MuleSoft®

What's new with Mule 4?

Contents

Introduction: Meet Mule 4

Mule achieved amazing popularity in the 7 years since Mule 3.0 was released. Today, Mule is used by over 1,100 enterprise customers located in more than 60 countries across every major industry, and has a community of over 175K developers. With these metrics in mind, the strong reputation of Mule 3 was top of mind as Mule 4 was being developed.

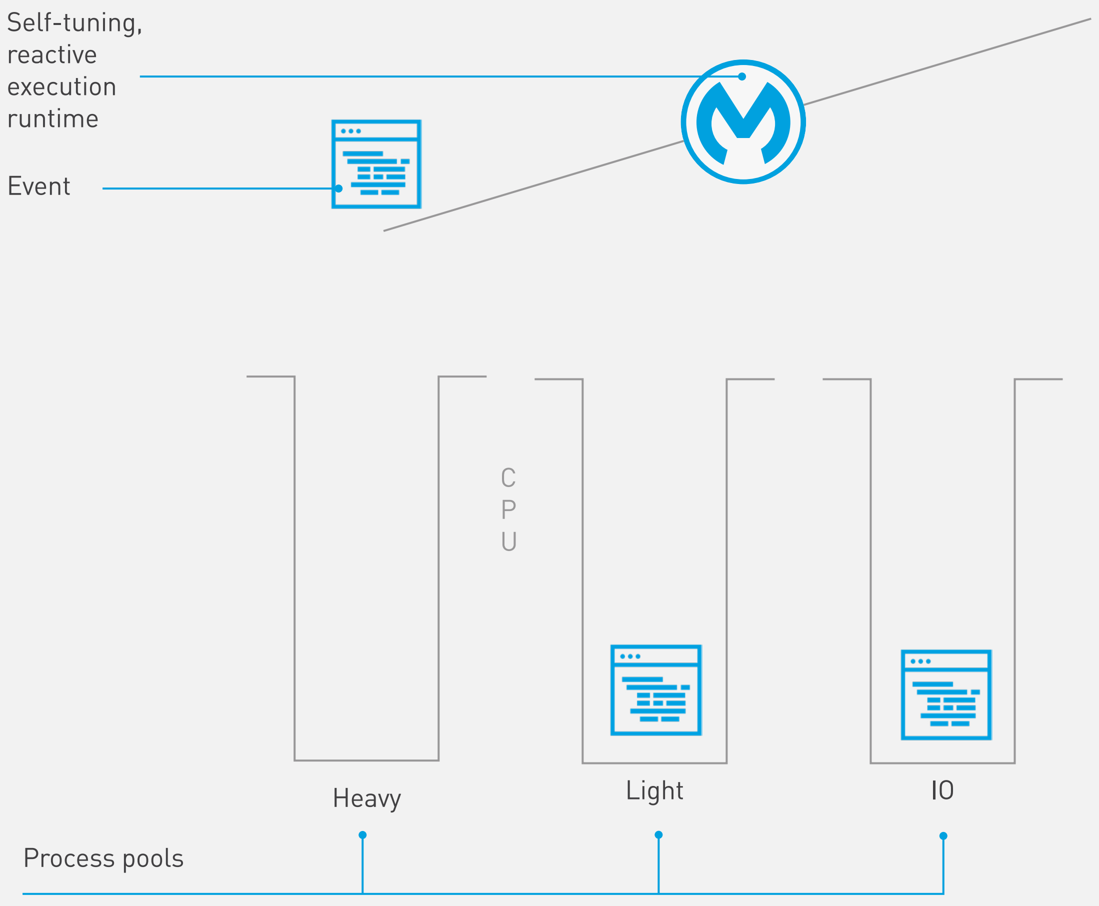
Mule 4 is a major evolution of the core runtime engine behind Anypoint Platform, and delivers innovation while keeping true to Mule fundamentals. Mule 4 enhances application development velocity with up to 50% fewer steps and concepts to learn, including: a simplified language for connectivity, a new error handling framework, consistent methods for accessing data, automatic tuning, and smoother upgrade processes.

Mule 4 is built with the fundamentals of Mule 3 in mind. The development evaluated ease of use, task simplification, runtime optimization and tuning, and long-term maintenance costs to ensure Mule 4 can power application networks by:

- Improving speed of delivery and accelerating developer on-ramp through a simplified language, error handling, and event and message model.
- Managing streams and larger-than-memory payloads transparently.
- Generating connectors automatically from RAML specs with REST connect.
- Creating powerful custom connectors and components with Mule SDK.
- Providing a guided development experience for integration novices with flow designer.
- Simplifying upgrades with classloader isolation.

Mule 4 is that next generation runtime engine, and we are excited by all that it can do.

Self-tuning engine



Mule 4 ships with a new reactive, non-blocking execution engine. This engine makes it possible to achieve optimal performance without having to do manual tuning steps, such as declaring exchange patterns, processing strategies or threading configuration. This is a task-oriented execution model that allows you to take advantage of non-blocking IO calls and avoid performance problems due to incorrect processing strategy configurations.

The Message processor can now inform the runtime if an operation is a CPU intensive, CPU light, or IO. Then the runtime self-tunes, or sizes the pools automatically, for different workloads dynamically. This removes the need for users to manage thread pools manually. As a result, Mule 4 removes complex tuning requirements to achieve optimum performance.

DataWeave

Mule 4 introduces DataWeave as the default expression language, replacing Mule Expression Language (MEL) with a scripting and transformation engine. In prior versions of Mule, there were a variety of evaluators with MEL to handle different inputs, like Groovy and JSON. While MEL handled these expressions and created a consistent experience to deal with these expressions, it did not handle transformations.

Extensive efforts were made by the DataWeave team on the language version 2.0. The team not only supported the runtime integration, but also worked to improve the language itself. This results in innovations like:

- *Reusing code broadly*: It is easy to now package and import scripts into others, enabling users to reuse and share code.
- *Access data without transformations*: Users no longer need to convert data into a set of objects to access it; instead, they can write expressions to access the data stored in CSV, JSON, XML or other forms directly.
- *Simplified syntax*: All operators are now functions, making DataWeave easier to learn.
- *Java interoperability*: Static methods can be executed through DataWeave.
- *New advanced capabilities*: Call Java functions directly, use multi-line comments, and define function types and variable types with type inference.

Transformers and DataMapper were the tools available for basic manipulation. But as the integration landscape grew, the amount of transformations and the complexity behind those transformations grew as well. DataWeave was introduced in 2015 to handle more complex transformations with high performance; with a highly performing transformation engine and rich querying capabilities—DataWeave became a hit.

With DataWeave, the expressions are focused on the structure of our data, rather than its format. This is because a Java array is the same as a JSON one in DataWeave; in other words, users no longer need different expressions to handle them. Users can now query and evaluate any expression without first transforming it into a Java object.

Binary data can now be accessed anywhere with larger-than-memory, random, and repeatable access.

Mule 4 runtime gives DataWeave all data regarding the current execution, including payload, variables, expected output, and metadata. This enables DataWeave to know, for example, whether a String or a map is required, how to handle each variable, whether to coerce a type, and more. Simple one-line expressions are written as:

```
#[payload ++ variables.myStatus]
#[attributes.statusCode]
```

In this example, the payload, variables, and attributes keywords will be interpreted as their respective form.

Set an HTTP request header with DataWeave

```
#[output application/java --- payload ++ {host: ‘httpbin.
org’}]
```

How does this work? DataWeave infers the output type, rather than relying on the user to explicitly declare the output format. While users can declare the output type, it is not necessary. A JSON payload now sets an HTTP request header, taking DataWeave’s existing map of headers and adding it to the script.

Now, when a request is made, the backend will answer with the received headers that contain the values sent to the HTTP listener as a body and the added host.

Full integration

Improvements around routing and attribute one-liners is not the only improvement with DataWeave. Another simplification is for flows. The number of transform elements needed have been decreased by defining content "inline." For example, when building the content of a file *inside* the File connector's write component, there is no need to use a 'transform' component in order to get the payload beforehand. The user does not need *additional steps* to iterate the received JSON payload; the new file path is determined with the expression:

```
#[payload.name ++ '.' ++ dataType.mimeType.subType]
```

Also, users can add a desired "date" attribute within the write operation, exactly where it's needed, setting the content to:

```
#[payload ++ { date : now() }]
```

That last expression is a great example of the output type being inferred. Since the user knows the payload is JSON, there is no need to specify the output; the generated files will be in that format as well.

This works for all new connectors since they're supported by the new Mule SDK. In the example below, the HTTP body of a request is set with a script, where users could take advantage of all of DataWeave's features—as in any transform component script:

```
#[
%dw 2.0

output application/json
---
payload ++ {location : 'LATAM', office : 'BA'}
]
```

Additionally, the listener response body can be configured with the expression:

```
#[payload.data]
```

This is because the backend server will return a JSON, where that attribute represents the payload sent to it. So, the data received by the listener will be modified to include some more attributes, and later forwarded back.

Compatibility with Mule Expression Language

DataWeave is the primary and default expression language for Mule 4. MEL is deprecate, however every expression can feature the `mel:` prefix to indicate that it should be evaluated with MEL.

For example, the HTTP listener below will answer whether a variable starts with "SUCCESS" or not, using MEL to configure the response body with the following expression:

```
#[mel:flowVars.receivedMessage.startsWith('SUCCESS').
toString()]
```

Error handling

Error handling in Mule 4 is different from Mule 3. Before, error handling was Java exception handling, without a way of communicating what kind of error each component threw. It was also a "trial and error" experience, where users would check source code frequently and force the error several times to see what happened.

Error handling with Mule 4 introduces a new experience

While Java Throwables are still available, the main error handling mechanism in Mule 4 is based on the Mule runtime engine's own errors: if something goes wrong with a component, Mule provides a clear error that is typed accordingly, with useful data on the problem. The problem can then be easily routed to a handler.

More importantly, each component declares the type of errors it may throw, so users are able to identify all potential errors at design time.

Mule errors

Execution failures are represented with Mule errors that have the following components:

- A description regarding the problem
- A type, used to characterize the problem
- A cause, the underlying Java Throwable that resulted in the failure
- An optional error message, which is used to include a proper Mule Message regarding the problem

For example, when an HTTP request fails with a 401 status code, a Mule Error provides the following information:

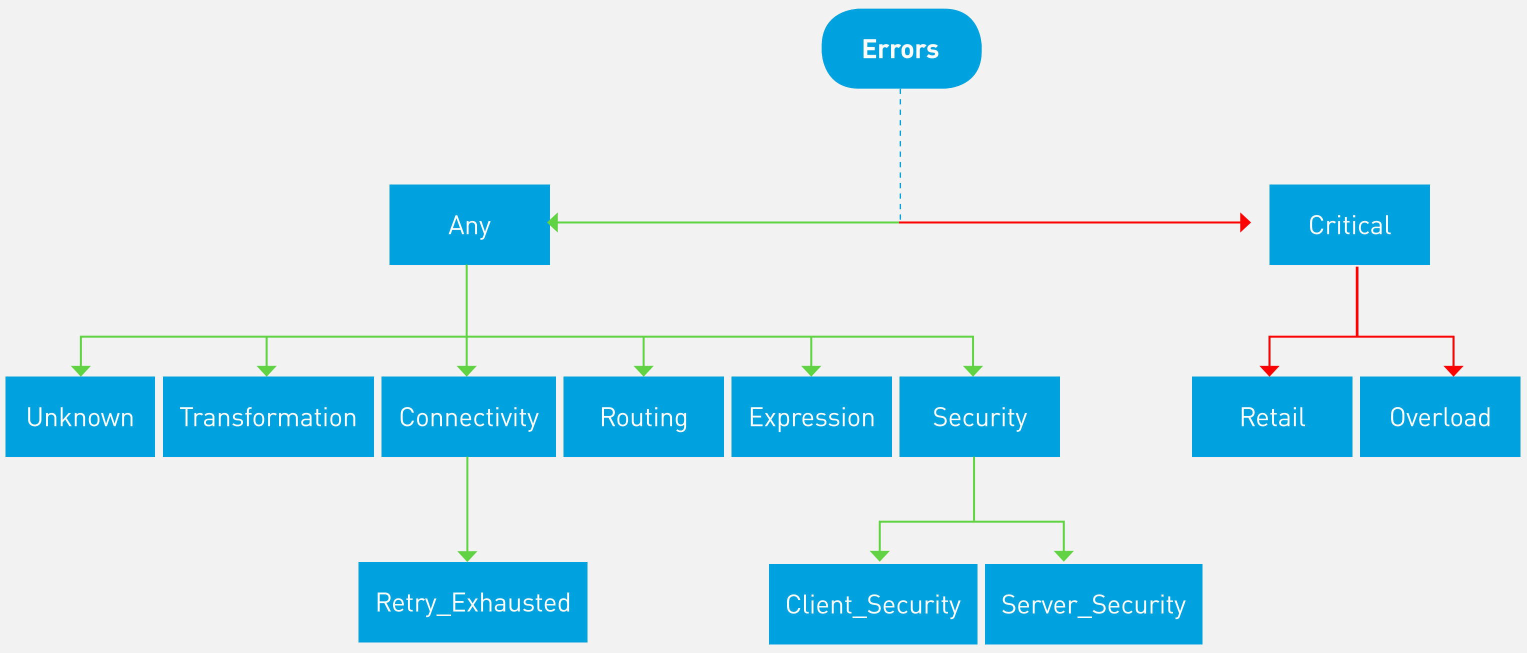
```
Description: HTTP GET on resource ‘http://localhost:36682/
testPath’ failed: unauthorized (401)

Type: HTTP:UNAUTHORIZED

Cause: a ResponseValidatorTypedException instance

Error Message: The entire HTTP 401 response received, including
the body and headers
```

Error types

In the example above, the error type is `HTTP:UNAUTHORIZED`, not just `UNAUTHORIZED`. This is because error types consist of a namespace and an identifier, allowing users to distinguish the types according to their domain, like `HTTP:NOT_FOUND` and `FILE:NOT_FOUND`. While connectors define their own namespace, core runtime errors have an implicit one: `MULE:EXPRESSION` and `EXPRESSION` are interpreted as one namespace.

Another important characteristic of error types is that they may have a parent type. For example, `HTTP:UNAUTHORIZED` has `MULE:CLIENT_SECURITY` as the parent, which, in turn, has `MULE:SECURITY` as the parent. This establishes error types as specifications of more global ones: an HTTP unauthorized error is a kind of client security error, which is a type of a more broad security issue.

These hierarchies mean routing can be more general, since, for example, a handler for `MULE:SECURITY` will catch HTTP unauthorized errors as well as OAuth errors. Below is a visual of can see what the core runtime’s hierarchy looks like:



All errors are either `GENERAL` or `CRITICAL`, the latter being so severe that they cannot be handled. At the top of the general hierarchy is `ANY`, which allows matching all types under it. The `UNKNOWN` type is used when no clear reason for the failure is found.

When it comes to connectors, each connector defines its error type hierarchy considering the core runtime one. However, `CONNECTIVITY` and `RETRY_EXHAUSTED` types are always present since they are common to all connectors.

The most important thing about these error types is that they are part of each component's definition, so during the design process, users will be aware of any failures that could occur.

Error handler

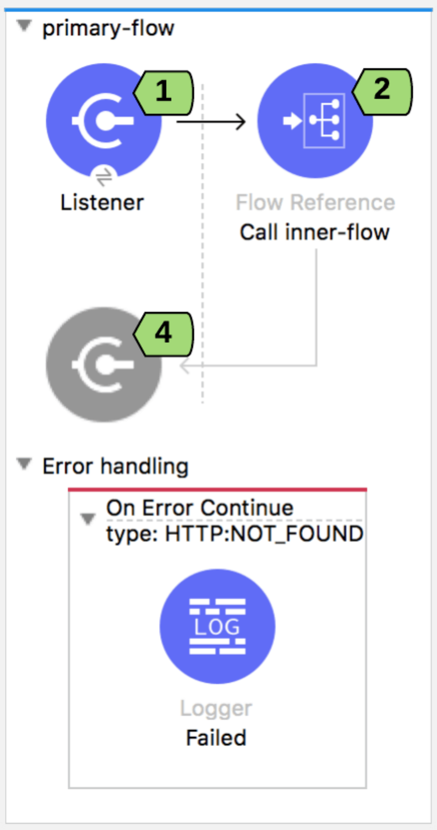
Mule 4 also introduces the error handler component, that can any number of internal handlers that route an error to the first one matching it.

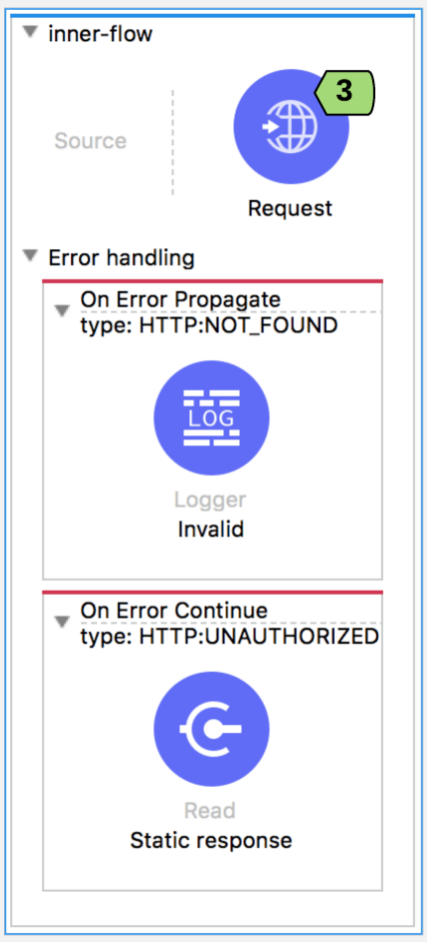
Such handlers are on-error-continue and on-error-propagate, which both allow conducting matching through an error type (or group of) or through an expression (*for advanced use cases*). These are similar but simpler to the choice, catch, and rollback exceptions strategies of Mule 3. If an error is raised within a flow, its error handler will be executed and the error will be routed to the matching handler. At this point, the error is available for inspection, so the handlers could execute and act accordingly:

- An on-error-continue will execute and use the result of the execution, as the result of its owner—as if the owner had actually completed the execution successfully. Any transactions at this point would be committed as well.
- An on-error-propagate will rollback any transactions, execute, and use that result to rethrow the existing error—meaning its owner will be considered as "failing."

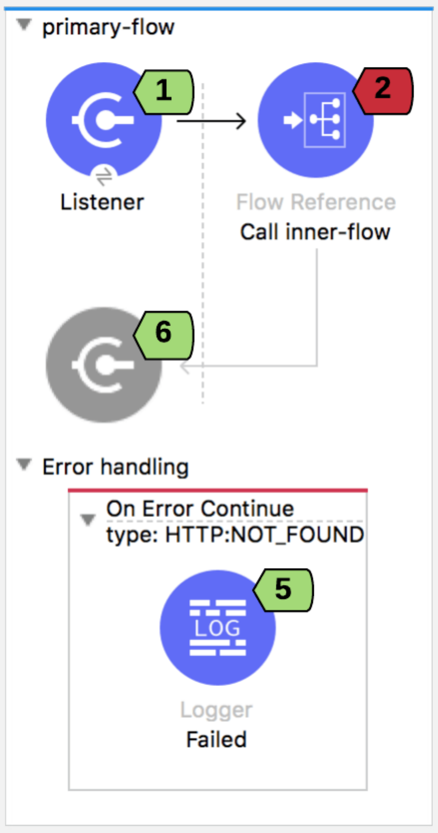
Consider a few examples to see how this works. First, consider the following application where an HTTP listener triggers a flow reference to another flow that performs an HTTP request.

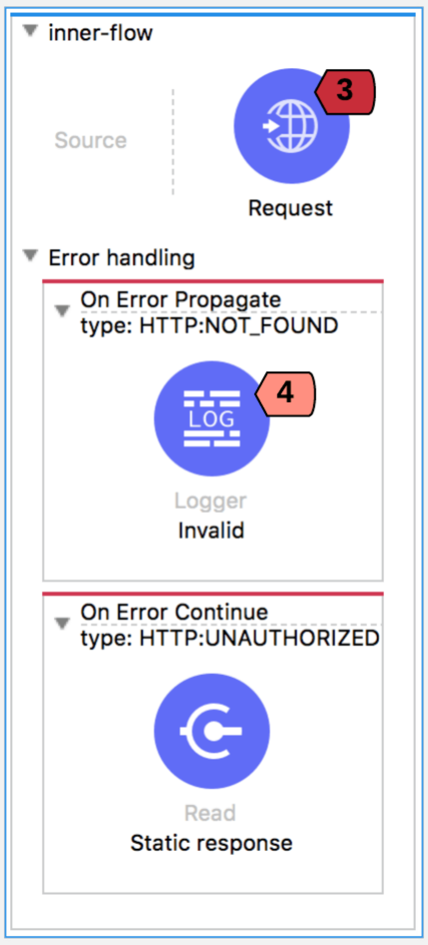
If everything goes right when a message is received, (1) the reference is triggered (2) the request performed (3), then a successful response is expected (4).



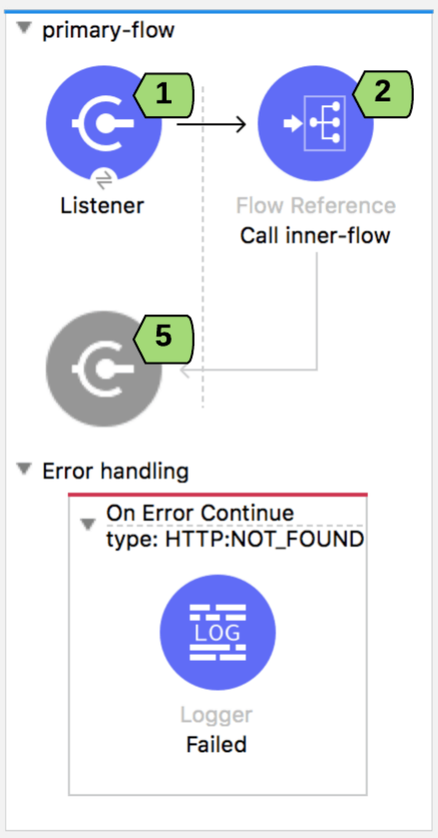


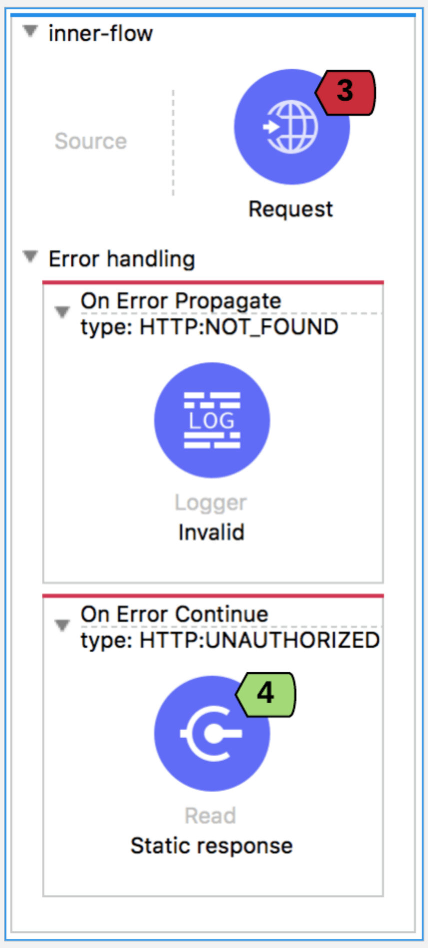
If the HTTP request fails with a not found error (3), because of the error handler setup of inner-flow, then the error will be propagated (4), and the flow reference will fail as well (2). However, since primary-flow is handling that with an on-error-continue, this will execute (5) and a successful response will be returned (6).



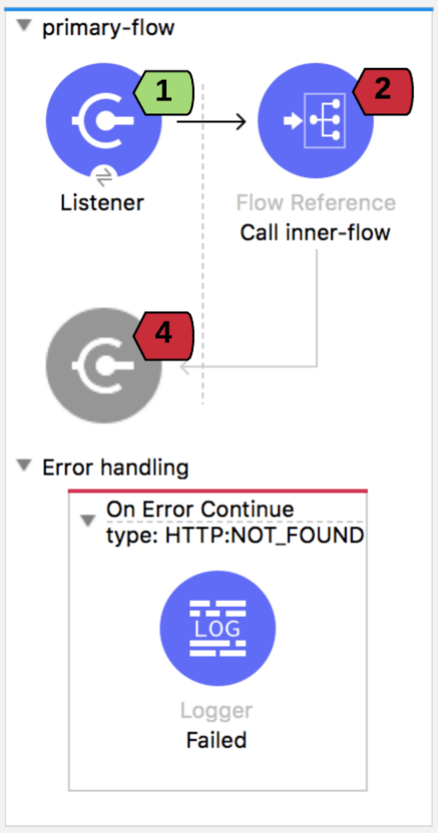


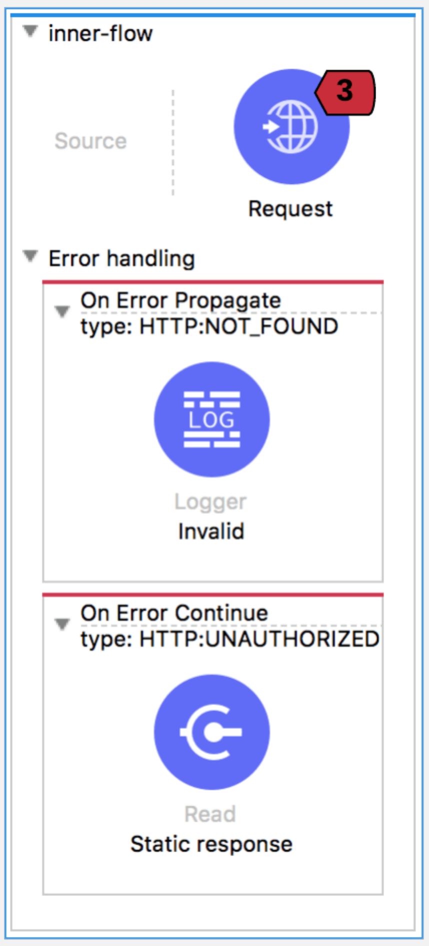
Should the request fail with an unauthorized error instead (3), then the inner-flow will handle it with an on-error-continue by retrieving static content from a file (4). Then, the flow reference will be successful as well (2) and a successful response will be returned (5).



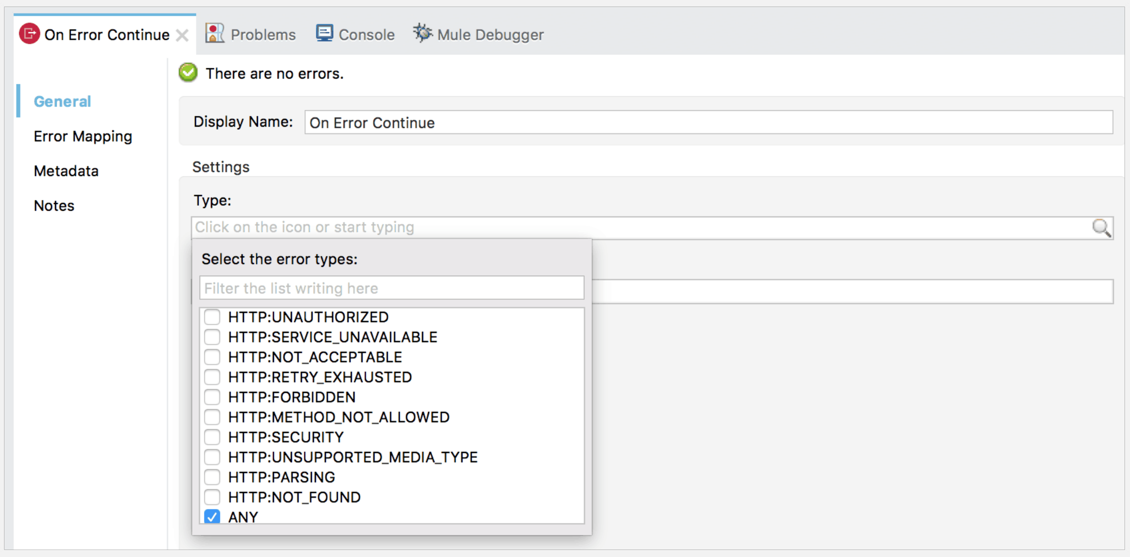


But what if another error occurred in the HTTP request? Although there are only handlers for “not found” and “unauthorized” errors in the flow, errors are still propagated by default. This means that if no handler matches the error that is raised, then it will be rethrown. For example, if the request fails with a “method not allowed” error (3), then it will be propagated causing the flow reference to fail (2), and that propagation will result in a “failure” response (4).





The scenario above could be avoided by making the last handler match `ANY`, instead of just `HTTP:UNAUTHORIZED`. Notice how, below, all the possible errors of an HTTP request are suggested:



There are no errors.

Display Name: On Error Continue

Settings

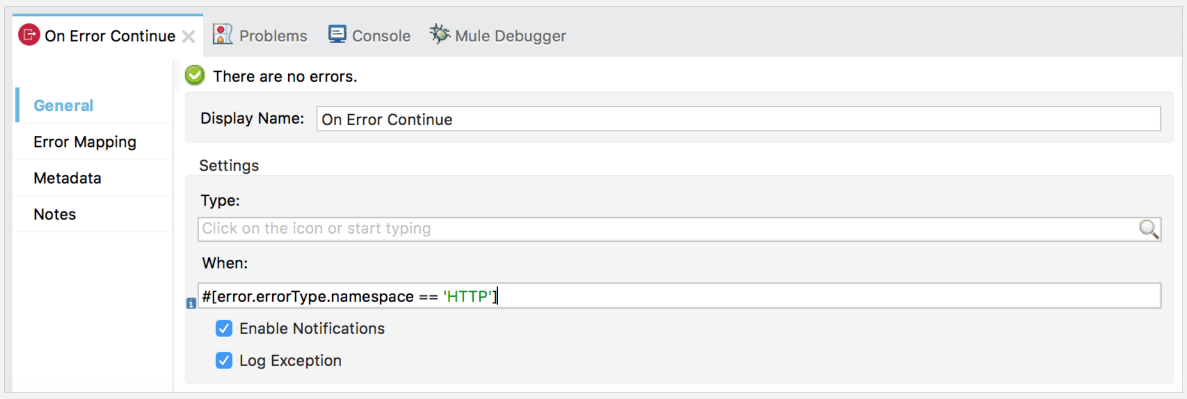
Type: Click on the icon or start typing

Select the error types:

Filter the list writing here

- [ ] HTTP:UNAUTHORIZED
- [ ] HTTP:SERVICE_UNAVAILABLE
- [ ] HTTP:NOT_ACCEPTABLE
- [ ] HTTP:RETRY_EXHAUSTED
- [ ] HTTP:FORBIDDEN
- [ ] HTTP:METHOD_NOT_ALLOWED
- [ ] HTTP:SECURITY
- [ ] HTTP:UNSUPPORTED_MEDIA_TYPE
- [ ] HTTP:PARSING
- [ ] HTTP:NOT_FOUND
- [x] ANY

Errors can also be matched through expressions. For example, since the Mule error is available during error handling, it can be used to match all errors whose namespace is HTTP:



There are no errors.

Display Name: On Error Continue

Settings

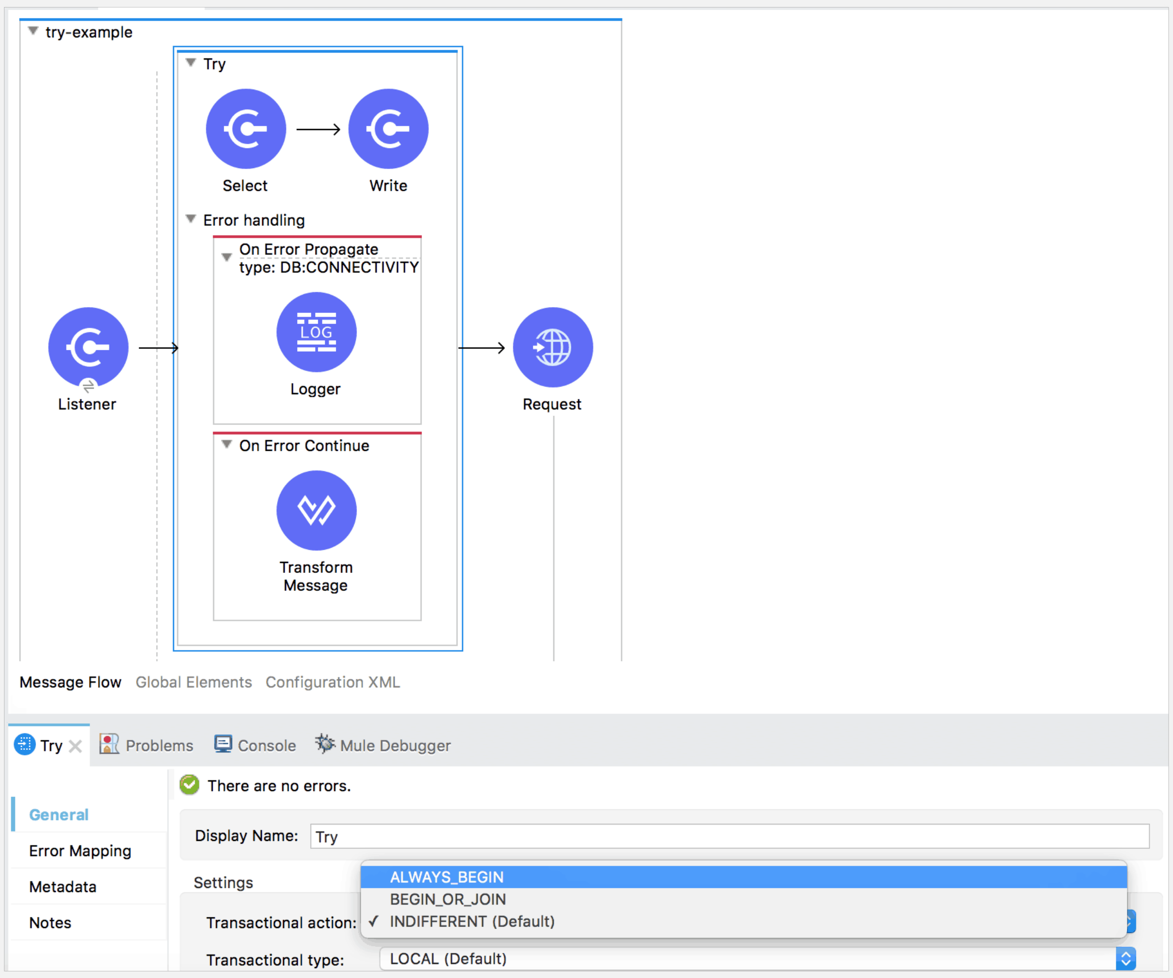
Type: Click on the icon or start typing

When: #[error.errorType.namespace == 'HTTP']

- [x] Enable Notifications
- [x] Log Exception

Try scope

Mule 4 provides more fine-grained error handling with the introduction of the "try" scope. Try scopes can be used within a flow to handle errors of inner components. The scope also supports transactions—replacing the old transactional scope.



There are no errors.

Display Name: Try

Transactional action: ✓ INDIFFERENT (Default)

Transactional type: LOCAL (Default)

The error handler behaves as explained earlier. In the example above, any database connection errors will be propagated, causing the try to fail and the flow's error handler to execute. In this case, any other errors will be handled and the try scope will be considered successful which, in turn, means that the following processor, an HTTP request, will continue executing.

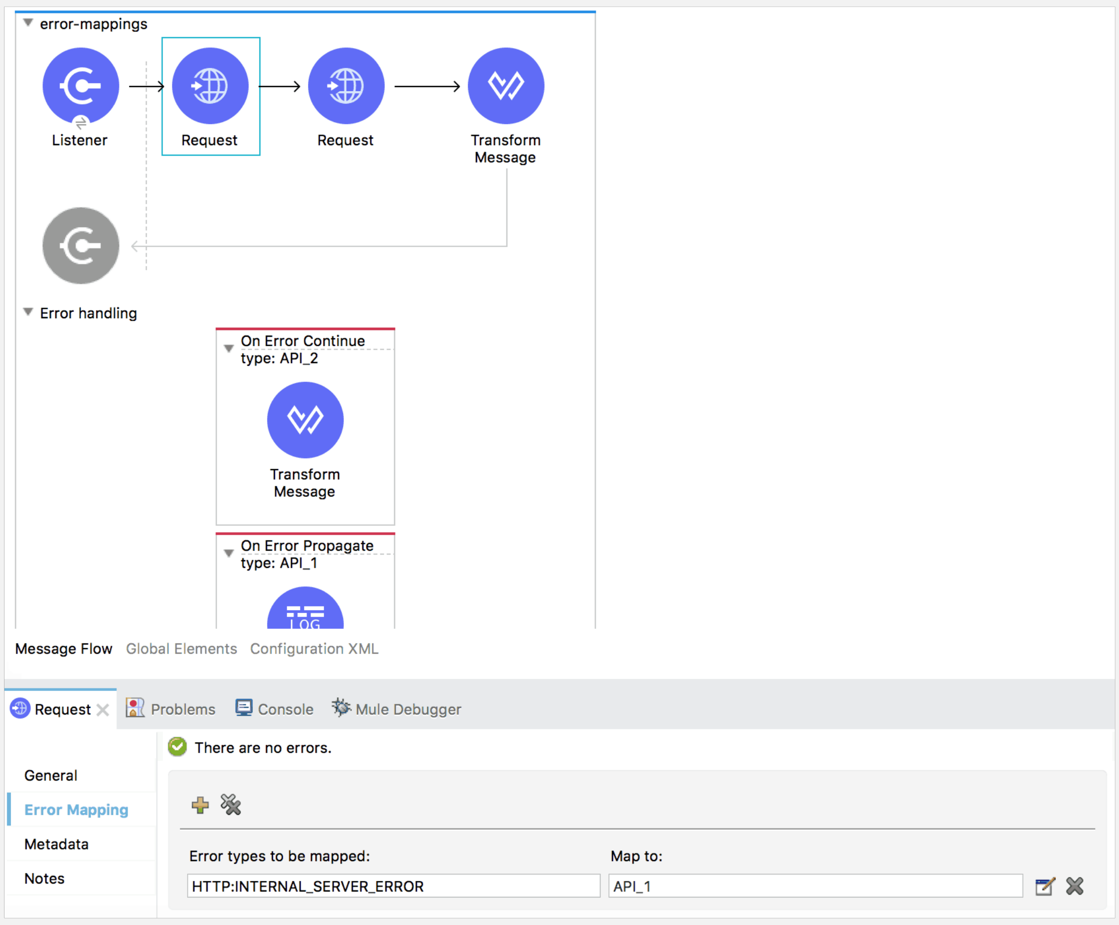
Error mappings

Error handling can also be mapped to custom errors. Error mappings are most useful when an app contains several equal components and there is a need to distinguish between each specific error.

By adding error mappings to each components, all specified types of errors will be mapped to another of the user's choosing. In doing so, the error handler can then digest accordingly. Consider if a flow is aggregating results from 2 APIs using an HTTP request component for each. The user may want to distinguish between the errors of API 1 and API 2, since by default their errors will be the same.

By mapping errors from the first request to a custom API_1 error and errors in the second request to API_2, users can route those errors to different handlers.

The example below shows the mapping of the `HTTP:INTERNAL_SERVER_ERROR`, where different handling policies can be applied in case an API goes down. It propagates the error in the first API and handles it in the second API.



There are no errors.

Error types to be mapped:	Map to:
HTTP:INTERNAL_SERVER_ERROR	API_1

Triggers and connectors

Anypoint Connectors now include new “triggers” for Mule flows. Rather than configuring logic to perform scheduled queries against target systems, the connector contains built-in logic to listen for data changes, such as when a new database row is created, a Salesforce lead is changed, or a file is created. Also, the File and FTP connectors have also been enhanced—providing users with the ability to append files, create directories, and perform other operations.

All connectors now follow the same model of operations and connector configurations because of the new Mule SDK. The File, FTP, JMS, and Email connectors have also been enhanced.

The features behind the File connector include:

- The ability to read files or fully list directories’ contents on demand; unlike the old transport, which only provided a polling inbound endpoint.
- Top level support for common File System operations such as copying, moving, renaming, deleting, creating directories, and more.
- Support for locking files on the file system level.
- Advanced file matching functionality.

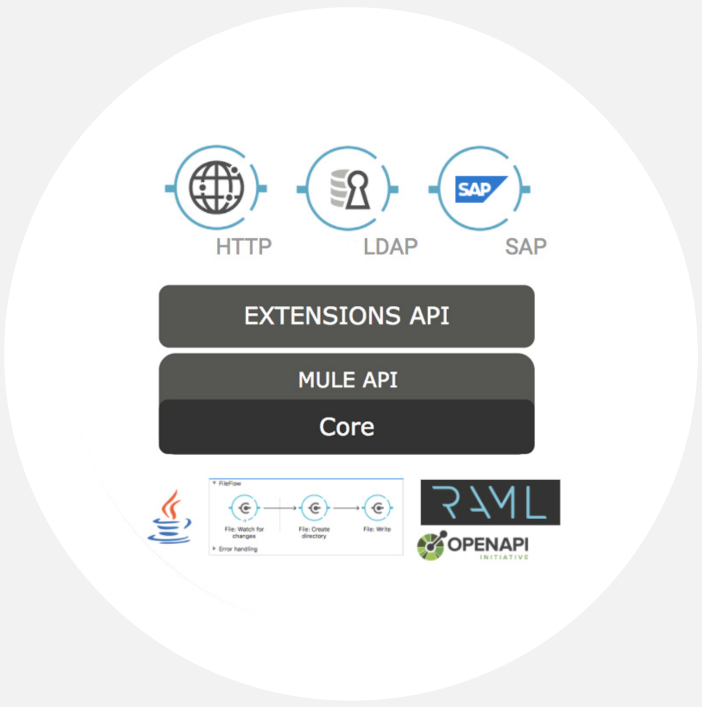
This connector was designed to be fully consistent with the FTP Connector. The same set of operations are available on both connectors in Anypoint Exchange and they behave *almost* the same.

Both the File and FTP connectors look exactly the same. Some of the main features of the FTP connector are:

- Read files or fully list directories contents on demand.
- Copy, move, rename, delete, and create directories quickly and consistently.
- Support for locking files.
- Advanced file matching functionality.

Mule SDK

Mule SDK is the successor to the Anypoint Connector DevKit that enables developers to easily extend Mule and create new Anypoint Connectors which can be shared in Anypoint Exchange. Mule SDK provides a simple annotation-based programming model for extending Mule and also offers enhanced forward compatibility with Mule versions, transactions support, and the ability to create routers.



Before, Anypoint Connector DevKit would take a Connector project and generate code around it, creating a wrapper that would then bridge the code to the Mule runtime internals. Now, the SDK relies on Mule APIs (*more precisely, the Extensions API*), and does not need to generate code between the connector and the runtime internals. This makes it easier to build connectors that are both backwards and forward compatible across version of the runtime. This also allows the runtime to inject cross-cutting functionality (such as streaming, media-type handling, reconnection, enrichment, etc.), across connector operations

Any artifact built with Mule SDK will be completely isolated from the runtime. This will allow users to use any dependency or library without worrying about version conflicts with those shipped in the runtime. The SDK automatically enforces consistency and adds cross-cutting functionality – such as repeatable streaming, embeddable DataWeave scripts, automatic static DataSense, and mimeTypes support – across connector operations and into existing modules without the need to rebuild them. For a list of functionalities, read the SDK documentation.

Transactions, error handling, and more

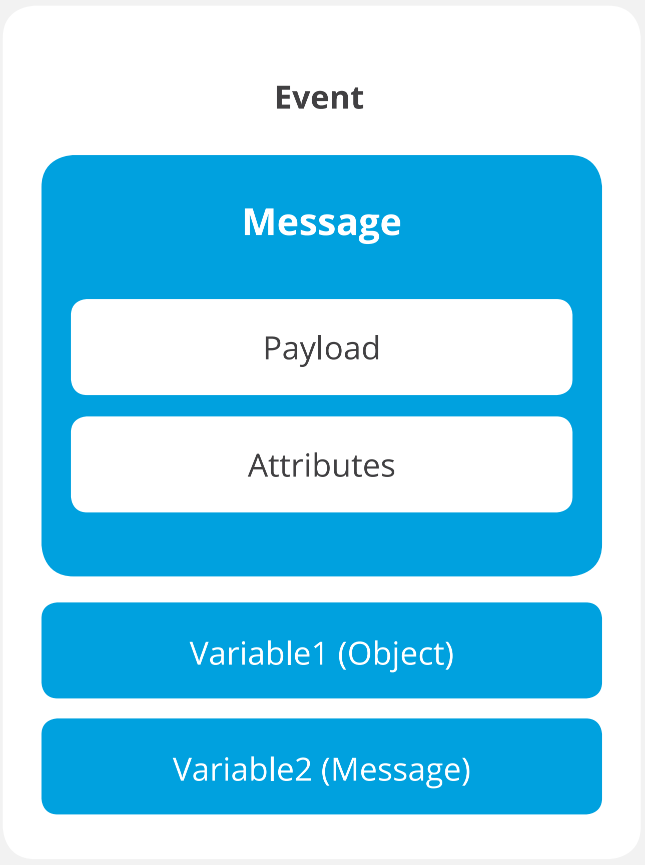
A transaction is a series of actions that work in an atomic way. For example, if a flow need to execute 5 steps, but the 4th fails, the first 3 steps are rollbacked. As a result, the system state does not change; instead, it prevents having an inconsistent state.

Transactions are not permanent actions until the entire sequence of events are complete. Users can now build both operations and message sources capable of participating in single resource or XA transactions. They can also develop non-blocking operations easily leveraging Mule 4's new reactive engine and use expressions on the module's config elements. This makes it super easy to develop modules which enable simultaneous connections to many different credentials.

Mule SDK also leverages the new error handling mechanisms, DataWeave 2.0, and DataSense. Each component in a module lists the full list of possible errors, making it easier for users to handle errors. Users can expose custom pieces of logic in their modules as DataWeave functions and by supporting dynamic DataSense, Mule SDK has a more powerful model to describe the dynamic types for every individual parameter and message sources.

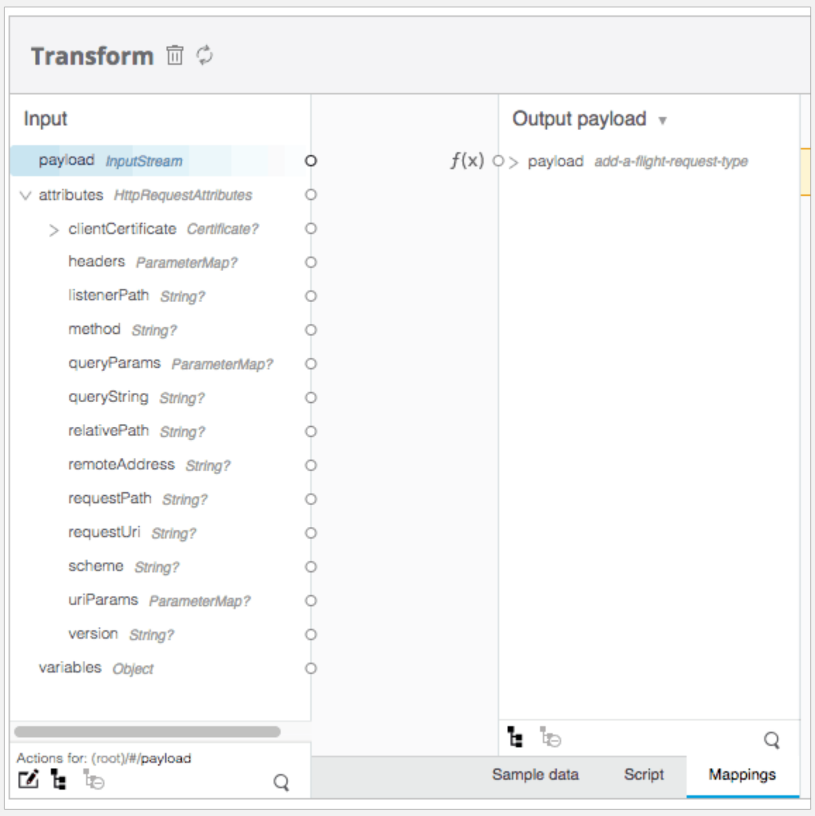
Message model

Mule 4 also introduces a simplified Message model which makes it easier to access content and metadata. Users no longer need to think about transport barriers because there is only one scope of properties. The Message model in Mule 4 and beyond looks like:



A Message in Mule 4 is made up of a Payload and Attributes. The new Attributes object replaces the inbound property scope in the Mule 3 Message model. This allows for easy differentiation of incoming HTTP headers and other metadata about the request provided by the HTTP listener. Also, as it is strongly typed, it is easy to discover and use the connector-populated metadata associated with each request, especially combined with DataSense support in Anypoint Studio.

When a Flow is triggered (e.g. a new HTTP request is received) the connector, in this case, the HTTP Listener, will create a Message that composes both the HTTP body and an instance of HttpAttributes. The HttpAttributes instance will contain both the incoming HTTP headers and all other metadata. Because it is typed, it is easier to use and self-discover.



In Anypoint Studio and Design Center—flow designer, and via DataSense, the attributes populated by the previous connector source or operation can be seen and used. The screenshot above depicts this in the context of the DataWeave Transform component. While this example is HTTP-centric, when a different source connector (e.g. JMS) is used, the attributes object will be a specific instance to the connector and will, in turn, provide access to connector-specific metadata, (e.g JMS correlation ID)

Outbound properties

In Mule 4, outbound properties no longer exist. Instead, the headers or properties (e.g. HTTP headers or JMS properties) are now configured explicitly as part of the connector operation configuration. Users can user variables to store the value of headers and use it later in flows as well.

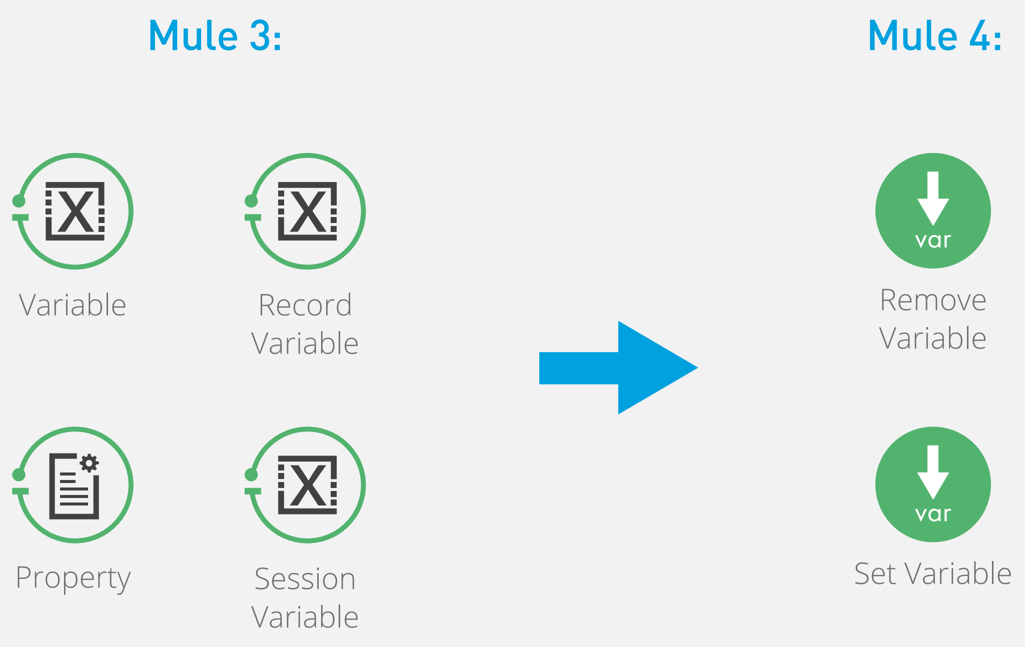
In Mule 3, it was possible to add outbound message properties anywhere in a flow. The next outbound endpoint or connector operation (e.g. an HTTP Requester) would then use the set of outbound properties present at that point in the flow and send them, along with the current payload, via the outbound end-point or connector (e.g. HTTP). While this approach seems very convenient, it resulted in a lot of issues where the behavior of flows and the outbound properties were not deterministic. Subsequently, the refactoring of a flow or the introduction of a new connector operation in an existing flow would significantly change the behavior of a flow. The improvements in the Mule 4 Message model avoid these issues with outbound properties in Mule 3.

To Mule 3 users: Impact on application design

The changes to Message properties will result in more deterministic flows. For Mule 3 users, the changes to Message properties will require a slight recalibration as to how users think about defining their integrations. When migrating existing applications to Mule 4 while using Mule 3 transports via the compatibility module, the Message properties will continue to work as they do in Mule 3. Only when using the new connectors in Mule 4 will you need to work with the new Attributes objects.

Variables

Flow variables remain largely untouched, with the major change being that they have been renamed to “variables” and that session variables have been removed.



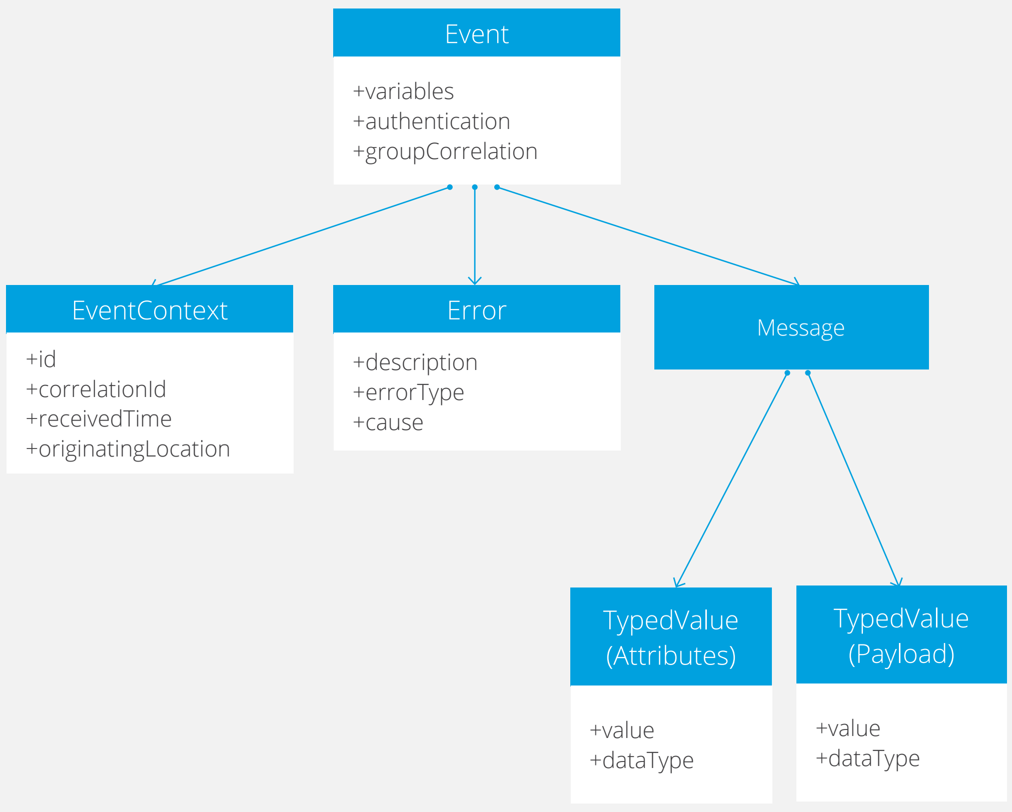
Mule 4 removes session variables for a couple of reasons. Firstly, session variables that were propagated between different flows didn't have anything to do with a stateful client session. Secondly, session variables were often seen as security concern because variables defined in a Flow were automatically being propagated via HTTP headers or JMS properties. Consequently, users were frequently disabling this functionality.

The more security conscious, and explicit, way of propagating values between different flows or systems is via the explicit use of HTTP headers or JMS properties.

Impact on application design

Other than the changes required in migrating to DataWeave expressions, there is nothing specific to take into account regarding variables. If you are migrating an application from Mule 3 that uses session variables, it is quite likely that you didn't ever need session variables, and flow variables will be sufficient. But, if you do need to propagate values between flows, then you should use set outbound and header/property as required in the source flow and then read this value from the attributes object in the destination flow.

The simplified updated model can be seen in the class diagram below:



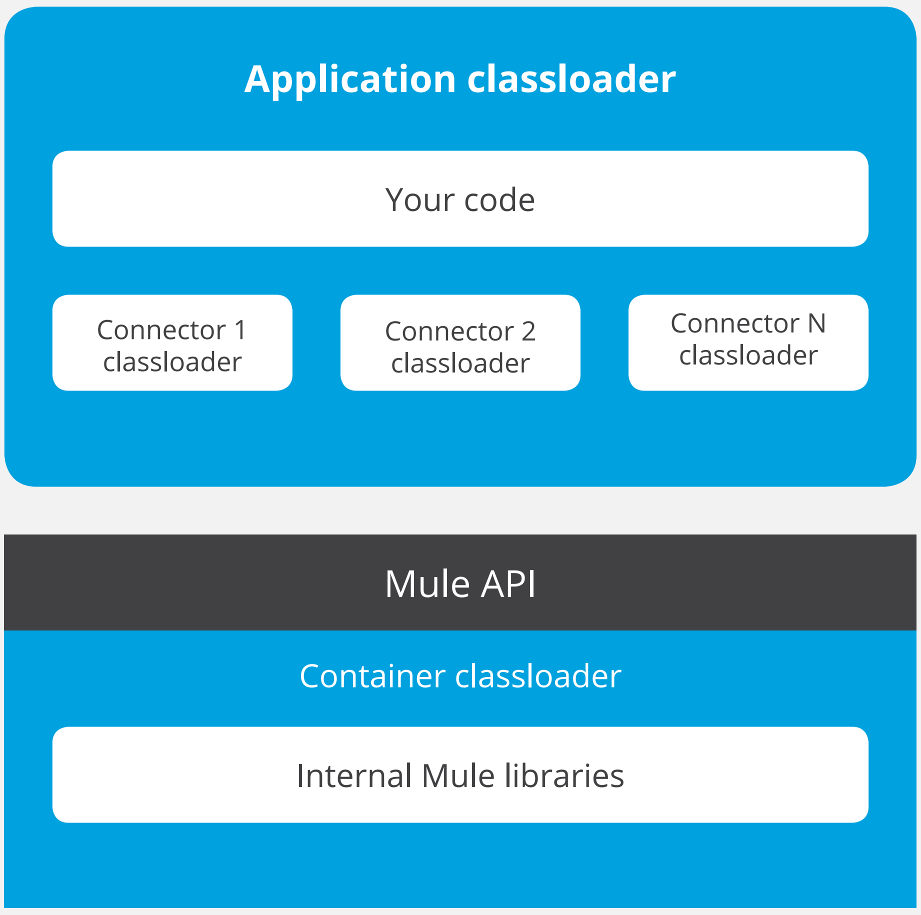
In summary:

- **Message:** The user message includes the payload, and an optional attributes instance. This is what users primarily work with and is what connector operations receive and return.
- **Event (*internal*):** The message, variables, authentication, and group correlation (which contains the sequence number and group size). This is internal to the runtime and updated during flow execution as the message and variables are mutated. The Event also has a reference to the current *EventContext*.
- **EventContext (*internal*):** Represents a single request or transaction with its correlationID and other details. The same *EventContext* instance is used throughout Flow execution.

Upgrade

The primary of objective when release Mule 4 was to make it easier for users to get new enhancements with little to no work on the user. This was difficult in prior versions of Mule because if users were dependent on Mule's internal Java libraries, then they were forced to upgrade as libraries often changed from release to release.

Classloader isolation



With new classloader isolation between Mule applications, the runtime, and Anypoint Connectors, any library changes that occur internally do not affect the application itself. There is now a well-defined Mule API, so users can be sure that they are using supported APIs. Connectors are distributed outside the runtime as well, making it possible to get connector enhancements and fixes without having to upgrade the runtime or vice versa.

Conclusion

Mule 4 is an incredible iteration of the Mule runtime. With all the improvements like the simplified development language of DataWeave and the ability to upgrade connectors and applications make it easy to create, collaborate, and connect.

Learn more

Blog: Mule 4

Read the Mule Runtime 4.1.1 Release Notes

Webinar: Mule 4 GA - Accelerate the speed of development

About MuleSoft

MuleSoft's mission is to help organizations change and innovate faster by making it easy to connect the world's applications, data and devices. With its API-led approach to connectivity, MuleSoft's market-leading Anypoint Platform™ is enabling over 1,000 organizations in more than 60 countries to build application networks. For more information, visit mulesoft.com.