**How to access inbound and outbound properties in dataweave**

**http://www.a2cart.com/mule-esb-interview-questions/**

**Choice Router :**

Do not use choice routers to conditionally set variables, instead opt for a set-variable component and conditionally set the value using DataWeave.

We can use like this

**<set-variable value='#[if(condition) "1" else "2"]' doc:name="x" variableName="x"/>**

OR MEL or dataweave not Choice

Choice router isn’t meant to be used in every situation you would use an if-then-else statement. Instead, the choice router is meant to be used when you need to use conditional logic that will dictate how your message will be routed through your program.

For conditional throwing exception use find suitable validator components

**Frequently used Connectors in Projects:**

Custom file filter for File connector

[Apache Commons IO](http://commons.apache.org/proper/commons-io/javadocs/api-2.5/index.html) ships with Mule runtime and provides various file filters that can easily be used in Mule application.

File inbound endpoint element only allows configuring filter that implements java.io.FileFilter or java.io.FilenameFilter.

Write your own java customer filter using extending these classes and declare as beans and refer it from connector as reference.

**<file:inbound-endpoint path="input" moveToDirectory="output" responseTimeout="10000" doc:name="File">**

**<filter ref="fileFilterWrapper"></filter>**

**</file:inbound-endpoint>**

<https://blogs.mulesoft.com/dev/connectivity-dev/using-advanced-file-filters-in-file-inbound-endpoint/>

<https://blogs.perficient.com/2017/02/28/enabling-https-for-mule-application/>

By default, http.port = 8081, https.port = 8082, Cloudhub forwards port 80 to http.port (8081), 443 to port https.port (8082) to your worker VM which runs your application.

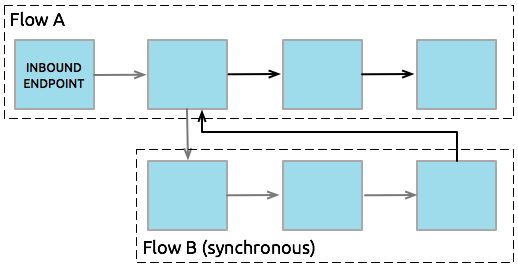
**Types of Flows**

When its execution is triggered by another flow in an application, a flow exists as one of three types:

|  |  |  |
| --- | --- | --- |
| **1** | **Subflow** | A subflow processes messages **synchronously** (relative to the flow that triggered its execution) and always inherits both the processing strategy and exception strategy employed by the triggering flow. While a subflow is running, processing on the triggering flow pauses, then resumes only after the subflow completes its processing and hands the message back to the triggering flow. |
| **2** | **Synchronous Flow** | A synchronous flow, like a subflow, processes messages **synchronously** (relative to the flow that triggered its execution). While a synchronous flow is running, processing on the triggering flow pauses, then resumes only after the synchronous flow completes its processing and hands the message back to the triggering flow. However, unlike a subflow, this type of flow *does not* inherit processing or exception strategies from the triggering flow.  This type of flow processes messages along a single thread, which is ideally suited to transactional processing. |
| **3** | **Asynchronous Flow** | An asynchronous flow simultaneously and **asynchronously** processes messages in parallel to the flow that triggered its execution. When a flow passes a message to an asynchronous flow, thus triggering its execution, it simultaneously passes a copy of the message to the next message processor in its own flow. Thus, the two flows – triggering and triggered – execute simultaneously and independently, each finishing on its own. This type of flow *does not* inherit processing or exception strategies from the triggering flow.  This type of flow processes messages along multiple threads. |

### About Synchronous Message Processing

When a flow triggers a synchronous flow or subflow, it passes programmatic control to the triggered flow and suspends its own message processing activity. For example, when the synchronous Flow B completes its sequence of message processing events, it passes programmatic control back to Flow A. The message that exits Flow B replaces the message in Flow A (see image below).



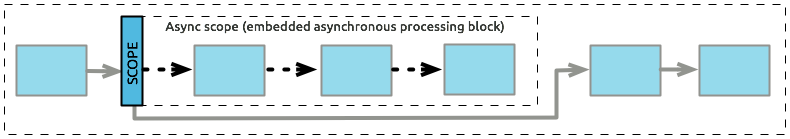
Since the Flow A and Flow B hand off programmatic control to each other and, by implication, all processing occurs on the same thread, each event in the message processing sequence can be tracked. This setup is is ideal for ensuring **transactional processing**.

| **Type of Flow** | **Component** | **Execution Relative to Triggering Flow** | **Exception and Processing Strategies** |
| --- | --- | --- | --- |
| Subflow | Flow Reference | synchronous | inherited |
| Synchronous Flow | Flow Reference | synchronous | not inherited |
| Asynchronous Flow | Flow Reference wrapped within an [Async Scope](https://docs.mulesoft.com/mule-runtime/3.8/async-scope-reference) | asynchronous | not inherited |

# Async Scope Reference

An **async scope** is a branch processing block that executes simultaneously with the parent message flow. This type of processing block can prove useful for executing time-consuming operations (such as printing a file or connecting to a mail server) — as long as those operations do not require sending a response back to the initiating flow. In other words, the main flow can continue execution while it initiates and processes the asynchronous scope; it does not have to pause until the last message processor embedded in the asynchronous flow has completed its task.

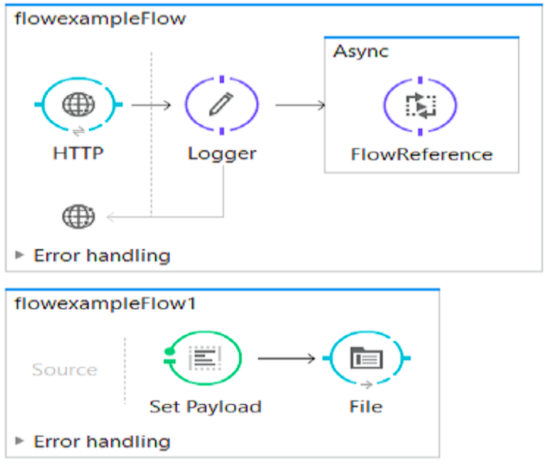
To facilitate this simultaneous branch processing, the async scope sends one copy of the message it has received to the first embedded message processor in its own processing block; at the same time it sends another copy of the message to the next message processor in the main flow (see below).



Since they operate on a copy of the message on a different thread, async scopes cannot, by definition, support request-response exchange patterns. Instead, they must implement one of several supported one-way processing strategies, as detailed in the configuration section, below.

If no processing strategy is configured for the async scope, Mule applies a queued-asynchronous processing strategy.

For asynchronous flows, the flow reference must be wrapped with async scope in the triggering flow.



## Async Scopes versus Asychronous Flows

An async scope is similar to an [**asynchronous flow**](https://docs.mulesoft.com/mule-runtime/3.8/flows-and-subflows) in that:

* It processes the message asynchronously with the main flow, so the message is simultaneously processed in the async scope without pausing the processing in the main flow thread
* Does not pass data from the scope back into the main flow thread
* It can have its own processing strategy

However, unlike an asynchronous flow, an async scope:

* Exists in-line with the main flow thread
* Is not called by a flow reference component
* Is not re-usable
* Cannot have its own exception handling strategy – it inherits this from the flow in which it resides

## Async Scopes versus Subflows

An async scope is similar to a [**subflow**](https://docs.mulesoft.com/mule-runtime/3.8/flows-and-subflows) in that it inherits the exception strategy of the main flow.

However, unlike a subflow, an async scope:

* Processes messages asynchronously
* Does not pass data back to the main flow
* Exists in-line with the main flow thread
* Is not called by a flow reference component
* Is not re-usable

\*Even though the Async scope receives a copy of the Mule message, the payload is not copied. The same payload objects are referenced by both Mule messages: One that continues down the original flow, and the one processed by the Async scope.

In other words, if the payload of your message is a mutable object (for example a bean with different fields in it) and a message processor in your async scope changes the value of one of the fields, the message processors outside of the Async scope see the changed values.

## Replacing versus Modifying Object References

If you replace, that is, change the reference completely inside the async scope, then both the payload and the flow variable in the original thread continue to have their original values.

If you modify, that is, make a change in the object referenced, but leave the same reference, the payload is modified for the original thread, but is preserved for the flow variable

**<flow name="replace">**

**<http:inbound-endpoint address="http://localhost:9000/replacepayload" exchange-pattern="request-response" />**

**<set-payload value="original payload" />**

**<set-variable value="original flowvar" variableName="testflowvar"/>**

**<logger level="WARN" message="original payload: #[payload]" />**

**<logger level="WARN" message="original flowvar: #[flowVars['testflowvar']]" />**

**<async>**

**<set-payload value="new payload" />**

**<set-variable value="new flowvar" variableName="testflowvar"/>**

**<logger level="WARN" message="Payload in async: #[payload]" />**

**<logger level="WARN" message="Flowvar in async: #[flowVars['testflowvar']]" />**

**</async>**

**<scripting:component>**

**<scripting:script engine="groovy">**

**<scripting:text>**

**Thread.sleep(3000)**

**return payload**

**</scripting:text>**

**</scripting:script>**

**</scripting:component>**

**<logger level="WARN" message="Payload after async: #[payload]" />**

**<logger level="WARN" message="Flowvar after async: #[flowVars['testflowvar']]" />**

**</flow>**

**<flow name="modify">**

**<http:inbound-endpoint address="http://localhost:9000/modifypayload" exchange-pattern="request-response" />**

**<set-payload value="#[['key':'originalvalue']]" />**

**<set-variable value="#[['key':'originalvalue']]" variableName="testflowvar"/>**

**<logger level="WARN" message="original payload: #[payload]" />**

**<logger level="WARN" message="original flowvar: #[flowVars['testflowvar']]" />**

**<async>**

**<set-payload value="#[payload.key = 'new payload'; return payload]" />**

**<set-variable value="#[['key':'new value']]" variableName="testflowvar"/>**

**<logger level="WARN" message="Payload in async: #[payload]" />**

**<logger level="WARN" message="Flowvar in async: #[flowVars['testflowvar']]" />**

**</async>**

**<scripting:component>**

**<scripting:script engine="groovy">**

**<scripting:text>**

**Thread.sleep(3000)**

**return payload**

**</scripting:text>**

**</scripting:script>**

**</scripting:component>**

**<logger level="WARN" message="Payload after async: #[payload]" />**

**<logger level="WARN" message="Flowvar after async: #[flowVars['testflowvar']]" />**

**</flow>**

### Configuring a Processing Strategy

Configuring a processing strategy is optional. Unless you explicitly define a different one, Mule applies the queued-asynchronous processing strategy to the scope. You can configure the **Processing Strategy** of the async scope to one of the following available processing strategies.

| **Strategy** | **Description** |
| --- | --- |
| Asynchronous Processing Strategy | Same as queued-asynchronous processing strategy (which is what Mule applies if no other processing strategy is configured) except that it doesn’t use a queue. Use this only if for some reason you do not want your processing to be distributed across nodes. |
| Custom Processing Strategy | A user-written processor strategy. |
| Queued-Asynchronous Processing Strategy | Uses a queue to decouple the flow’s receiver from the rest of the steps in the flow. It works the same way in a scope as in a flow. Mule applies this strategy unless another is specified. Select this if you want to fine-tune this processing strategy by:   * Changing the number of threads available to the flow. * Limiting the number of messages that can be queued. * Specifying a queue store to persist data. |
| Queued Thread Per Processor Processing Strategy | Not applicable to most use cases. Writes messages to a queue, then every processor in the scope runs sequentially in a different thread. |
| Thread Per Processor Processor Strategy | Not applicable to most use cases. Every processor in the scope runs sequentially in a different thread. |

# Transformers

A Transformer prepares a message to be processed through a Mule flow by enhancing or altering the message header or message payload. For example, if the message source that triggers your flow receives data in XML format, but a downstream message processor requires JSON-formatted data, one or more transformers positioned between the message source and the message processor can achieve the necessary translation. (Since no XML-to-JSON transformer exists, this particular example calls for a XML-to-Object transformer chained to an Object-to-JSON transformer.)

### Script Transformers

This type of transformer integrates a script to perform the transformation. One transformer is provided for each of the four supported scripting languages, and a fifth, generic transformer can implement a script written in any of the four languages such as a Groovy, Javascript, Python, or Ruby.

### Java Object Transformers

Each transformer in this group changes a Java object into another Java object, a Java object into some other data type (such as an HTTP request), or some non-Java data type (such as an HTTP response) into a Java object.

You can configure a transformer locally or globally.

You can chain transformers together so that the output from one transformer becomes the input for the next. To chain transformers, you create a space-separated list of transformers in the transformer-refs or responseTransformer-refs attributes or by creating multiple <transformer> elements as shown above.

transformer-refs="ByteArrayToString StringToObject ObjectToInputStream"

You could also configure this as follows:

<transformer ref="ByteArrayToString"/>

<transformer ref="StringToObject"/>

<transformer ref="ObjectToInputStream"/>

# Native Support for JSON

JSON is now natively supported in Mule, meaning you can work with JSON documents and bind them automatically to Java objects.

## Examples

For example, using AJAX, you usually receive JSON. From here, you can get a request for a javabean from the server side, and you can convert that automatically to JSON.

Another example, if you get a request from outside, such as a web service request, your REST type content could be JSON or XML, while internally the components would be javabeans.

In this case, the feature would automatically respond to a JSON request with a JSON response.

**Java Transformer :**

public class helloWorldTransformer extends AbstractMessageTransformer{

/\*\*

\* @param args

\*/

public Object transformMessage(MuleMessage message, String outputEncoding) throws TransformerException {

return "Hello World!";

}

}

<custom-transformer class="practice.SayHello" doc:name="Java" returnClass="java.lang.String"/>

There is another way to call Java classes using component.

**Invoke Component Reference**

public class GreetingService {

public String sayHello(String name) {

return String.format("Hello %s!", name);

}

public String sayGoodbye(String name) {

return String.format("Goodbye %s!", name);

}

}

**<spring:beans>**

**<spring:bean name="greetingService" lass="org.mule.invoke.GreetingService"/>**

**</spring:beans>**

<http:listener-config name="HTTP\_Listener\_Configuration" host="localhost" port="8081" doc:name="HTTP Listener Configuration"/>

<flow name="demoFlow">

<http:listener config-ref="HTTP\_Listener\_Configuration" path="/greeting" doc:name="HTTP"/>

**<invoke object-ref="GreetingService" method="sayHello" doc:name="Invoke"/>**

</flow>

</mule>

# Error Handling

From a high level perspective, errors that occur in Mule fall into one of two categories: **System Exceptions**, and **Messaging Exceptions**.

## System Exceptions

Mule invokes a **System Exception Strategy** when an exception is thrown at the system-level (that is, when no message is involved, exceptions are handled by system exception strategies). For example, system exception strategies handle exceptions that occur:

* During application start-up
* When a connection to an external system fails

When a system exception strategy occurs, Mule sends an exception notification to registered listeners, logs the exception, and — if the exception was caused by a connection failure — executes the [reconnection strategy](https://docs.mulesoft.com/mule-runtime/3.8/configuring-reconnection-strategies). System Exception Strategies are not configurable in Mule.

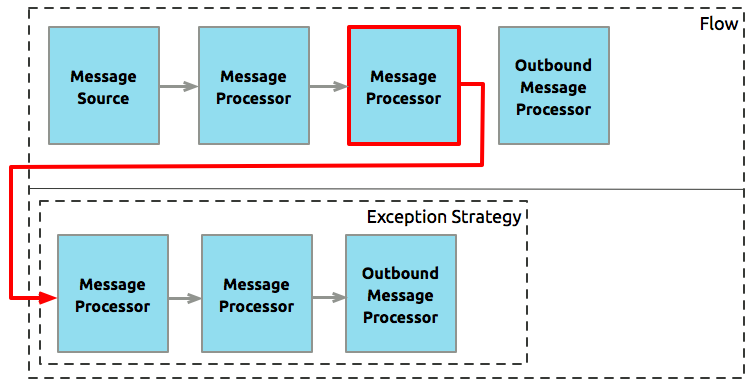
### Example Scenario

Mule establishes a connection to a JMS broker in order to receive a message. When Mule attempts to use the connection to consume a message the connection fails, which causes Mule to invoke the system exception strategy. Because the failure occurred before any message was received for processing, Mule invoked the system, rather than messaging, exception strategy.

## Messaging Exceptions

Mule invokes a **Messaging Exception Strategy** whenever an exception is thrown within a flow (i.e., whenever a message is involved, exceptions are handled by messaging exception strategies).

When a message being processed through a Mule flow throws an exception, normal flow execution stops and processes transfers to the message processor sequence within the exception strategy. You can incorporate any number of message processors – and in one case, other exception strategies – into an exception strategy to handle the exception precisely as you wish. The diagram below illustrates what happens when a message throws an exception.



Mule supports five types of messaging exception strategies, each of which is capable of handling errors that occur in flows which process transactions:

| **Exception Strategy** | **Use** | **Transaction Error Handling** |
| --- | --- | --- |
| [Default Exception Strategy](https://docs.mulesoft.com/mule-runtime/3.8/error-handling#default-exception-strategy) | Defined and implicitly applied by default to handle all messaging exceptions that are thrown in Mule applications | When a message throws an exception, the default exception strategy rolls back the message and logs the exception. |
| [Catch exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/catch-exception-strategy) | Define a catch exception strategy to customize the way Mule handles any exception. Catch exception strategies consume inbound messages. | When a message throws an exception, the catch exception strategy always commits the transaction and consumes the message. |
| [Rollback exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/rollback-exception-strategy) | Define a rollback exception strategy to ensure that a message that throws an exception in a flow is rolled back for reprocessing (if the message source supports redelivery). Rollback exception strategies do not consume inbound messages. | When a message throws an exception, the rollback exception strategy makes one or more attempts to rollback the message and redeliver it for processing (if the message source supports redelivery). If the message exceeds its redelivery attempts, then the rollback exception strategy takes the message from its inbound source and consumes the message. |
| [Reference exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/reference-exception-strategy) | Define a reference exception strategy to refer and adhere to the error handling parameters defined in a global catch, rollback or choice exception strategy. | When a message throws an exception, the reference exception strategy refers and adheres to the error handling parameters defined in a global catch, rollback or choice exception strategy. (The reference exception strategy itself never actually performs any rollback, commit, or consume activities.) |
| [Choice exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/choice-exception-strategy) | Define a choice exception strategy to customize the way Mule handles a message that throws an exception based on the message’s content at the moment it throws the exception. | When a message throws an exception, the choice exception strategy makes a decision about where to route the message for further processing. (The choice exception strategy itself never actually performs any rollback, commit, or consume activities.) |

### Characteristics of Messaging Exception Strategies

* Each flow can contain only one exception strategy. However this can be a choice exception strategy that then refers to other nested exception strategies.
* Choice exception strategies can contain one or more catch and/or rollback exception strategies. (Rollback and catch exception strategies cannot, however, contain other exception strategies.)
* Each exception strategy can contain any number of message processors.
* The exception strategy message processors should not throw exceptions, because you cannot create a nested exception strategy for these message processors.

## Other Ways of Handling Errors

### Until Successful Scope

**Until Successful** behaves similarly to a rollback exception strategy. This scope attempts to route a message through its child flow until the message is processed successfully. However, you can define the maximum number of processing attempts the Until Successful scope undertakes before it reverts to handling the message as though it were an exception.

### Exception Filter

Mule’s Exception filter stops normal flow execution when it discovers a message that contains a message in the exceptionPayload field. By comparison, an exception strategy typically stops normal flow execution when a message throws an exception in the flow.

# Rollback Exception Strategy

You can define a rollback exception strategy to ensure that a message that throws an exception in a flow is rolled back for reprocessing. Use a rollback exception strategy when you cannot correct an error that occurs in a flow. Usually, you use a rollback exception strategy to handle errors that occur in a flow that involve a [transaction](https://docs.mulesoft.com/mule-runtime/3.8/transaction-management). If the transaction fails, that is, if a message throws an exception while being processed, then the rollback exception strategy rolls back the transaction in the flow. If the inbound connector is transactional, Mule delivers the message to the inbound connector of the parent flow again to reattempt processing (that is, message redelivery).

Beyond managing transactional errors, you can use a rollback exception strategy to:

* Manage unhandled exceptions—​exceptions that the application fails to catch.
* Put in flows in which messages require redelivery.

A rollback exception strategy has the potential to introduce an infinite loop of activity within a flow: a message throws an error, the rollback exception strategy catches the exception and rolls the message back for reprocessing; the message throws an error again, the rollback exception strategy catches the exception again, and rolls the message back for reprocessing, and so on.

To avoid this infinite loop and responsibly manage unresolvable errors, you can apply two limitations to a rollback exception strategy:

* Define the maximum number of times that the rollback exception strategy attempts to redeliver the message for processing.
* Define a flow to handle messages that exceed the maximum number of redelivery attempts.

Mule attempts message redelivery when your flow uses one of the following two types of transports: **transactional** or **reliable**.

## Logger Component

If nothing is specified in the Message field, the logger logs the entire Mule message, including all session, inbound and outbound properties, as well as flow variables. As it could be very verbose to show all the contents of everything, the contents of the payload aren’t shown, only its type.

## Best Practice - Diferentiate Instances

If a flow uses several loggers, add some fixed text in the logger’s message to identify where it was genereated:

<flow name="FlowWithLoggers">

<http:listener config-ref="HTTP\_Listener\_Configuration1" path="/hello" doc:name="HTTP"/>

<logger level="INFO" message="Message before base64: #[message]" doc:name="Log message before"/>

<base64-encoder-transformer/>

<logger level="INFO" message="Message after base64: #[message]" doc:name="Log message after"/>

<vm:outbound-endpoint path="next.in.line" />

</flow>

## Using Script Component as a Logger

If you need more details about the message, a simple scripted logging component like the following can come handy:

<scripting:script name="Logger" engine="groovy">

<scripting:text>log.info(message); log.info(payload); message</scripting:text>

</scripting:script>

You can reference the script component from anywhere in your flow(s) using the name you give it, in thise case "Logger":

<flow name="FlowWithLoggers">

<http:listener config-ref="HTTP\_Listener\_Configuration1" path="hello" doc:name="HTTP"/>

<scripting:component script-ref="Logger" />

<base64-encoder-transformer/>

<scripting:component script-ref="Logger" />

<vm:outbound-endpoint path="next.in.line" />

</flow>

## Routers Reference Table

| **Message Processor** | **Description** |
| --- | --- |
| [All](https://docs.mulesoft.com/mule-runtime/3.8/routers#all) (Deprecated) | Broadcast a message to multiple targets |
| [APIkit Router](https://docs.mulesoft.com/apikit/4.x/) | Based on an API [RAML file](https://raml.org), it routes arriving calls to the corresponding flow depending on the resource and method. See [APIkit documentation](https://docs.mulesoft.com/apikit/4.x/). |
| [Async](https://docs.mulesoft.com/mule-runtime/3.8/routers#async) | Run a chain of message processors in a separate thread |
| [Choice](https://docs.mulesoft.com/mule-runtime/3.8/routers#choice) | Evaluates a message against specified criteria, then sends it to the first message processor that matches those criteria. |
| [Collection Aggregator](https://docs.mulesoft.com/mule-runtime/3.8/routers#collection-aggregator) | Checks the group tag (known as a Correlation ID) attached to each message in a group to create a collection of messages which share the same Correlation ID. |
| [Collection Splitter](https://docs.mulesoft.com/mule-runtime/3.8/routers#collection-splitter) | Accepts a collection of messages (or parts of messages), splits them into individual messages, then sends each new message, in sequence, to the next message processor in a flow. |
| [Custom Aggregator](https://docs.mulesoft.com/mule-runtime/3.8/routers#custom-aggregator) | Lets you write you own Java code to determine how messages are constructed and sent. |
| [Custom Processor](https://docs.mulesoft.com/mule-runtime/3.8/routers#custom-processor) | A custom-written message processor |
| [First Successful](https://docs.mulesoft.com/mule-runtime/3.8/routers#first-successful) | Sends a message to the next message processor within a "circular" list of processor targets. |
| [Idempotent Message Filter](https://docs.mulesoft.com/mule-runtime/3.8/routers#idempotent-message-filter) | Filter out duplicate message by message ID |
| [Idempotent Secure Hash Message Filter](https://docs.mulesoft.com/mule-runtime/3.8/routers#idempotent-secure-hash-message-filter) | Filter out duplicate message by message content |
| [Message Chunk Aggregator](https://docs.mulesoft.com/mule-runtime/3.8/routers#message-chunk-aggregator) | Checks the group tag (Correlation ID) of each message in a collection, selects all the messages whose group tag matches the specified value, then combines those messages into a single message which is then sent to the next message processor in an application flow. This is particularly useful for re-assembling the segments of a long message that has been received as multiple messages, each one consisting of a segment of fixed length created and sent by the Message Chunk Splitter. |
| [Message Chunk Splitter](https://docs.mulesoft.com/mule-runtime/3.8/routers#message-chunk-splitter) | Sections a message into segments of a specified length, then sends each segment, in sequence, to the next message processor in a flow. This is particularly useful when the message recipient cannot accept messages longer than a specified length. |
| [Message Filter](https://docs.mulesoft.com/mule-runtime/3.8/routers#message-filter) | Filter messages using a filter |
| [Processor Chain](https://docs.mulesoft.com/mule-runtime/3.8/routers#processor-chain) | Create a message chain from multiple targets |
| [Recipient List](https://docs.mulesoft.com/mule-runtime/3.8/routers#recipient-list) | Send a message to multiple connectors |
| [Request Reply](https://docs.mulesoft.com/mule-runtime/3.8/routers#request-reply) | Receive a message for asynchronous processing and accept the asynchronous response on a different channel |
| [Resequencer](https://docs.mulesoft.com/mule-runtime/3.8/routers#resequencer) | Accepts a collection of messages, then uses the Sequence ID of each message to reorder those messages. It then sends the messages (in order of their new sequence), to the next message processor in an application flow. |
| [Round Robin](https://docs.mulesoft.com/mule-runtime/3.8/routers#round-robin) | Iterates through a list of two or more message processors, sending successive messages to the next message processor on the list. When it reaches the end of the list, it jumps to the start of the list and resumes the iteration. |
| [Until Successful](https://docs.mulesoft.com/mule-runtime/3.8/until-successful-scope) | Repeatedly attempt to process a message until successful |
| [Scatter Gather](https://docs.mulesoft.com/mule-runtime/3.8/scatter-gather) | Sends a request message to multiple targets concurrently. It collects the responses from all routes, and aggregates them into a single message. |
| [SOAP Router](https://docs.mulesoft.com/apikit/3.x/apikit-for-soap) | Based on a WSDL file, it routes arriving calls to the corresponding flow depending on the resource and method. See [APIkit for SOAP documentation](https://docs.mulesoft.com/apikit/3.x/apikit-for-soap). |
| [Splitter](https://docs.mulesoft.com/mule-runtime/3.8/routers#splitter) | Evaluates an expression which determines how it sections a message into two or more parts. The Splitter then sends each of these message parts, in sequence, to the next message processor in an application flow. |
| [WireTap](https://docs.mulesoft.com/mule-runtime/3.8/routers#wiretap) | Send a message to an extra message processor as well as to the next message processor in the chain |

## Scopes Available in Studio

|  | **Scope** | **Description** |
| --- | --- | --- |
| async icon | Async | Creates a block of message processors that execute asynchronously while the rest of the flow continues to execute in parallel. For instance, you can populate an Async scope with a sequence of processors that perform logging so that logging does not slow down the rest of the application flow.  For specific information on configuring the Async Scope, see the [Async Scope Reference](https://docs.mulesoft.com/mule-runtime/3.8/async-scope-reference) page. |
| cache icon | Cache | Caches data produced by part of a flow. Wrap a cache scope around message processors in your flow so that it caches the response events produced within the scope. For specific information on configuring the Cache Scope, see the [Cache Scope](https://docs.mulesoft.com/mule-runtime/3.8/cache-scope) page. |
| composite source icon | Composite Source | To accept incoming messages from multiple input channels, place two or more message sources (also known as receivers) into a Composite Source. A message entering the Composite Source on any supported channel triggers the processing flow. |
| foreach icon | Foreach | Splits any type of message collection apart into individual messages for processing, and then aggregate them again at the end of the scope. For specific information on configuring the Foreach Scope, see the [Foreach Scope](https://docs.mulesoft.com/mule-runtime/3.8/foreach) page. |
| message enricher icon | Message Enricher | Appends information to a message, often using an expression to determine what part of the payload to evaluate so as to return an appropriate value to append to that payload. For example, the expression can evaluate a ZIP code and then append the associated City and State to the payload. The message processor is executed and the enricher scope uses the result of that execution to enrich the message coming into the scope. |
| poll icon | Poll | Periodically polls an embedded message receiver for new messages. For example, set a Poll to retrieve email at regular intervals by placing a request-response connector such as SMTP within the Poll processing block. |
| sub flow icon | Sub Flow | A flow that is called by another flow. Sub flows inherit their properties from the flow reference and are always synchronous. This type of scope can be very useful when you need to reuse code at several points within the same flow. Simply place (and configure) Flow Reference Components wherever you want the sub flow processing block to execute. |
| transactional icon | Transactional | Mule applies the concept of [transactions](http://en.wikipedia.org/wiki/Transaction_processing) to operations in application for which the result cannot remain indeterminate. In other words, where a series of steps in flow must succeed or fail as one unit, Mule uses a transaction to demarcate such a unit. |
| until successful icon | Until Successful | Attempts, at a specified interval, to route a message to an embedded message processor until one of the following occurs:   * The message processor succeeds * The maximum number of retries is reached * An exception is thrown   Thus, Until Successful can prove useful in sending messages to resources, such as shared printers, which might not always be immediately available. |

# Shared Resources

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 the elements in it. Once defined, any Mule application associated with a particular domain can access resources in this file. Note that Mule applications can be associated with only one domain at a time.

Shared resources allow multiple development teams to work in parallel using the same set of reusable connectors. Defining these connectors as shared resources at the domain level allows the team to:

* Expose multiple services within the domain through the same port
* Share the connection to persistent storage
* Share services between applications through a well-defined interface
* Ensure consistency between applications upon any changes, as the configuration is only set in one place

<https://docs.mulesoft.com/mule-runtime/3.8/shared-resources>

# Object Scopes

A scope, also referred to as cardinality, describes how Mule creates and manages objects in a Mule container.

Object scopes defined in Mule:

* Singleton - Only one object creates for all requests. Singleton objects must be thread-safe, because multiple threads access the same object. Therefore, any member variables need to be guarded when writing to ensure only one thread at a time changes the data.
* Prototype - A new object creates for every request or every time the object is requested from the registry. Objects that are given prototype scope are created for each request on the object, so the object does not need to be thread-safe, since there can only ever be one thread accessing it. However, the object must be stateless, because member variables only exist for the lifetime of the request.

See Also: [About Transformers](https://docs.mulesoft.com/mule-runtime/3.8/object-scopes#about-transformers)

* Pooled - Only applies to component objects, but these are stored in a pool that guarantees that only one thread can access an object at a time. Pooled objects are thread safe since Mule guarantees that only one thread can access the object at a time. Pooled objects can’t easily maintain state on the object itself since multiple instances are created. The advantage of pooled over prototype is when an object may be expensive to create, and creating a new instance for every message it receives, slows down the application.

<component >

<spring-object ref="myBean">

</component>

## Example: Singleton

<component >

<singleton-object class="com.foo.Bar"/>

</component>

## Example: Prototype

<component >

<prototype-object class="com.foo.Bar">

</component>

<!—or short form -->

<component class="com.foo.Bar"/>

## Example: Pooled

<pooled-component class="com.foo.Bar"/>

## Advantages of Anypoint Enterprise Security

Anypoint Enterprise Security adds new features on top of Mule Enterprise’s existing security capabilities. Mule already provides the following security features:

* [Mule Security Manager](https://docs.mulesoft.com/mule-runtime/3.8/configuring-the-spring-security-manager), client authentication and authorization on inbound requests as well as credential mapping for outbound calls
* [LDAP](https://docs.mulesoft.com/mule-runtime/3.8/setting-up-ldap-provider-for-spring-security) and third party identity management system integration
* [Validation of inbound requests](https://docs.mulesoft.com/mule-runtime/3.8/enabling-ws-security) through the SAML 2.0 federated identity standard
* [Secure FTP (SFTP) Transport](https://docs.mulesoft.com/mule-runtime/3.8/sftp-transport-reference) that enables Mule flows to read and write to remote directories over the SSH protocol.

# Tuning Performance

A Mule application is a collaboration of a set of flows. Conceptually, messages are processed by flows in three stages:

1. Message receipt by the inbound connector
2. Message processing
3. Message transmission an outbound connector

**Stage 1** always happens first. Stages 2 and 3 can be interleaved, since a flow can intermix message processors and outbound endpoints.

Tuning performance in Mule involves analyzing and improving these three stages for each flow. You can start by applying the same tuning approach to all flows and then further customize the tuning for each flow as needed.

## About Threads in Mule

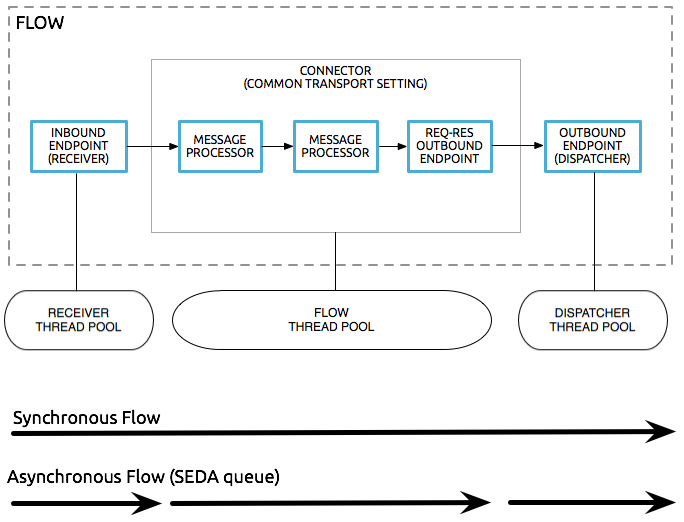
Each request that comes into Mule is processed on its own thread. A connector’s receiver has a thread pool with a certain number of threads available to process requests on the inbound endpoints that use that connector.

Keep in mind that Mule can send messages asynchronously or synchronously. Messages are processed asynchronously, unless one of the following is true:

* The flow uses the synchronous processing strategy
* The flow takes part in a transaction
* The inbound endpoint which received the message uses the request-response message exchange pattern

**Note that several kinds of threads are used to process a message:**

* The receiver thread, which originally receives the message, either:
  + Processes the entire flow (synchronous), or
  + Ends by writing the message to a SEDA queue (asynchronous)
* The flow thread, which processes the bulk of the flow (asynchronous)
* Dispatcher threads, which send messages to one-way endpoints (asynchronous)
* The following diagram illustrates these threads.



# Choosing the Right Clustering Topology

You can deploy Mule in many different topologies. As you build your Mule application, it is important to think critically about how best to architect your application to achieve the desired availability, fault tolerance, and performance characteristics. This page outlines some of the solutions for achieving the right blend of these characteristics through clustering. There is no one correct approach for everyone, and designing your system is both an art and a science.

Deploying an application into a cluster is useful for achieving the following:

* High availability (HA): making your system continually available in the event that one or more servers in the cluster, or a data center, fails.
* Fault tolerance (FT): ensuring recovery from failure of an underlying component. Typically, the recovery is achieved through transaction rollback or compensating actions.
* Scaling: ensuring that your application can scale horizontally to meet increased demand.

Seamless failover is made possible by a distributed memory store that shares all transient state information among clustered Mule instances, such as:

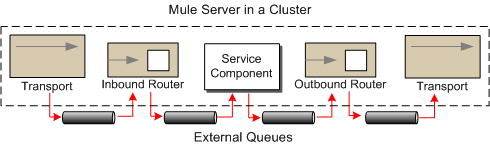
* SEDA service event queues
* In-memory message queues

Mule High Availability is currently available for the following transports:

* HTTP (including CXF Web Services)
* JMS
* WebSphere MQ
* JDBC
* File
* FTP
* Clustered (replaces the local VM transport)

## JMS Queues

JMS can be used to achieve HA & FT by routing messages through JMS queues. In this case, each message is routed through a JMS queue whenever it moves from component to component.



**Load Balancers**

Load balancers simply route requests to different servers based on the the current load of each server and which servers are currently up. Load balancers can be software or hardware based. This approach is commonly used with clustered databases (see below).

Pros:

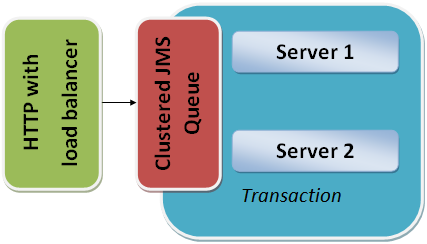
* Straightforward to do
* Well understood by developers

Cons:

* Not a complete solution on its own, doesn’t provide fault tolerance.

## Example

In this example architecture, HTTP requests come in through a load balancer and are immediately put on a JMS queue. The JMS queue is clustered between the different servers. A server will start processing a message off the JMS queue and wrap everything in a transaction.



If the server goes down, the transaction will roll back and another server will pick up the message and start processing it.

## Maintaining State in a Database

If you have a clustered database, one option for your application is to simply store all state in the database and rely on it to replicate the data consistently across servers.

**Handling Stateful Components**

While most applications can be supported by the above techniques, some require sharing state between JVMs more deeply.

One common example of this is an aggregator component. For example, assume you have an aggregator that is aggregating messages from two different producers. Producer #1 sends a message to the aggregator, which receives it and holds it in memory until Producer #2 sends a message.

Producer #1 ---> |----------|

|Aggregator| --> Some other component

Producer #2 ---> |----------|

If the server with the aggregator goes down between Producer #1 sending a message and Producer #2 sending a message, Producer #2 can’t just send its message to a different server because that server will not have the message from Producer #1.

The solution to this is to share the state of the aggregator component across different machines through clustering software such as Terracotta, Tangosol Coherence, JGroups, etc. By using one of these tools, Producer #2 can simply fail over to a different server. Note that MuleSoft has not tested Mule with these tools and does not officially support them.

Pros:

* Works for all clustering cases
* Can work as a cache as well

Cons:

* Not officially supported by MuleSoft
* Requires performance tuning to get things working efficiently

# Reliability Patterns

## Overview

A high-reliability application (one that has zero tolerance for message loss) not only requires the underlying ESB to be reliable, but that reliability needs to extend to individual connections. If your application uses a transactional transport such as JMS, VM, or JDBC, reliable messaging is ensured by the built-in support for transactions in the transport. This means, for example, that you can configure a transaction on a JMS inbound endpoint that makes sure messages are only removed from the JMS server when the transaction is committed. By doing this, you ensure that if an error occurs while processing the message, it will still be available for reprocessing.

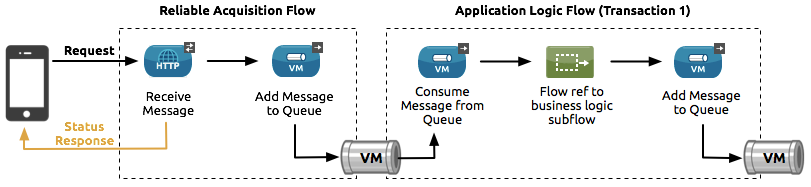
In other words, the transactional support in these transports ensures that messages are delivered reliably from an inbound endpoint to an outbound endpoint or between processors within a flow. Note though that if you want to move messages between different transports that support transactions you need to use XA transactions to ensure that both transports' transactions are committed as one atomic unit. See [Transaction Management](https://docs.mulesoft.com/mule-runtime/3.8/transaction-management) for more information on XA and other types of transactions.

However, suppose you have a web application that uses a non-transactional transport such as HTTP. How do you ensure reliable messaging for your application? The answer is to follow a reliability pattern.

Note that VM file persistency is disabled on clusters, so VM endpoints persist in-memory in a clustered topology.

**What is a Reliability Pattern?**

A **reliability pattern** is a design that results in reliable messaging for an application even if the application receives messages from a non-transactional transport. A reliability pattern couples a reliable acquisition flow with an application logic flow, as shown in the following diagram.



The **reliable acquisition flow** (that is, the left-hand part of the diagram) delivers a message reliably from an inbound endpoint to an outbound endpoint, even though the inbound endpoint is for a non-transactional transport. The outbound endpoint can be any type of transactional endpoint such as VM or JMS. If the reliable acquisition flow cannot deliver the message, it ensures that the message isn’t lost:

* For socket-based transports like HTTP, this means returning an "unsuccessful request" response to the client so that the client can retry the request.
* For resource-based transports like File or FTP, it means not deleting the file, so that it can be reprocessed.

The **application logic flow** (that is, the right-hand side of the diagram) delivers the message from the inbound endpoint (which uses a transactional transport) to the business logic for the application.

**General Considerations**

Here are a number of things to consider in implementing the reliability pattern:

* Always use a transaction when the transport allows you to do so.
* Always use a synchronous processing strategy in the acquisition flow.
* Use an XA transaction for bridging transports, that is, where you want to enlist multiple managed resources within the same transaction.
* The reliability of JMS is tied to the MQ implementation and how it is configured. Most MQ implementations allow you to configure whether messages are to be stored in memory only or to be persisted. You can achieve reliability only if you configure the MQ server to persistently store messages before sending them forward. Otherwise, you risk losing messages in case of an MQ server crash.
* Reliability has performance implications.
* If the outbound transport in the reliable acquisition flow is not transactional (for example, a flow from file-to-FTP), the only way to ensure message delivery is to turn off threading on the respective connector. To understand this, imagine if an exception occurs while sending the message to the outbound endpoint (this might happen if the FTP server is down). If threading is not turned off, the caller may not notice the exception. That’s because the exception occurred in a different thread, and there is no way that one thread can see exceptions that occur in another thread. The following example shows how to turn off threading in the connector:

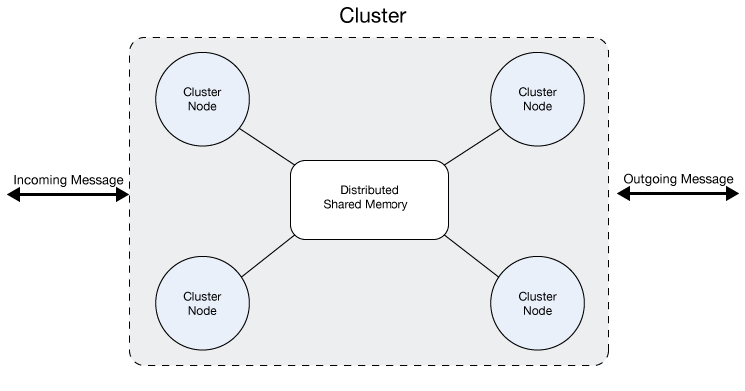
<ftp:connector name="ftpConn">

<dispatcher-threading-profile doThreading="false"/>

</ftp:connector>

# Mule Runtime High Availability (HA) Cluster Overview

A **cluster** is a set of Mule runtimes that acts as a unit. In other words, a cluster is a virtual server composed of multiple nodes. The nodes (Mule runtimes) in a cluster communicate and share information through a distributed shared memory grid. This means that the data is replicated across memory in different physical machines.



MuleSoft Homepage

