**Mule-Project:**Mule project is a normal MuleSoft project which have all the dependencies defined in itself

**Domain-Project:**Mule supports the ability to define selected connectors as common resources and expose them to all applications deployed under a same domain. These resources are known as shared resources, to host these you must create a Mule Domain Project and then reference it on each of the projects that are meant to use an elements in it. That is the mule domain project.

**Use:** In simple words, we can say Mule Domain Project is implemented to configure the resources that are shared among other applications.

**Note:** One Mule application can be associated to one domain only and Domain Project is only applicable for On-Premises project.

**Logging:**

Logging helps you to monitor and troubleshoot your application and server.  
If you are running you application in your local machine, then you can see the logs in your console but debugging in the production doesn’t seems to be same. So there we can use Logging. We need to log in such a way, so that no important information is lost which is desired to troubleshoot application whenever any issue comes. There are many ways to log some are log4j, slf4j, JSON Logger. Mule internally use slf4j logging.

We can log message Synchronously as well as Asynchronously.

**Synchronous Logging:**

The execution of flow will halt until logger logs the message. Same thread as of

processing will be used.

**Asynchronous Logging:**

By default Mule supports this logging depending on the log level like info, debug. In

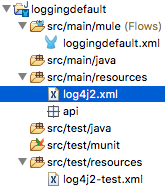
this separate thread will be created for logging which is just like fire and forget and

and the flow will continue.

**Log4j :**

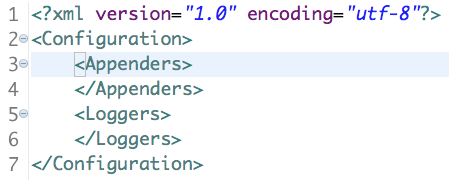
In a default Mule 4 project, located in the *src/main/resources* folder, there is a file called *log4j2.xml*.The naming and location of this file is significant, as Mule 4 projects have a *Maven*-based foundation.

The log4j2.xml file is also based on a standard that is not Mule-specific. It is a Java logging framework provided by the [Apache logging services](https://logging.apache.org/) in the [Apache Log4j](https://logging.apache.org/log4j/2.x/) subproject and is the preferred logging framework. This is largely because Mule applications are built on the Mule runtime which is developed in the Java programming language.



## ****The default log4j2.xml file****

Rather than displaying the default contents of the complete log4j2.xml file here, let’s split it into its two sections; **Appenders**and **Loggers.**



## ****Loggers****

Let’s first look at the configuration of the **Loggers**element.

*Loggers* can be configured to determine what specific *packages* are to be logged — and to what depth — where packages are a grouped set of classes and interfaces, represented by a *namespace*.

Collectively, packages are the foundation of the logic for your Mule applications. For example, you might choose to record everything from the **DEBUG**level upwards for a specific package, while the default configuration for all other packages might be to record only events from the **ERROR** level upwards.

The levels are defined in terms of severity of the loggable event and range in value from **FATAL**(most severe) to **ALL**(least severe). The amount of log data output varies with severity, thus the **FATAL**level outputs the least amount of data while **ALL**logs the most.

Meanwhile, back in a Mule 4 project’s default log4j2.xml file configuration, one of the main considerations for the **Loggers**section is to choose whether to log your events *synchronously* or *asynchronously*. It’s recommended to select *asynchronous* logging in production to improve the application’s performance, i.e. low latency.



|  |  |
| --- | --- |
| 1  2  3  4  5 | <Loggers>  <AsyncRoot level="INFO">       <AppenderRef ref="file" />  </AsyncRoot>  </Loggers> |

In the above code snippet, notice the *AppenderRef* element and its **ref**value “file.” This references the *name* element value from the appender, and thus links an appender configuration (discussed next) to the logger configuration, i.e. it instructs upon the logger the log’s output destination, format, and its configuration.

## ****Appenders**:**

Appenders essentially describe how to deliver the logging events to a destination. Think about where you typically see log events outputted from your Mule applications — the Console view in Anypoint Studio and the log files of the application itself — both of these are examples of *Appenders*.

Note: The appender that generates the log output for Anypoint Studio’s Console view is specified in a log4j2.xml file located in the application’s *conf* folder. I will not be modifying that log’s configuration in this blog.

Let’s peruse the default configuration of the**Appenders** section:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <Appenders>    <RollingFile name="file"  fileName="${sys:mule.home}${sys:file.separator}logs${sys:file.separator}loggingproject.log"  filePattern="${sys:mule.home}${sys:file.separator}logs${sys:file.separator}loggingproject-%i.log">            <PatternLayout pattern=                "%-5p %d [%t] [event: %X{correlationId}] %c: %m%n" />            <SizeBasedTriggeringPolicy size="10 MB" />            <DefaultRolloverStrategy max="10"/>      </RollingFile>  </Appenders> |

As you can see, there is only one pre-configured child element in the **Appenders**element: **RollingFile**.

The RollingFile appender ensures that all log events are routed to a specified file, but there are alternative options, such as writing events to a relational database, submitting messages to topics for Kafka integration, or over SMTP.

## App Logs

Every app that you build in Studio comes with its own log4j2.xml file. The log contains information about any errors raised in the app (unless you have app logic to handle those errors). It also contains anything you want to explicitly log, if you build the logic in the app.

Mule automatically logs multiple messages and specific elements in your app flows to help you debug and keep track of events. You can also include the [Logger component](https://docs.mulesoft.com/mule-runtime/4.3/logger-component-reference) anywhere in a flow and set it up to output any message you want.

You can view an app log as follows:

* If you’re running an app from Anypoint Studio, the output from the app log is visible in Anypoint Studio’s console window.
* If you’re using Mule from the command line to run an app, the app log is visible in your OS console.

You can also view the app log in this default location, unless the log file path was customized in the app’s log file (log4j2.xml):

MULE\_HOME/logs/<app-name>.log

## Runtime Log

The runtime log (mule\_ee.log) contains information about app and lifecycle events. For example, the log records an entry when a Mule service or app starts, deploys, stops, or undeploys.

The runtime log configuration is located in the log4j2.xml file, in the /conf directory. You can customize this file when running the server in standalone mode.

### View the Runtime Log in Anypoint Studio:

Click **Anypoint Studio > About Anypoint Studio > Installation Details > Configuration > View Error Log**:

**Message Id & Correlation Id:**

When you use Splitter the “Message” is split in to multiple new Messages, each with it’s own uniq message-Id and payload. The correlation-Id is used to correlate these messages. The Message-Id of the Message before splitting is set to the correlation-Id of messages produced by the Splitter.

By design, Correlation Ids cannot be changed within a flow in Mule 4 applications and can be set only at source. This ID is part of the Event Context and is generated as soon as the message is received by the application. When a HTTP Request is received, the request is inspected for "X-Correlation-Id" header.

* If "X-Correlation-Id" header is present, HTTP connector uses this as the Correlation Id.
* If "X-Correlation-Id" header is NOT present, a Correlation Id is randomly generated.

**For Incoming HTTP Requests:**  
In order to set a custom Correlation Id, the client invoking the HTTP request must set "X-Correlation-Id" header. This will ensure that the Mule Flow uses this Correlation Id.  
  
**For Outgoing HTTP Requests:**  
You can also propagate the existing Correlation Id to downstream APIs. By default, all outgoing HTTP Requests send "X-Correlation-Id" header. However, you can choose to set a different value to "X-Correlation-Id" header or set "Send Correlation Id" to NEVER.

**Versioning:**

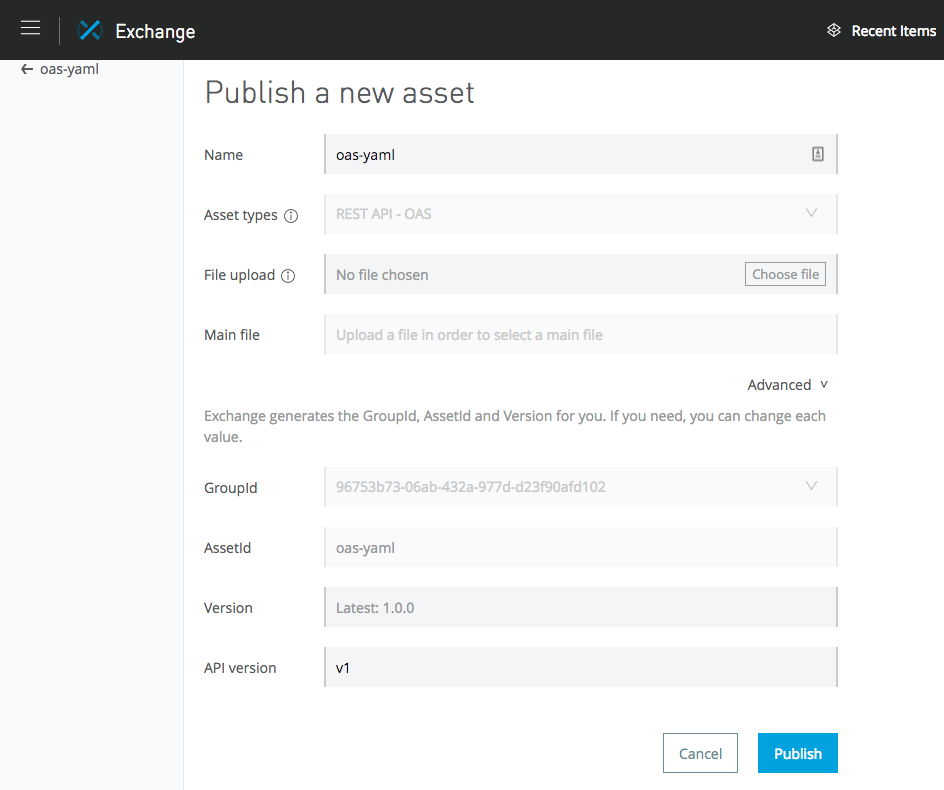
Exchange asset versions follow the [Semantic Versioning](https://semver.org/) model of major, minor, and patch releases. For example, if an asset is version 2.4.6, then its major version is 2.x.x, its minor version is 2.4.x, and its patch version is 2.4.6.

An HTTP API asset is an endpoint, to which you can provide only policies or proxies. HTTP APIs have only major version increases and do not have minor or patch version changes.

When creating a new version of an asset, the name, icon, and description of the previous version are used. To change these properties use the instructions in [Describe an Asset](https://docs.mulesoft.com/exchange/to-describe-an-asset). Any changes to the name, icon, or description apply to all versions of the asset.

**Deploy a New API Asset**

You can deploy an API asset directly into Exchange using the New asset button. In the Publish a new asset window, the Advanced section enables you to specify the version of the asset in the Version field, and the version of the API specification in the API version field.



Only change the API version when an updated API specification has changes that are not backward compatible from the new version to the old version.

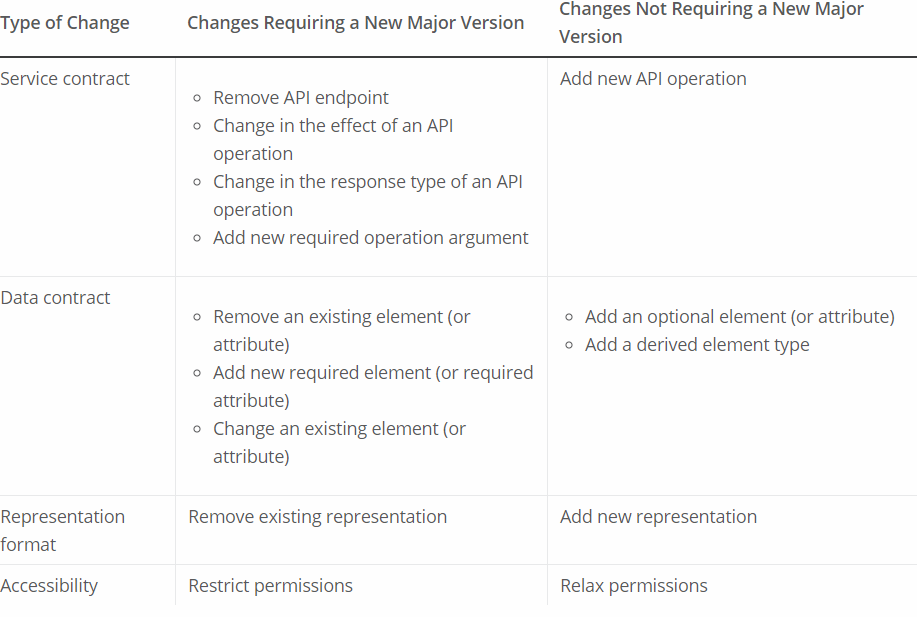
## Best Practices for API Versioning:

An API should be designed for long term use, but because change is inevitable, the API can never be completely stable.

What is important is to manage the changes through a comprehensive versioning strategy. This includes a multiple month deprecation schedule and documentation that is up to date and describes the change in sufficient level of detail.

Not all changes require a new version. APIs designed for backward compatibility normally do not require major changes and should use minor or patch changes.

Nevertheless, it is not always possible to design APIs in that way. The table below outlines when to introduce a new version and when a new version is not required. Typically when one clicks **Add new API operation** a minor version increase occurs.

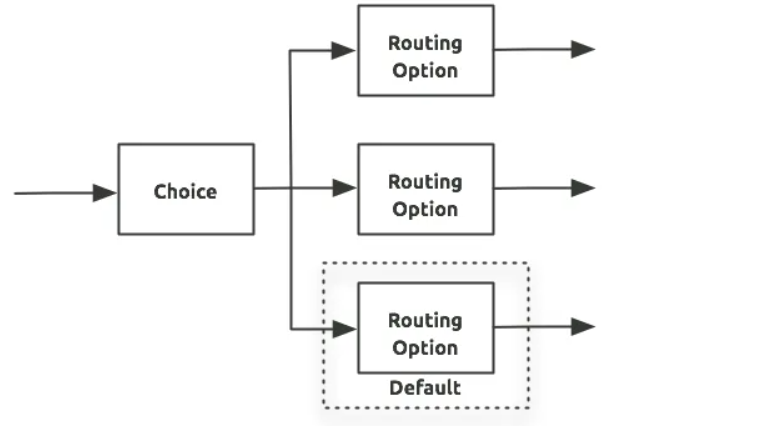


**Control Flows:**

The main task of Flow Control component is to take the input Mule event and route it to one or more separate sequences of components. It is basically routing the input Mule event to other sequence(s) of components. Therefore, it is also called as Routers. Choice and Scatter-Gather routers are the most used routers under Flow Control component.

**Choice** :The Choice router dynamically routes messages through a flow according to a set of DataWeave expressions that evaluate message content.

The Choice router enables [content-based routing](http://www.enterpriseintegrationpatterns.com/ContentBasedRouter.html), which is a common way to introduce routing logic based on content of the current message.



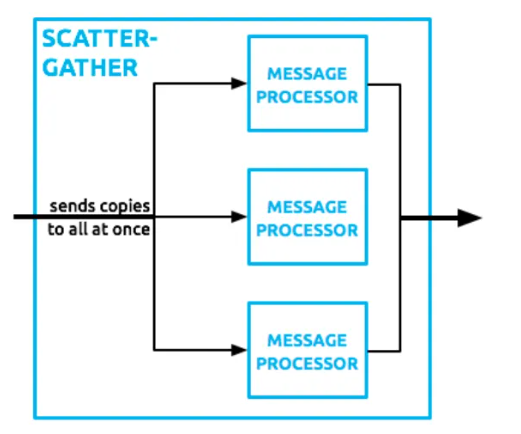
The choice route sends the event to one route based on the conditional logic.

The conditions are written with dataweave

**First Successful**: The First Successful router iterates through a list of configured processing routes until one of the routes executes successfully. If any processing route fails execution (throws an error), the router executes the next configured route.If none of the configured routes execute successfully, the First Successful router throws an error.

**Round Robin:** The Round Robin router iterates through a list of two or more routes in order, but it only routes to *one* of the routes each time it is executed. It keeps track of the previously selected route and never selects the same route consecutively. For example, the first time Round Robin executes, it selects the first route. The next time, it selects the second route. If the previously selected route is the last route in the list, Round Robin jumps to the first route.

**Scatter Gather**: The routing message processor **Scatter-Gather** sends a request message to multiple targets concurrently. It collects the responses from all routes, and aggregates them into a single message.



As its name implies, it works on the fundamentals of scatters (copy) and Gather (Consolidates). We can understand its working with the help of following two points −

* First, this router copies (Scatter) a Mule event to two or more parallel routes. The condition is that each route must be a sequence of one or more event processors which is like a sub-flow. Each route in this case will create a Mule event by using a separate thread. Every Mule event will have its own payload, attributes as well as variables.
* Next, this router gathers the created Mule events from each route and then consolidates them together into a new Mule event. After this, it passes this consolidated Mule event to the next event processor. Here the condition is that the S-G router will pass a consolidated Mule event to the next event processor only when every route is completed successfully.

**Error Handling by Scatter-Gather Router**

First, we must have knowledge on the kind of error that can be generated within Scatter-Gather component. Any error might be generated within event processors leading the Scatter-Gather component to throw an error of type Mule: COMPOSITE\_ERROR. This error will be thrown by the S-G component only after every route either fails or completes.

To handle this error type, a try scope can be used in each route of Scatter-Gather component. If the error is successfully handled by try scope, then the route will be able to generate a Mule event, for sure.

**Dataweave Operator: Using Map to Return an Array:**

(':array', ':function') ⇒ :array

Returns an array that is the result of applying a transformation function (lambda) to each of the elements. The lambda is invoked with two parameters: index and the value. If these parameters are not named, the index is defined by default as $$ and the value as $.

**Transform**

**%dw 1.0**

**%output application/json**

**---**

**users: ["john", "peter", "matt"] map  upper $**

**Output**

**{**

**"users": [**

**"JOHN",**

**"PETER",**

**"MATT"**

**]**

**}**

**Using Map on an Object:**

**(':object', ':function') ⇒ ':array'**

Returns an array with the values that result out of applying a transformation function (lambda) to each of the values in the object. The keys of the original object are all ignored by this operation and the object is treated as an array. To have access to the keys, you can use the operation mapObject instead. The lambda is invoked with two parameters: index and the value. If these parameters are not named, the index is defined by default as $$ and the value as $. The index refers to the position of a key:value pair when the object is treated as an array.

**Map Object**

(':object', ':function') ⇒ ':object'

Similar to Map, but instead of processing only the values of an object, it processes both keys and values as a tuple. Also instead of returning an array with the results of processing these values through the lambda, it returns an object, which consists of a list of the key:value pairs that result from processing both key and value of the object through the lambda.

The lambda is invoked with two parameters: key and the value. If these parameters are not named, the key is defined by default as $$ and the value as $.

**FlatMap**

Iterates over each item in an array and flattens the results.

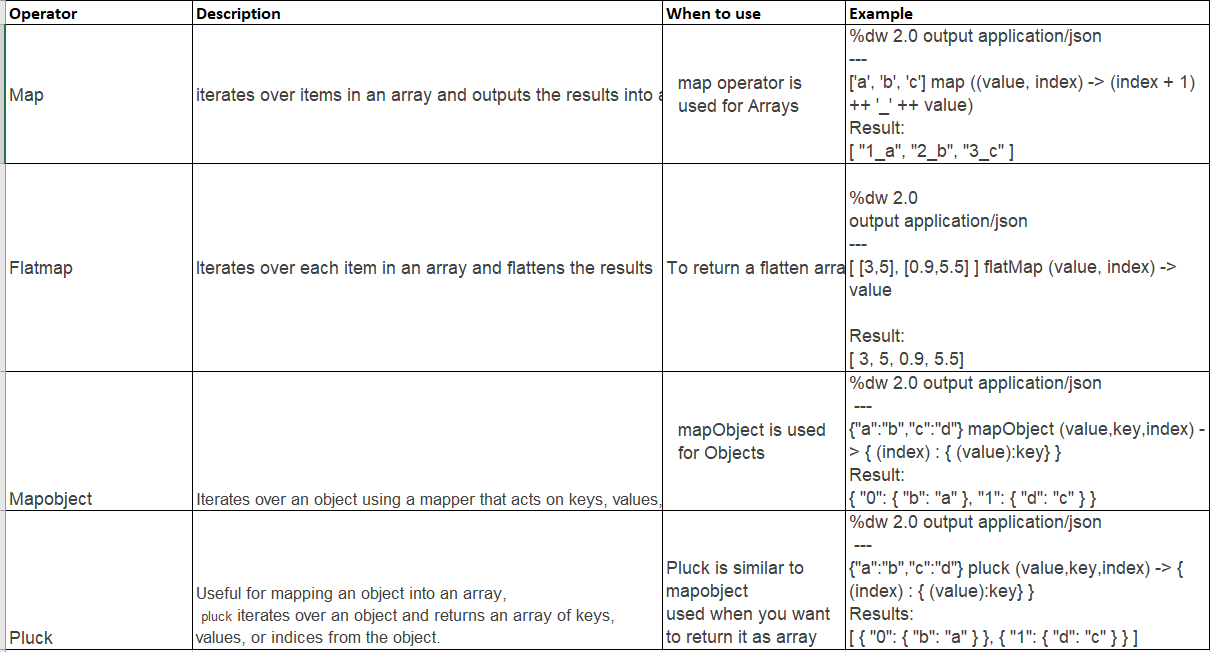
Instead of returning an array of arrays (as map does when you iterate over the values within an input like [ [1,2], [3,4] ]), flatMap returns a flattened array that looks like this: [1, 2, 3, 4]. flatMap is similar to flatten, but flatten only acts on the values of the arrays, while flatMap can act on values and indices of items in the array.

**Pluck**

(':object', ':function') ⇒ ':array'

Pluck is useful for mapping an object into an array. Pluck is an alternate mapping mechanism to mapObject. Like mapObject, pluck executes a lambda over every key:value pair in its processed object as a tuple, but instead of returning an object, it returns an array, which may be built from either the values or the keys in the object.

The lambda is invoked with two parameters: key and the value. If these parameters are not named, the key is defined by default as $$ and the value as $.



**Exception handling:**

Messaging error occurs when a event is processed through mule flow that throws an error

Normal execution stops and an event is passed through the first processor in the error handler.

If there is no error handler defined mule default error handler is use.

There are broadly two ways to define the errors in Mulesoft as below.

**Message Level Errors –** Message level error occurs when mule message is involved. e.g. error occurred when calling a database or error occurred (http 401 unauthorized or 503 service unavailable etc.) while calling an external API

**System Level Error –** System level error occurs when there is no mule message involved. E.g. connectivity issue to a JMS provider, connectivity issue to a Database. You cannot handle system level

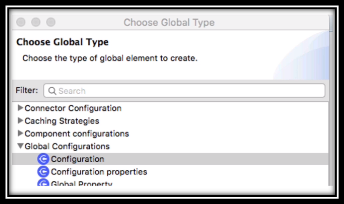
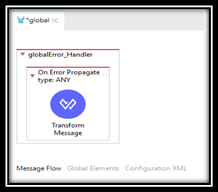
**Message Exception**: The message exceptions can now be handled at 3 different levels:

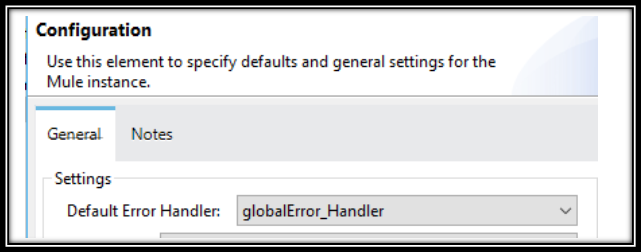
If there is no error handling written at any of the levels, the Mule Default Error Handler is used, which stops the execution of the flow and logs the exception.

The Mule Default Error Handler is not configurable but can be replaced our own Global error handler by creating a Configuration global element.

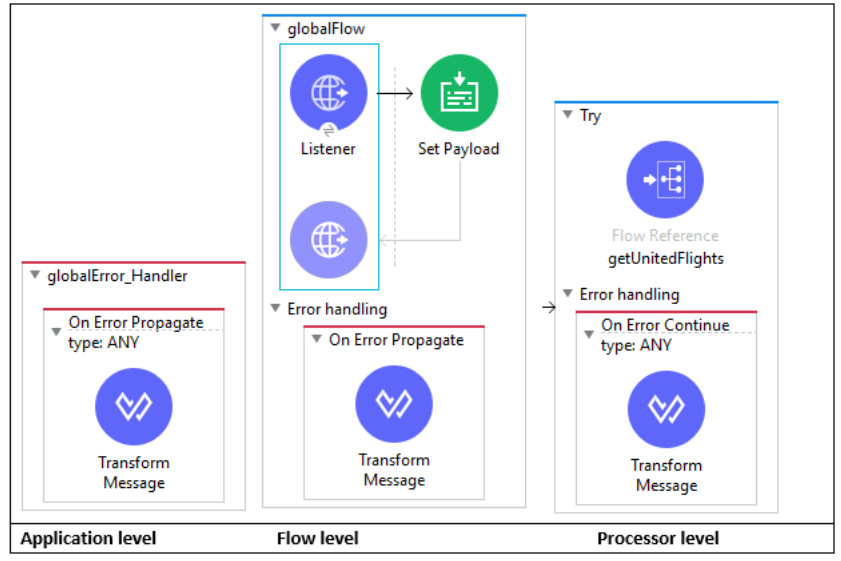
1. **Application level**-The handlers written at the application level are global handlers, which can be used to handle the errors thrown by any flow, which doesn't have its own error handling. Defining a default error handler for an application.

* Add a error handler in the global configuration file





1. **Flow level:** The errors that are handled in the flow of the application.Flow level handlers can be added to regular Flows and private flows, but not to Subflows.
2. **Processor Level:** The errors that are handled at the processor level.To handle the errors at the processor level, add one or more processors into a Try Scope and handle it using the On Error Propagate or On Error Continue Scopes.



**Error Handler Scopes:**

The error handling mechanisms in Mule 4.

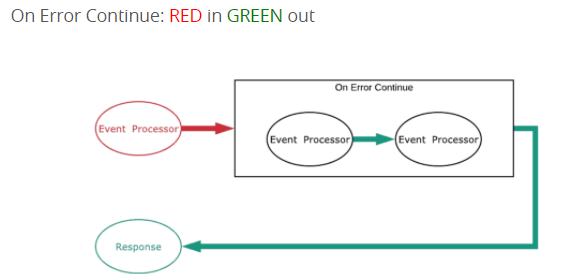
* On-Error Continue
* On-Error Propagate
* Try Catch Scope
* Raise Error

**On-Error Continue**

On-Error Continue catches the error and does not report it as an error, thus the processing of the flow continues even after the error has occurred. This error handler can be used in flows where you don’t want to stop the flow processing even if an error has occurred.

error continue scope handled as below:

* All processors in error handling scope are executed
* At the end of the scope
  + The rest of the scope that threw the error is not executed
  + The event is  passed to the next level as if the flow execution has completed successfully
* The HTTP listener returns a successful response

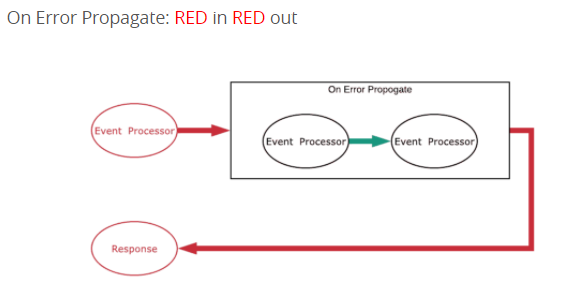


**On-Error Propagate**

In case of any errors, On-Error Propagate processes the error message and re-throws the error to its parent flow. No further processing is done on that particular flow.

 error propagate scope handled as below:

* All processors in error handling scope are executed
* At the end of the scope
  + The rest of the scope that threw the error is not executed
  + The error is rethrown upto the next level and handled there
* The HTTP listener returns an error response.



**Try-Catch Scope**

Try-catch scope can be used within a flow to do error handling of just inner components. Try-catch scope can be very useful in cases where we want to add a separate error processing strategy for various components in the flow.

**Custom error type:**

This core component generates a Mule error, as if a failure had occurred, which allows you to customize its description and type.

**Raise Core Runtime Error Types**

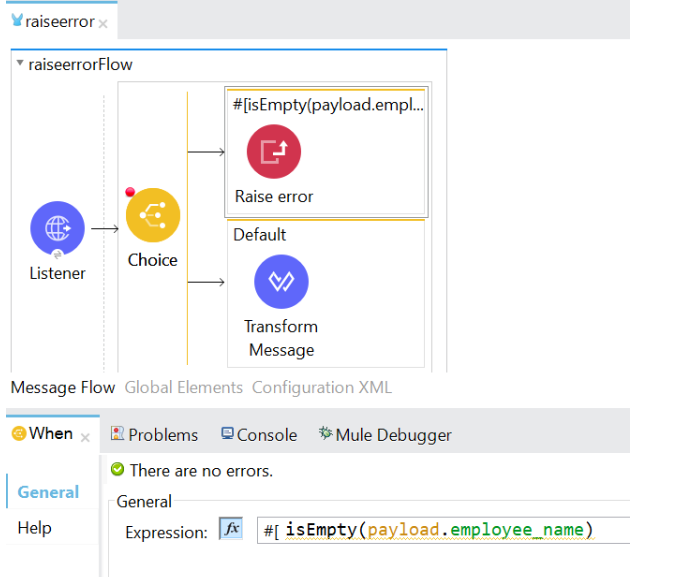
For core runtime error types, you must use the implicit namespace and identifier, you can only customize the error’s description message. For example:

<raise-error type=“MULE:CONNECTIVITY” description=“Error description message”/>

**Raise Custom Error Types**

For custom error types, declare a new namespace. The namespace of an error type should help you identify the origin of an error. For example:

<raise-error type=“ORDER:INVALID\_DATA” description=“Email is invalid. Cannot complete transaction”/>



**Transaction Types:**

Transactions are operations in a Mule app for which the result cannot remain indeterminate. When a series of steps in a flow must succeed or fail as one unit, Mule uses a transaction to demarcate that unit.

For example, you might use a transaction to encapsulate several steps in a flow that result in committing information to a database. In this type of scenario, the commit (or transaction) is either entirely complete and succeeds, or is incomplete and fails. Even if partially complete, the commit fails. When a transaction fails, Mule rolls back the operations within the transaction so that no part results in partial completion.

Implement a [Reliability Pattern](https://docs.mulesoft.com/mule-runtime/4.3/reliability-patterns) to design your application so it is capable of reliable messaging, even if the application receives messages from a non-transactional connector.

You can also configure Bitronix to manage transactions in your Mule application.

Mule supports Single Resource (Local, the default) and Extended Architecture (XA) transaction types (transactionType). The only components that can define the transaction type are message sources (For example, jms:listener and vm:listener) and the Try scope.

The following table describes the characteristics of each transaction type and the requisites for an operation to join the transaction:

