



# Mule Development Best Practices

- Objectives
- Audience
- Best Practices
  - Anypoint Studio
    - Studio Updates
  - Mule Runtime
  - Mule Projects in Anypoint Studio
    - Project Files Structure
    - XML Indentation and Formatting
    - Naming Conventions
    - Properties per Environment
    - Secure Application Properties
  - Mule Components
    - Dataweave
    - HTTP(s) Listener
    - HTTP Request
      - Request to APIs with self-signed certificates
      - HTTP Request Status Code Validator
      - HTTP Request to API with SNI validation
    - Web Service Consumer
    - Database Connector
    - APIKit
    - Validator
    - Flows Reusability
    - Java Custom Code/Scripting
    - Cache
  - Unit and Integration Testing
  - Mule Deploy Maven Plugin
  - Debugging & Troubleshooting
    - Wire Logging

---

# 1 Objectives

The following guide is intended to set and describe best practices and standards when Developing Mule Applications

---

## 2 Audience

- C4E Architect
- C4E Core Developers
- Architects
- Developers

---

## 3 Best Practices

### 3.1 Anypoint Studio

#### 3.1.1 Studio Updates

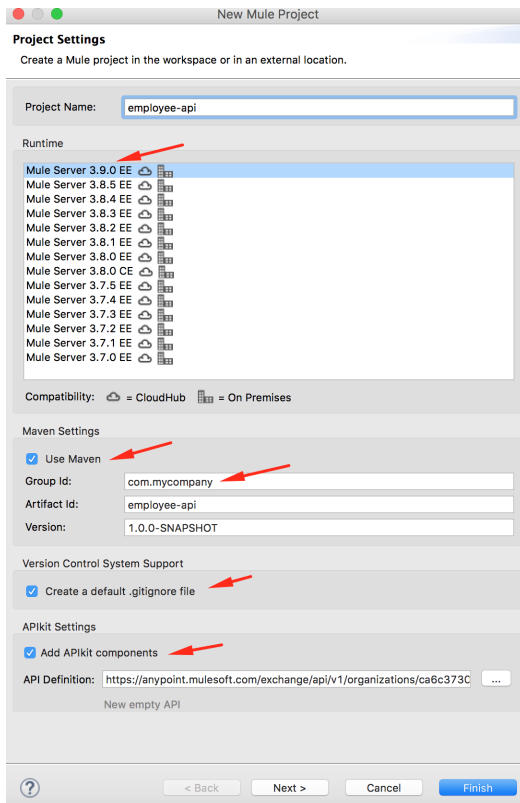
- Keep your Anypoint Studio up-to-date (Install Studio Updates when available).
- Use the recommended Java version for the specific Studio version.

### 3.2 Mule Runtime

- It's really important the coherence of the Mule Runtimes version within the SDLC. If you developed and tested your application using a specific Mule Runtime version, deploy it to a worker with the same Runtime version.
- Upgrade your projects constantly to the latest patch-version (Install the new Runtimes from the Studio Update Site or from the Mule Runtime Update Site).
- If you want to migrate your applications to a minor-new-version or a major-new-version follow the migration guides provided in the MuleSoft public documentation
  - <https://docs.mulesoft.com/release-notes/>

### 3.3 Mule Projects in Anypoint Studio

- Use Maven
- Include a .gitignore file
- Add APIKit components getting the RAML from Design Center

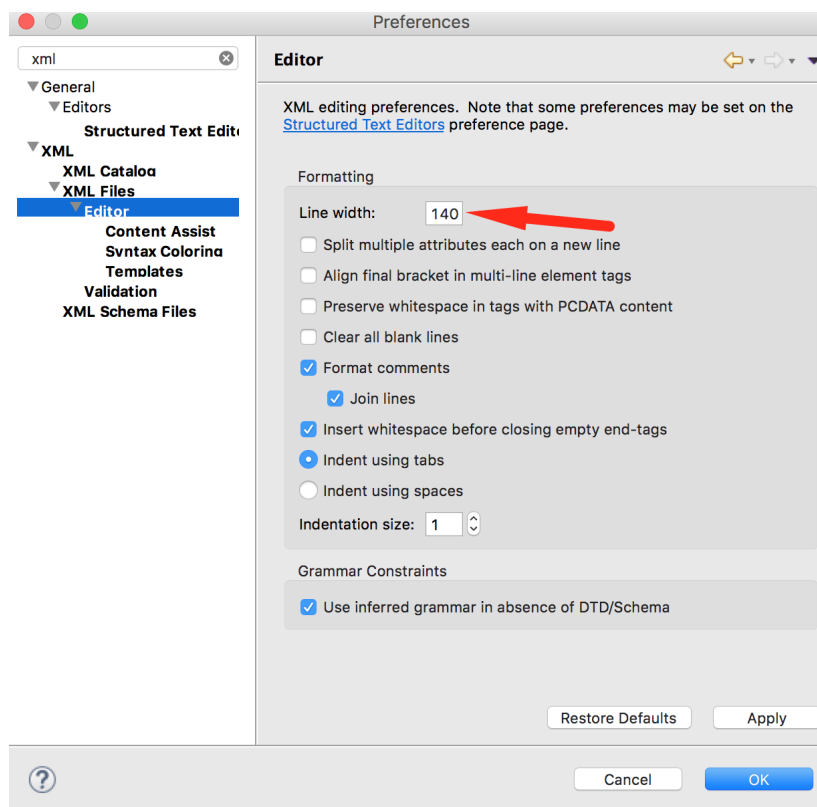


### 3.3.1 Project Files Structure

- Create a separate XML for the global elements (e.g. configuration elements)
- Create a separate XML for each use-case or resource-implementation
- Create separate XMLs for common structures/logic
- Create different packages for the resources (DataWeave, WSDLs, examples, etc)

### 3.3.2 XML Indentation and Formatting

- Define a line width in your Anypoint Studio XML editor preferences, e.g. 140



- Indent all your XML (mule XMLs, pom.xml, log4j2.xml, etc.) files before committing to the source code repository.

### 3.3.3 Naming Conventions

Define naming conventions, the following is just an example:

Category	Convention	Pattern	Examples	Comments
Mule Projects	kebab-case	{domain}-{subdomain}-{layer}-{type}	employee-sys-api customer-sys-api cloudhub-log-aggregator	The project name should match the API name In the case of polling/batch processes, not API related projects, is enough to use a descriptive name with kebab-case
Maven Group Id	fixed name	{com.your.company}	com.customer	
Mule XML files	kebab-case	{significant-name}.xml	employee-api.xml get-employee.xml common.xml	
Global Elements	kebab-case	{significant-name}	http-listener-config	

Category	Convention	Pattern	Examples	Comments
Mule properties files	dot separated	{domain}. {subdomain}.{layer}. {environment}.properties	employee.sys .dev.properties employee.sys .prod.properties	Lower case
Mule properties keys	dot separated	-	http.host core.utils.api. path	Lower case
Dataweave resources packages	package notation	{dw}.{classifier} {dw}.{plural} _{classifier}	dw dw.employee dw.employee. address dw.health_check	e.g. complete path: src/main/resources/dw/employee/address
Dataweave Scripts	kebab-case	{significant-name}.dwl	build-error-message.dwl	
JSON Examples	kebab-case	{method}-{significant-name}-{request response}-example	get-employee-response-example	
Mule flows	kebab-case	{significant-description}-flow {significant-description}-sub-flow	send-user-to-queue-flow validate-message-sub-flow	
API HTTP base path (On-Prem)	-	{domain}/ {subdomain}/{layer}/ {version}/*	/employee/ sys/v1/*	On-premise applications will be on the same server and the APIs will differentiate with the base path
API HTTP base path (Cloudhub)	-	-	/api/*	Each Cloudhub application has its own server

### 3.3.4 Properties per Environment

Use property placeholders to externalise properties. Use a placeholder in the location-name of the file to identify the environment.

```
<context:property-placeholder location="employee.api.${mule.env}.properties"/>
```

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/deploying-to-multiple-environments>

#### Update for Mule 4

This is not recommended with Mule 4. The recommended way is managing properties via ARM:

- <https://docs.mulesoft.com/mule4-user-guide/v/4.1/intro-configuration>
- <https://docs.mulesoft.com/mule-runtime/4.1/configuring-properties>



### 3.3.5 Secure Application Properties

Sensitive application properties such as access credentials and passwords can be obfuscated in CloudHub by using the Secure Application Properties feature.

Such feature can be used in conjunction with the Mule Credentials Vault that allows users to encrypt properties within the application properties files.

- Use AES algorithm with a CBC mode.  
Define 128 bits key, if you want to use 256 bits key, the **JCE Unlimited Strength Jurisdiction Policy Files** need to be installed in each workstation
  - For more information: <http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/mule-credentials-vault>
- Mule 4: <https://docs.mulesoft.com/mule-runtime/4.1/secure-configuration-properties>

For CloudHub deployment, there is no need for encryption. Use hidden properties instead: <https://docs.mulesoft.com/runtime-manager/secure-application-properties>

## 3.4 Mule Components

### 3.4.1 Dataweave

Use Dataweave:

- To transform/enrich the data, dates transformation and build error/response messages instead of custom scripting code (Java, Groovy, etc)
- When dealing with large payloads, include the directive `indent=false` to improve the client's parsing performance and to reduce the response payload size.
- Define the input content-type to avoid verbose messages in the Log e.g.

```
<dw:transform-message doc:name="Build Approval Message">
  <dw:input-payload mimeType="application/java"/>
  <dw:set-payload resource="classpath:dw/employee/build-employee-response.dwl"/>
</dw:transform-message>
```

- Use inline DataWeave scripts on the development phase, then pass it to an external file to have a clean XML and reusable scripts.



For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/dataweave>

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/dataweave-examples>
- Mule 4: <https://docs.mulesoft.com/mule4-user-guide/v/4.1/dataweave>

### 3.4.2 HTTP(s) Listener

Use the following placeholders depending on the deployment model

- `${http.port}` when deploying to CloudHub/Hybrid using HTTP (`http.port` is a reserved word in CloudHub that resolves to 8081)
- `${https.port}` when deploying to CloudHub/Hybrid using HTTPS (`https.port` is a reserved word in CloudHub that resolves to 8082)
- `${http.private.port}` when deploying to CloudHub using HTTP and using a Dedicated Load Balancer (`http.private.port` is a reserved word in CloudHub that resolves to 8091)
- `${https.private.port}` when deploying to CloudHub using HTTPS and using a Dedicated Load Balancer (`https.private.port` is a reserved word in CloudHub that resolves to 8092)

If HTTPS is required (e.g. in the DLB upstream protocol or for the CloudHub shared load balancer) use the HTTPS Listener with a TLS Context reference, configuring the key-store with key-pair values generated with Keytool.

```
<tls:context name="TLS_Context" doc:name="TLS Context">
  <tls:key-store type="jks" path="company-keystore.jks" alias="${jks.alias}" keyPassword="${jks.key.password}" password="${jks.password}"/>
</tls:context>
```

SSL can also be terminated at Load Balancer level when using a Dedicated Load Balancer in CloudHub or if there's an on-prem load balancer (e.g. F5) for hybrid scenarios. If this is the case, the HTTPS configuration can be avoided

- For more information:
  - Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/http-listener-connector#https-protocol-configuration>
  - Mule 4: <https://docs.mulesoft.com/connectors/http-connector>

### 3.4.3 HTTP Request

#### Request to APIs with self-signed certificates

When consuming APIs that are exposed with a self-signed certificate, use the HTTP Requester with a TLS Context reference, configuring the trust store with the self-signed certificate or including the `"insecure=true"` attribute to avoid the certificate validation (useful when running locally, don't use this in prod).

```
<tls:context name="tls.context.internal" doc:name="TLS Context">
  <tls:trust-store path="company-keystore.jks" password="${internal.jks.password}" type="jks"
  insecure="true"/>
</tls:context>
```

## HTTP Request Status Code Validator

By default the HTTP Request expects a 2xx status code from the target system, if there's a different status code in the response it will throw an Exception. If you want to control different responses by adding some business logic, you have to configure the range of status codes considered as 'valid', by doing that you can put some logic after the HTTP Request, e.g. a Choice to make decisions based on the status code.

```
<http:request config-ref="HTTP_Request_Configuration" .....>
  <http:success-status-code-validator values="200..599"/>
</http:request>
```

## HTTP Request to API with SNI validation

Sometimes, when consuming external APIs, in the server side of the external system there's an **SNI** validation. The HTTP Request component handles this automatically behind the scenes sending the 'Host' header including the port (e.g. [api.external.com:443](http://api.external.com:443)). Sometimes these external systems validate the host without the port, so you could end up with an error response. The solution to this is to override the behaviour, just including a "Host" header in the HTTP Requester with the expected host.

```
<http:header headerName="Host" value="api.external.com"/>
```

## Web Service Consumer

Import WSDLs including all references (XSDs) in your project, use the Webservice Consumer module and point the configuration to the local WSDL.

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/web-service-consumer>
- Mule 4: <https://docs.mulesoft.com/connectors/web-service-consumer>

## Database Connector

- Use parameterized queries
- Use dynamic queries only if you don't have another choice, remember to validate inputs previously (using dynamic queries could cause SQL-Injection vulnerability [https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection))

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/database-connector>
- Mule 4: <https://docs.mulesoft.com/connectors/db-connector-index>

## APIKit

Use APIKit features to generate the flows automatically

Use APIKit Exception Strategy to catch connector related exceptions

It's a good practice to disable the APIKit Console under all exposed environments (e.g. CloudHub environments, onPrem production environments) as:

- If the API Console is used without caution, you can affect real objects/data due to the fact that you don't have a 'mocking-services' option. When using API Portals (Exchange Public/Private Portal) you can enable mocking services.
- If an Exchange - API Spec is published on the Public Portal, the API Console is a redundancy.
- If there are no policies applied, the API Console is a risk, due to the point #1.

For more information:

- Mule 3: <https://docs.mulesoft.com/apikit/3.x/apikit-3-index>
- Mule 4: <https://docs.mulesoft.com/apikit/4.x/apikit-4-index>

## Validator

Use Validator instead of throwing exceptions with Groovy Scripts

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/validations-module>
- Mule 4: <https://docs.mulesoft.com/connectors/validation/validation-connector>

## Flows Reusability

Use Flow references to separate and reuse common logic. The flow diagram should be clear, showing the steps of the use case.

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/flow-reference-component-reference>
- Mule 4: <https://docs.mulesoft.com/mule-runtime/4.1/flowref-about>

## Java Custom Code/Scripting

Only if there is no other choice.

## Cache

Use Cache scope whenever possible, to wrap HTTP calls/flows/logic that obtain information that doesn't change frequently.

Cache-scope != caching in general.

- Use object-store for typical caching use-cases.
- Use cache-scope for response-replay.
  - be aware that cache-scope caches the entire message - perfect for response-replay, but this is not desirable for typical cache-data use-cases.

Define a TTL (time to live) based on an analysis, it shouldn't be arbitrary

For more information:

- Mule 3: <https://docs.mulesoft.com/mule-user-guide/v/3.9/cache-scope>
- Mule 4: <https://docs.mulesoft.com/mule-runtime/4.1/cache-scope>

## 3.5 Unit and Integration Testing

Create significant **Unit tests** using MUnit

Use **MUnit**, mock all Connectors/Transports that make an outbound call.

- If there are query parameters, URI parameters or headers, use the set-message component to mock the incoming parameters
- Use Dataweave to simulate requests

Naming conventions

- Test flows: {name-of-the-flow}-unit-test
- Files: {name-of-the-mule-file}-unit-test

For more information:

- MUnit for Mule 3: <https://docs.mulesoft.com/munit/v/1.3/>
- MUnit for Mule 4: <https://docs.mulesoft.com/munit/v/2.1/>

Create significant **Integration tests** using MUnit

Use **MUnit**, don't mock Connectors/Transports.

- Be sure to not affect real data or to rollback the transaction after executing operations like WRITE, UPDATE, DELETE.

Naming conventions

- Test flows: {name-of-the-flow}-integration-test
- Files: {name-of-the-mule-file}-integration-test

For more information:

- MUnit for Mule 3: <https://docs.mulesoft.com/munit/v/1.3/>
- MUnit for Mule 4: <https://docs.mulesoft.com/munit/v/2.1/>

## 3.6 Mule Deploy Maven Plugin

Prepare your applications for automated deployments (executed by CI/CD processes), including the Mule Maven Deploy Plugin

- Mule 3 Maven Plugin: CloudHub deployments: <https://docs.mulesoft.com/mule-user-guide/v/3.9/mule-maven-plugin#cloudhub>
- Mule 3 Maven Plugin: OnPrem deployments: <https://docs.mulesoft.com/mule-user-guide/v/3.9/mule-maven-plugin#runtime-manager>
- **Mule 4** Maven Plugin: <https://docs.mulesoft.com/mule-runtime/4.1/mmp-deployment-concept>

## 3.7 Debugging & Troubleshooting

Use Anypoint Studio Debugger to debug your Mule Applications.

For more information:

- Studio 6: <https://docs.mulesoft.com/anypoint-studio/v/6.5/studio-visual-debugger>
- Studio 7: <https://docs.mulesoft.com/anypoint-studio/v/7.1/visual-debugger-concept>

### 3.7.1 Wire Logging

To enable more detailed logging to see all the HTTP requests and responses, configure the following loggers in `src/main/resources/log4j2.xml`

Mule 3

```
<AsyncLogger name="org.mule.module.http.internal.HttpMessageLogger" level="DEBUG" />  
<AsyncLogger name="com.ning.http" level="DEBUG" />
```

Mule 4

```
<AsyncLogger name="org.mule.service.http.impl.service.HttpMessageLogger" level="DEBUG" />
```

Complete reference: <https://support.mulesoft.com/s/article/How-to-Enable-HTTP-Wire-Logging>

## 4 Error Handling in API-led Connectivity Approach Best Practices

- API-led Connectivity
- REST API Error handling
  - HTTP Status
  - Error Types
  - Error Response
  - APIKit Error Handling
  - Validation Module
  - Transaction Identifier
    - Transaction Identifier in API-led Connectivity
  - HTTP Request error
    - Response Validator Exception
    - Capture HTTP Response
    - Branch based on HTTP Status
    - Log System Error
- Example
  - Legacy Application
  - System Customer API
    - Successful Request
    - Customer Not Found
    - Bad Request from Legacy System
  - Experience Customer API
    - Successful Request
    - Customer Not Found
    - Bad Request from Legacy System
- Customize Error Handling

### 4.1 API-led Connectivity

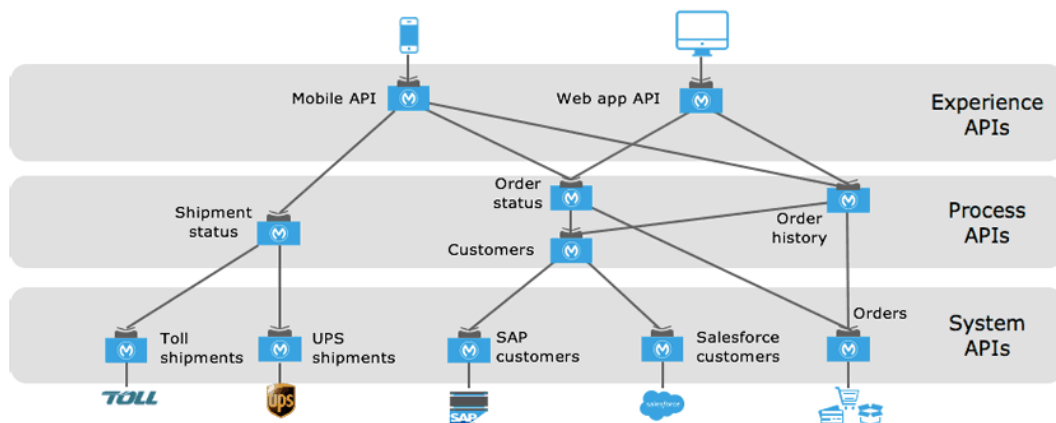
API-led connectivity is a methodical way to connect data to applications through reusable and purposeful APIs. These APIs are developed to play a specific role – unlocking data from systems, composing data into processes, or delivering an experience.

The APIs used in an API-led approach fall into three categories:

- **System APIs:** these usually access the core systems of record and provide a means of insulating the user from the complexity or any changes to the underlying systems. Once built, many users can access data without any need to learn the underlying systems and can reuse these APIs in multiple projects.
- **Process APIs:** These APIs interact with and shape data within a single system or across systems (breaking down data silos) and are created here without a dependence on the source systems from which that data originates, as well as the target channels through which that data is delivered.

- **Experience APIs:** Experience APIs are the means by which data can be reconfigured so that it is most easily consumed by its intended audience, all from a common data source, rather than setting up separate point-to-point integrations for each channel. An Experience API is usually created with API-first design principles where the API is designed for the specific user experience in mind.

A set of APIs following this approach will look like this:



Based on this, we should consider how we'll handle the possible errors that will be raised during the APIs execution. For example, a user consumes the Web App, calling the Experience API and creating an order. The Experience API calls the "Order Status" API (Process). This API consumes the "Customers" process API and retrieves a customer from the "Salesforce customers" API.

Now, if the customer sent by the user in the Web App has been deleted from Salesforce, the Salesforce connector will fail, and the "Salesforce customers" API will return an error. We have to analyze what should be returned to the Web App user, and how the error will be propagated. The Salesforce error message won't be designed for an end user and could contain sensitive information.

## 4.2 REST API Error handling

REST APIs use the Status of an HTTP response message to inform clients of their request's result.

### 4.2.1 HTTP Status

HTTP RFC defines over 40 standard status codes that can be used to convey the results of a client's request. The status codes are divided into the five categories presented here:

**1xx:** Informational - Communicates transfer protocol-level information.

**2xx:** Success - Indicates that the client's request was accepted successfully.

**3xx:** Redirection - Indicates that the client must take some additional action in order to complete their request.

**4xx:** Client Error - This category of error status codes points the finger at clients. It's commonly used for business validations and business errors.

**5xx:** Server Error - The server takes responsibility for these error status codes. These errors are not expected and should never happen.

The most common HTTP error codes are:



- **400 - Bad Request:** An error in the client request (Mostly due to validations)
- **401 - Unauthorized:** User can't be authenticated
- **403 - Forbidden:** The server cannot give access to the resource
- **404 - Not Found:** The resource defined in the URL doesn't exist
- **412 - Precondition Failed:** One of the validations in the request failed (Sometimes used instead of 400)
- **500 - Internal Server Error:** The server encountered an unexpected condition
- **502 - Bad Gateway:** The server, while acting as a gateway or proxy, received an invalid response from an inbound server
- **504 - Gateway Timeout:** The server tried to access an upstream service and it took more than the expected

## 4.2.2 Error Types

For the purposes of this guide, it's important to distinguish between two kinds of errors.

### Business Error

It's an error expected by the business domain, for example, a validation not being satisfied, a required field not being sent, or a record not being found. The API should notify the user about this error in a proper way.

### System Error

These kinds of errors aren't expected. Development support or operation teams have to be notified when one of this errors is raised. These exceptions could be thrown when a Database connection fails, an upstream service returns a server error, or due to a code error.

## 4.2.3 Error Response

A REST API should reply with an error code and a message to be shown to the consumer. To add more information, a description field could be returned. A transaction or request identifier could be useful to provide traceability, but we will see this later.

Here are some examples of how an error response should be:

### 404 - Not Found

```
{
  "code": "RESOURCE_NOT_FOUND",
  "message": "Resource not found",
  "description": "The resource with the ID 12 cannot be found",
  "transactionId": "803ec5202bd54576992d082da70e6338"
}
```

### 500 - Internal Server Error

```
{
  "code": "INTERNAL_SERVER_ERROR",
  "message": "Internal Server Error",
  "description": "Internal Server error",
}
```

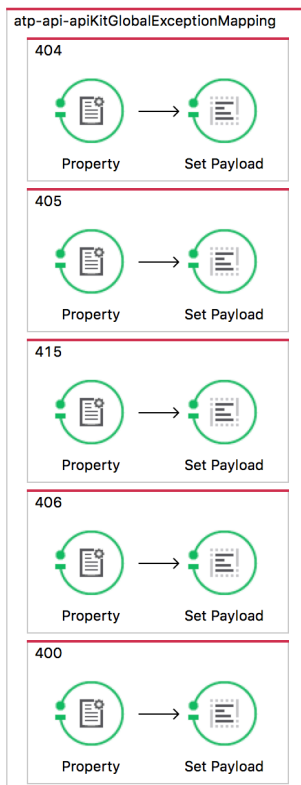
```

    "transactionId": "803ec5202bd54576992d082da70e6338"
  }

```

## 4.2.4 APIKit Error Handling

When creating a new APIKit project in Anypoint Studio, an error handler will be created by default:



This component will map a set of exceptions to specific error codes. The exceptions set by default are:

- `org.mule.module.apikit.exception.NotFoundException`
- `org.mule.module.apikit.exception.MethodNotAllowedException`
- `org.mule.module.apikit.exception.UnsupportedMediaTypeException`
- `org.mule.module.apikit.exception.NotAcceptableException`
- `org.mule.module.apikit.exception.BadRequestException`

These exceptions are thrown by the API Router when validating the HTTP request against a RAML. We can adapt this component to map most of the exceptions raised by our Mule Application and connectors. For example:

### APIKit Exception Handling

```

<apikit:mapping-exception-strategy name="api-kit-global-exception-strategy">
  <apikit:mapping statusCode="404">
    <apikit:exception value="org.mule.module.apikit.exception.NotFoundException"/>
    <apikit:exception value="com.mulesoft.services.exceptions.ResourceNotFoundException"/>
    [...]
  
```

```

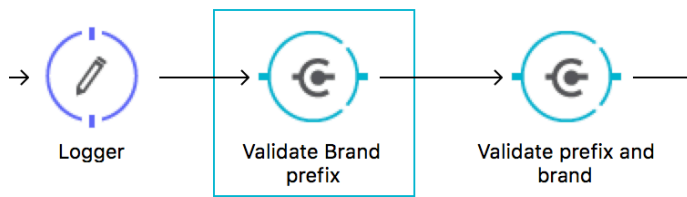
</apikit:mapping>
<apikit:mapping statusCode="405">
    <apikit:exception value="org.mule.module.apikit.exception.MethodNotAllowedException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="415">
    <apikit:exception value="org.mule.module.apikit.exception.UnsupportedMediaTypeException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="406">
    <apikit:exception value="org.mule.module.apikit.exception.NotAcceptableException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="400">
    <apikit:exception value="org.mule.module.apikit.exception.BadRequestException"/>
    <apikit:exception value="com.mulesoft.services.exceptions.PrefixDoesNotMatchException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="401">
    <apikit:exception value="org.springframework.security.authentication.BadCredentialsException"/>
    <apikit:exception value="org.mule.api.security.UnauthorisedException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="502">
    <apikit:exception value="java.net.ConnectException"/>
    <apikit:exception value="java.net.SocketException"/>
    <apikit:exception value="org.mule.module.http.internal.request.ResponseValidatorException"/>
    <apikit:exception value="com.mulesoft.services.exceptions.SoapErrorMessage"/>
    <apikit:exception value="org.mule.module.ws.consumer.SoapFaultException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="504">
    <apikit:exception value="java.util.concurrent.TimeoutException"/>
    [...]
</apikit:mapping>
<apikit:mapping statusCode="500">
    <apikit:exception value="java.lang.Exception"/>
    [...]
</apikit:mapping>
</apikit:mapping-exception-strategy>

```

## 4.2.5 Validation Module

Out of the box, Mule provides the Validation Module, useful to check business rules and throw an exception when they are not satisfied. The exception being thrown by this component generally represents a business error. The validation module lets us define a message that can be understood by the user.

The best practice is to create a custom exception and use it in the validation component. Then, this exception can be mapped to a specific HTTP Status.



Validate Brand prefix x Mule Debugger api APIkit Consoles (atp-price-api) Error Log Console api APIkit Consoles (dev-bridgestoneamericas-order) api APIkit Consoles (dev-bridgestoneamericas-order)

There are no errors.

Display Name: Validate Brand prefix

Basic Settings

Configuration:

Validator: Is True

Results Settings

Message: The partNumber prefix does not belong to any brand - partNumber: #[payload.partNumber] - prefix: #[flowVars.partNumberPrefix]

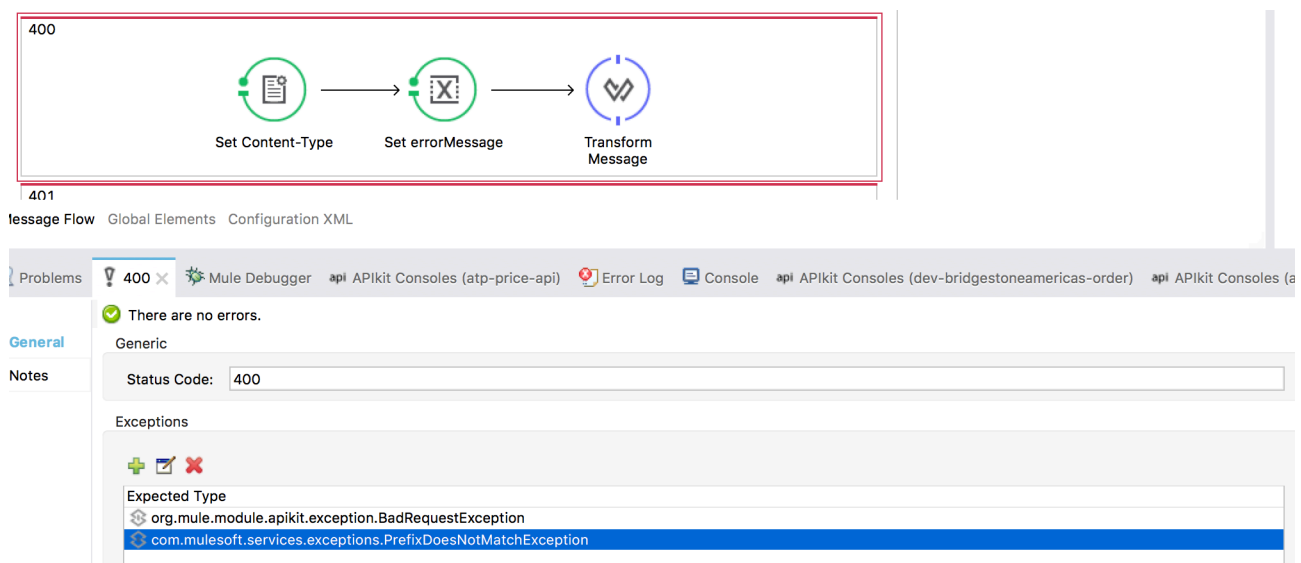
Exception Class: com.mulesoft.services.exceptions.PrefixDoesNotMatchException

Validator Settings

Expression: #[flowVars.partNumberPrefix == "" || flowVars.brandFromPrefix != ""]

Referenced Libraries

- src/main/app (Flows)
- src/main/api
- src/main/java
  - com.mulesoft.services.exceptions
    - PrefixDoesNotMatchException.java
    - ResourceNotFoundException.java
    - SoapErrorMessage.java
- src/main/resources
- src/test/java
- src/test/munit
- src/test/resources
- JRE System Library [Java SE 8 [1.8.0\_151]]
- Mule Server 3.9.0 EE
- src
- target
- mule-project.xml [Mule Server 3.9.0 EE]
- pom.xml



The screenshot shows the MuleSoft IDE interface. At the top, a message flow diagram is visible, enclosed in a red box. The diagram consists of three steps: 'Set Content-Type', 'Set errorMessage', and 'Transform Message'. Below the diagram, the '401' status code is displayed. The main panel shows the 'Error Log' tab, which indicates 'There are no errors.' The 'Status Code' is set to '400'. The 'Exceptions' section shows a list of expected types, with 'com.mulesoft.services.exceptions.PrefixDoesNotMatchException' selected.

Then, the service will return a response similar to this one:

**Error Response - Bad Request**

```
{
  "code": "BAD_REQUEST",
  "message": "Bad request",
  "description": "The partNumber prefix does not belong to any brand - partNumber: AZ 000009 - prefix: AZ.",
  "transactionId": "d713c1187f1148c7b224403308b5596c"
}
```

## 4.2.6 Transaction Identifier

For security reasons, it is not recommended to return to the user all the information we have when a system error is raised. We should only provide basic information, the error type, or an error code. The stack trace and the exception chain has to be accessible only to the developers, operations, support, or admin teams. The information can be sent to the logs or a Cloudhub notification.

If the user receives an error message, it will be very difficult to find the exception log or Cloudhub notification that describes what happened with that specific request. Creating a transaction or request identifier is a good practice. This transactionId can be logged and sent in the Cloudhub notification. Then, the user will have this transactionId, and the entire execution of the user request can be obtained from the logs.

The identifier should be generated at the beginning of the HTTP Request, so any validation error will log and return a transaction identifier.

For example, if the API returns this error message:

**Error Response - Bad Request**


```
{
```

```


"code": "BAD_GATEWAY",
"message": "Bad Gateway",
"description": "Upstream service failed",
"transactionId": "3af8be1c58694ce892c74eb630bef14f"
}

```


We can go to the application log and look for the transactionId:

● tirehub-atp-api-qa-v1 | Search Results 

3af8be1c58694ce892c74eb630bef14f X Search Advanced ▾

17:22:15.780	05/09/2018	Worker-0	[tirehub-atp-api-qa-v1].https-listener-config.worker.01	INFO
transactionId: 3af8be1c58694ce892c74eb630bef14f - Request - method: GET - URI: /api/inventory?dealerCode=&supplierCode=&partNumber=BR%20000009&locationId=100&brand=&width=&aspectRatio=&rimDiameter=&inputQuantity=				
17:22:15.782	05/09/2018	Worker-0	[tirehub-atp-api-qa-v1].https-listener-config.worker.01	INFO
transactionId: 3af8be1c58694ce892c74eb630bef14f - Retrieve Inventory - dealerCode: - supplierCode: - partNumber: BR 000009 - locationId: 100 - brand: - width: - aspectRatio: - rimDiameter: - inputQuantity:				
17:22:15.783	05/09/2018	Worker-0	[tirehub-atp-api-qa-v1].https-listener-config.worker.01	INFO
transactionId: 3af8be1c58694ce892c74eb630bef14f - Retrieve Inventory - brand: - origPartNumber: BR 000009 - partNumber: 000009 - partNumberPrefix: BR - brandFromPrefix: BRIDGESTONE				
	17:22:17.073	05/09/2018	Worker-0	[tirehub-atp-api-qa-v1].https-listener-config.worker.01 ERROR
transactionId: 3af8be1c58694ce892c74eb630bef14f - System error - class: class org.mule.module.ws.consumer.SoapFaultException - message: Bad Gateway: Server was unable to process request. ---> Object reference not set to an instance of an object.. - details: org.mule.module.ws.consumer.SoapFaultException: Server was unable to process request. ---> Object reference not set to an instance of an object.. - org.apache.cxf.binding.soap.SoapFault: Server was unable to process request. ---> Object reference not set to an instance of an object.				
17:22:17.263	05/09/2018	Worker-0	[tirehub-atp-api-qa-v1].https-listener-config.worker.01	INFO
transactionId: 3af8be1c58694ce892c74eb630bef14f - System error - Notification sent to Cloudhub				

If your application sends a Cloudhub notification for each system error, you can trace the Cloudhub notification matching the transactionId too.



tirehub-atp-api-qa-v1.cloudhub.io Today 05:22 pm Insight X

TransactionId: 3af8be1c58694ce892c74eb630bef14f - System error - Class: class org.mule.module.ws.consumer.SoapFaultException - Message: [more](#)

## Transaction Identifier in API-led Connectivity

When following the API-led Connectivity approach, Mule APIs will consume another APIs in the Anypoint Platform. The consumer will call the API in the Experience layer, while the error could be originated in the System API. For example, in the case explained at the beginning of this document, the execution flow will be:

User → UX Web App API → Process Order Status API → Process Customer API → System Salesforce Customer API

If each API in this flow generates its own transaction identifier, we won't be able to match an error received by the user with an exception thrown by the System Salesforce API. This is why, in the identifier generation, we expect to receive this identifier as a header in the HTTP Request. Now, the UX API will generate the TransactionId, will pass it as a header to the Process API, and so on.

To do this, the TransactionId generation expression looks like this:

**Mule Expression Language - Set Transaction ID**

```
<set-variable variableName="transactionId" value="#[(message.inboundProperties['x-transaction-id'] != null) ? message.inboundProperties['x-transaction-id'] : message.rootId.replace('-', ' ')]" doc:name="Set Transaction ID" />
```

The API will expect a header name "x-transaction-id", if it isn't sent, a new UUID will be adopted (pre-computed, from the message root id).



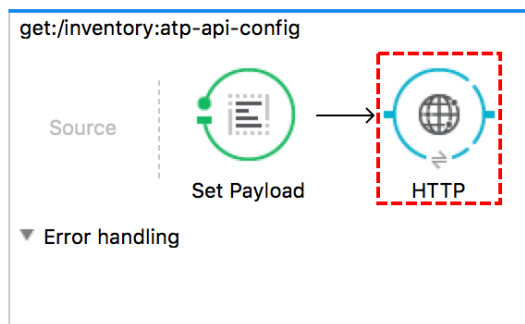
A nice side effect of using the message root id is that this is the same value used in Mule for Insights (in case you are using that functionality)

## 4.2.7 HTTP Request error

### Response Validator Exception







By default, the HTTP Connector validates the status of the HTTP Response. When the status is **4XX** or **5XX**, it throws an exception of the type "ResponseValidatorException". After the exception is thrown, the payload will contain the HTTP Response.

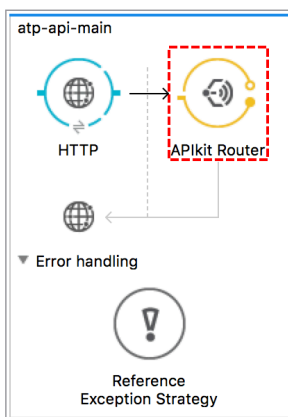
One caveat of the error handling is that the Payload is lost when the default exception handler rethrows an exception. In this case, the payload is really important, because it contains the error description returned by the REST API being consumed.



Message Flow Global Elements Configuration XML









Name	Value	Type
▶  <b>DataType</b>	SimpleDataType{type=org.mule.tran...	org.mule.transformer.types.SimpleD...
▶  <b>Exception</b>	null	
▶  <b>exceptionThrown</b>	org.mule.module.http.internal.reque...	org.mule.module.http.internal.reque...
▶  <b>Message</b>		org.mule.DefaultMuleMessage
▶  <b>Message Processor</b>	/get:/inventory:atp-api-config/proce...	org.mule.module.http.internal.reque...
▶  <b>Payload (mimeType="*/*", enc...</b>	org.glassfish.grizzly.utils.BufferInput...	org.glassfish.grizzly.utils.BufferInput...



atp-api-console

Message Flow Global Elements Configuration XML

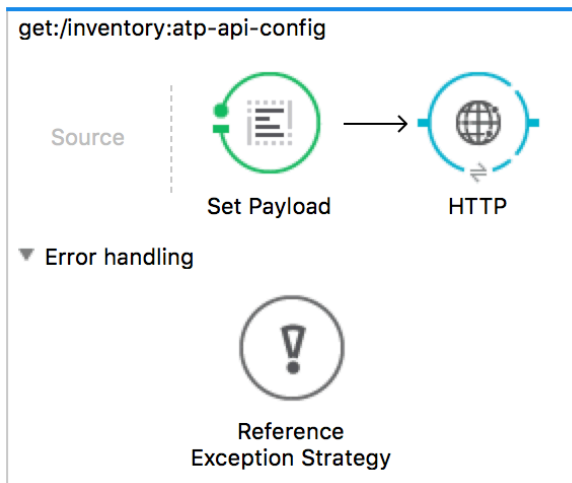


Name	Value	Type
▶  <b>DataType</b>	SimpleDataType{type=org.mule.tran...	org.mule.transformer.types.SimpleD...
▶  <b>Exception</b>	null	
▶  <b>exceptionThrown</b>	org.mule.module.http.internal.reque...	org.mule.module.http.internal.reque...
▶  <b>Message</b>		org.mule.DefaultMuleMessage
▶  <b>Message Processor</b>	APIKit Router	org.mule.module.apikit.Router
▶  <b>Payload (mimeType="*/*", enc...</b>	{NullPayload}	org.mule.transport.NullPayload

Try to use sub-flows instead of flows when it possible, given that the HTTP Response should reach the APIKit Exception Handler.

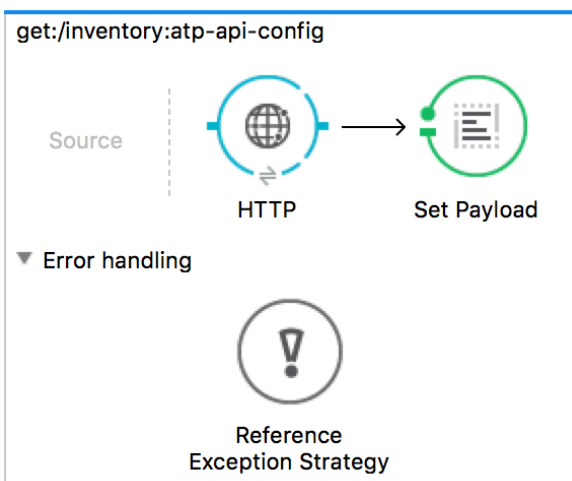


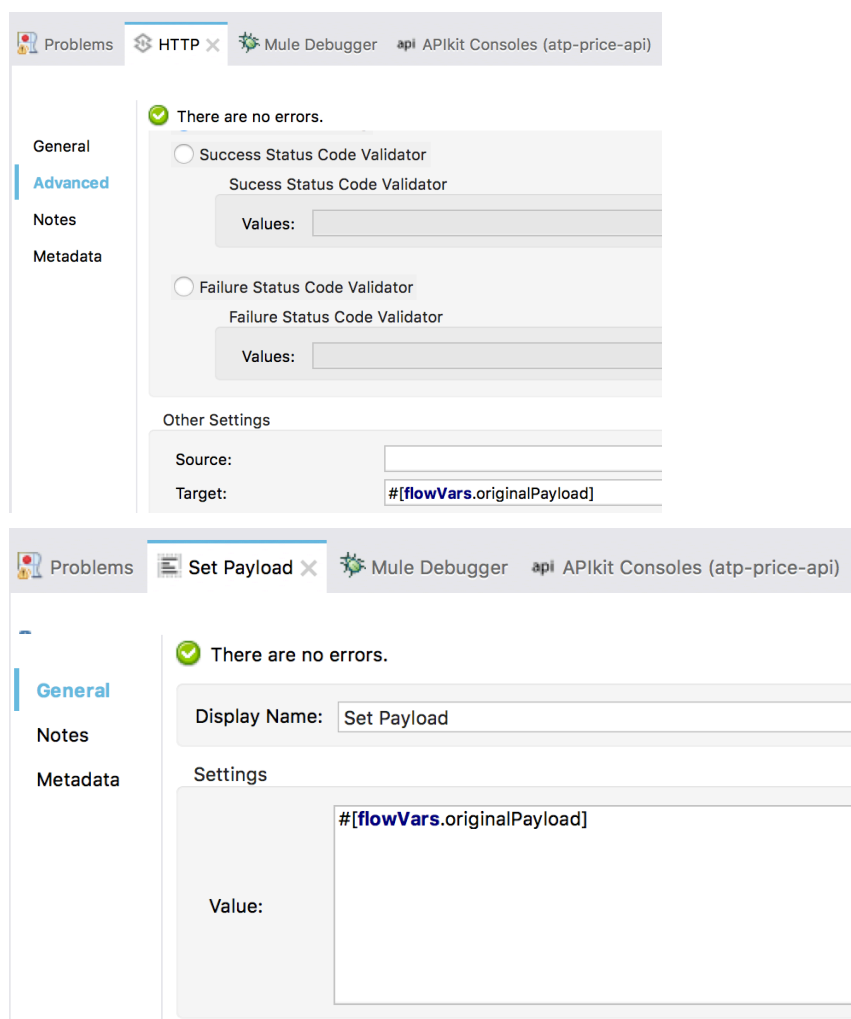
It's a good practice to add a Reference Exception Strategy at the Endpoint flow level:



## Capture HTTP Response

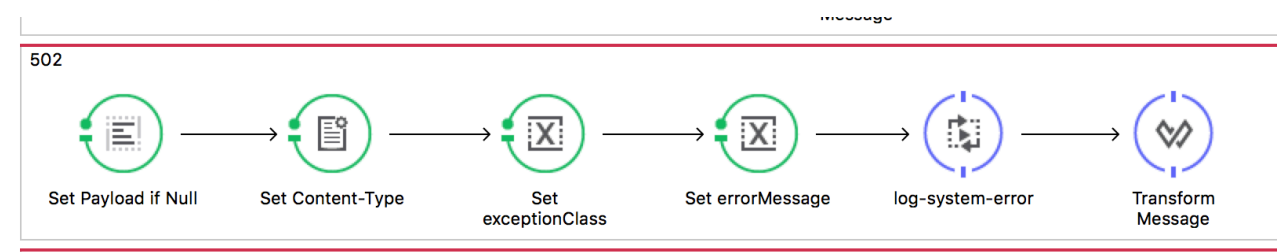
Sometimes we must use a flow and the payload containing the HTTP Response is lost when being raised to the Error Handler. One way to keep the HTTP Response is storing it in a variable.





When the HTTP Response is received, being a success or a failure response, it will be stored in the variable "originalPayload". If the request is successful, Mule will restore the payload.

If the request fails and the `ResponseValidatorException` is thrown, the exception handler will receive the null payload and the flow variable. That's why we added a Set Payload component that will put the variable in the payload if it exists.



### Set Payload if Null

```
#[flowVars.originalPayload != null ? flowVars.originalPayload : payload]
```

## Branch based on HTTP Status

The APIKit exception mapping relates a list of exception types to one status code.

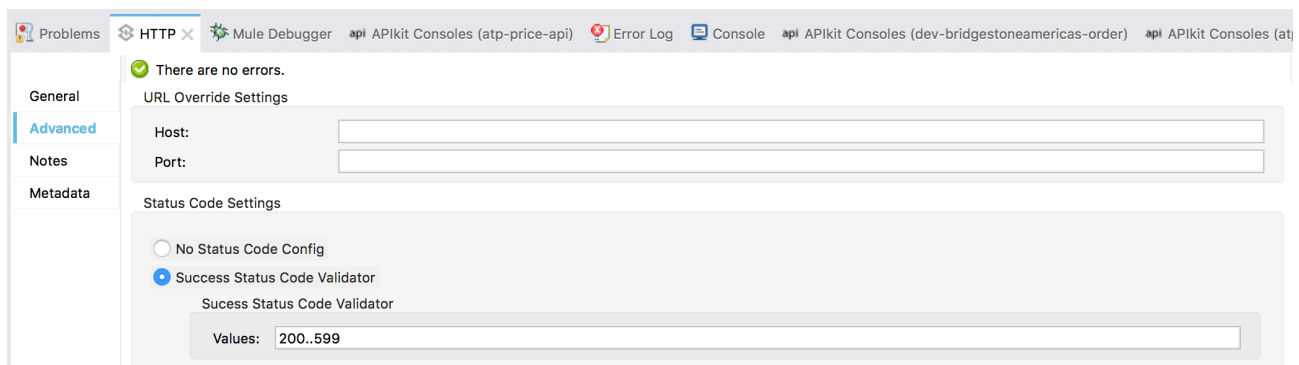
Based on the use case, we would want to return different status codes based on the response of the inner HTTP Request, e.g., going back to the previous example:

The "Process Customers API" calls "Salesforce Customers API". When the last one returns an HTTP 400, it means that the process API is doing something wrong and it will require the intervention of a developer. So, the Process Customers API will return HTTP 500.

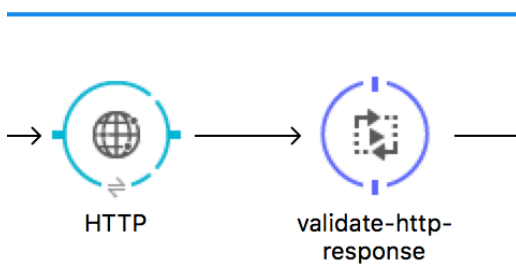
Now, when the Salesforce API returns an HTTP 404, the Process layer wants to return a 404, because it's an expected behaviour, the customer doesn't exist. This isn't possible with the `ResponseValidatorException`. In both cases, the exception is of the same type, and one Exception can be mapped only to one HTTP Status.

That's why we decide to avoid the `ResponseValidatorException` and make a custom validation.

First, we configure the HTTP Connector so that any status between 200 and 599 means a valid response.



Right after the HTTP component, we validate the response and use the Validation Module



### Flow validate-http-response

```
<sub-flow name="validate-http-response">
  <choice doc:name="Choice">
    <when expression="#[message.inboundProperties['http.status'] == 404]">
      <dw:transform-message doc:name="Transform Message">
        <dw:input-payload mimeType="application/java"/>
        <dw:set-variable variableName="errorMessage"><![CDATA[%dw 1.0
%output application/java
---

```

```

"HTTP " ++ (inboundProperties['http.status' default ""]) ++ ": " ++ payload]]></dw:set-variable>
    </dw:transform-message>
    <validation:is-true message="#[flowVars.errorMessage]"
exceptionClass="com.mulesoft.services.exceptions.HttpNotFoundException" expression="#[false]"
doc:name="Validation"/>
    </when>
    <when expression="#[message.inboundProperties['http.status'] >= 400]">
        <dw:transform-message doc:name="Transform Message">
            <dw:input-payload mimeType="application/java"/>
            <dw:set-variable variableName="errorMessage"><![CDATA[%dw 1.0
%output application/java
---
"HTTP " ++ (inboundProperties['http.status' default ""]) ++ ": " ++ payload]]></dw:set-variable>
            </dw:transform-message>
            <validation:is-true message="#[flowVars.errorMessage]"
exceptionClass="com.mulesoft.services.exceptions.HttpResponseException" expression="#[false]"
doc:name="Validation"/>
                </when>
                <otherwise>
                    <logger message="HTTP Response: #[message.inboundProperties['http.status']]" level="TRACE"
doc:name="Logger"/>
                </otherwise>
            </choice>
        </sub-flow>

```

In this flow, we obtain the error message from the HTTP Response and send this as part of the exception message. Based on the HTTP Status, we throw two exceptions: **HttpNotFoundException** and **HttpResponseException**. Previously, we created these Java exceptions, e. g.:

#### HttpResponseException

```

package com.mulesoft.services.exceptions;

public class HttpResponseException extends Exception {

    private static final long serialVersionUID = -7859285731473424800L;

    public HttpResponseException() {
    }

    public HttpResponseException(String message) {
        super(message);
    }

    public HttpResponseException(Throwable cause) {
        super(cause);
    }

    public HttpResponseException(String message, Throwable cause) {
        super(message, cause);
    }

}

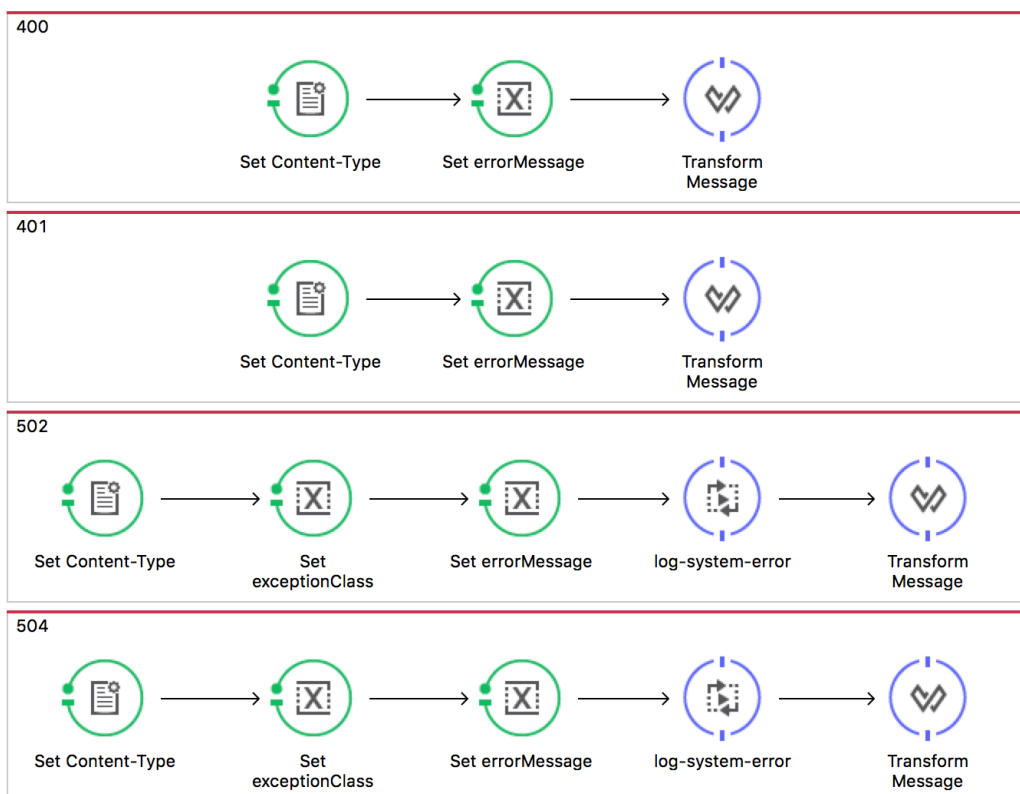
```

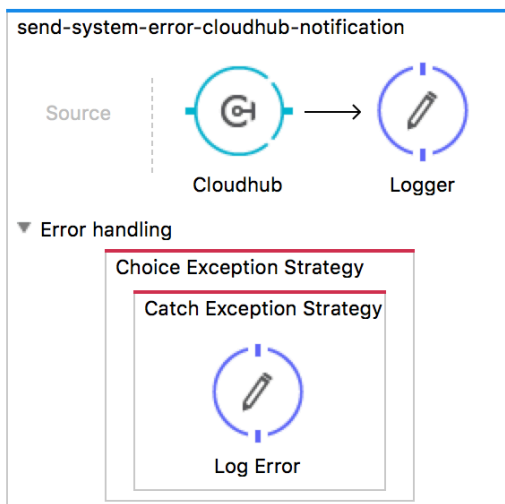
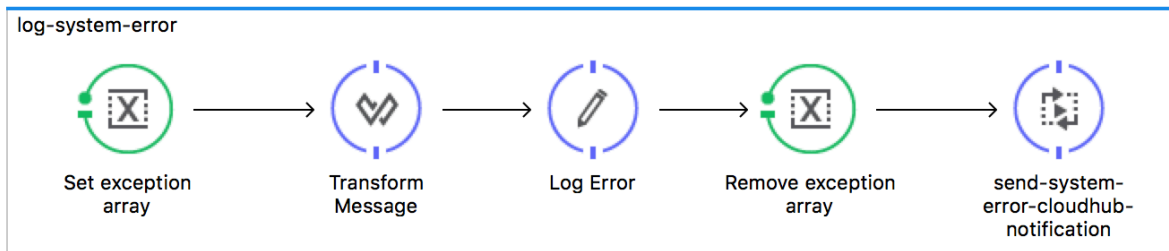
Then, we set these exceptions in the APIKit exception mapping, to return a 404 (Not Found) or 500 (Internal Server Error).

## Log System Error

As it was explained before, we differentiate two kinds of exceptions: business and system errors. In the example shown before `HttpNotFoundException` and `HttpResponseException` are business and system exceptions respectively.

For business errors, it's not a good practice to log the entire stack trace, given that they could happen frequently and don't demand an intervention from the developers, they are expected errors. At the other side, for each system exception being raised, we should log as much information as it's possible. This is why the APIKit exception mapping sections for system errors contain a reference to a flow name "log-system-error".





This flow will create an array containing all the levels of the exception chain. Sometimes the message describing the real issue is only in the inner exceptions. We do this with a MEL Expression because Dataweave can't access the exceptions. Then, a Dataweave component will create a String containing the information obtained from all the exceptions and log it. In the end, this information will be sent in a Cloudhub notification.

#### Log System Error

```

<sub-flow name="log-system-error">
  <set-variable variableName="exceptionChain" value="#[exception.?toString(),exception.?cause.?
  toString(),exception.?cause.?cause.?toString(),exception.?cause.?cause.?cause.?toString()]" doc:name="Set
  exception array"/>
  <dw:transform-message doc:name="Transform Message">
    <dw:set-variable variableName="errorDetailedMessage"><![CDATA[%dw 1.0
%output application/java
---
(flowVars.exceptionChain filter $ != null) joinBy "\n- "]]></dw:set-variable>
  </dw:transform-message>
  <logger message="transactionId: #[flowVars.transactionId] - System error - class:
#[flowVars.exceptionClass] - message: #[flowVars.exceptionMessage] - details:
#[flowVars.errorDetailedMessage]" level="ERROR" doc:name="Log Error"/>
  <remove-variable variableName="exceptionChain" doc:name="Remove exception array"/>
  <flow-ref name="send-system-error-cloudhub-notification" doc:name="send-system-error-cloudhub-
  notification"/>
</sub-flow>
<flow name="send-system-error-cloudhub-notification">

```

```
<cloudhub:create-notification config-ref="Cloudhub__Inhered_Token_Authentication" domain="system-
customer-api.services.mulesoft.com" message="TransactionId: #[flowVars.transactionId] - System error -
Class: #[flowVars.exceptionClass] - Message: #[flowVars.exceptionMessage]" priority="ERROR"
doc:name="Cloudhub">
    <cloudhub:custom-properties ref="#[&quot;transactionId&quot;; flowVars.transactionId,
&quot;exceptionClass&quot;; flowVars.exceptionClass, &quot;errorMessage&quot;; flowVars.exceptionMessage,
&quot;errorDetailedMessage&quot;; flowVars.errorDetailedMessage]"/>
</cloudhub:create-notification>
<logger message="transactionId: #[flowVars.transactionId] - System error - Notification sent to
Cloudhub" level="INFO" doc:name="Logger"/>
<choice-exception-strategy doc:name="Choice Exception Strategy">
    <catch-exception-strategy when="exception.causedBy(java.lang.NullPointerException)"
logException="false" doc:name="Catch Exception Strategy">
        <logger message="transactionId: #[flowVars.transactionId] - System error - Error sending
notification to Cloudhub" level="ERROR" doc:name="Log Error"/>
    </catch-exception-strategy>
</choice-exception-strategy>
</flow>
```

## 4.3 Example

This Github repository contains three Mule Applications that serve as an example of how error handling should be applied when we have multiple layers of APIs.

<https://github.com/mulesoft-consulting/api-led-error-handling-example>

These applications are:

- **Legacy Application:** It's an application that will return an error or a not found error for specific customer ids. The idea behind this application is to mock a legacy system that will return different error messages or code without the standard HTTP Status.
- **System Customer API:** It's a system level API that will provide connectivity to the legacy system described before. Based on the error codes, it will return an HTTP Status. It can be 200, 404 or 500.
- **Experience Customer API:** It represents the API consumed by the user. It validates the Status of the HTTP Response manually and throws custom exceptions to return status 404 or 500.

Each of these applications runs on different ports, so the three of them can be run simultaneously from Anypoint Studio. The ports are 8083, 8082 and 8081.

### 4.3.1 Legacy Application

This application represents a legacy system that we're integrating with. Some legacy systems don't use the HTTP Status codes and it's why we have to create a System API.

#### Legacy - Successful Request

```
$ curl -i http://localhost:8081/customers/1
HTTP/1.1 200
Transfer-Encoding: chunked
```

```
Content-Type: application/json; charset=UTF-8
Date: Wed, 16 May 2018 21:02:57 GMT

{
  "id": "1",
  "name": "Name"
}
```

**Legacy - Customer Not Found**

```
$ curl -i http://localhost:8081/customers/2
HTTP/1.1 200
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8
Date: Wed, 16 May 2018 21:10:07 GMT

{
  "error": "NOT_FOUND"
}
```

**Legacy - Bad Request**

```
$ curl -i http://localhost:8081/customers/-1
HTTP/1.1 200
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8
Date: Wed, 16 May 2018 21:11:16 GMT

{
  "error": "BAD_REQUEST"
}
```

## 4.3.2 System Customer API

It's the API at the system layer. It will parse the error messages returned by the legacy system and return the proper HTTP Status.

### Successful Request

**System - Successful Request**

```
$ curl -i http://localhost:8082/api/customer/1
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8
Date: Wed, 16 May 2018 21:20:33 GMT

{
  "id": "1",
```



```
"name": "Name"
}
```

## Customer Not Found

### System - Not Found

```
$ curl -i http://localhost:8082/api/customer/2
HTTP/1.1 404 Not Found
Transfer-Encoding: chunked
Content-Type: application/json
Date: Wed, 16 May 2018 21:21:37 GMT

{
  "code": "RESOURCE_NOT_FOUND",
  "message": "Resource not found",
  "description": "Customer with this id was not found.",
  "transactionId": "aa5b4c5df9ca481fb9a6bf6f7f9c3229"
}
```

### Log - Customer Not Found

```
INFO 2018-05-16 18:21:37,549 [[system-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: aa5b4c5df9ca481fb9a6bf6f7f9c3229 - Request -
method: GET - URI: /api/customer/2
ERROR 2018-05-16 18:21:37,572 [[system-customer-api].httpListenerConfig.worker.02]
org.mule.module.apikit.MappingExceptionListener:
*****
Message           : Customer with this id was not found.
Payload           : {error=NOT_FOUND}
Payload Type      : java.util.LinkedHashMap
Element           : /get:\customer\{customerId}:apiConfig/processors/2 @ system-customer-api
-----
Root Exception stack trace:
com.mulesoft.services.HttpNotFoundException: Customer with this id was not found
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
    at
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
    at
    at org.mule.extension.validation.internal.DefaultExceptionHandler$DirectMessageConstructorDelegate.createException(DefaultExceptionHandler.java:202)
    [...]
```

## Bad Request from Legacy System

### System - Bad Request

```
$ curl -i http://localhost:8082/api/customer/-1
HTTP/1.1 500 Server Error
Transfer-Encoding: chunked
Content-Type: application/json
Date: Wed, 16 May 2018 21:25:22 GMT
Connection: close

{
  "code": "INTERNAL_SERVER_ERROR",
  "message": "Internal Server Error",
  "description": "Internal Server error",
  "transactionId": "9f01af17641c4ec193718e4e8e8c0816"
}
```

### Log - Bad Request

```
INFO 2018-05-16 18:25:22,310 [[system-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: 9f01af17641c4ec193718e4e8e8c0816 - Request -
method: GET - URI: /api/customer/-1
ERROR 2018-05-16 18:25:22,339 [[system-customer-api].httpListenerConfig.worker.02]
org.mule.module.apikit.MappingExceptionListener:
*****
Message           : Error from Upstream Service: BAD_REQUEST.
Payload           : {error=BAD_REQUEST}
Payload Type      : java.util.LinkedHashMap
Element           : /get:/customer/{customerId}:apiConfig/processors/3 @ system-customer-api
-----
Root Exception stack trace:
com.mulesoft.services.HttpErrorException: Error from Upstream Service: BAD_REQUEST
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
    at
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
    at
    at org.mule.extension.validation.internal.DefaultExceptionFactory$DirectMessageConstructorDelegate.createException(DefaultExceptionFactory.java:202)
    [...]
ERROR 2018-05-16 18:25:22,341 [[system-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: 9f01af17641c4ec193718e4e8e8c0816 - System
error - class: class org.mule.api.MessagingException - message: Internal Server Error:Error from Upstream
Service: BAD_REQUEST. - details: org.mule.api.MessagingException: Error from Upstream Service: BAD_REQUEST.
- com.mulesoft.services.HttpErrorException: Error from Upstream Service: BAD_REQUEST
ERROR 2018-05-16 18:25:22,344 [[system-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: 9f01af17641c4ec193718e4e8e8c0816 - System
error - Error sending notification to Cloudhub
```

### 4.3.3 Experience Customer API

It's the API consumed by the user, based on the Status returned by the System API, will return an HTTP Status to the user. When the response received is a server error, it will send the error message received to Cloudhub (Only works for Cloudhub deployments) and log it with the stack trace.

#### Successful Request

##### Experience - Successful Request

```
$ curl -i http://localhost:8083/api/customer/1
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/json;charset=UTF-8
Date: Thu, 17 May 2018 12:48:14 GMT

{
  "customerId": "1",
  "customerName": "Name"
}
```

#### Customer Not Found

##### Experience - Not Found

```
$ curl -i http://localhost:8083/api/customer/2
HTTP/1.1 404 Not Found
Transfer-Encoding: chunked
Content-Type: application/json
Date: Thu, 17 May 2018 12:55:28 GMT

{
  "code": "RESOURCE_NOT_FOUND",
  "message": "Resource not found",
  "description": "Customer with this id was not found..",
  "transactionId": "9d7ff24a37db463b920fe7f7cb0362bd"
}
```

##### Log - Customer Not Found

```
INFO 2018-05-17 09:54:30,327 [[experience-customer-api].httpListenerConfig.worker.01]
org.mule.api.processor.LoggerMessageProcessor: transactionId: 9d7ff24a37db463b920fe7f7cb0362bd - Request -
method: GET - URI: /api/customer/2
ERROR 2018-05-17 09:55:28,207 [[experience-customer-api].httpListenerConfig.worker.01]
org.mule.module.apikit.MappingExceptionListener:
*****
Message : Customer with this id was not found..
Payload : com.mulesoft.weave.reader.DefaultSeekableStream@2fb8e4d0
```

```

Element          : /get:\customer\[customerId]:apiConfig/processors/1/validate-http-response/
subprocessors/0/0/1 @ experience-customer-api
-----
Root Exception stack trace:
com.mulesoft.services.HttpNotFoundException: Customer with this id was not found.
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
    at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
[...]
```

## Bad Request from Legacy System

### Experience - Bad Request

```

$ curl -i http://localhost:8083/api/customer/-1
HTTP/1.1 500 Server Error
Transfer-Encoding: chunked
Content-Type: application/json
Date: Thu, 17 May 2018 13:25:49 GMT
Connection: close

{
  "code": "INTERNAL_SERVER_ERROR",
  "message": "Internal Server Error",
  "description": "Internal Server error",
  "transactionId": "e674266e8e7745dba5c36d91bd612f42"
}
```

### Log - Bad Request

```

INFO 2018-05-17 10:25:28,863 [[experience-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: e674266e8e7745dba5c36d91bd612f42 - Request -
method: GET - URI: /api/customer/-1
ERROR 2018-05-17 10:25:48,887 [[experience-customer-api].httpListenerConfig.worker.02]
org.mule.module.apikit.MappingExceptionListener:
*****
Message          : HTTP 500: Internal Server error.
Payload          : com.mulesoft.weave.reader.DefaultSeekableStream@5cb05e5f
Element          : /get:\customer\[customerId]:apiConfig/processors/1/validate-http-response/
subprocessors/0/1/1 @ experience-customer-api
-----
Root Exception stack trace:
com.mulesoft.services.HttpErrorException: HTTP 500: Internal Server error
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
    at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
```

```

        at
org.mule.extension.validation.internal.DefaultExceptionFactory$DirectMessageConstructorDelegate.createException(DefaultExceptionFactory.java:202)
        at
org.mule.extension.validation.internal.DefaultExceptionFactory.createException(DefaultExceptionFactory.java:88)
        at
org.mule.extension.validation.internal.DefaultExceptionFactory.createException(DefaultExceptionFactory.java:102)
        at org.mule.extension.validation.internal.ValidationSupport.validateWith(ValidationSupport.java:45)
        at
org.mule.extension.validation.internal.CommonValidationOperations.isTrue(CommonValidationOperations.java:56)
[...]
ERROR 2018-05-17 10:25:48,969 [[experience-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: e674266e8e7745dba5c36d91bd612f42 - System
error - class: class org.mule.api.MessagingException - message: Internal Server Error:HTTP 500: Internal
Server error. - details: org.mule.api.MessagingException: HTTP 500: Internal Server error.
- com.mulesoft.services.HttpErrorException: HTTP 500: Internal Server error
ERROR 2018-05-17 10:25:49,025 [[experience-customer-api].httpListenerConfig.worker.02]
org.mule.api.processor.LoggerMessageProcessor: transactionId: e674266e8e7745dba5c36d91bd612f42 - System
error - Error sending notification to Cloudhub

```

In this API, the original error is not going to be logged. The System API only returned an "Internal Server Error" for security reasons. Thanks to the Transaction Identifier, the error can be traced down to the System API.

**Note:** When you run the three applications locally, all the logs will be mixed in the console. You can go to the **.mule** directory inside the Anypoint Studio workspace to see the real log files for each application.

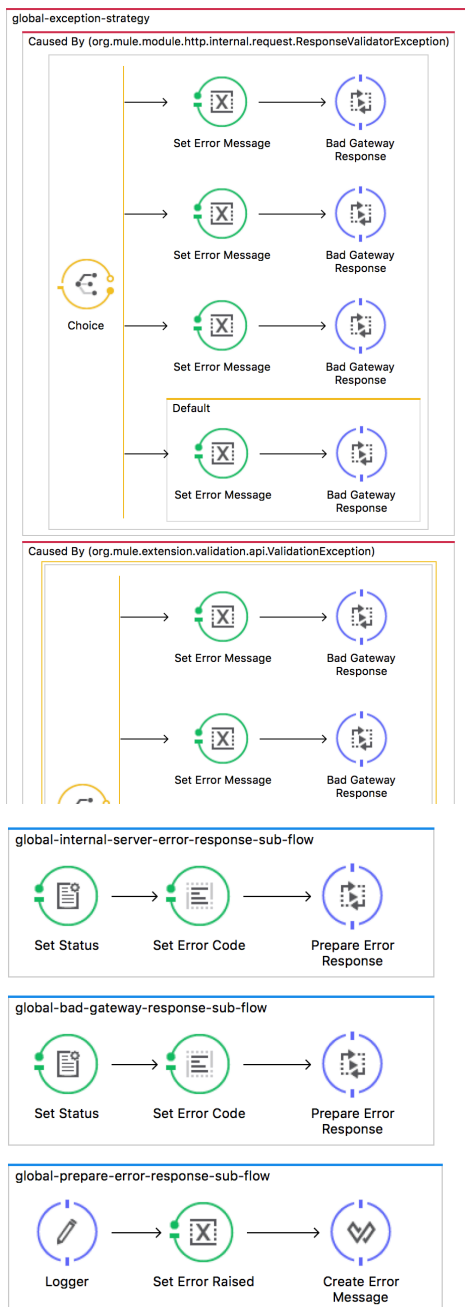
## 4.4 Customize Error Handling

The APIKit exception handling has some limitations. An important one is that one exception can only be mapped to a specific HTTP Status. For example, if we don't care about losing the HTTP Response, the `ResponseValidatorException` could only be mapped to one Status code. An alternative to this is creating a global Choice Exception Strategy that maps a given Exception class.

This error handling strategy has been applied to the API Template located in this Github repository:

<https://github.com/mulesoft-consulting/api-template>

In this case, the HTTP Status can be set based on the Exception, variables and properties the message contains:



An example of this Choice Exception Strategy would be:

#### Choice Exception Strategy

```
<choice-exception-strategy name="global-exception-strategy">
  <catch-exception-strategy
    when="#[exception.causedBy(org.mule.module.http.internal.request.ResponseValidatorException)]"
    doc:name="Caused By (org.mule.module.http.internal.request.ResponseValidatorException)">
    <choice doc:name="Choice">
      <when expression="#[message.inboundProperties['http.status'] == 401]">
```

```

        <set-variable variableName="errorMessage" value="Upstream service did not authorize the
request." doc:name="Set Error Message"/>
        <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
    </when>
    <when expression="#[message.inboundProperties['http.status'] == 403]">
        <set-variable variableName="errorMessage" value="Access to the upstream service is
forbidden." doc:name="Set Error Message"/>
        <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
    </when>
    <when expression="#[message.inboundProperties['http.status'] < 500]">
        <set-variable variableName="errorMessage" value="Upstream service unable to fulfil
request." doc:name="Set Error Message"/>
        <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
    </when>
    <otherwise>
        <set-variable variableName="errorMessage" value="Upstream service internal error."
doc:name="Set Error Message"/>
        <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
    </otherwise>
</choice>
</catch-exception-strategy>
<catch-exception-strategy
when="#[exception.causedBy(org.mule.extension.validation.api.ValidationException)]" doc:name="Caused By
(org.mule.extension.validation.api.ValidationException)">
    <choice doc:name="Choice">
        <when expression="#[message.inboundProperties['http.status'] == 401]">
            <set-variable variableName="errorMessage" value="Upstream service did not authorize the
request. #[exception.cause.message]" doc:name="Set Error Message"/>
            <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
        </when>
        <when expression="#[message.inboundProperties['http.status'] == 403]">
            <set-variable variableName="errorMessage" value="Access to the upstream service is
forbidden. #[exception.cause.message]" doc:name="Set Error Message"/>
            <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
        </when>
        <when expression="#[message.inboundProperties['http.status'] < 500]">
            <set-variable variableName="errorMessage" value="Upstream service unable to fulfil
request. #[exception.cause.message]" doc:name="Set Error Message"/>
            <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
        </when>
        <otherwise>
            <set-variable variableName="errorMessage" value="Upstream service internal error.
#[exception.cause.message]" doc:name="Set Error Message"/>
            <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
        </otherwise>
    </choice>
</catch-exception-strategy>
<catch-exception-strategy when="#[exception.causedBy(java.util.concurrent.TimeoutException)]"
doc:name="Caused By (java.util.concurrent.TimeoutException)">
    <set-variable variableName="errorMessage" value="Unable to connect to upstream service. Request
timed out." doc:name="Set Error Message"/>
    <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
</catch-exception-strategy>
<catch-exception-strategy when="#[exception.causedBy(java.util.NoSuchElementException)]"
doc:name="Caused By (java.util.NoSuchElementException)">

```

```

        <set-variable variableName="errorMessage" value="Unable to connect to upstream service."
doc:name="Set Error Message"/>
        <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy when="#[exception.causeMatches('java.net')]" doc:name="Cause Matches
('java.net')">
        <set-variable variableName="errorMessage" value="Unable to connect to upstream service."
doc:name="Set Error Message"/>
        <flow-ref name="global-bad-gateway-response-sub-flow" doc:name="Bad Gateway Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy
when="#[exception.causedBy(org.mule.module.apikit.exception.NotFoundException)]" doc:name="Caused By
(org.mule.module.apikit.exception.NotFoundException)">
        <flow-ref name="global-resource-not-found-response-sub-flow" doc:name="Resource Not Found
Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy
when="#[exception.causedBy(org.mule.module.apikit.exception.MethodNotAllowedException)]" doc:name="Caused
By (org.mule.module.apikit.exception.MethodNotAllowedException)">
        <flow-ref name="global-method-not-allowed-response-sub-flow" doc:name="Method Not Allowed
Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy
when="#[exception.causedBy(org.mule.module.apikit.exception.UnsupportedMediaTypeException)]"
doc:name="Caused By (org.mule.module.apikit.exception.UnsupportedMediaTypeException)">
        <flow-ref name="global-unsupported-media-type-response-sub-flow" doc:name="Unsupported Media
Type Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy
when="#[exception.causedBy(org.mule.module.apikit.exception.NotAcceptableException)]" doc:name="Caused By
(org.mule.module.apikit.exception.NotAcceptableException)">
        <flow-ref name="global-not-acceptable-response-sub-flow" doc:name="Not Acceptable Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy
when="#[exception.causedBy(org.mule.module.apikit.exception.BadRequestException)]" doc:name="Caused By
(org.mule.module.apikit.exception.BadRequestException)">
        <set-variable variableName="errorMessage" value="#[exception.cause.message]" doc:name="Set
Error Message"/>
        <flow-ref name="global-bad-request-response-sub-flow" doc:name="Bad Request Response"/>
    </catch-exception-strategy>
    <catch-exception-strategy when="#[exception.causedBy(java.lang.RuntimeException)]" doc:name="Caused
by (java.lang.RuntimeException)">
        <expression-component doc:name="Set Error Info"><![CDATA[if (flowVars.httpStatus == null ||
flowVars.httpStatus == '') {
            message.outboundProperties['http.status'] = "500";
} else {
            message.outboundProperties['http.status'] = flowVars.httpStatus;
}

if (flowVars.errorMessage == null || flowVars.errorMessage == '') {
    flowVars.errorMessage = "Unable to fulfil request due to internal error.";
}

if (flowVars.errorDescription == null || flowVars.errorDescription == '') {
    flowVars.errorDescription = null;
}]]></expression-component>
    </catch-exception-strategy>

```



```
}

if (flowVars.errorCode == null || flowVars.errorCode == '') {
    payload = app.registry.statuses.get(message.outboundProperties['http.status']);
} else {
    payload = flowVars.errorCode;
}

return payload;
]]></expression-component>
    <flow-ref name="global-prepare-error-response-sub-flow" doc:name="Prepare Error Response"/>
</catch-exception-strategy>
<catch-exception-strategy doc:name="Default">
    <set-variable variableName="errorMessage" value="Unable to fullfil request due to internal
error." doc:name="Set Error Message"/>
    <flow-ref name="global-internal-server-error-response-sub-flow" doc:name="Internal Server
Response"/>
</catch-exception-strategy>
</choice-exception-strategy>
```

The exceptions caught in this block are:

- org.mule.module.http.internal.request.ResponseValidatorException
- org.mule.extension.validation.api.ValidationException
- java.util.concurrent.TimeoutException
- java.util.NoSuchElementException
- java.net.\* (Network related exceptions)
- org.mule.module.apikit.exception.NotFoundException
- org.mule.module.apikit.exception.MethodNotAllowedException
- org.mule.module.apikit.exception.UnsupportedMediaTypeException
- org.mule.module.apikit.exception.NotAcceptableException
- org.mule.module.apikit.exception.BadRequestException
- java.lang.RuntimeException

The APIKit exceptions are the ones set by default in the APIKit exception mapping and can be thrown by the APIKit Router when it validates the HTTP Request.

## 5 Error Handling in API-led Connectivity Approach Best Practices - Mule 4

- [Introduction](#)
- [Mule Error concept](#)
  - [Error Types](#)
  - [Guidelines](#)
- [REST API Error handling](#)
  - [HTTP Status](#)
  - [Error Response Message](#)
- [APIKit Error Handler](#)
- [Error Handler](#)
- [Try Scope and On Error component](#)
  - [Try Scope](#)
  - [On Error component](#)
- [Validation Module](#)
- [Mapping Errors](#)
- [HTTP Response Validation](#)
- [Transaction Identifier](#)
  - [CorrelationID](#)

### 5.1 Introduction

This document is an introduction to Error Handling for Mule 4. Based on the official documentation and field experience with Mule 4.

### 5.2 Mule Error concept

Mule 4 includes a simplified way to manage errors. Instead of dealing with Java exceptions directly, there is now an Error concept built directly into Mule. Mule Modules and Connectors declare what Errors may occur for any given operation. This makes it easy for you to discover possible errors at design time and catch them.

Mule throws a messaging error whenever a problem occurs within a flow. From a high-level perspective, errors that occur in Mule fall into one of two categories: system errors and messaging errors. Base documentation: <https://docs.mulesoft.com/mule-runtime/4.1/intro-error-handlers>

We should consider how we'll handle the possible errors that will be raised during the APIs execution.

For example, a user consumes the Web App, calling the Experience API and creating an order. The Experience API calls the "Order Status" API (Process). This API consumes the "Customers" process API and retrieves a customer from the "Salesforce customers" API.

Now, if the customer sent by the user in the Web App has been deleted from Salesforce, the Salesforce connector will fail, and the "Salesforce customers" API will return an error. We have to analyze what should be returned to the Web App user, and how the error will be propagated. The Salesforce error message won't be designed for an end user and could contain sensitive information.

## 5.2.1 Error Types

In Mule, all Errors have a type, consist of a namespace and an identifier. Base documentation: <https://docs.mulesoft.com/mule-runtime/4.1/mule-error-concept#about-error-types>

Each component declares the type of errors it can throw, so you can identify potential errors at design time.

## 5.2.2 Guidelines

It is recommended to follow the following guidelines when applying error handling:

- Whenever it can be applied, it is advisable to use the “try scope” to treat components that may cause errors (with their known types).
- There must be a handler responsible for processing all those exceptions that have not been caught by the try scope.
  - Use the Error Handler of the flow.
  - Use the Error Handler created by APIKit.
  - Use a global Error Handler.
- Each error message that needs to be returned must support a standard format and be created within dataweave.
  - Use the Correlation ID as a field in the standard error message providing traceability.
- When an error occurs the payload is available for analyze and use.

## 5.3 REST API Error handling

REST APIs use the Status of an HTTP response to message and inform clients of their request's result.

### 5.3.1 HTTP Status

HTTP RFC defines over 40 standard status codes that can be used to convey the results of a client's request. The status codes are divided into five categories:

- 1xx: Informational - Communicates the transfer protocol-level information.
- 2xx: Success - Indicates that the client's request was accepted successfully.
- 3xx: Redirection - Indicates that the client must take additional action in order to complete their request.
- 4xx: Client Error - Indicates the client made some sort of error. It is most commonly used for business validations and business errors.
- 5xx: Server Error - Indicates the server takes responsibility for the error status codes. These errors are not expected and should never happen.

The most common HTTP error codes are:

- 400 - Bad Request: An error in the client request (Mostly due to validations)
- 401 - Unauthorized: User can't be authenticated
- 403 - Forbidden: The server cannot give access to the resource
- 404 - Not Found: The resource defined in the URL doesn't exist

- 412 - Precondition Failed: One of the validations in the request failed (Sometimes used instead of 400)
- 500 - Internal Server Error: The server encountered an unexpected condition
- 502 - Bad Gateway: The server, while acting as a gateway or proxy, received an invalid response from an inbound server
- 504 - Gateway Timeout: The server tried to access an upstream service and it took longer than expected

it's important to distinguish between two kinds of errors:

- **Business Error:** An error expected by the business domain. This may include a validation not being satisfied, a required field not being sent, or a record not being found. The API should notify the user about this error.
- **System Error:** These types of errors are not expected. System errors may occur when a Database connection fails, an upstream service returns a server error, or a code error is present. Development support or operation teams have to be notified when these types of errors occur.

### 5.3.2 Error Response Message

When an error occurs, A REST API should reply with an error code and sent a message to the client. To provide traceability, a transaction identifier could be used.

Here are some examples of how an error response should look:

#### 404 - Not Found

```
{
  "transactionId": "0-55558d60-f74e-11e8-b992-8c8590b35d7d",
  "code": "RESOURCE_NOT_FOUND",
  "message": "Resource not found",
  "description": "Resource not found - /values/addressTypess"
}
```

#### 500 - Internal Server Error

```
{
  "transactionId": "0-55558d60-f74e-11e8-b992-8c8590b35d7d",
  "code": "INTERNAL_SERVER_ERROR",
  "message": "Internal Server Error",
  "description": "Internal Server error"
}
```

## 5.4 APIKit Error Handler

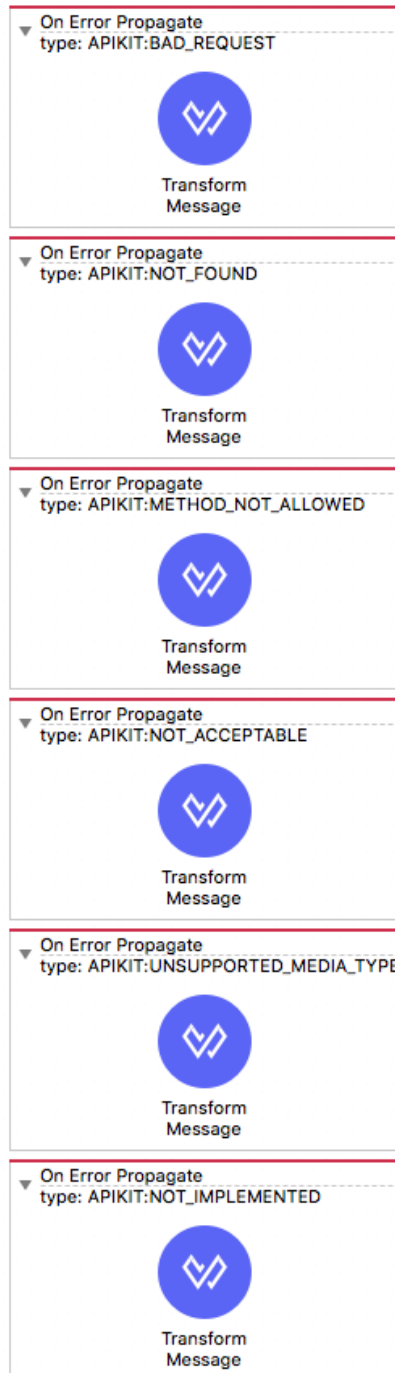
When creating a new APIKit project in Anypoint Studio, an error handler will be created by default for the most widely-used HTTP status code responses, which are mapped to the types and error messages.

The status codes are:

- 400: Bad request - Type: APIKIT:BAD\_REQUEST

- 404: Resource not found - Type: APIKIT:NOT\_FOUND
- 405: Method not allowed - Type: APIKIT:METHOD\_NOT\_ALLOWED
- 406: Not acceptable - Type: APIKIT:NOT\_ACCEPTABLE
- 415: Unsupported media type - Type: APIKIT:UNSUPPORTED\_MEDIA\_TYPE
- 501: Not implemented - Type: APIKIT:NOT\_IMPLEMENTED

#### ▼ Error handling



XML configuration

```

<error-handler>
  <on-error-propagate type="APIKIT:BAD_REQUEST">
    <ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
      <ee:message>
        <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{message: "Bad request"}]]></ee:set-payload>
      </ee:message>
      <ee:variables>
        <ee:set-variable variableName="httpStatus">400</ee:set-variable>
      </ee:variables>
    </ee:transform>
  </on-error-propagate>
  <on-error-propagate type="APIKIT:NOT_FOUND">
    <ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
      <ee:message>
        <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{message: "Resource not found"}]]></ee:set-payload>
      </ee:message>
      <ee:variables>
        <ee:set-variable variableName="httpStatus">404</ee:set-variable>
      </ee:variables>
    </ee:transform>
  </on-error-propagate>
  <on-error-propagate type="APIKIT:METHOD_NOT_ALLOWED">
    <ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
      <ee:message>
        <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{message: "Method not allowed"}]]></ee:set-payload>
      </ee:message>
      <ee:variables>
        <ee:set-variable variableName="httpStatus">405</ee:set-variable>
      </ee:variables>
    </ee:transform>
  </on-error-propagate>
  <on-error-propagate type="APIKIT:NOT_ACCEPTABLE">
    <ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
      <ee:message>
        <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{message: "Not acceptable"}]]></ee:set-payload>
      </ee:message>
      <ee:variables>
        <ee:set-variable variableName="httpStatus">406</ee:set-variable>

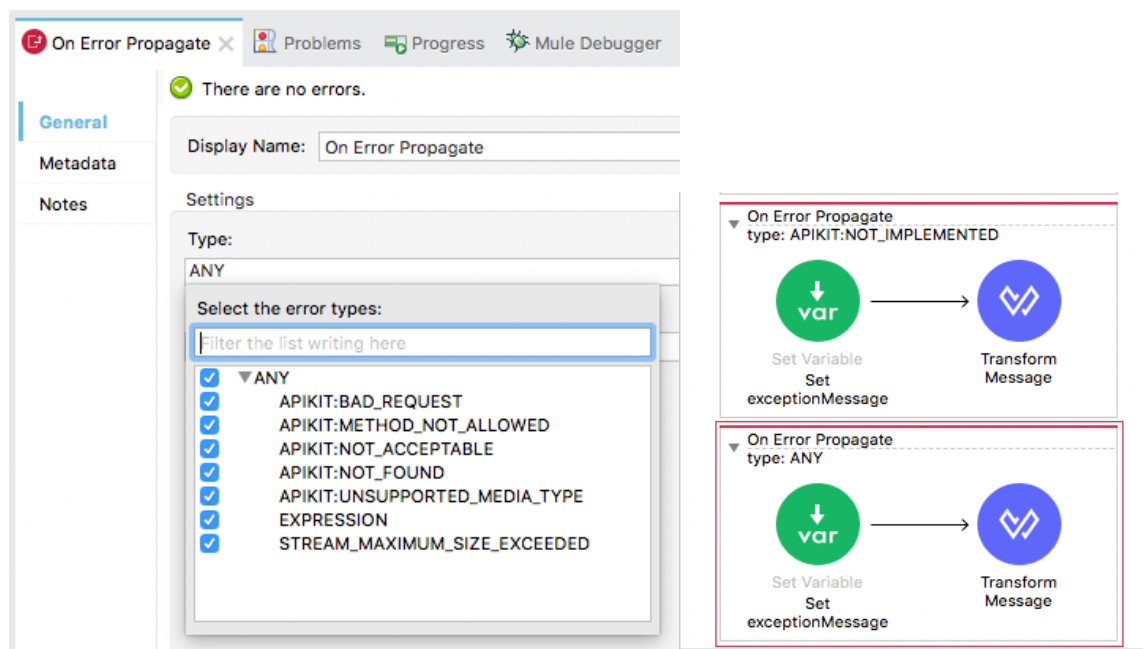
```

```

        </ee:variables>
    </ee:transform>
</on-error-propagate>
<on-error-propagate type="APIKIT:UNSUPPORTED_MEDIA_TYPE">
    <ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
        <ee:message>
            <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{message: "Unsupported media type"}]]></ee:set-payload>
        </ee:message>
        <ee:variables>
            <ee:set-variable variableName="httpStatus">415</ee:set-variable>
        </ee:variables>
    </ee:transform>
</on-error-propagate>
<on-error-propagate type="APIKIT:NOT_IMPLEMENTED">
    <ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">
        <ee:message>
            <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{message: "Not Implemented"}]]></ee:set-payload>
        </ee:message>
        <ee:variables>
            <ee:set-variable variableName="httpStatus">501</ee:set-variable>
        </ee:variables>
    </ee:transform>
</on-error-propagate>
</error-handler>

```

You can add more On-Error components for catching others error, even Custom Errors using mappings. For example, by selecting ANY, you are catching all the errors that are not being caught for the previous On-Error components.



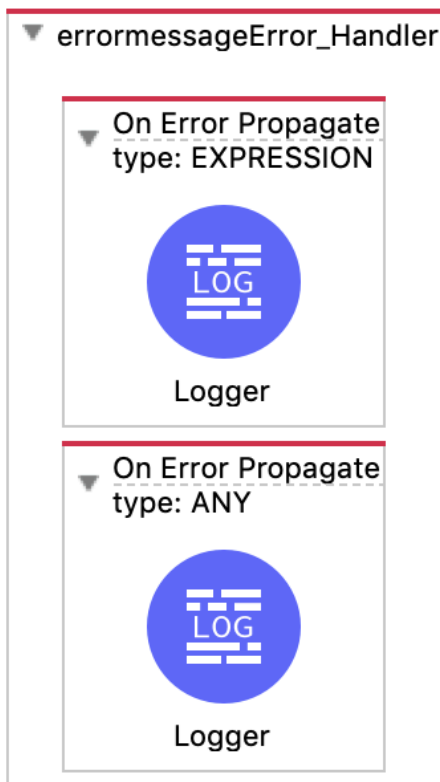
The screenshot displays the Mule IDE interface for configuring an 'On Error Propagate' handler. The left sidebar shows the 'General' tab with the 'Display Name' set to 'On Error Propagate' and the 'Type' set to 'ANY'. A dropdown menu is open, showing a list of error types: APIKIT:BAD\_REQUEST, APIKIT:METHOD\_NOT\_ALLOWED, APIKIT:NOT\_ACCEPTABLE, APIKIT:NOT\_FOUND, APIKIT:UNSUPPORTED\_MEDIA\_TYPE, EXPRESSION, and STREAM\_MAXIMUM\_SIZE\_EXCEEDED. The right pane shows two flow diagrams. The top diagram is for 'On Error Propagate type: APIKIT:NOT\_IMPLEMENTED' and the bottom diagram is for 'On Error Propagate type: ANY'. Both diagrams show a 'Set Variable' (var) and 'Set exceptionMessage' step followed by a 'Transform Message' step.

## 5.5 Error Handler

Mule provides an Error Handler that allows you to define the global error handling. Can contain any number of internal handlers and can route an error to the first one matching it, as the automatically generated by APIKit.

That can be used when APIKit doesn't exist in the project, or when the APIKit Error Handling is not used.

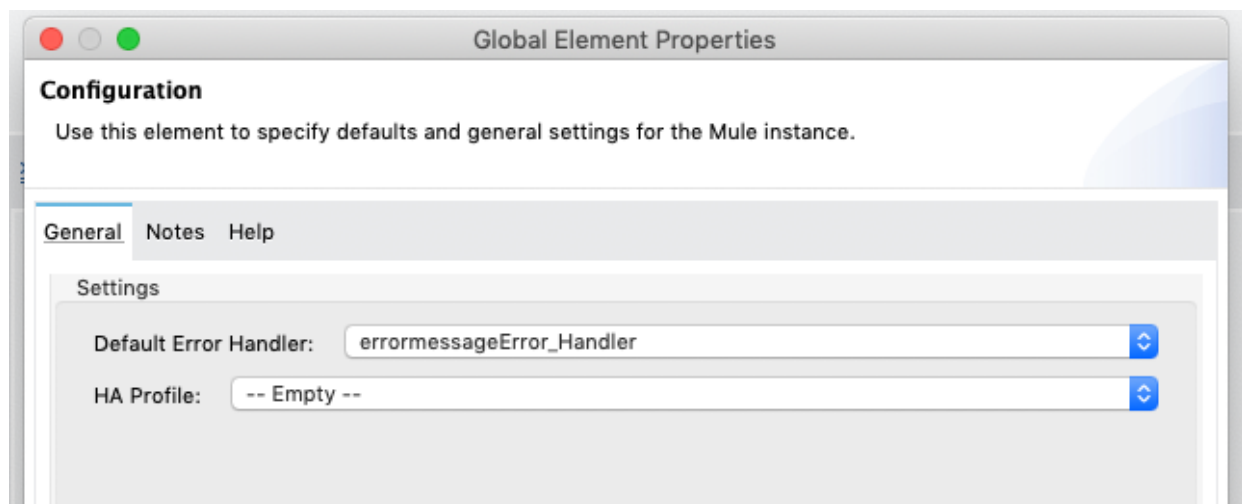




In Global Elements add a Configuration referencing this Error Handler.

Global Configuration Elements			
Type	Name	Description	
Configuration (Configuration)	Configuration		<div>Create</div> <div>Edit</div> <div>Delete</div>

Select the Default Error Handler, in this case, the previously created "errormessageErrpr\_Handler".



As in the Error Handling generated by APIKit, you can add On-Error components for catching errors. For example, by selecting ANY, you are catching all the errors that are not being caught for the previous On-Error components.

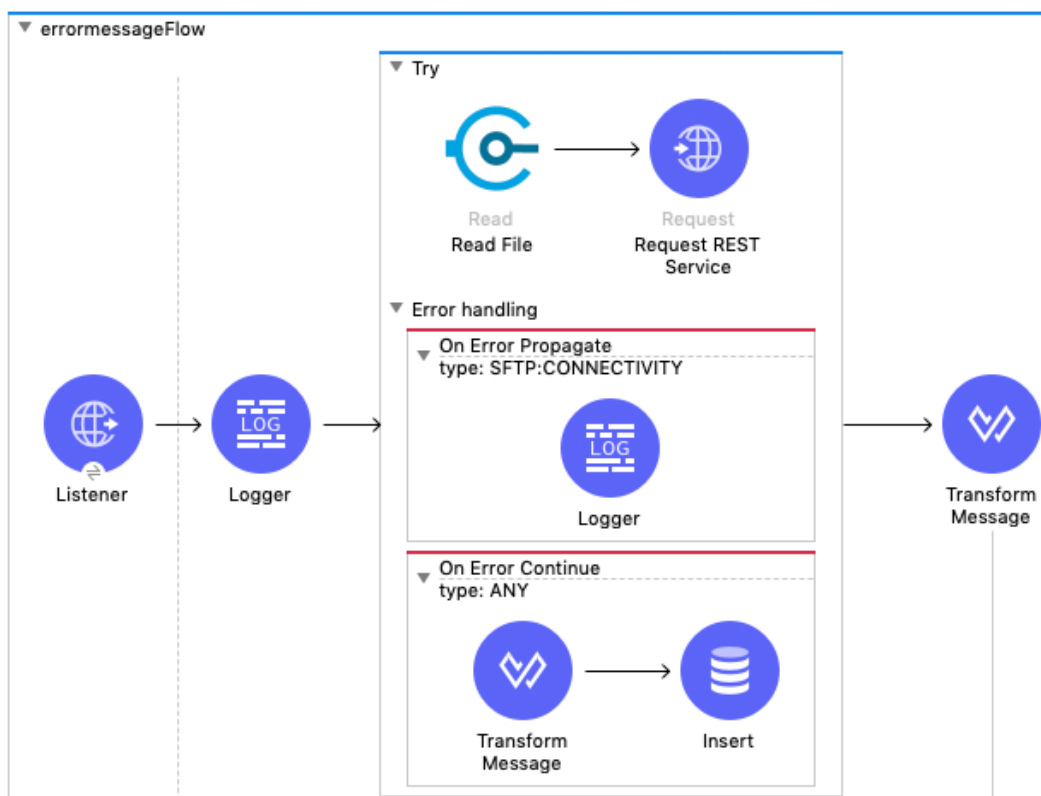
## 5.6 Try Scope and On Error component

Similar to Error Handling in Java, Mule 4 uses try/catch to handle exceptions thrown during the execution of the try blocks. <https://docs.mulesoft.com/mule-runtime/4.1/try-scope-concept>

Allows you to catch errors in the middle of flow without having to create a new flow, specifically to catch that error.

### 5.6.1 Try Scope

Allow you to handle errors that may occur when attempting to execute any of the components inside the Try scope.



In the example, the SFTP Read and HTTP Request processors are wrapped by the Try scope. If an error is raised by the SFTP Read or the HTTP request, then the Try scope's error handler is executed, and the error is routed to the matching handler.

There are two types of handling behaviors: on-error-continue and on-error-propagate

## 5.6.2 On Error component

You can configure your error handlers to catch errors so that the flow can keep processing (On Error Continue), or they can be re-propagated (On Error Propagate). Base documentation: <https://docs.mulesoft.com/mule-runtime/4.1/on-error-scope-concept>

- **On Error Propagate:** Executes but propagates the error to the next level up, breaking the owner's execution. If a transaction is being handled, it's rolled back.
- **On Error Continue:** Executes and uses the result of the execution, as the result of its owner, as if the owner had actually completed the execution successfully. If a transaction is being handled, it would be committed as well

On Error components define the types of errors that they handle.

You can select the type or types that the On Error component will handle, selecting the **type** (or types) or using the **when** attribute to define the expression will be evaluated to determine if the On Error will be executed.

**Using attribute type**

```
<on-error-continue enableNotifications="true" logException="true" doc:name="On Error Continue"
type="DB:CONNECTIVITY, HTTP:CONNECTIVITY, SFTP:CONNECTIVITY" />
<on-error-continue enableNotifications="true" logException="true" doc:name="On Error Continue" type="ANY" /
>
```

Use the error namespace to handle all error of a type with a single On Error component. In this example all HTTP errors will be handled by a single On Error component, configure the **when** attribute with `#[error.errorType.namespace == 'HTTP']`.

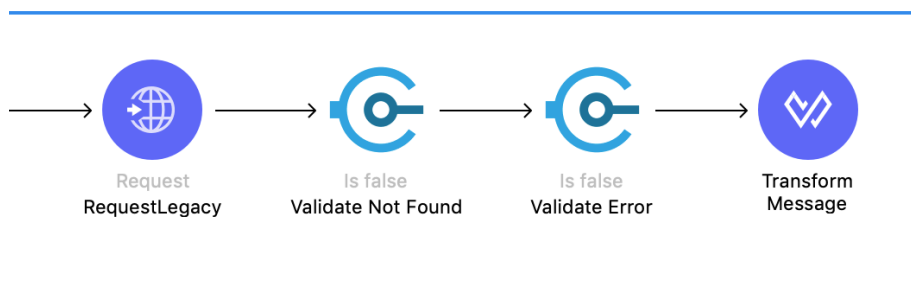
**Using attribute when**

```
<on-error-propagate enableNotifications="true" logException="true" doc:name="On Error Propagate - All HTTP
Errors" when="#[error.errorType.namespace == 'HTTP'] /">
```

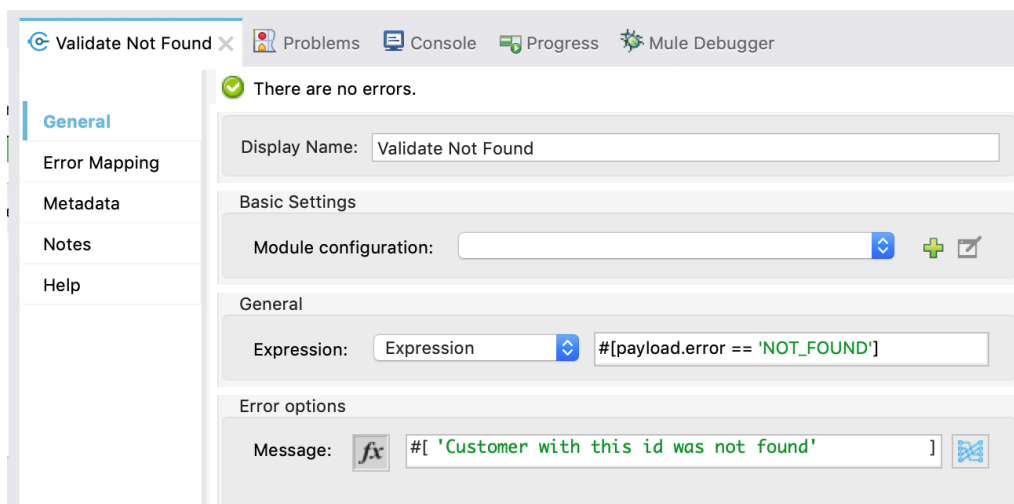
## 5.7 Validation Module

Mule provides a Validation Module that allows you to check business rules and throw an exception when they are not satisfied. The validation module allows you to define a message that can be understood by the user.

The exception being thrown by this component generally represents a business error.



In this example, if the error field of the payload from the response of the HTTP Request is "NOT\_FOUND" the validator will return false and generate an error "VALIDATION:INVALID\_BOOLEAN".



You can handle this VALIDATION error with an On Error component

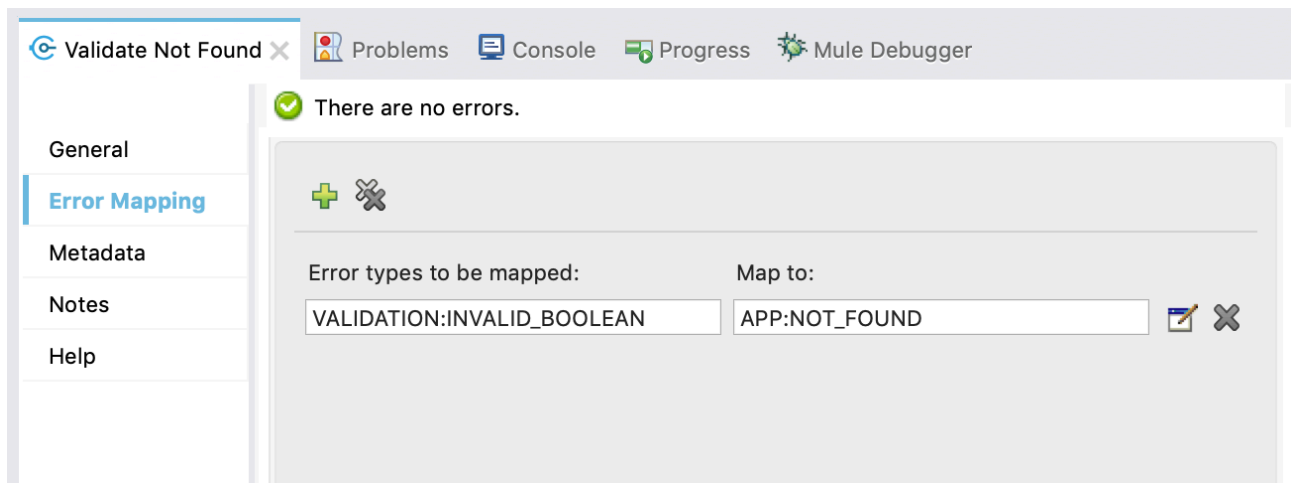
## 5.8 Mapping Errors

In the previous example, the Validation Module throws a Validation Error. It's useful to manage these generic errors (to differentiate them) using Custom Errors.

For our application, the custom error is APP:NOT\_FOUND, and this error can be handled by a new or existing On-Error component, that created from the APIKit, global error handler or another one.

In the example below the "VALIDATION:INVALID\_BOOLEAN" is mapped to the custom error "APP:NOT\_FOUND" so, the Validator processor will throw an APP:NOT\_FOUND error.

Errors can be mapped to a Custom Error. In the Error Mapping tab of the processor configuration, you can create one or more error mapping.

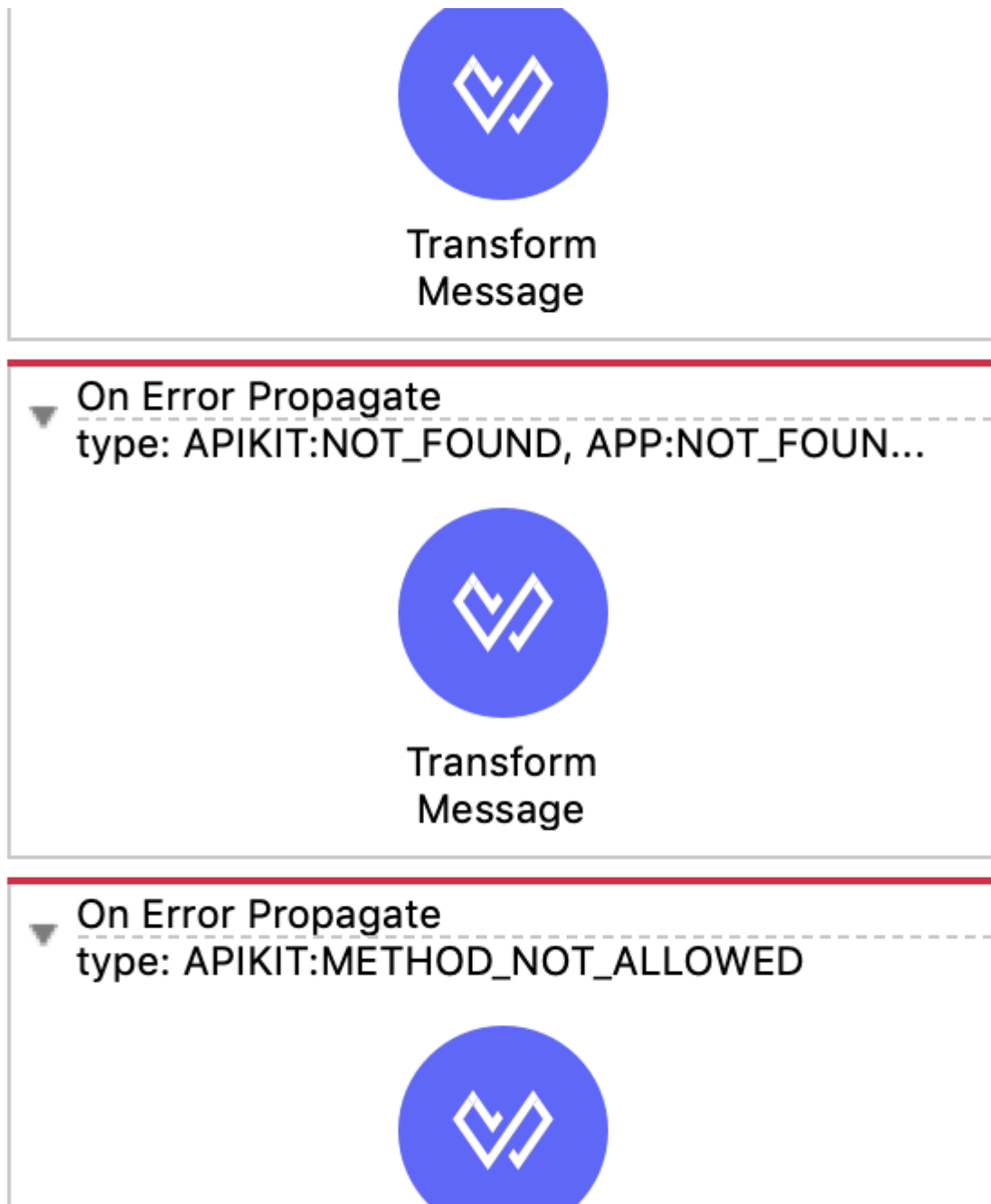


XML configuration

```
<validation:is-false doc:name="Validate Not Found" message="#['Customer with this id was not found']"
expression="#[payload.error == 'NOT_FOUND']">
  <error-mapping sourceType="VALIDATION:INVALID_BOOLEAN" targetType="APP:NOT_FOUND" />
</validation:is-false>
```

```
</validation:is-false>
```

In this case, the APP:NOT\_FOUND error is mapped to an APIKit NOT\_FOUND generated On-Error component and it will be handled by the On Error Propagate created from APIKit:



XML configuration

```
<on-error-propagate type="APIKIT:NOT_FOUND, APP:NOT_FOUND" enableNotifications="true" logException="true">
```

```

<ee:transform xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core" xsi:schemaLocation="http://
www.mulesoft.org/schema/mule/ee/core http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd" >
  <ee:message>
    <ee:set-payload><![CDATA[%dw 2.0
output application/json
---
{
transactionId: correlationId,
code: error.errorType.identifier as String,
message: "Resource not found",
description: error.description as String
}
]]>
    </ee:set-payload>
  </ee:message>
  <ee:variables>
    <ee:set-variable variableName="httpStatus"><![CDATA[404]]></ee:set-variable>
  </ee:variables>
</ee:transform>
</on-error-propagate>

```

## 5.9 HTTP Response Validation

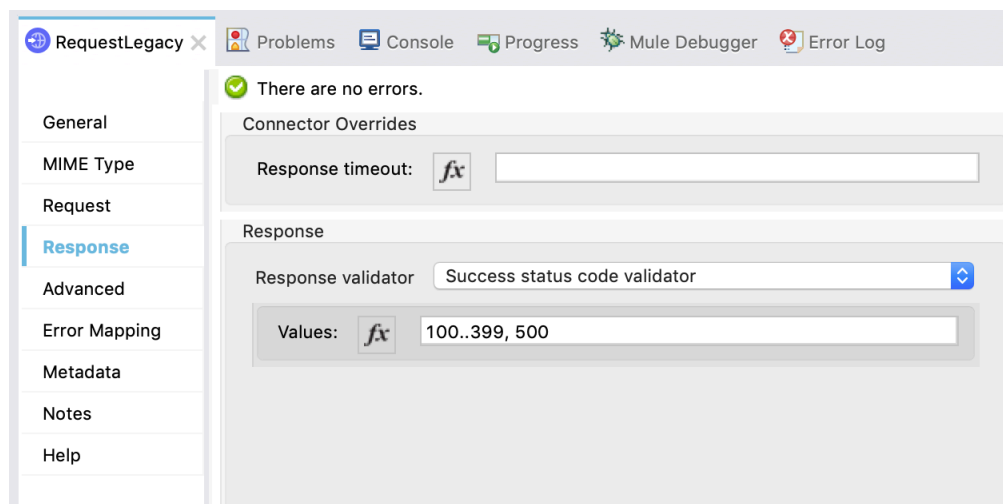
When the HTTP request operation receives an HTTP response, it validates the response through its status code. By default, it throws an error when the status code is higher than or equal to 400. Consequently, if the server returns a 404 (Resource Not Found) or a 500 (Internal Server Error) a failure occurs and error handling is triggered.

You can change the set of valid HTTP response codes in the tab Responses the attribute "Response Validator".

- **None:** When the status code is lower to 400 is considered valid, when the status code is higher than or equal to 400 is considered invalid and an error is thrown (default behavior).
- **Success Status Code Validator:** All the status codes defined within this element are considered valid; the request throws an error for any other status code.
- **Failure Status Code Validator:** All the status codes defined within this element are considered invalid and an error is thrown; the request is considered valid with any other status code.

After the HTTP Request, you can use a Choice to select the flow depending on the status code returned.

In the example below, the HTTP Request "RequestLegacy" has the status code 100 to 399 and 500 configured as a success response.

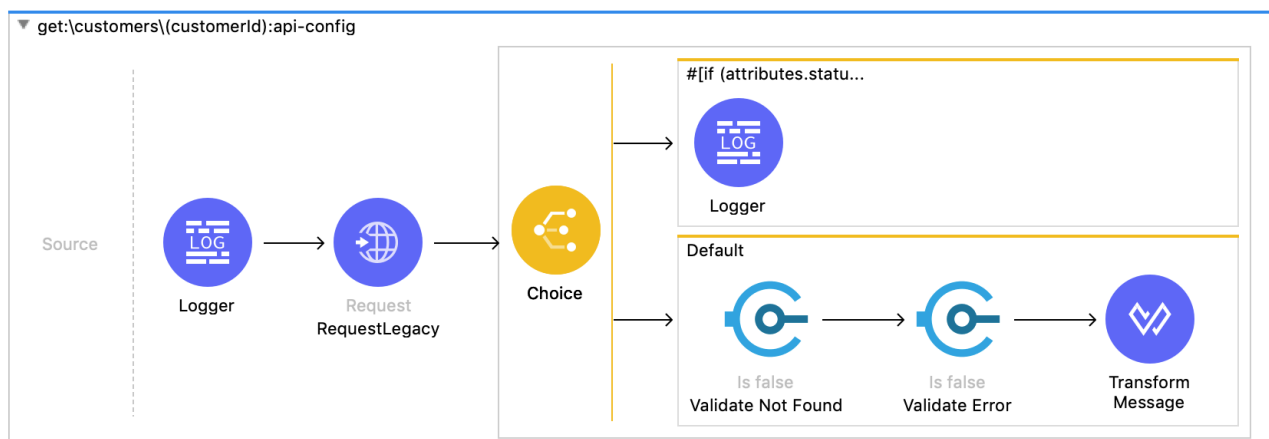


## XML configuration

```
<http:request method="GET" doc:name="RequestLegacy" config-ref="HTTP_Request_configuration" path="/
customers/{customerId}">
  <http:uri-params><![CDATA[#[output application/java --- {customerId :
attributes.uriParams.customerId}]]]></http:uri-params>
  <http:response-validator >
    <http:success-status-code-validator values="100..399, 500" />
  </http:response-validator>
</http:request>
```

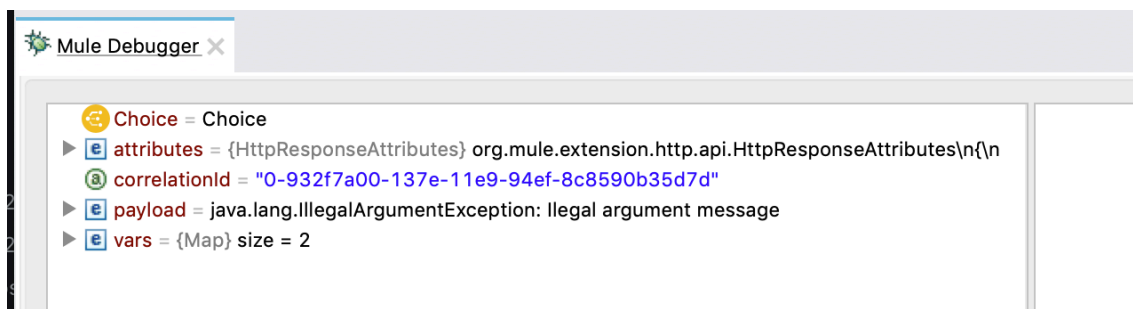
After the HTTP Request the Choice component will check the Status Code:

- if is different to 500 and not an error will take the default path.
- if is 500 will take the path to the logger, because the status code = 500 is configured as successful and doesn't generate an error
- Another status code upper or equal to 400, except 500, will generate an Error in the HTTP Request and this will be handled by the Error Handler defined for the flow. The choice will not be evaluated.

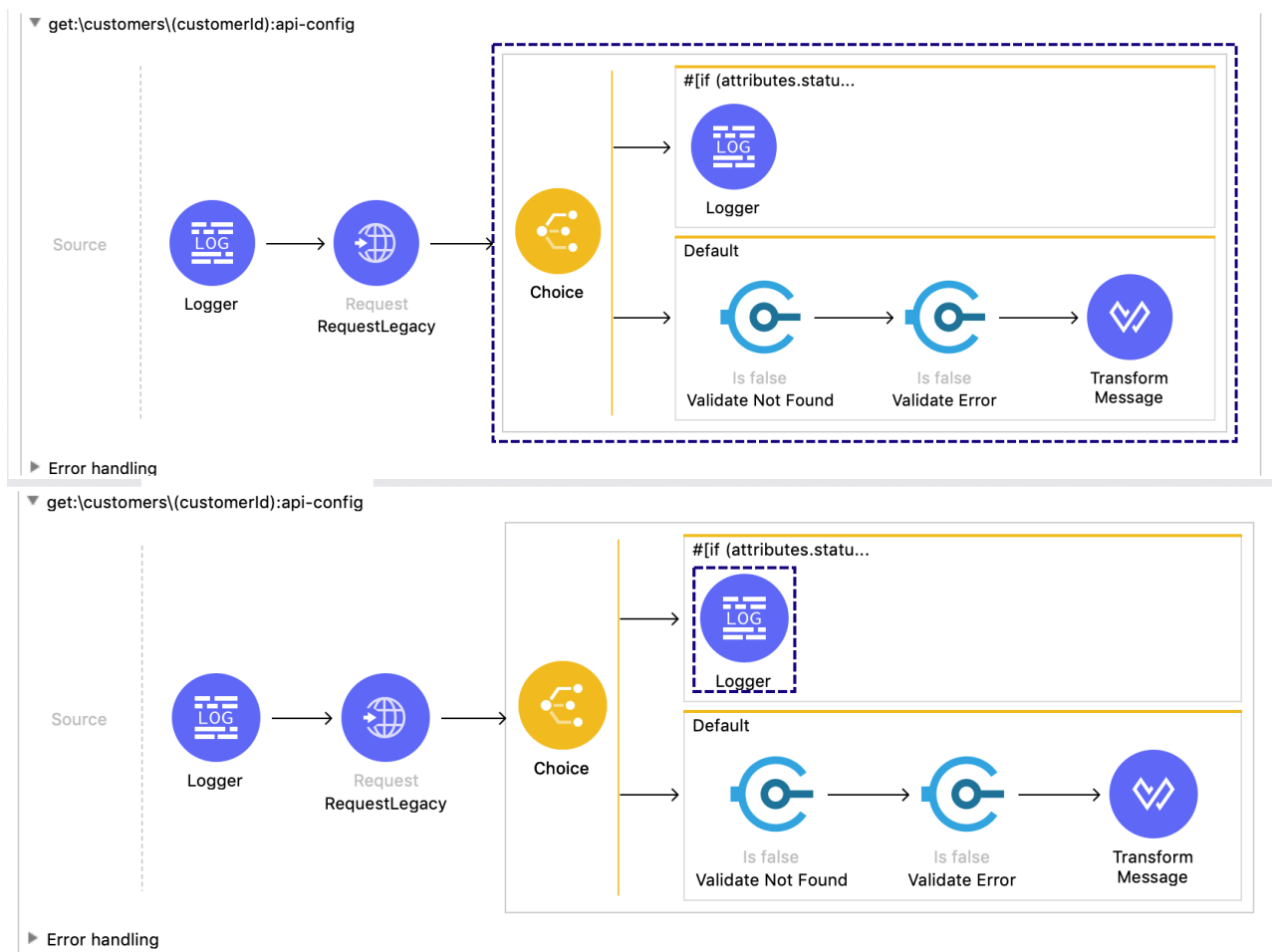


For example, the service Legacy behind the HTTP Request generates an "IllegalArgumentException" with Status Code = 500, the HTTP Request take this response as successful and doesn't generate an error,





The flow continues normally, the HTTP Request passes the payload received from the service Legacy as is (the error message from the service Legacy requested) to the Choice component where the status code will be checked and will take the Logger path.



## 5.10 Transaction Identifier

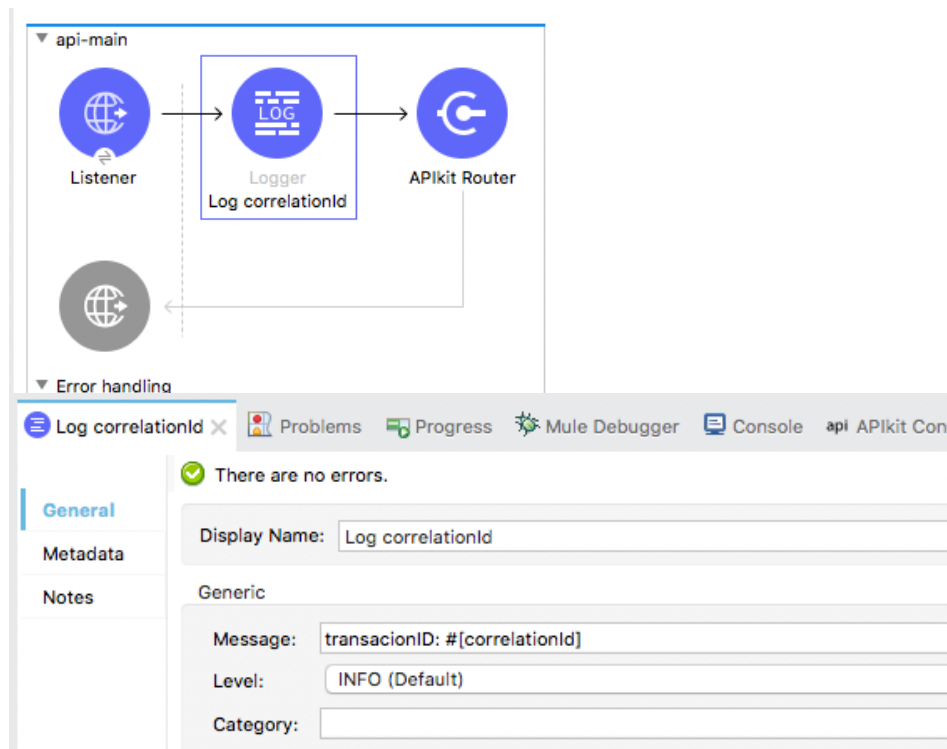
When a system error is raised, it is recommended to only provide basic information to the user, such as the error type and error code. The stack trace and exception chain should only be accessible to the developers, operations, support, or admin teams. The information can also be sent to the logs.

The best practice is to use a transaction or request identifier when giving a user an error message. From there, the transaction ID can be logged.

The identifier should be generated at the beginning of the HTTP Request, so any validation error will log and return a transaction identifier.

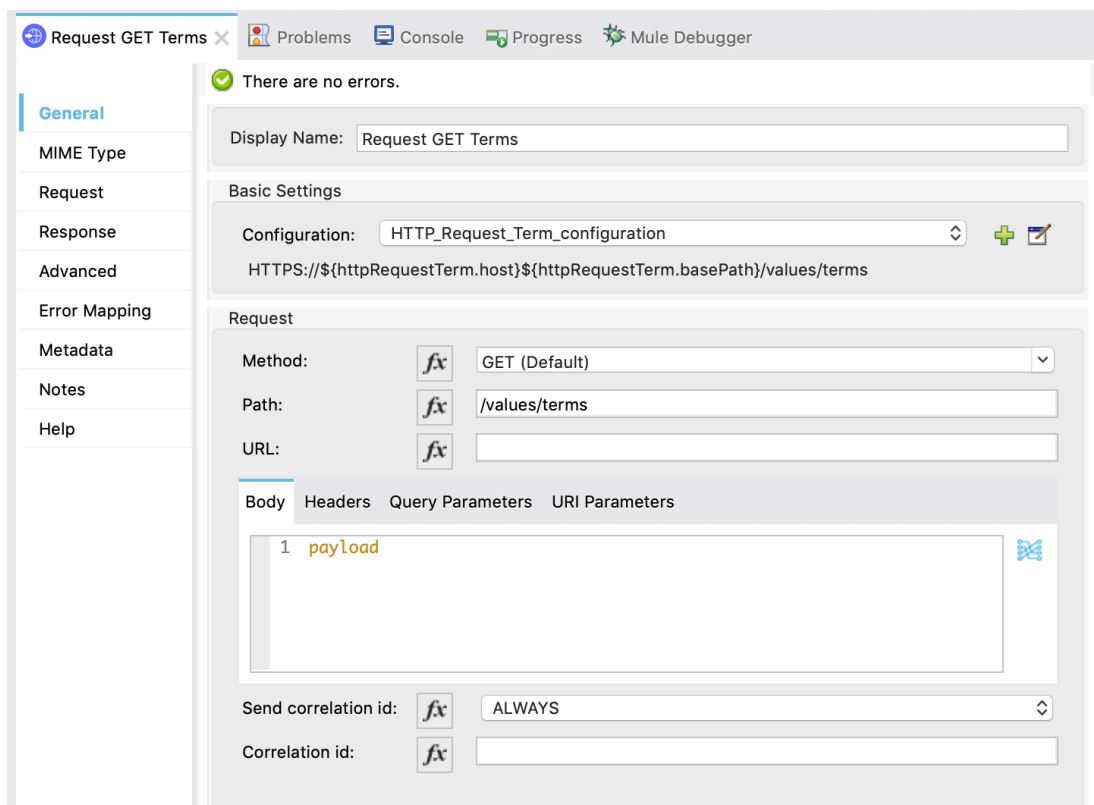
### 5.10.1 CorrelationID

The correlation ID is always accessible via the DataWeave expression `#[correlationId]`. The HTTP listener will always set that correlation ID when it receives an HTTP request. If the HTTP request has the HTTP request header `"x-correlation-id"`, the correlation ID will be set from the value of that header. If that header is not in the incoming HTTP request, then the HTTP listener generates a new correlation ID as a UUID.



All outgoing HTTP requests sent by a Mule app automatically get that HTTP request header "x-correlation-id" set to the current value of the correlation ID. So in other words, the correlation ID is automatically propagated to downstream APIs.

Send the correlation ID when using an HTTP Request processor, can be done setting the request header `"x-correlation-id"` or using the option **Send Correlation id**, set on ALWAYS in the example below:



Request GET Terms x Problems Console Progress Mule Debugger

General

MIME Type

Request

Response

Advanced

Error Mapping

Metadata

Notes

Help

There are no errors.

Display Name: Request GET Terms

Basic Settings

Configuration: HTTP\_Request\_Term\_configuration

HTTPS://\$(httpRequestTerm.host)\$(httpRequestTerm.basePath)/values/terms

Request

Method: GET (Default)

Path: /values/terms

URL:

Body Headers Query Parameters URI Parameters

1 payload

Send correlation id: ALWAYS

Correlation id:

Defining the attribute **Correlation id** you can create your custom correlation ID value and send it in the HTTP Request.

In the example below the correlation ID used is a custom variable called "customTransactionId", configure the **correlationId** attribute with `#[vars.customTransactionId]`.

```
<http:request method="GET" doc:name="Request GET Terms" config-ref="HTTP_Request_Term_configuration"
path="/values/terms" sendCorrelationId="ALWAYS" correlationId="#[vars.customTransactionId]" />
```