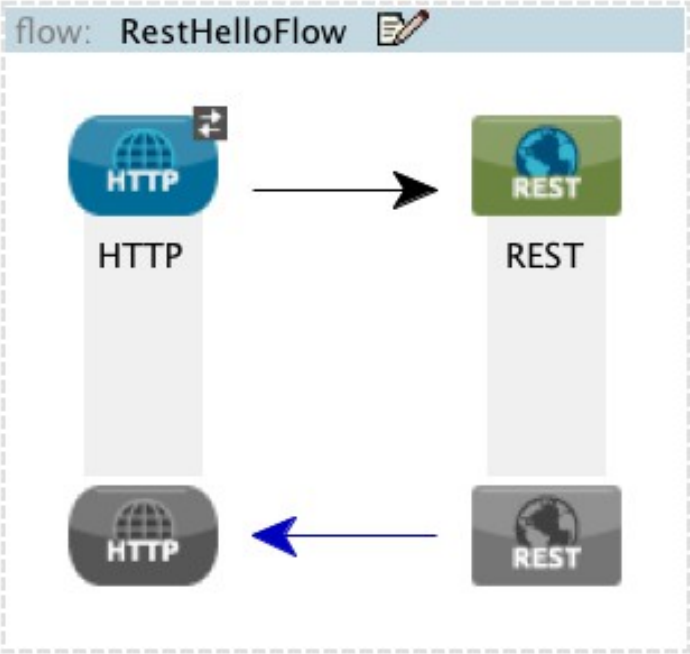
Creating JAX-RS Resource

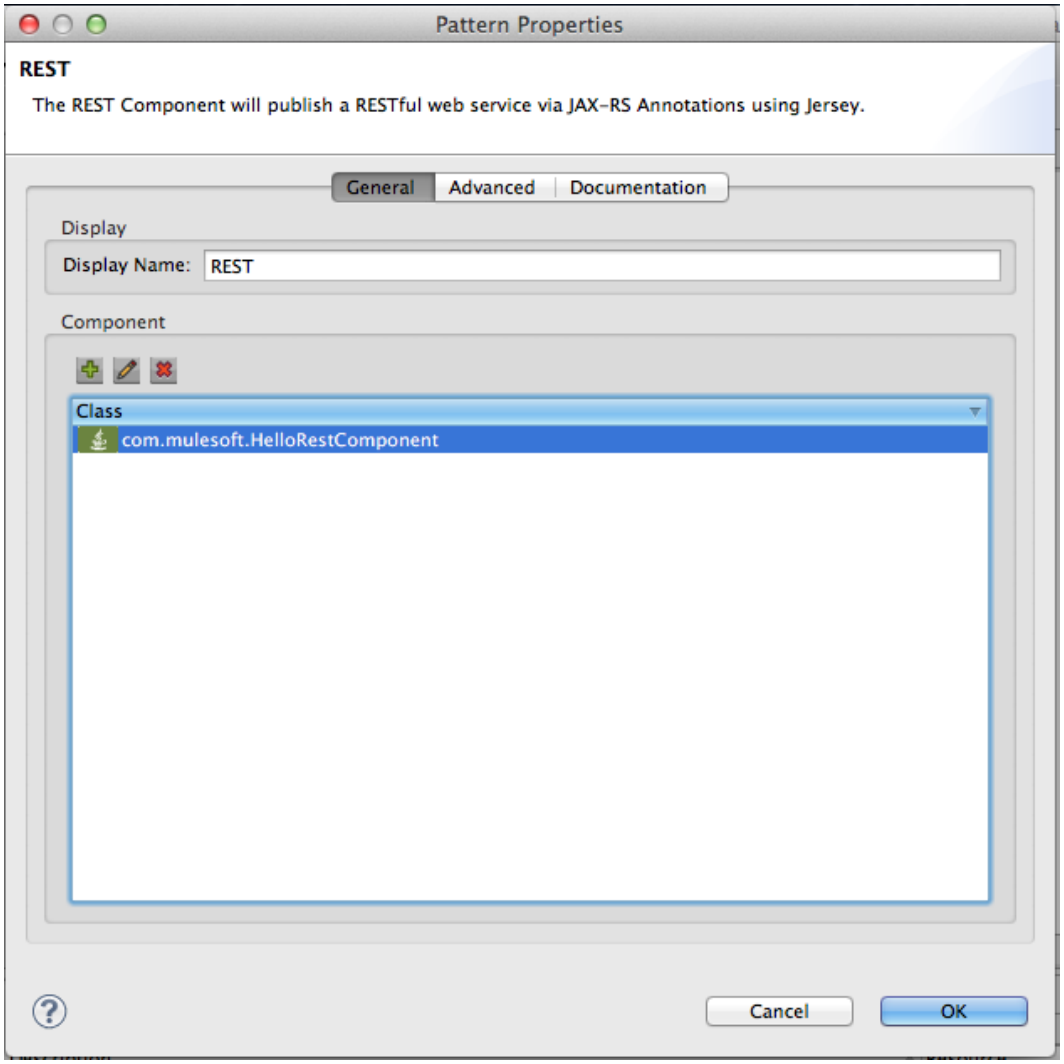
MuleSoft

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

@Path("/")
public class HelloRestComponent {
    @GET
    @Produces("text/plain")
    @Path("/greeting/{name}")
    public String greet(@PathParam("name") String input)
    {
        return "Hello "+ input;
    }
}
```

http://localhost:8080/{endpointpath}/{classpath}/{methodpath}/{value}

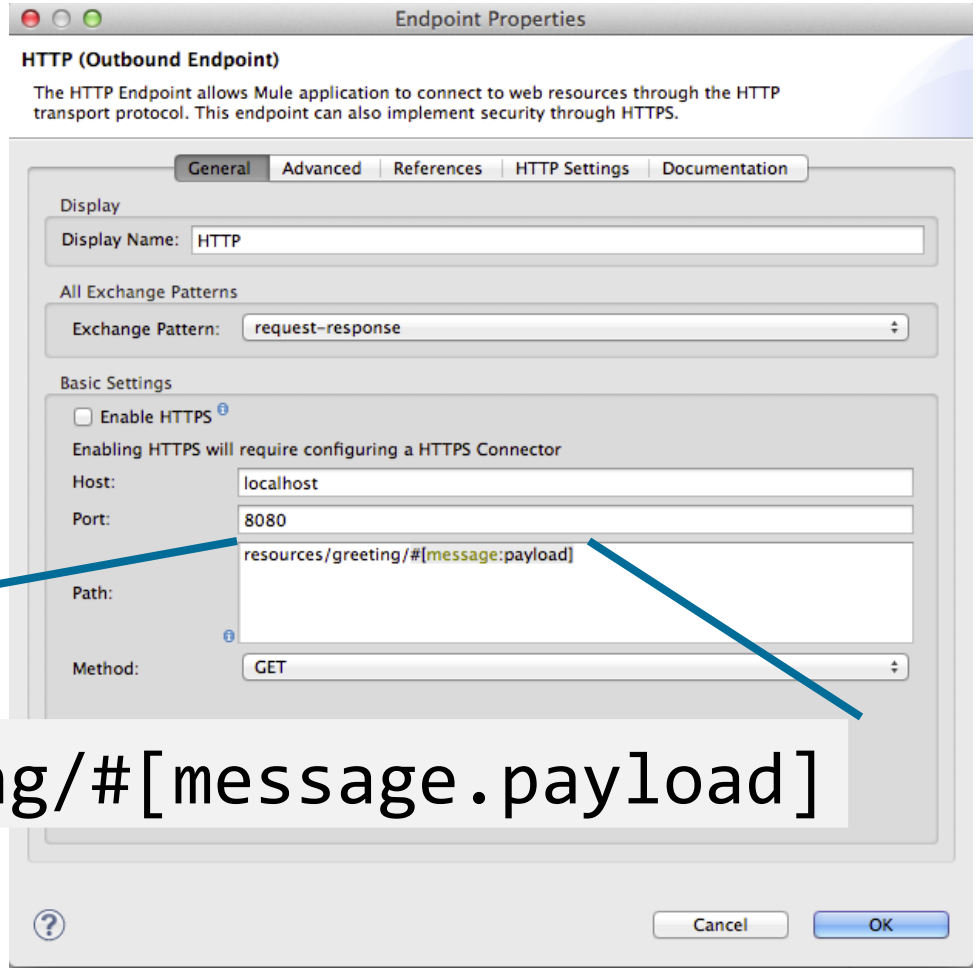
Deploying Jersey Resource





Consuming a RESTful Web Service

- Use **HTTP endpoint**
- Remember to set the correct **REST method**
- **Dynamic endpoints** are very useful




```
/resources/greeting/#[message.payload]
```

JSON Support

- **Jersey**:
  - Specify **@Produces/@Consumes** (application/json)
- **JSON module**:
  - uses **Jackson** to support binding JSON data to objects
  - **JSON to Object** transformer:
    - set **returnClass** attribute
    - JSON engine can be configured with **jsonConfig** attribute
  - **Object to JSON** transformer:
    - a map of JSON object attributes can be configured using **mapper-ref** attribute, usually default is enough
- Querying JSON payloads with **JSONPath**:
  - similar to how XPath can query XML documents

Summary

- **Creating** a **JAX-RS** resource
- **Deploying** a **JAX-RS** resource:
  - using Jersey resource
  - using simple service
- **Consuming** a **REST** resource
- **JSON capabilities** in Mule

Resources

- Using Mule with Web Services documentation

http://www.mulesoft.org/documentation/display/current/Using+Mule+with+Web+Services

- REST Component Reference documentation

http://www.mulesoft.org/documentation/display/current/REST+Component+Reference

JMS

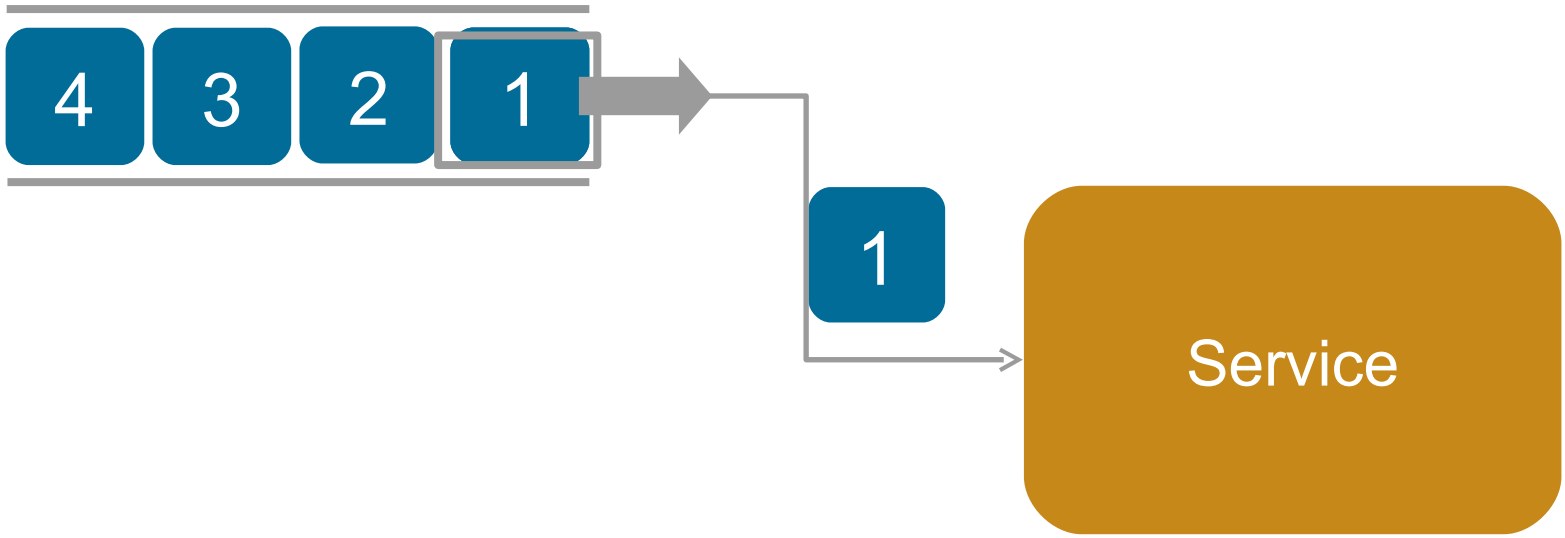
Java Message Service (JMS)

MuleSoft

- Message-oriented **middleware**:
  - *aka* "Enterprise Messaging"
- Allows for **exchange** of data and events:
  - supports creation of message-based applications
- Messages can be sent/received **asynchronously**:
  - supports loosely-coupled middleware
- Messages can participate in **transactions**
- API defined by the JMS **standard**
  - multiple implementations
  - WebLogic, WebSphere, MQSeries, ActiveMQ
  - each implementation has its own server & client libs

JMS messaging models

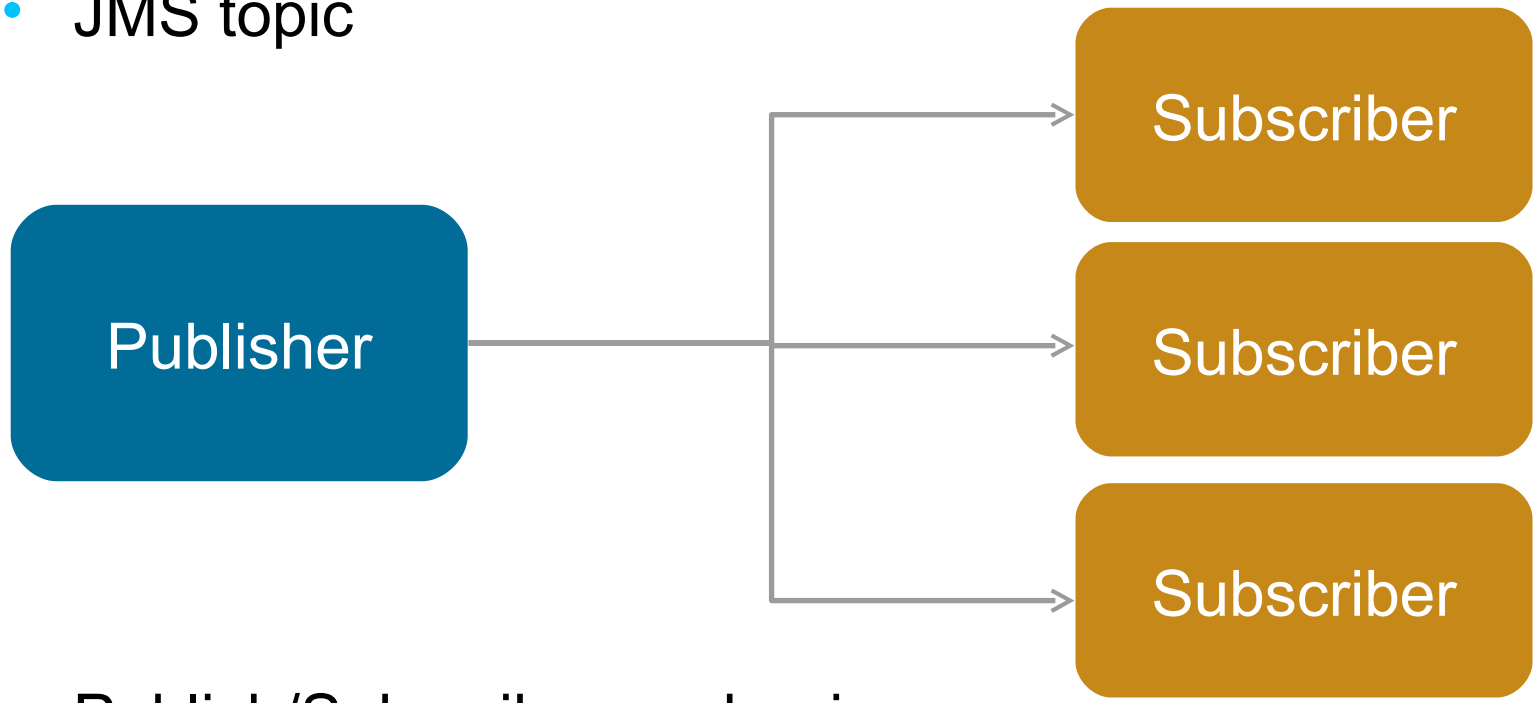
- JMS queue



- Standard queue concept
- First in, first out (FIFO)
- Single Consumer, potentially many Producers
- VM one-way endpoint acts as simple in memory Queue

JMS messaging models

- JMS topic



- Publish/Subscribe mechanism
- Multiple subscribers allowed
- Messages published while subscribers disconnected
- Similar to RSS feeds

JMS elements in Mule

- Types of JMS element

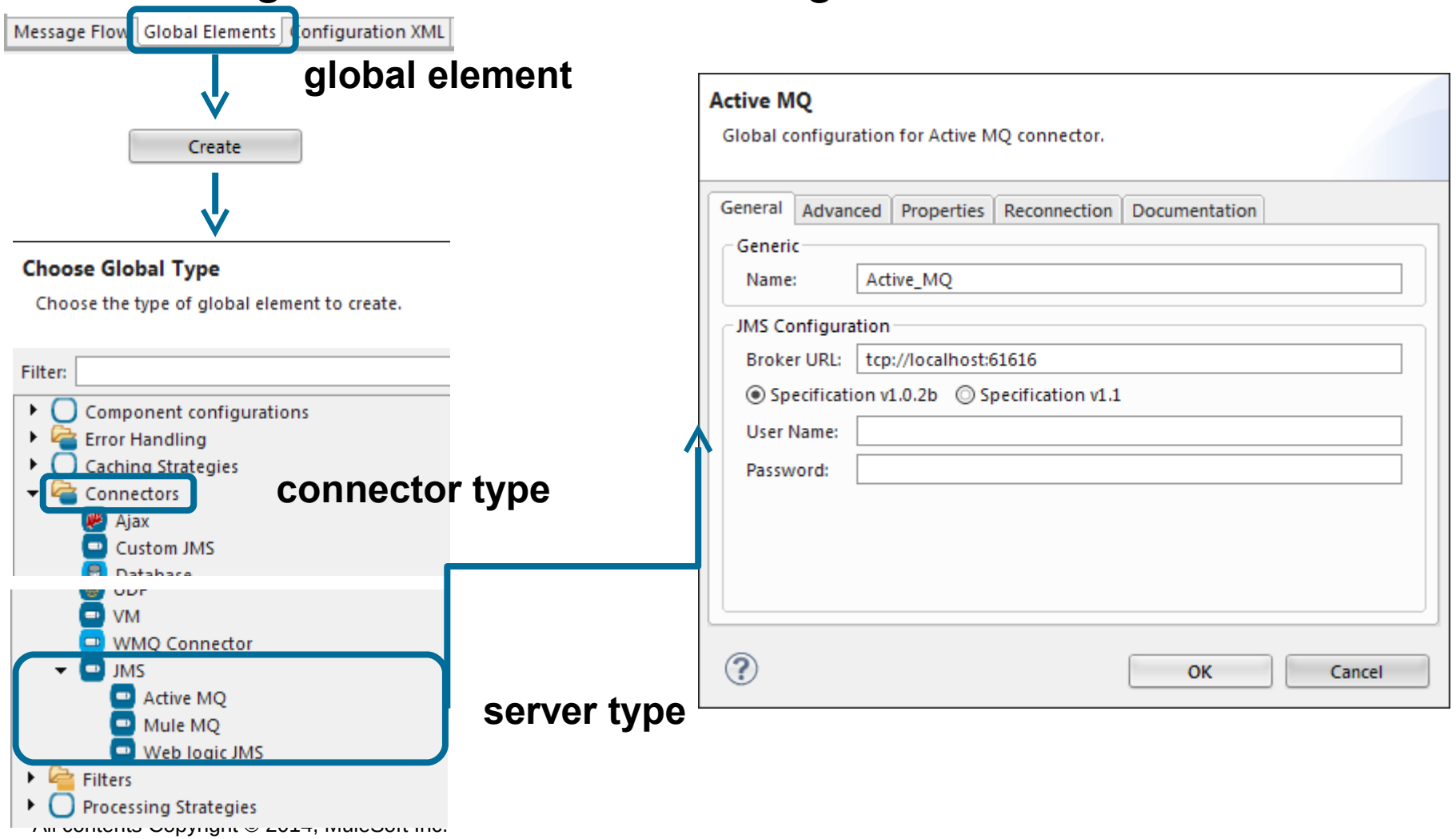
- JMS **connector**
- JMS **endpoints**
- JMS **transformers**
- JMS **selector**

JMS elements in Mule Studio

- JMS transport package supports JMS elements
- Connect to **any JMS-compliant server**:
  - **org.mule.transport.jms** for JMS transport protocol
  - some servers require specific configuration
- Read & write **JMS messages**:
  - **message models**
    - can **consume** from both **queues** and **topics**
    - can **produce** or **publish** to both queues and topics
  - **message serialization**
    - **serialize** data automatically
    - **de-serialize** data on demand
- **EE** only:
  - enhanced support for WebSphereMQ, including transactions

JMS connector in Mule Studio

- Creating a JMS connector as a global element

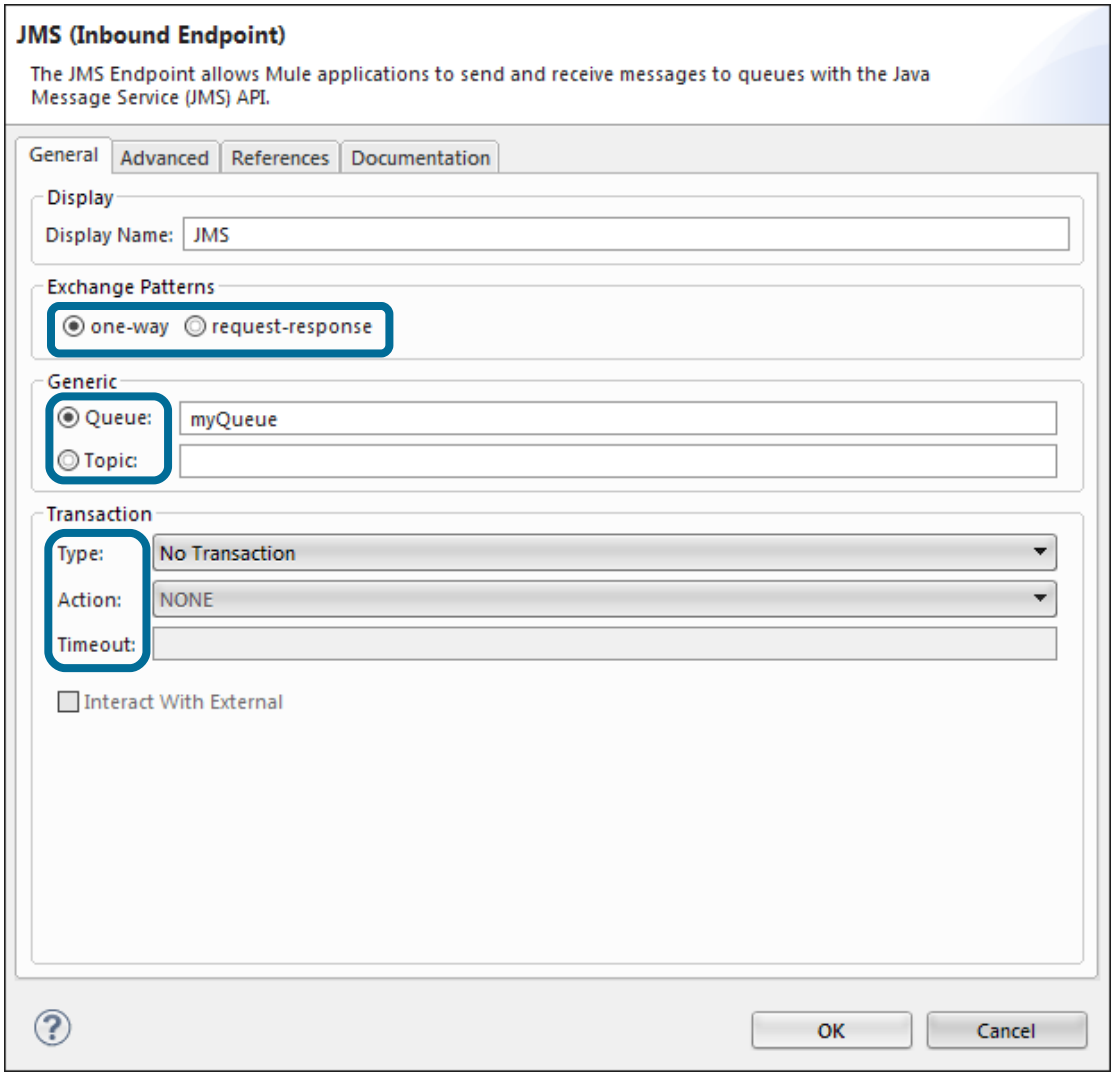


JMS inbound endpoint in Mule Studio

exchange pattern

queue or topic

transaction



JMS transformers in Mule Studio

MuleSoft

- Other filters, transformers and components can access **JMS header properties** on `MuleMessage`
- In **Java**:

```
String corrId = (String)
muleMessage.getInboundProperty("JMSCorrelationID");
boolean redelivered = (Boolean)
muleMessage.getInboundProperty("JMSRedelivered");
```

- In **MEL**:

```
#[message.inboundProperties['JMSCorrelationID']]

#[message.inboundProperties['JMSRedelivered']]
```

JMS selectors in Mule

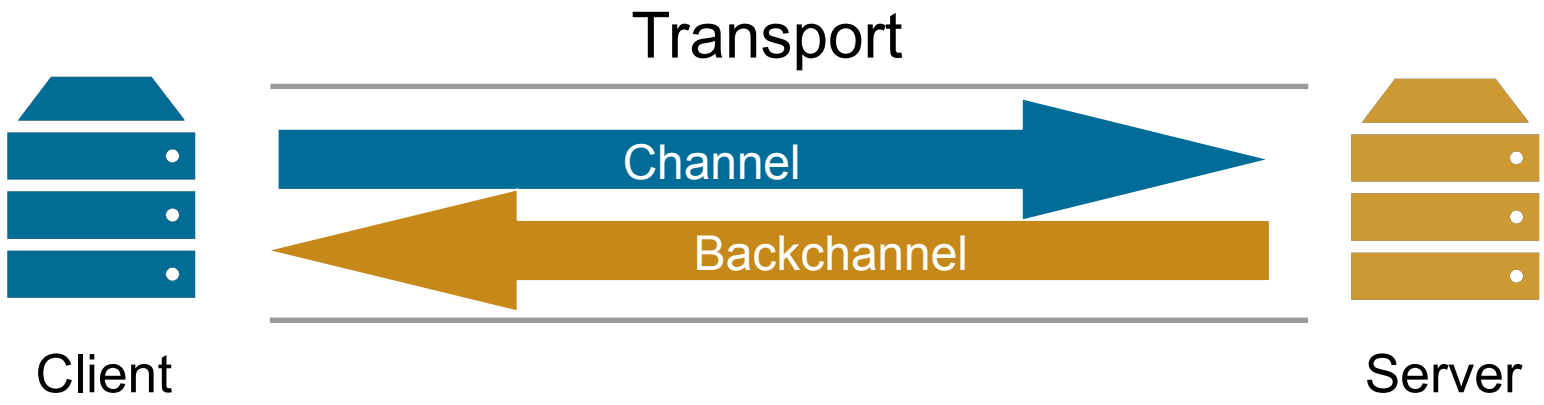
- JMS-specific **filters**
- Added in **XML only**

```
<jms:inbound-endpoint queue="myQueue">
    <jms:selector expression="JMSPriority=9" />
</jms:inbound-endpoint>
```

- Can access **JMS header properties**
- Filter applied before message is de-queued
  - non-matching messages remain available on queue

Backchannel

- A channel which allows data to be **returned back** in a **synchronous** message flow or **response** message:



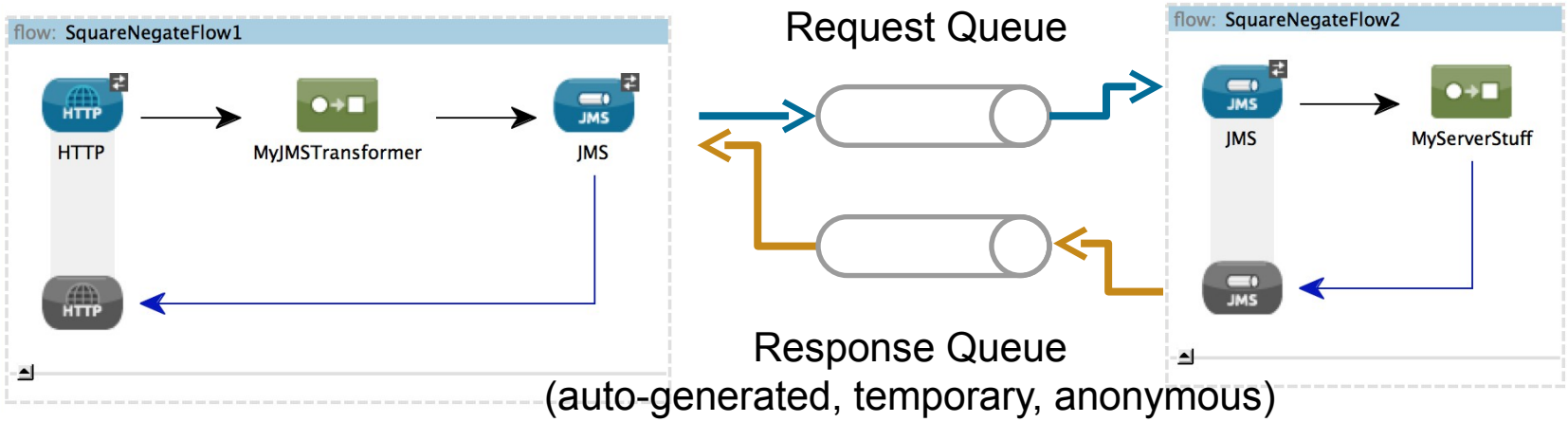
- Not all transports support backchannels:
  - file transport (for local files) has no back channel
  - FTP transport has no back channel

JMS backchannels in Mule

MuleSoft

- **JMS does not provide** backchannels *automatically* or *by default*
- A Mule application must **create its own** backchannel using JMS queue features built into Mule elements
- Two options for element:
  1. **outbound endpoint** with **request-response exchange pattern**
     - just as with HTTP or VM transports
  2. using **<request-reply> message processor** in XML
     - gives more explicit control over message routing
  - each has **pros** and **cons** (*see next*)
- The "**reply-to**" header tells Mule where to send a response
  - e.g. **JMS_REPLY_TO**="jms://myResponseQueue"
  - responses are sent after the message has finished processing
- **JMSCorrelationId** ensures proper sequencing

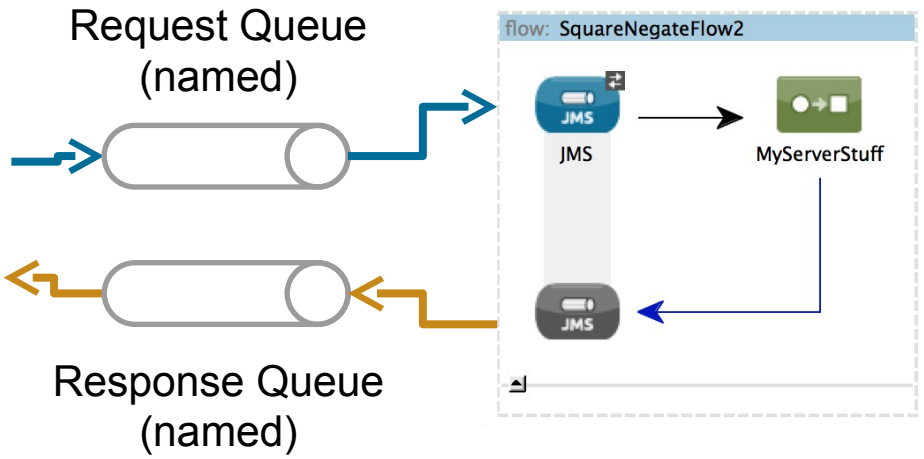
Request-response JMS endpoints



- Request-response **endpoint** automatically sets up a temporary **queue** for a backchannel
  - responses are sent through this queue
  - response queue is **temporary** and **anonymous**
  - reply-to header JMS_REPLY_TO is set automatically

Request-reply message processor

```
<request-reply>
<jms:outbound-endpoint queue="request" />
<jms:inbound-endpoint queue="response" />
</request-reply>
```



- **Request-reply message processor** (*not* an endpoint)
  - sends requests through a **nested outbound-endpoint**
  - receives responses thorough a **nested inbound-endpoint**
- Both **request** and **response** queues are set explicitly, as **named queues**
- No Mule Studio **icon** for now – must **edit XML**

Summary

- **J**ava **M**essage **S**ervice (**JMS**)
- JMS **messaging models**
  - **queues** and **topics**
- JMS elements in Mule
  - JMS **connector** as global element
  - JMS **inbound** and **outbound endpoints**
  - JMS **transformers**
  - JMS **selectors**
- JMS **backchannels**
  - **outbound endpoint** with **request-response** exchange pattern
  - **request-reply** XML element

Lab - JMS

Handling Errors

Mule's Exception Types

- **System** exception
  - thrown when *no* message is processed (e.g. by connector)
  - e.g. application start-up or connection problem
- **Message** exception
  - thrown when a message is processed (e.g. by transformer)
  - e.g. invalid data or unexpected application scenario

System Exceptions

- Thrown at **system level**
- Standard behavior (*not* configurable):
  1. **notifies** registered listeners
  2. **logs** the exception
  3. if connection failure, executes a **reconnection strategy**
     - **configurable** in XML…

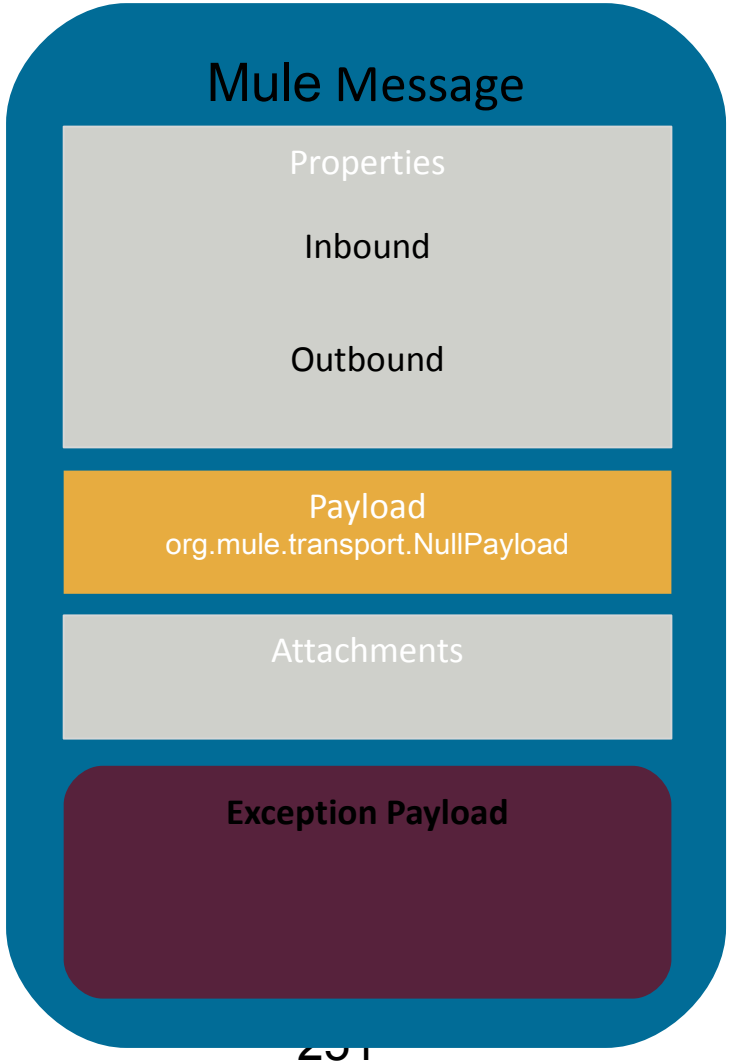
System Exception Reconnection Strategies

- ***Per* connector** or global **default** (in <configuration>)
- **Blocking** or non-blocking
  - <reconnect **block**="TRUE"/>
- Attempt **count** and **frequency** in ms
  - <reconnect **count**="5" **frequency**="1000"/>
  - <reconnect-**forever**…>
- **Notify** listeners of attempts (<**reconnect-notifier**/>)
- **Transactions**
  - **BEGIN** when initiated in a flow
  - **ROLLBACK** if connection fails or statements issued fail
  - **COMMIT** if connection succeeds and statements issued succeed

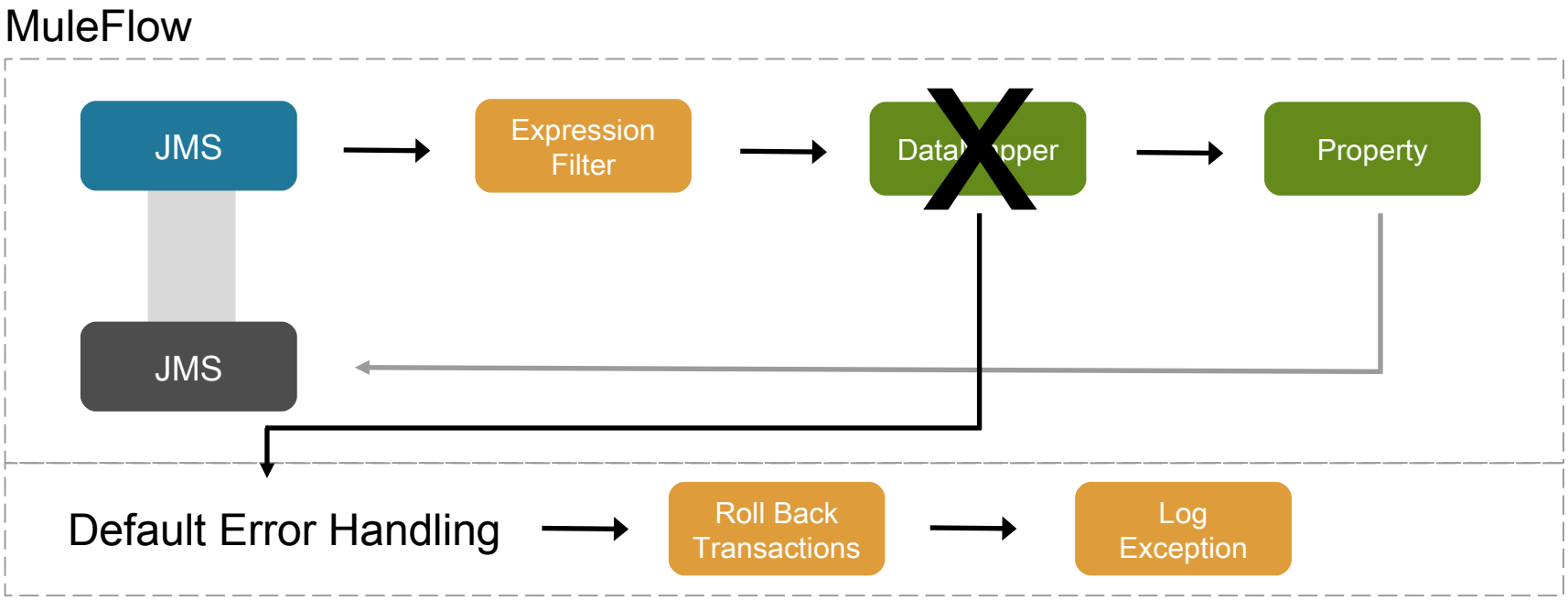
Exception strategies

- Default exception strategy

- **Standard behavior**
  - **aborts** flow execution
  - **logs** exception
  - **rolls back** transaction
  - **returns** control to caller
  - typically, **transport** maps **exception**
- Sets **payload** to null
  - **org.mule.transport.NullPayload**
  - no operations allowed
- Sets **exception payload**
  - original **message**
  - exception **stack trace**
  - exception **message**
- **Custom behavior** configurable

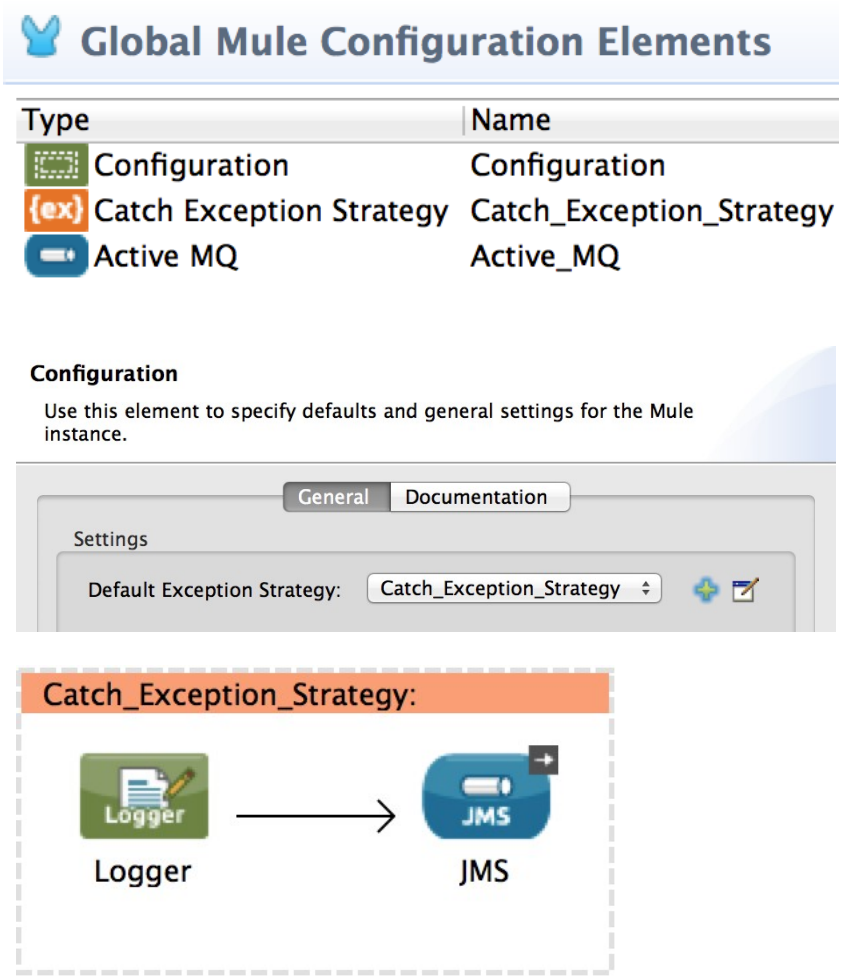


Message Exception Strategy



- Global Default:
  - Injected into Mule Flows implicitly
  - When an exception is thrown:
    - Rolls back transactions (if transactions are configured)
    - Logs the exception

Custom Global Exception Strategy



- Create a global exception strategy
  - Catch
  - Rollback
  - Choice
- Create a global configuration element which references the global exception strategy
- Add message processor(s) in the global exception strategy

Custom Global Exception Strategy - XML

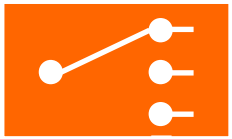
```
<mule . . . >
1 <catch-exception-strategy name="Catch_Exception_Strategy">
3     <logger level="INFO" doc:name="Logger"/>
      <jms:outbound-endpoint …… />
</catch-exception-strategy>
<configuration doc:name="Configuration" doc:description="References the default
    custom exception strategy"
2   defaultExceptionStrategy-ref="Catch_Exception_Strategy"/>
```

1. Global catch exception strategy defined
2. Global configuration element w/ defaultExceptionStrategy-ref pointed to global catch exception strategy
3. Message processors configured in global exception strategy

Error Handling Components

Available Error Handling

Catch
Exception Strategy



Choice
Exception Strategy

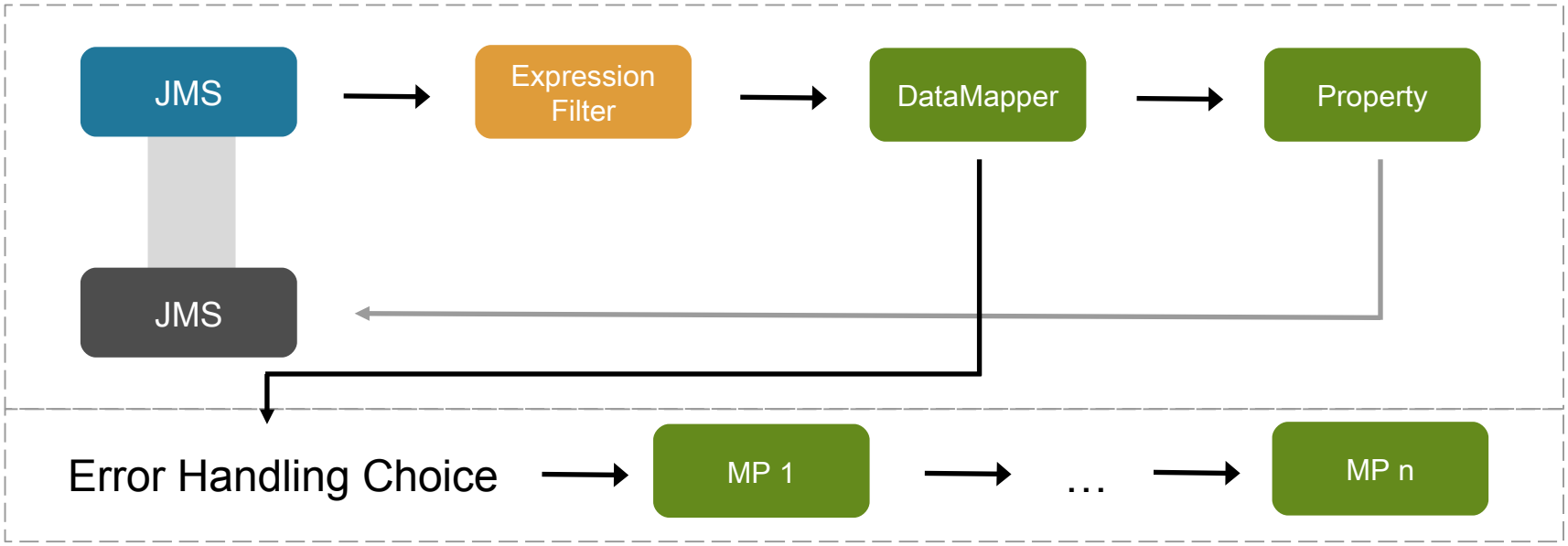
Mapping
Exception Strategy

Reference
Exception Strategy

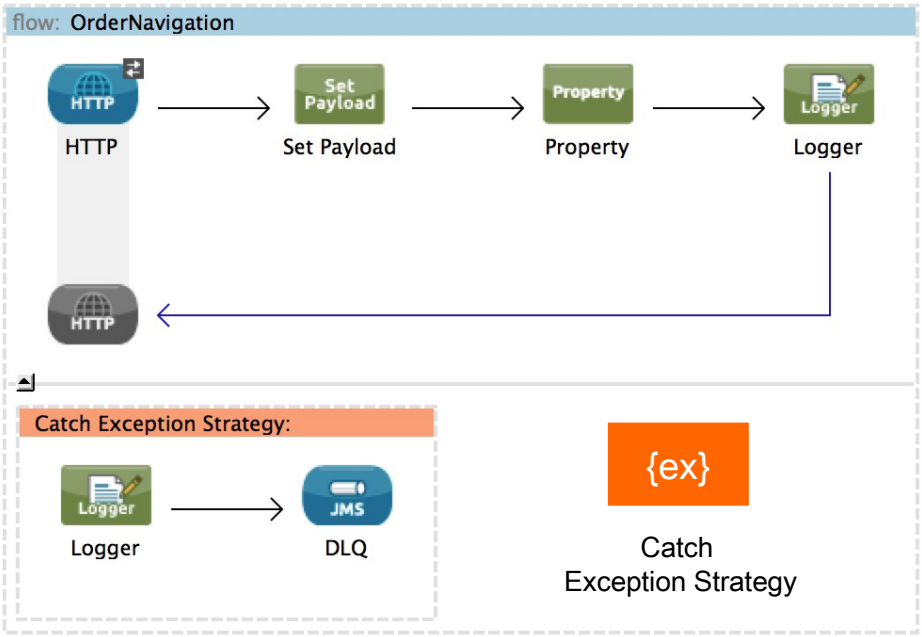
Rollback
Exception Strategy

- Overrides default exception strategy for a flow
- Each flow can contain one exception strategy

Custom Error handling

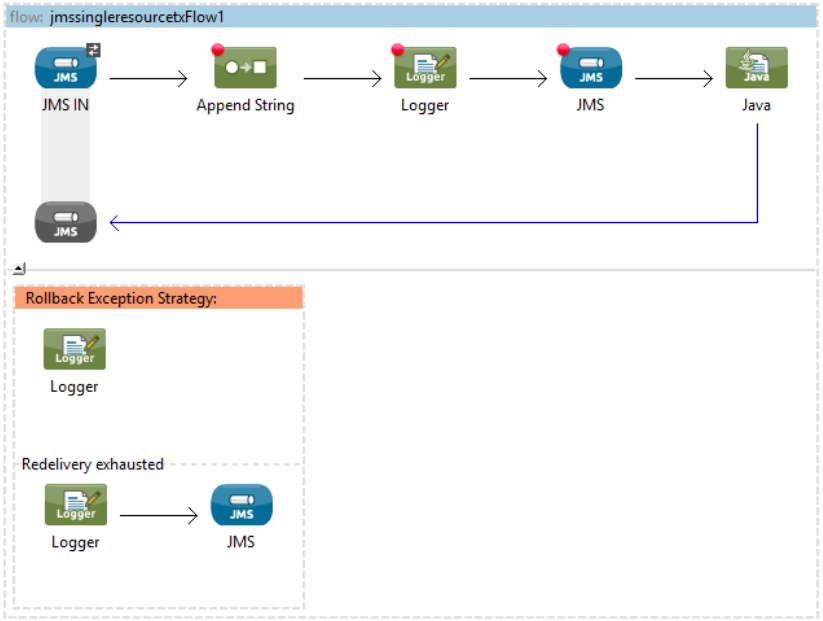


Catch Exception Strategy



- Use
  - Avoid propagating an exception or rolling back a transaction
  - Change an exception type
- Behavior
  - logs exception
  - preserves the message payload (e.g. does not set to null)
  - delivers result from child message processors to caller
  - exception is represented *only* in exception payload

Rollback Exception Strategy



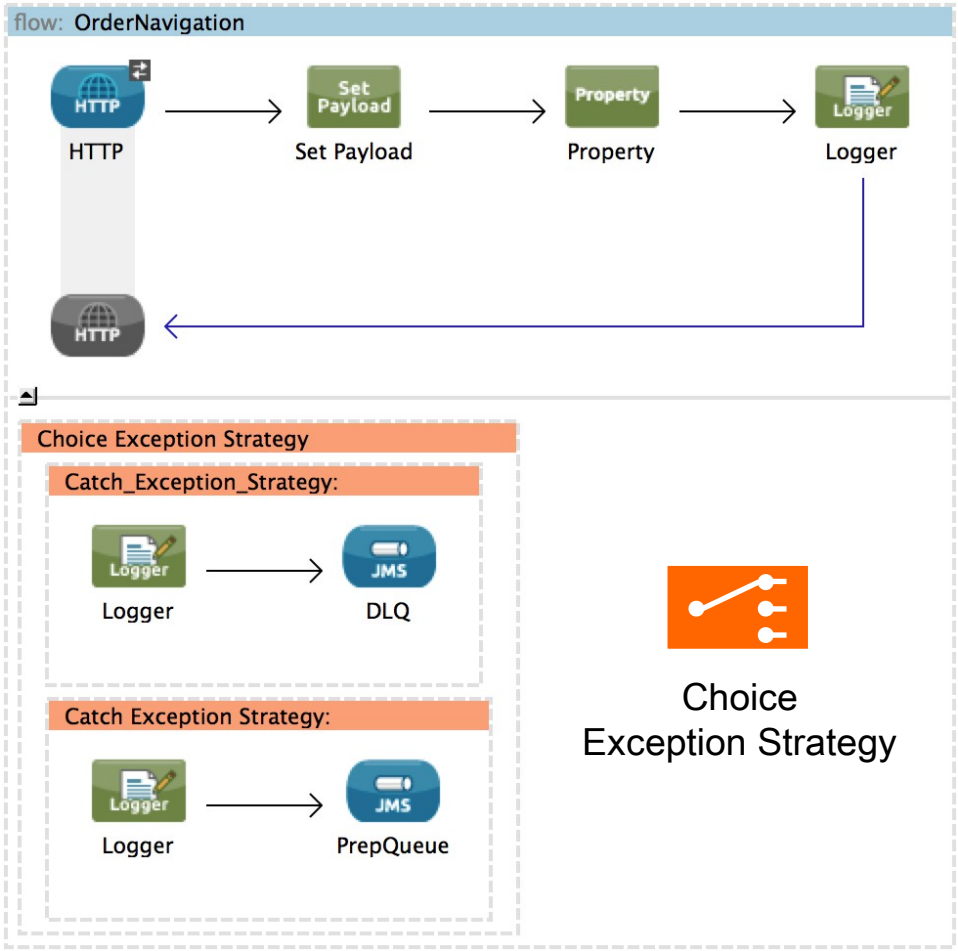
- Use
  - handle **transactions**
  - support **reliable** transport
  - manage **redelivery**
  - **route** unhandled exceptions

Rollback
Exception Strategy

Behavior

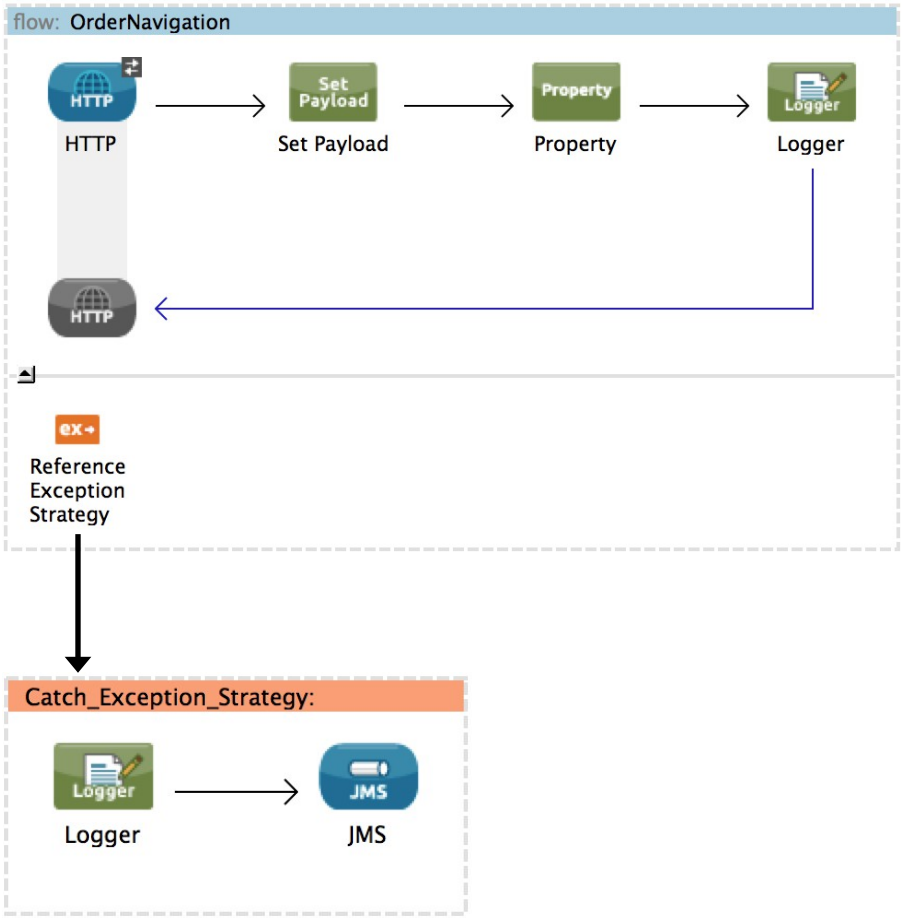
- **rolls back** transactions
- uses copy of **original message** for re-delivery
- supports **re-delivery policy** with:
  - **limit** on re-delivery attempts (after which "exhausted")
  - **alternate flow** called when re-delivery attempts exhausted
- supports differently **routing** unhandled exceptions
  - for one-way vs. request-response exchange patterns

Choice Exception Strategy



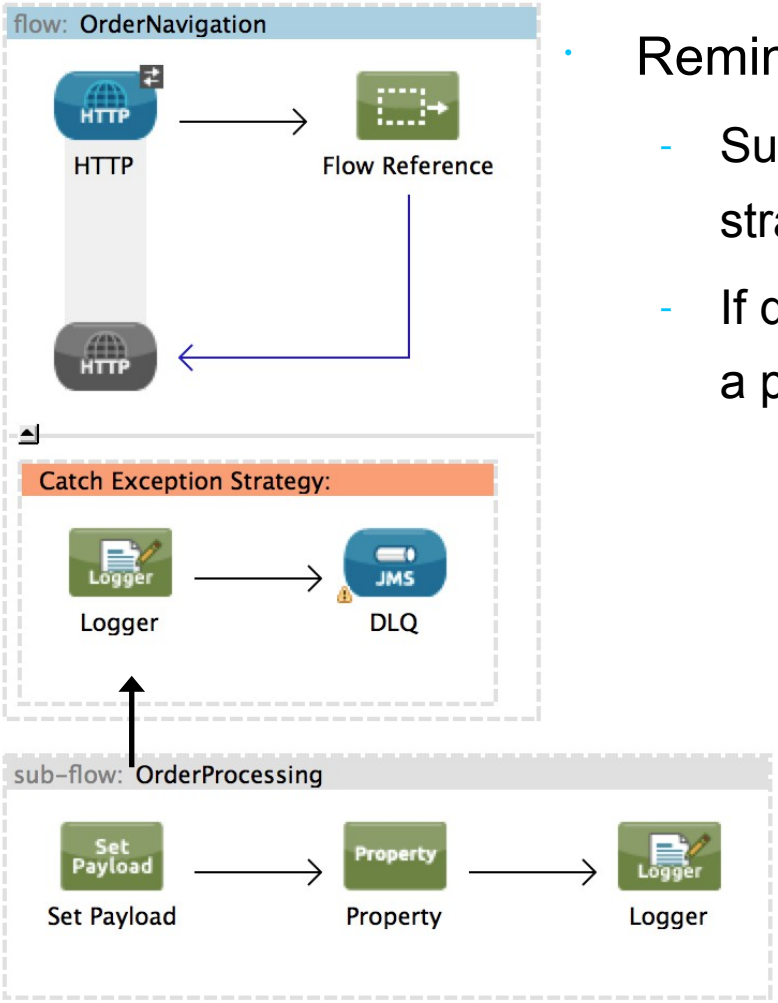
- Use
  - Support multiple exception strategies
  - Need for routing error messages to specific route
- Behavior
  - Evaluates expression to determine which nested strategy to trigger
  - First strategy evaluated to 'true' is used
  - Behavior of nested strategy takes over

Reference Exception Strategy

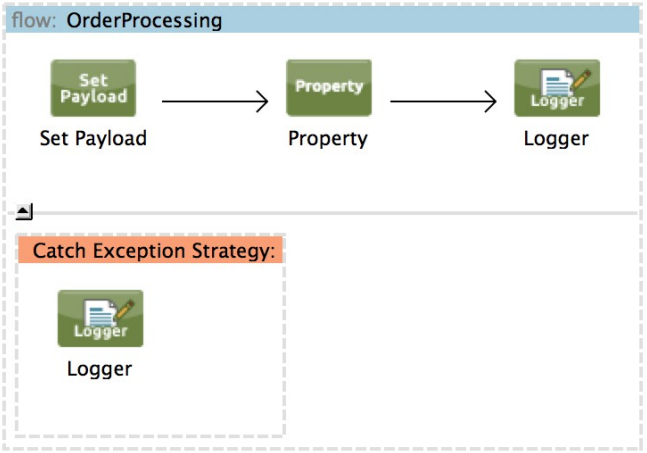


- Use
  - Reference a globally defined exception strategy
  - Global strategies can be:
    - Catch
    - Rollback
    - Choice
- Behavior
  - Behavior of referenced strategy is used

Subflows



- Reminder:
  - Subflows inherit the parent flow's exception strategy
  - If dedicated exception strategy is required use a private flow.

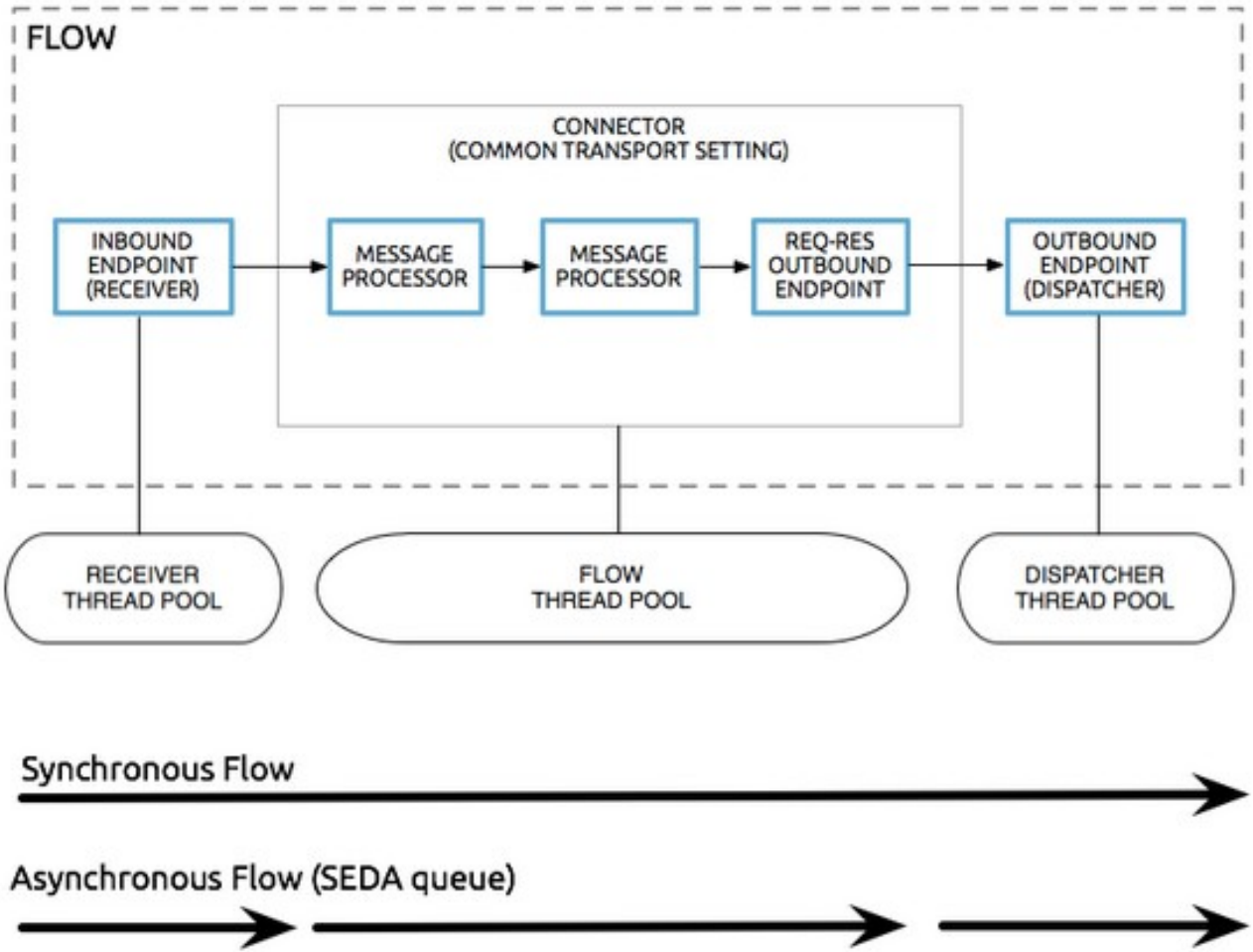


Lab - Exceptions

Lab – Local Tx

Flow Processing Strategies

Thread Pool



Flow Processing Strategies

MuleSoft

- Determine **processing model** used for a flow
- Set automatically by Mule
  - can sometimes be **changed** or **fine-tuned**
- Normal strategy options are:
  - **synchronous**
  - **queued-asynchronous** (with Mule messages queued)
- Set automatically based on:
  - flow **exchange pattern**
  - whether or not the flow is **transactional**
- Support **fine-tuning** of queued-asynchronous strategies by:
  - number of **threads** *per* flow
  - number of **messages** queued
  - **persistent queues** to store data

Setting the flow processing strategy

Exchange Pattern	Transactional?	Strategy
Request-Response	Yes	Synchronous
Request-Response	No	Synchronous
One-way	Yes	Synchronous
One-way	No	Queued-Asynchronous

- Flow processing strategy is **set automatically** as above by Mule, based on:
  - **exchange pattern**
  - whether or not the flow is **transactional**
- **Note** that queued-asynchronous strategies are used only for *one-way, non-transactional* exchange patterns

Changing the flow processing strategy

MuleSoft

Mule strategy	Change strategy?	Fine-tune?	Different strategy in a processing block?	Custom strategy?
Synchronous	No	No	Yes	Yes
Queued Asynchronous	Yes	Yes	Yes	Yes

- When the flow strategy is **asynchronous**, you can:
  - **change** the strategy to **synchronous**
  - reference a **synchronous subflow** (i.e. change strategy in a processing block)
  - **fine-tune** the strategy
- When the flow strategy is **synchronous**, you can use an **asynchronous scope** for some message processors

Changing the flow processing strategy

MuleSoft

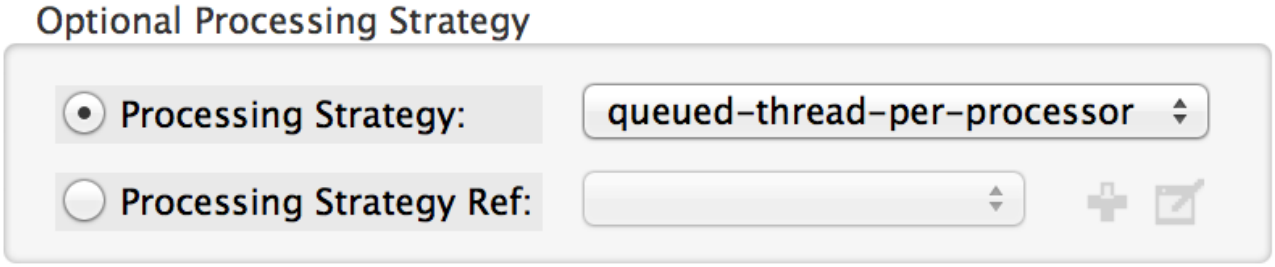
- Asynchronous flow processing strategy options

Note these options are *not applicable to most use cases*

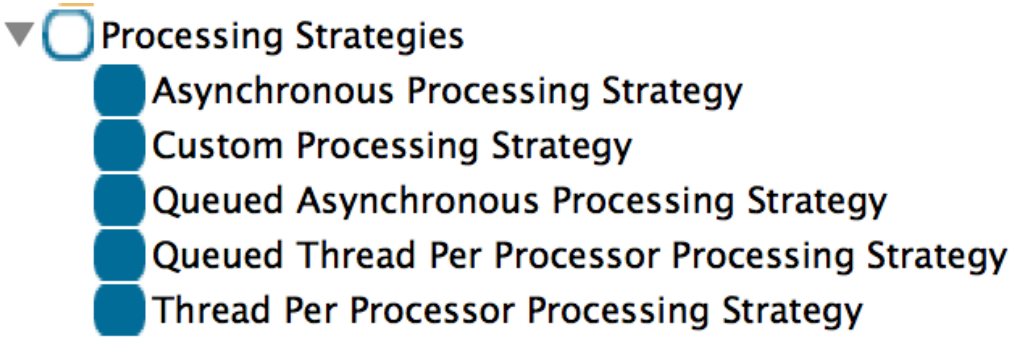
- **Asynchronous** but "**queue-less**"
  - same as queued-asynchronous, but doesn't use a queue.
  - used only when processing is not distributed to another node
- **Queued, thread-per-processor**
  - writes messages to a queue
  - each processor in scope runs in a different thread
- **Thread-per-processor**
  - no queue
  - each processor runs sequentially in a different thread

Changing the flow processing strategy

- Change options
- Change strategy **flow configuration attributes**



- Create a standard processing strategy **global element**
- Define a **custom processing strategy** as a global element



Fine-tuning the flow processing strategy

MuleSoft

- Threading & queueing profile

profile attribute	values	setting
poolExhaustedAction	enum	**WAIT**, **DISCARD**, **DISCARD_OLDEST**, **ABORT**, **RUN**
maxThreadsActive	integer	maximum number of **active** threads
maxThreadsIdle	integer	maximum number of **idle** threads
threadTTL	integer	"**t**ime **t**o **l**ive" for an idle active thread in ms
threadWaitTimeout	integer	time in ms to **wait** for an available thread
maxBufferSize	integer	size of **buffer** memory for waiting jobs
maxQueueSize	integer	maximum number of **messages** queued
queueTimeout	integer	timeout on taking events from the queue
doThreading	boolean	use threading or not

Fine-tuning the flow processing strategy

- Queue store options

Queue store nested element	Meaning
simple-in-memory-queue-store	Simple in-memory queue store
default-in-memory-queue-store	Default for non-persistent queues
default-persistent-queue-store	Default for persistent queues
file-queue-store	Simple file queue store
queue-store	Reference to queue store defined elsewhere
custom-queue-store	Custom, defined with Spring properties

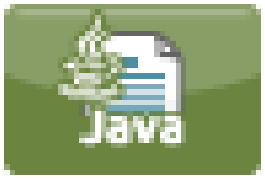
- Defined as **<•••-queue-store>** nested element of referenced processing strategy global element
- Configurable in processing strategy global element dialog

Custom Components

Custom objects

MuleSoft

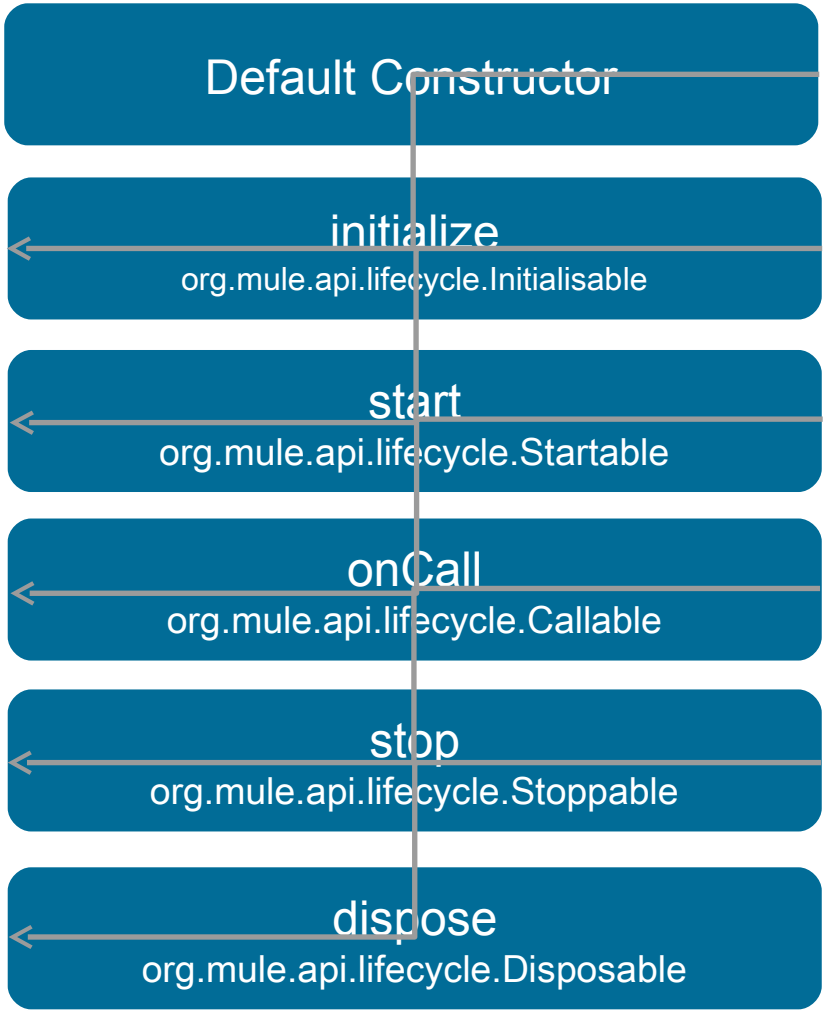
- We can implement multiple items in mule
  - Java components
  - **transformers**
  - Java **Beans**
  - **custom elements** of almost any type in Mule
- So how does Mule handle all this?
  - how many **instances** of our objects?
  - what about **object lifecycle**?
  - which **method** to call in our code?



Java

Object lifecycle

- Components in Mule follow a preset **object lifecycle**
- Can participate by implementing a **lifecycle interface**
- **Callable interface** commonly implemented:
  - **onCall method** called when message is received
- Support for **JSR-250**:
  - **@PostConstruct**
  - **@PreDestroy**



Callable example

```
import org.mule.api.lifecycle.Callable;
public class CallableComponent implements Callable {
  public Object onCall(MuleEventContext eventContext) throws
    Exception {
      MuleMessage message   =eventContext.getMessage();
      Object payload        =message.getPayload();
      Object myVar          =message.getInvocationProperty("MyFlowVar");
      String correlationId  =message.getCorrelationId();
      FlowConstruct flow    =eventContext.getFlowConstruct();
      MessageExchangePattern exchangePattern
            =eventContext.getExchangePattern();
      …
      return message;
    }
}
```

Component Bindings

Component bindings

MuleSoft

- Ability to **call Mule flows** without using Mule APIs:
  - use interface instead of API
  - add interface to custom component as a property
- **Mule creates a proxy** and injects it into the component:
  - application calls interface methods
  - Mule sends messages through **endpoints**
- Still "independent" of Mule:
  - implement the interface
  - configuration-driven

Component binding and interface

- External resource **interface declaration**

```
public interface ExternalResourceInterface {
        public String invoke(String request);
}
```

- Component **class implementation**

```
public class MyService {
        ExternalResourceInterface externalResource;
        public String processRequest(String request)
        {…
                String result = externalResource.invoke(request); …
                return result;
        }
}
```

Configuring a component binding in XML

```
<flow name="ComponentBindingFlow">
    <vm:inbound-endpoint path="ComponentBindingFlowRequest"/>
    <component class="com...MyService">
      <binding interface="com...ExternalResourceInterface">
          <vm:outbound-endpoint queue="Queue1" … >
        </vm:outbound-endpoint>
      </binding>
    </component>
</flow>
```

Executed when called from within component

Custom Transformers

Custom transformers

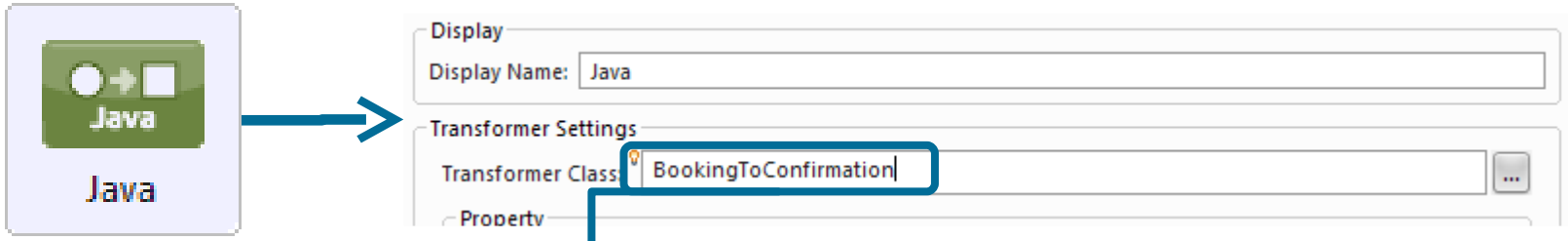
- **Convert data** from one type to another:
  - one type of **object** to another
  - one type of **structure** to another
- Standard or custom:
  - **transport-specific** transformations are provided
  - **application-specific** transformations are needed
- Implement a Mule **interface:**

	operates on:
org.mule.api.transformer.Transformer	**payload** *only*
org.mule.api.transformer.MessageTransformer	**MuleMessage**

- Or extend an **abstract class:**

org.mule.transformer.AbstractTransformer	**payload** *only*
org.mule.transformer.AbstractMessageTransformer	**MuleMessage**

Custom transformer example




```
public class BookingToConfirmation extends AbstractTransformer {

@Override
protected Object doTransform(Object src, String encoding) throws
TransformerException
{
      Booking booking = (Booking) src;
      return "Dear " + booking.passengerName
            +"We would like to inform you that your seat on"
            +" flight "+booking.flightNumber
                              +" leaving "+booking.originAirport
                              +" on "+booking.departureDate
                              +" at "+booking.departureTime
                              +" is confirmed.";
}
}
```

Custom Filters

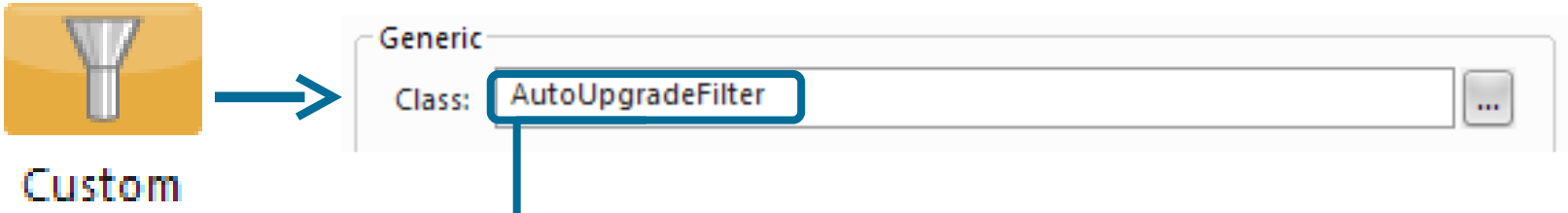
Custom filters

MuleSoft

- Filter **returns boolean** value:

- Each filter will check a **MEL expression**:
  - expression coded in configuration
- Filter operates on a **MuleMessage**:
  - can inspect **properties** or **payload**
- **Interface**: **org.mule.api.routing.filter.Filter**
  - **method**: boolean **accept**(MuleMessage message)

Custom filter example



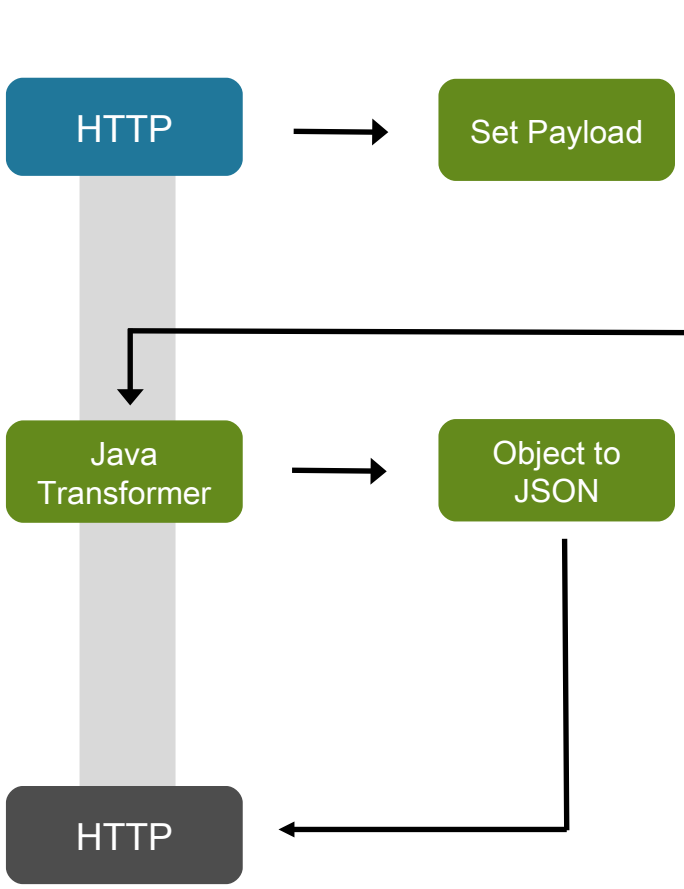

```
public class AutoUpgradeFilter implements Filter {

    public boolean accept(MuleMessage message) {
        Booking booking;
        if (message.getPayload() instanceof Booking) {
            booking = (Booking) message.getPayload();
            return (booking.getFrequentFlyer() != null)
                && (booking.getFlyerMiles() >= threshold);
        } else {
            return false;
        }
    }
}
```

Lab – Sorting Results using custom transformer

Lab 13.0 Goals

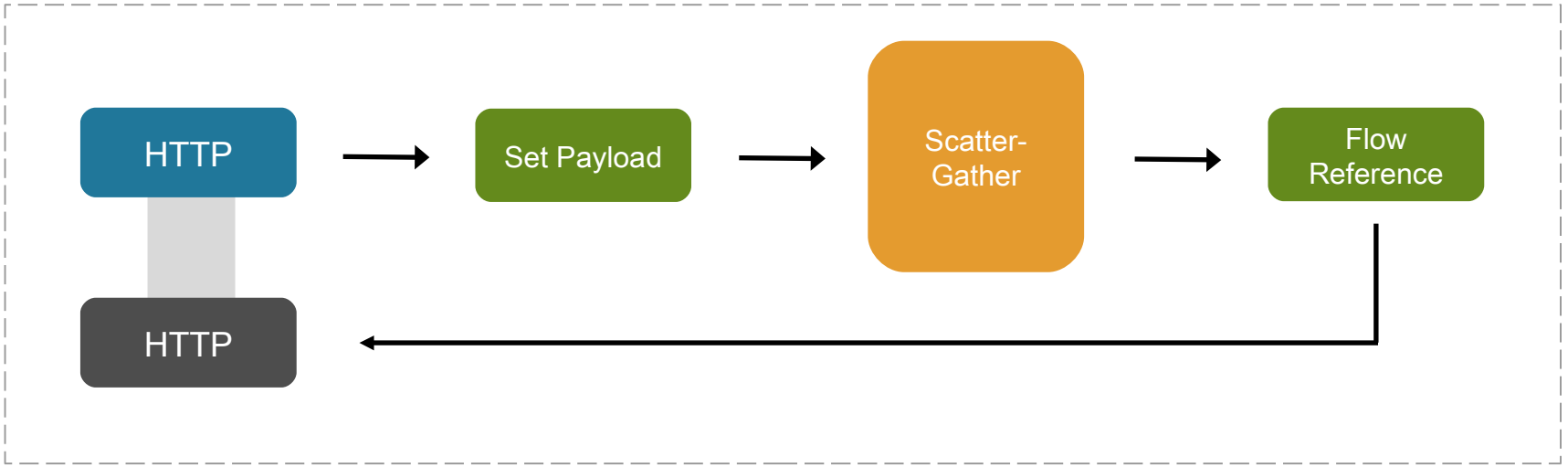
- Implement a Java transformer within a Mule Flow
- Implement an interface or extend an abstract class which will allow you to create this transformer
- Based on the price of each ticket option, return a sorted payload

Lab scenario



- Sort the ArrayList<Flight>s by **price**
- Multiple ways to sort this ArrayList
  - *Note that 'Flight' implements Comparable.*
- Ensure the sorted result is still returned as JSON

Lab scenario



CombineSortTransformSubflow

- MUA intends to call this logic from other flows in the future.
- Ensure it is callable via a flow reference

Lab Results