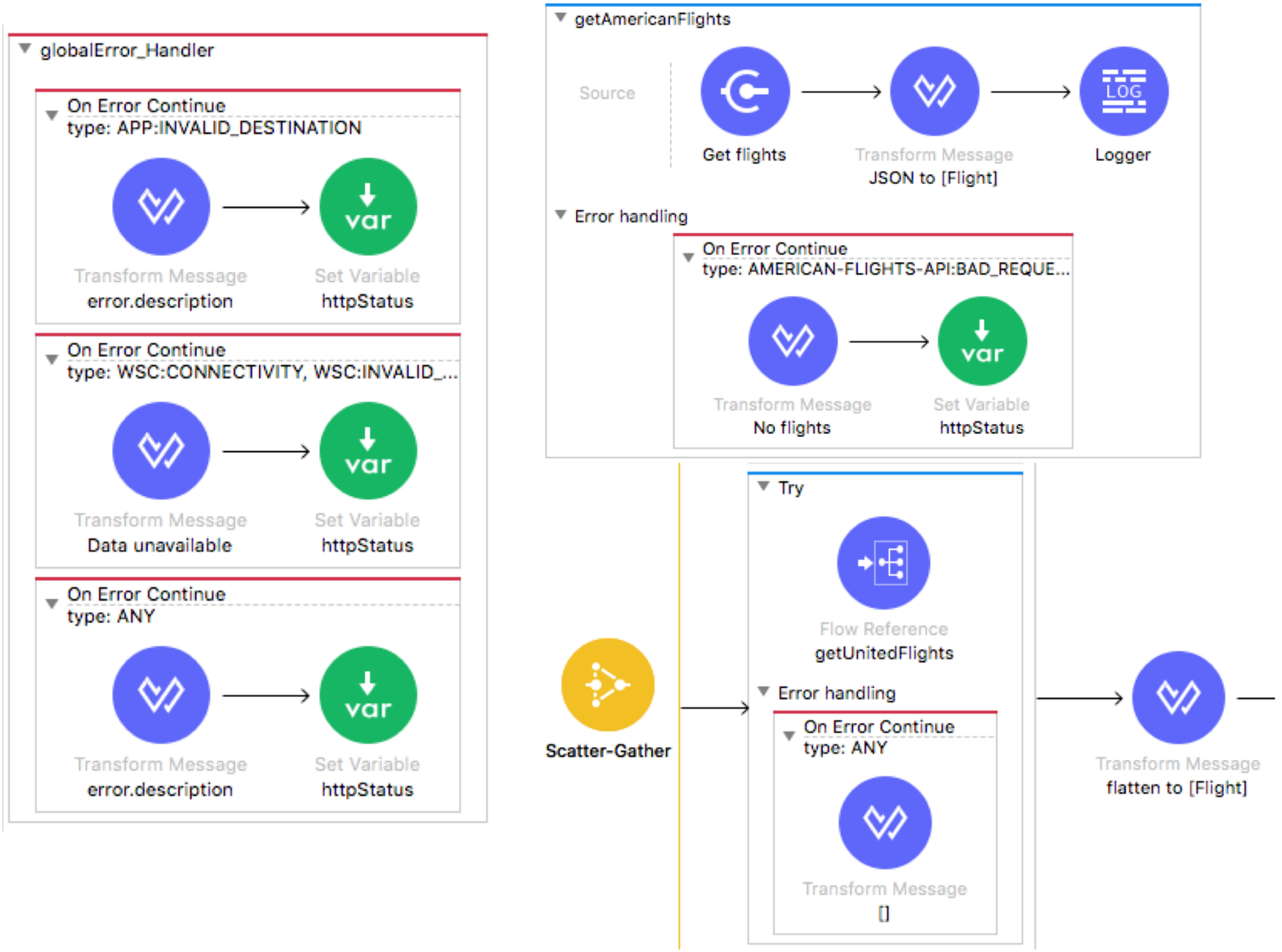
Module 10: Handling Errors

Goal



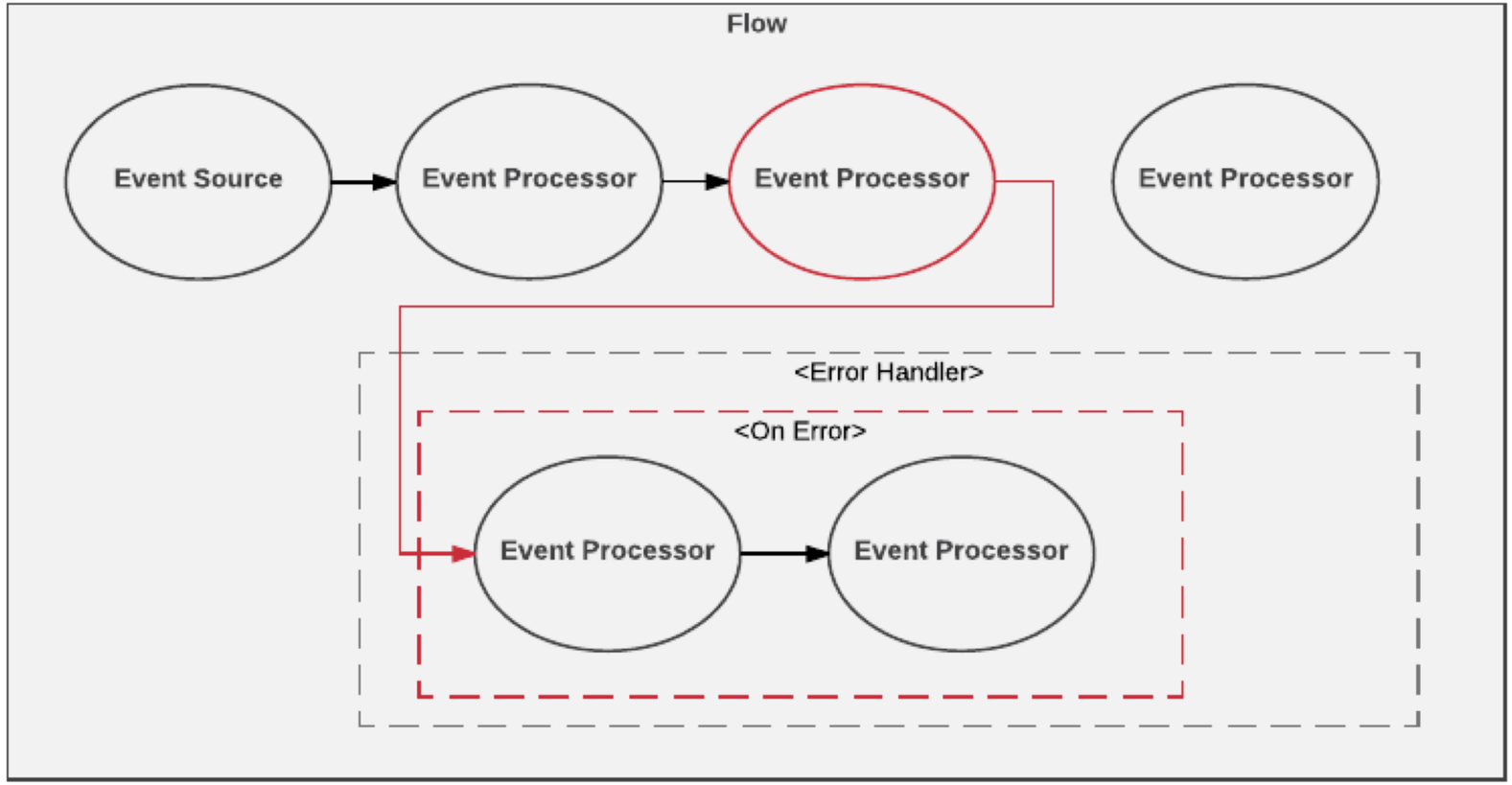
At the end of this module, you should be able to

- Handle messaging errors at the application, flow, and processor level
- Handle different types of errors, including custom errors
- Use different error scopes to either handle an error and continue execution of the parent flow or propagate an error to the parent flow
- Set the success and error response settings for an HTTP Listener
- Set reconnection strategies for system errors

Reviewing the default handling of messaging errors

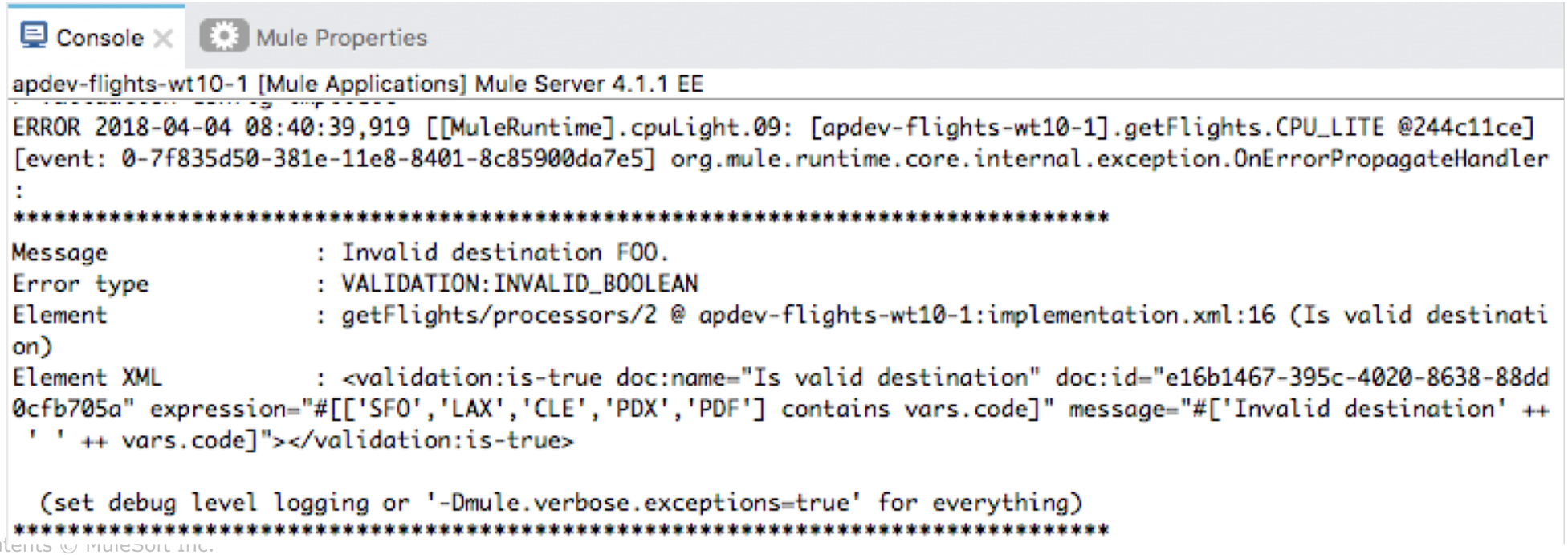
Handling messaging errors

- When an event is being processed through a Mule flow that throws an error
  - Normal flow execution stops
  - The event is passed to the first processor in an error handler



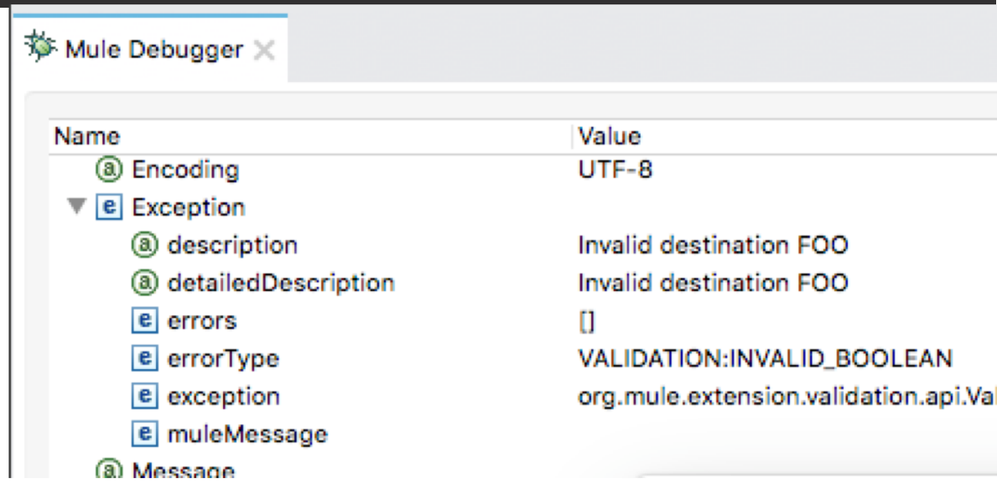
Default error handler behavior

- If there is no error handler defined, a **Mule default error handler** is used
  - Implicitly and globally handles all messaging errors thrown in Mule applications
  - Stops execution of the flow and logs information about the error
  - Cannot be configured



Information about the error

- When an error is thrown, an **error** object is created



- Two of its properties include
  - error.**description** – a string
  - error.**errorType** – an object

- Error types are identified by a namespace and an identifier
  - HTTP:UNAUTHORIZED, HTTP:CONNECTIVITY, VALIDATION:INVALID_BOOLEAN

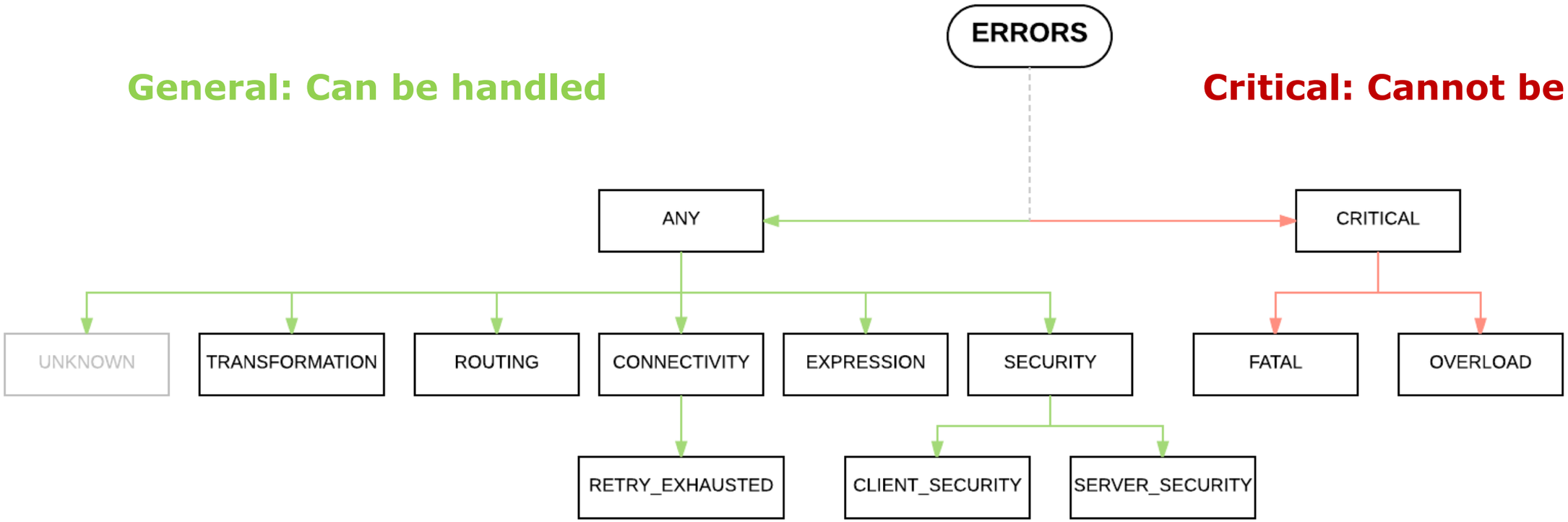
namespace identifier

Error types follow a hierarchy

- Each error type has a parent
  - HTTP:UNAUTHORIZED has MULE:CLIENT_SECURITY as the parent, which has MULE:SECURITY as the parent
  - VALIDATION:INVALID_BOOLEAN has VALIDATION:VALIDATION as the parent, which has MULE:VALIDATION as the parent
- The error type ANY is the most general parent

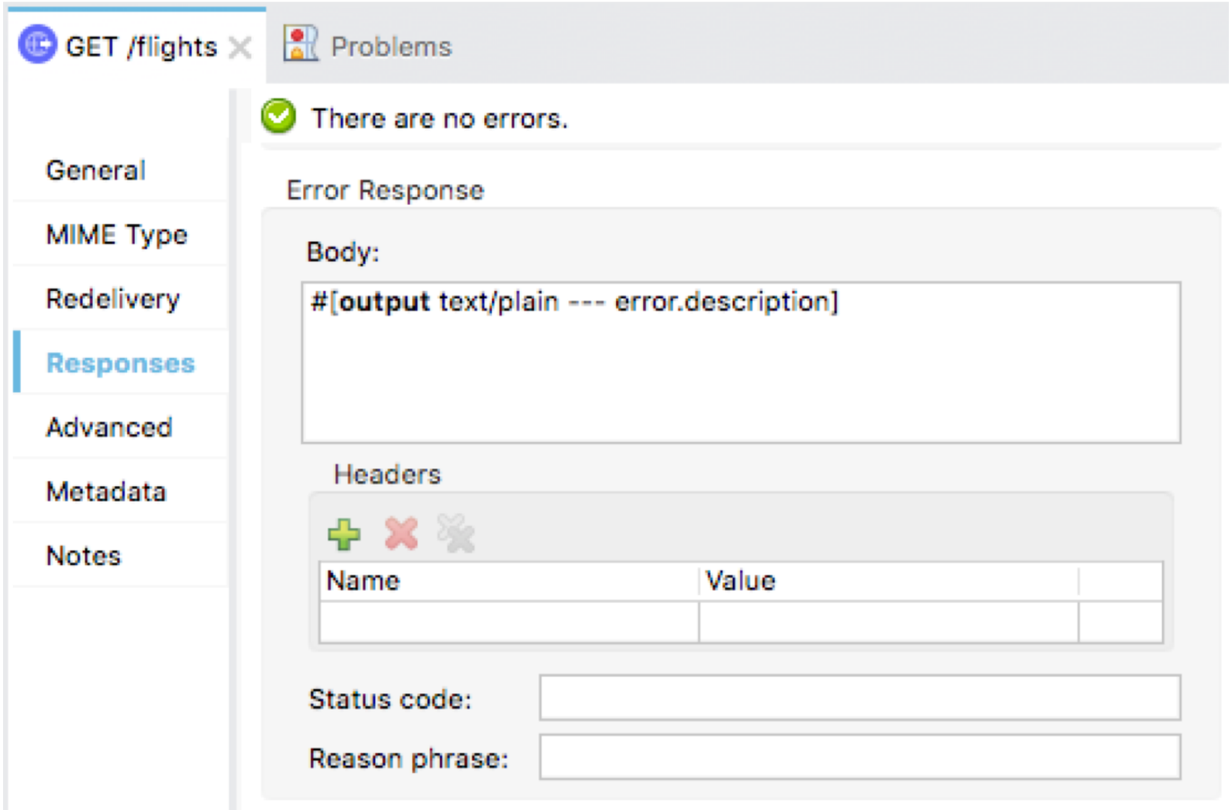

```
{
  "parentErrorType": {
    "parentErrorType": {
      "parentErrorType": {
        "parentErrorType": null,
        "namespace": "MULE",
        "identifier": "ANY"
      },
      "namespace": "MULE",
      "identifier": "VALIDATION"
    },
    "namespace": "VALIDATION",
    "identifier": "VALIDATION"
  },
  "namespace": "VALIDATION",
  "identifier": "INVALID_BOOLEAN"
}
```

Error type hierarchy reference



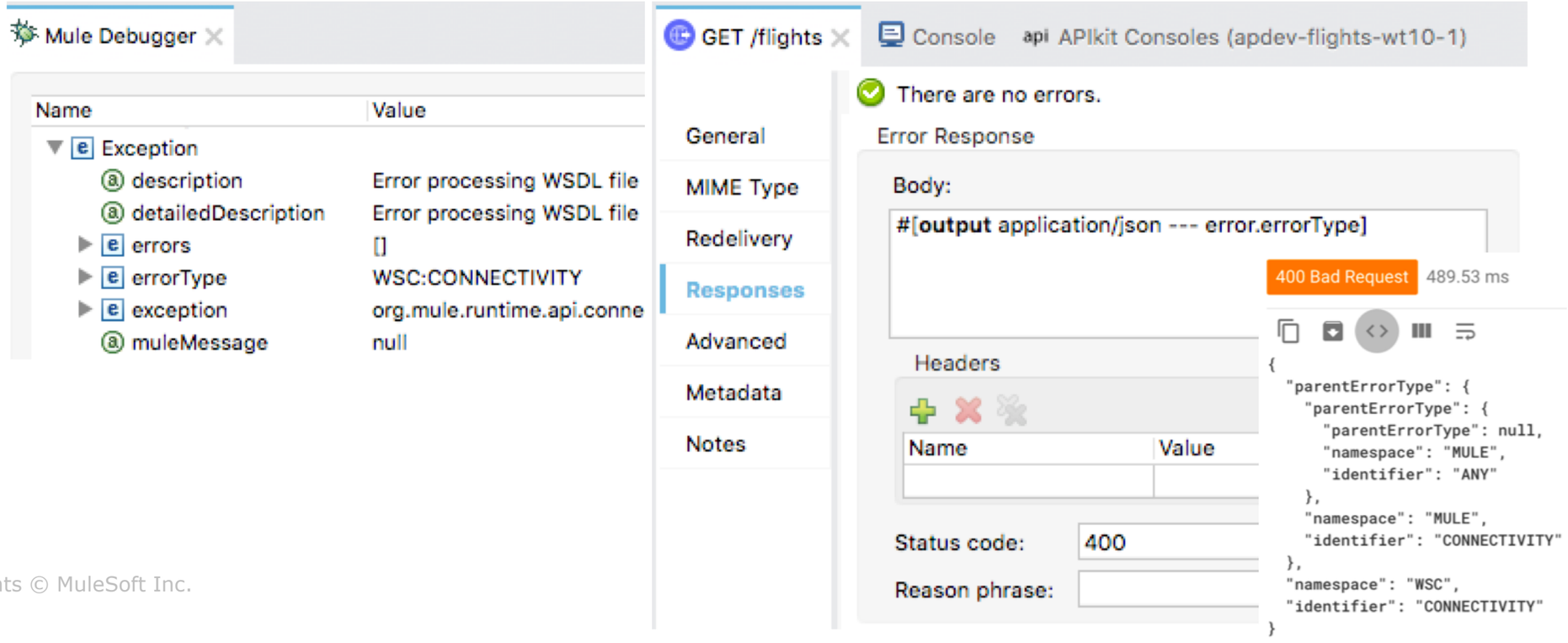
Information returned from HTTP Listeners

- By default, for a **success response**
  - The payload
  - A status code of 200
- By default, for an **error response**
  - The error description
  - A status code of 500
- You can override these values for an HTTP Listener



Walkthrough 10-1: Explore default error handling

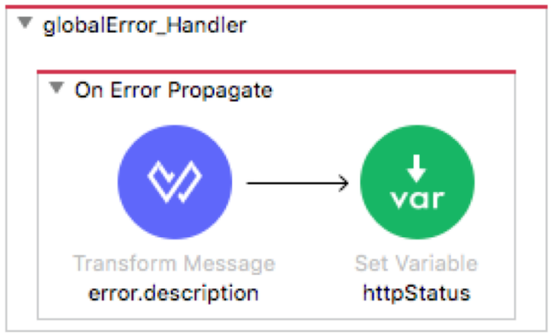
- Explore information about different types of errors in the Mule Debugger and the console
- Review the default error handling behavior
- Review and modify the error response settings for an HTTP Listener

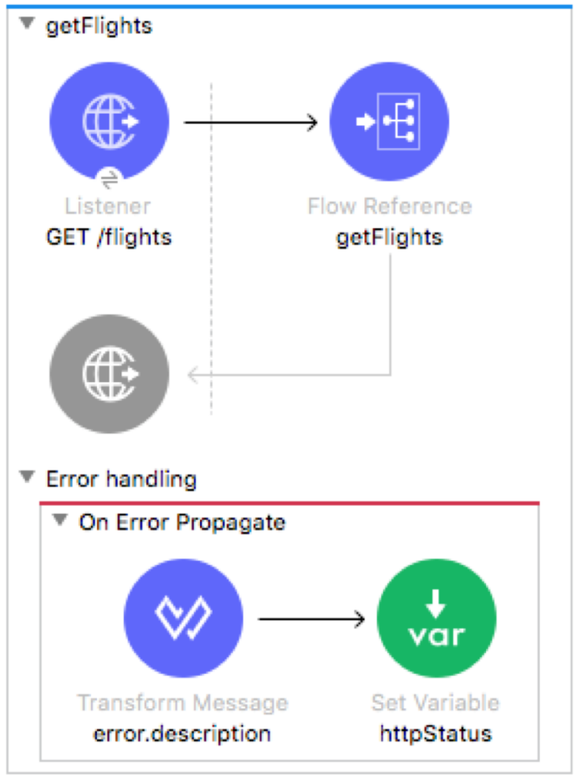


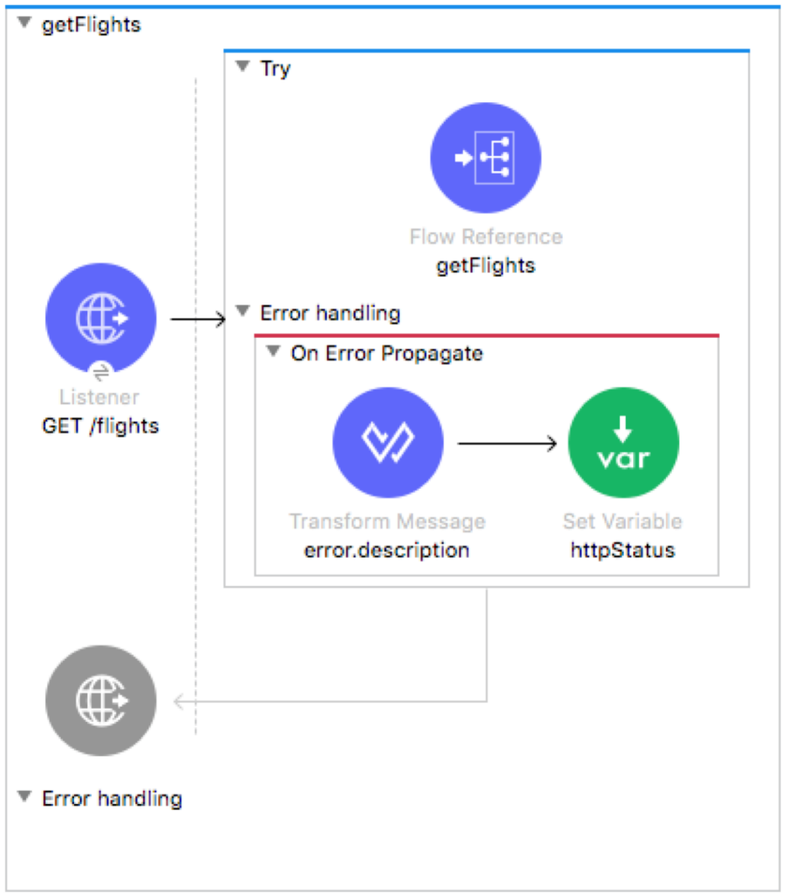
Creating error handlers

Creating error handlers

- Error handlers can be added to
  - An application (outside of any flows)
  - A flow
  - A selection of one or more event processors

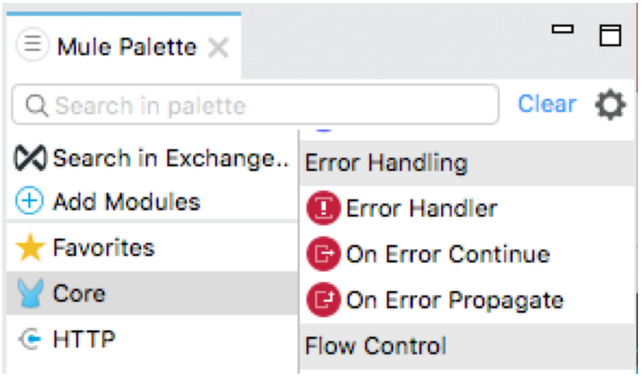




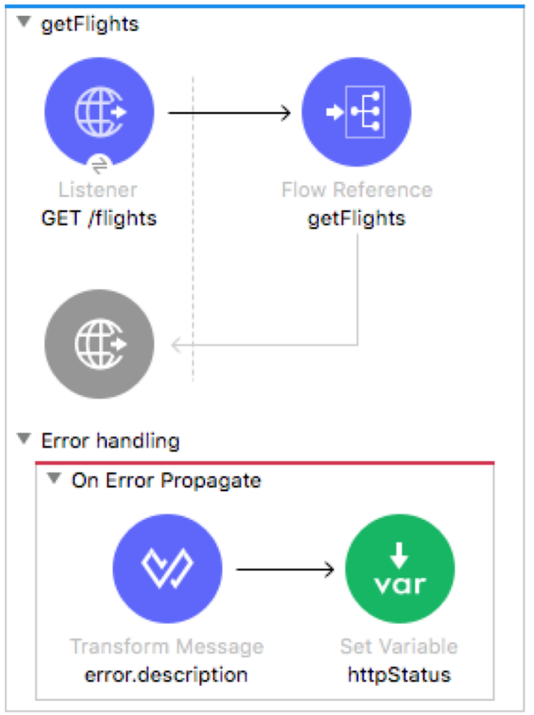


Adding error handler scopes

- Each error handler can contain one or more error handler scopes
  - **On Error Continue**
  - **On Error Propagate**



- Each error scope can contain any number of event processors

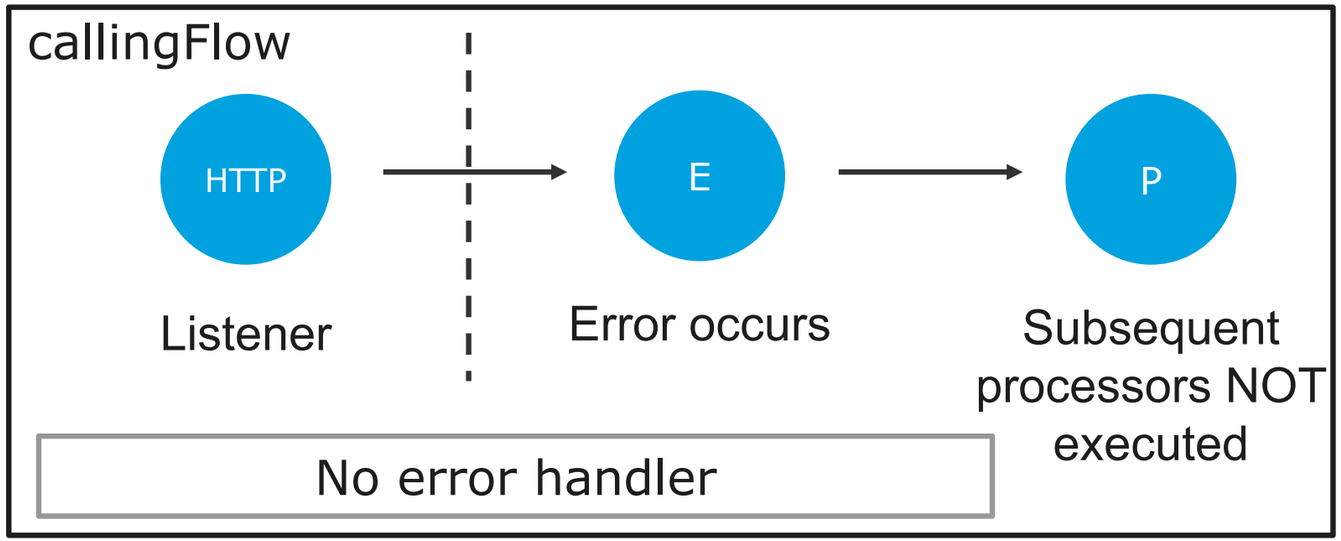


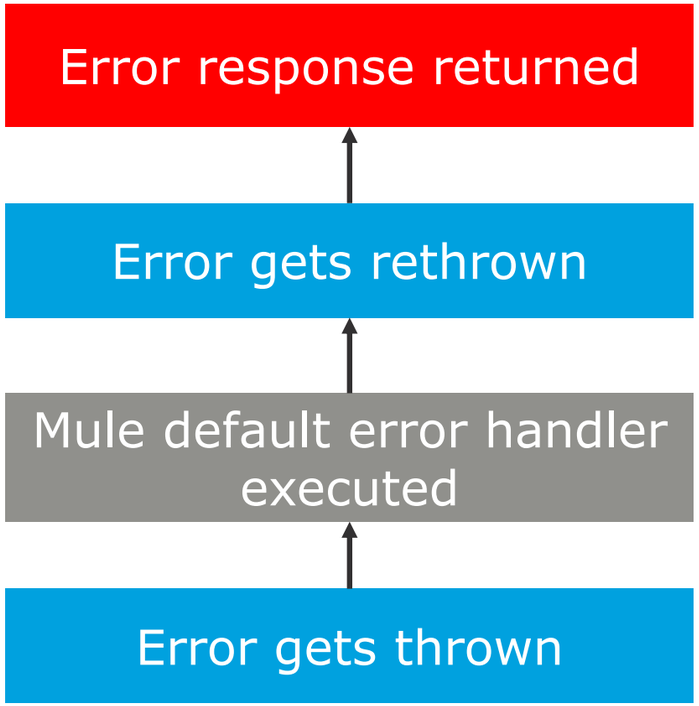
Two types of error handling scopes

- **On Error Propagate**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the flow that threw the error is not executed
    - *The error is rethrown up to the next level and handled there*
  - An HTTP Listener returns an ***error*** response

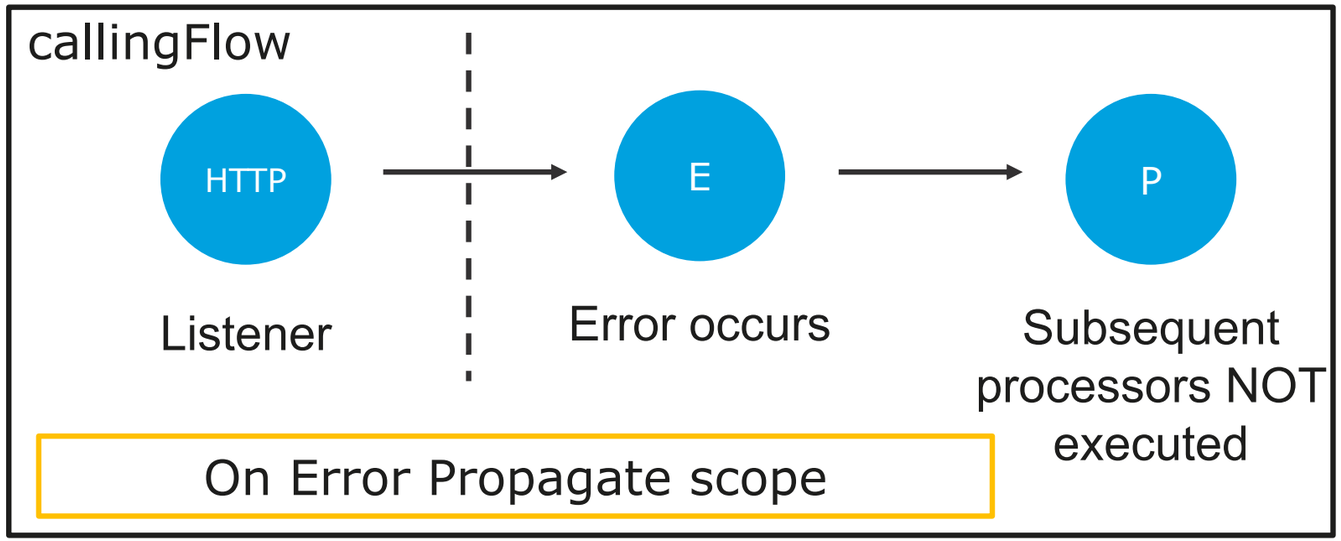
- **On Error Continue**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the flow that threw the error is not executed
    - *The event is passed up to the next level as if the flow execution had completed successfully*
  - An HTTP Listener returns a ***successful*** response

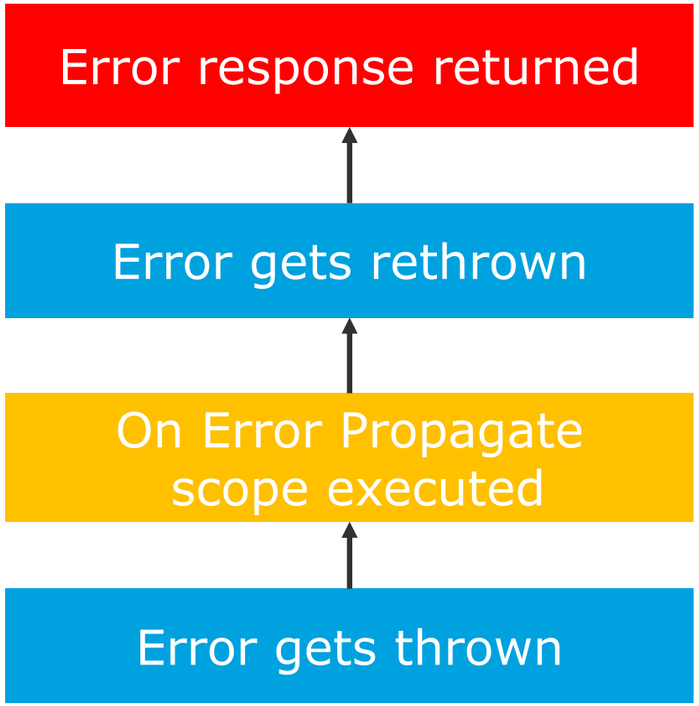
Error handling scenario 1



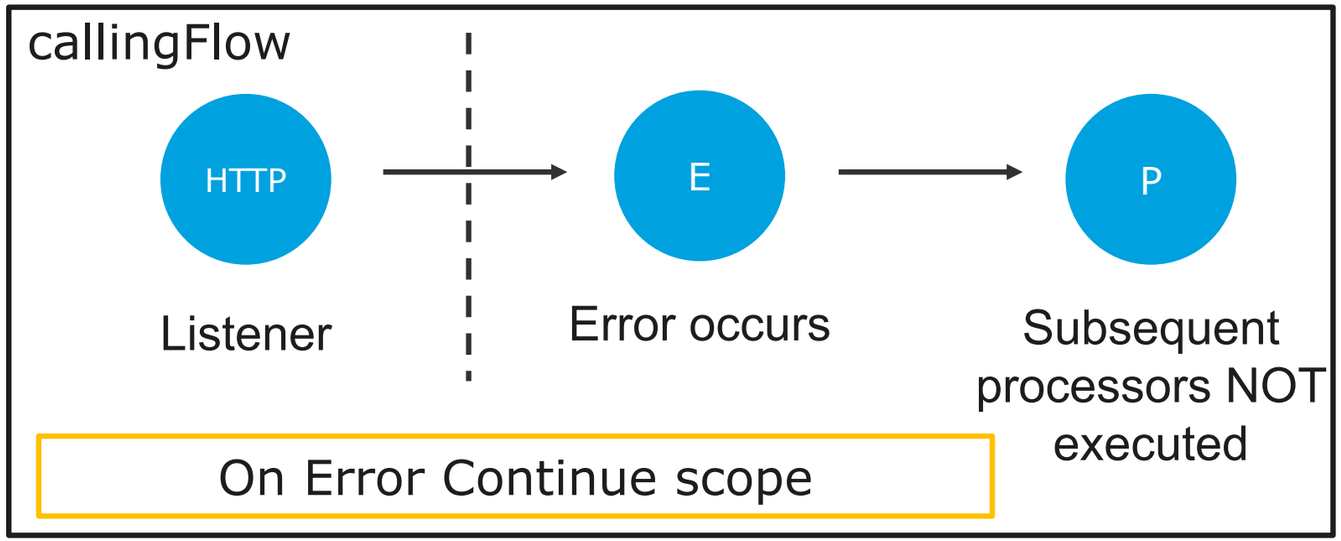


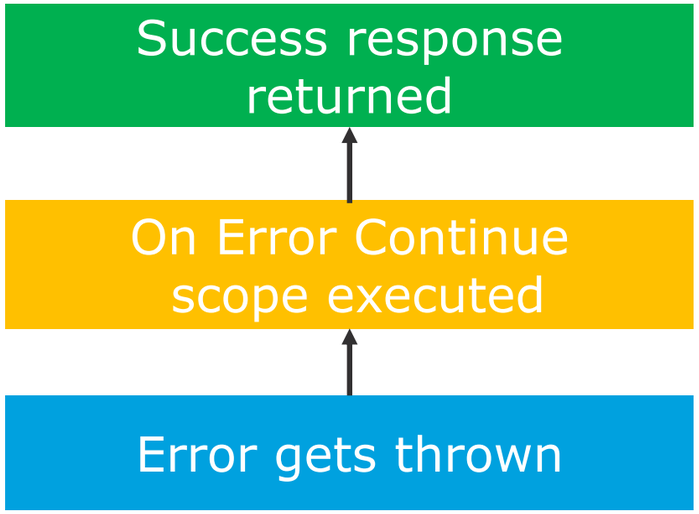
Error handling scenario 2



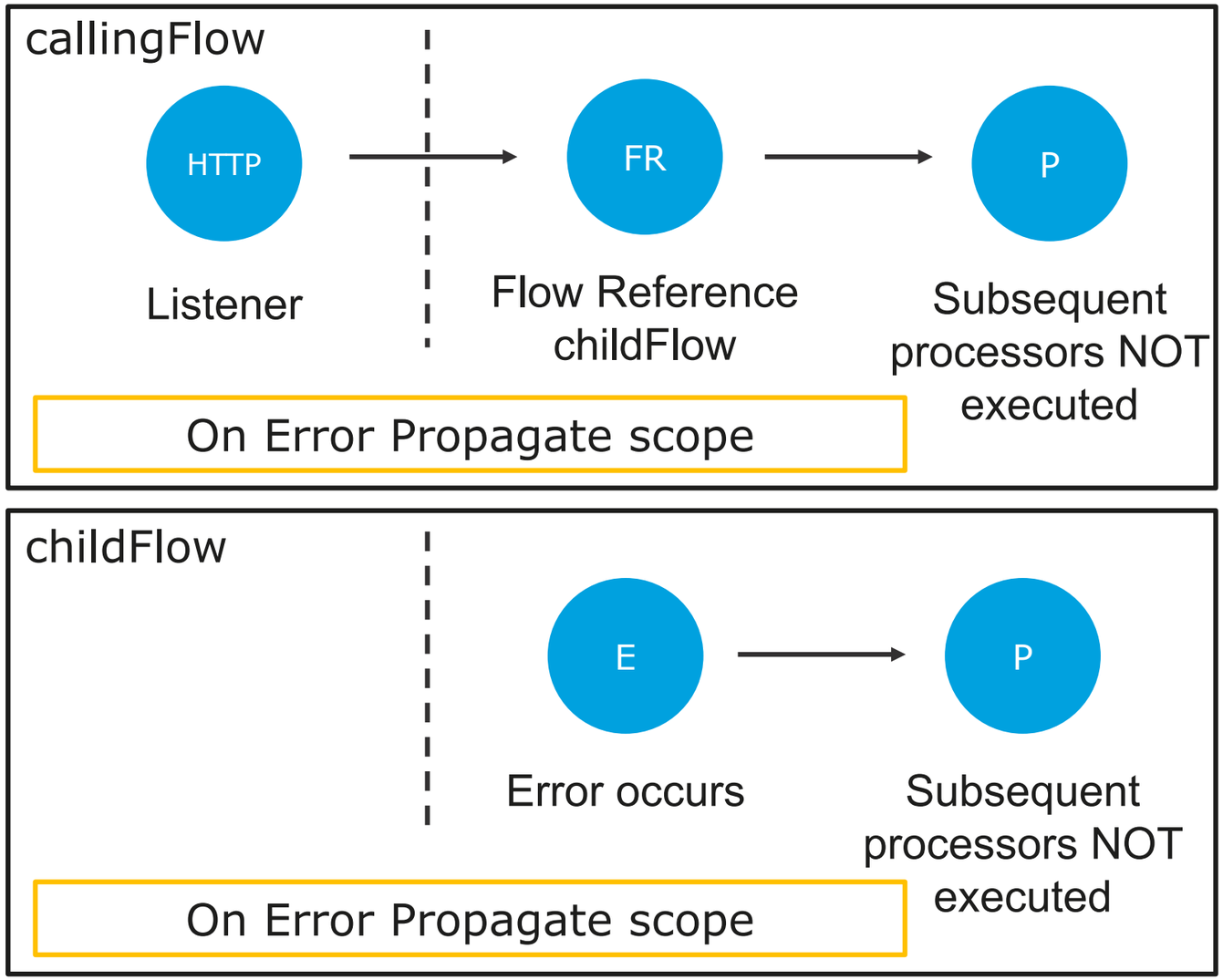


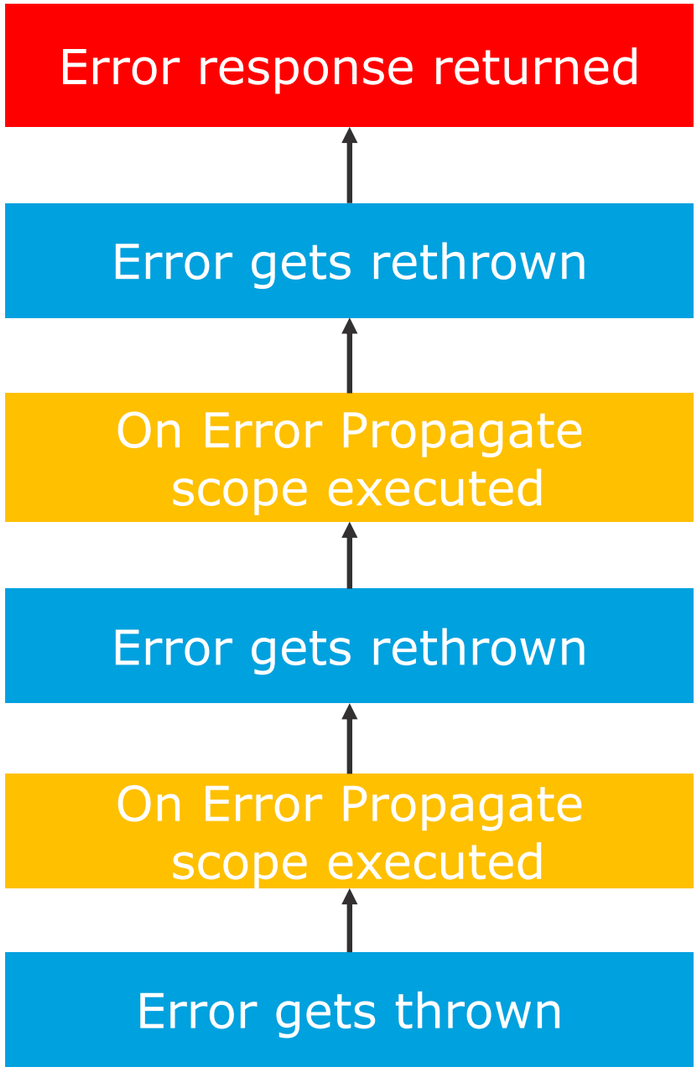
Error handling scenario 3



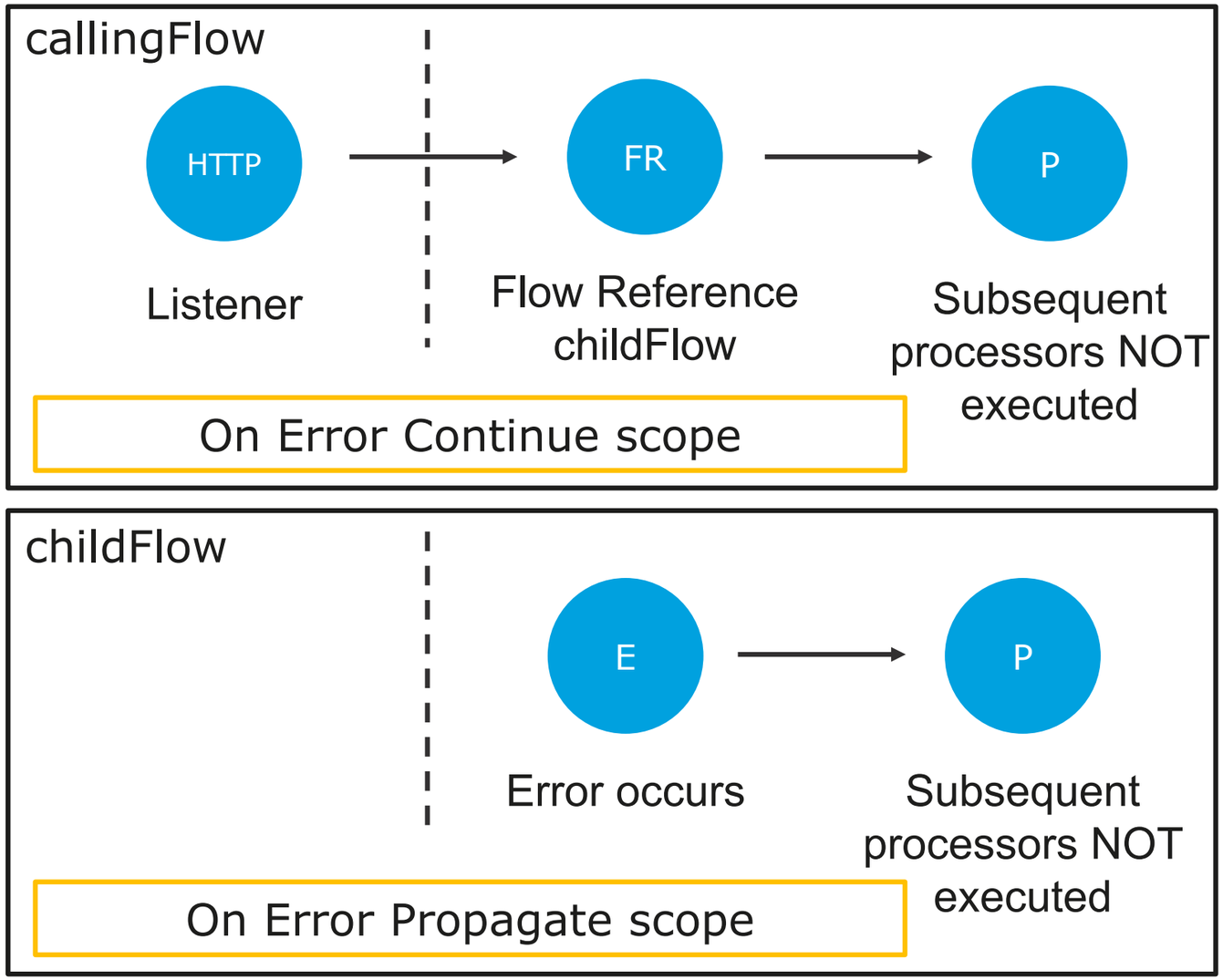


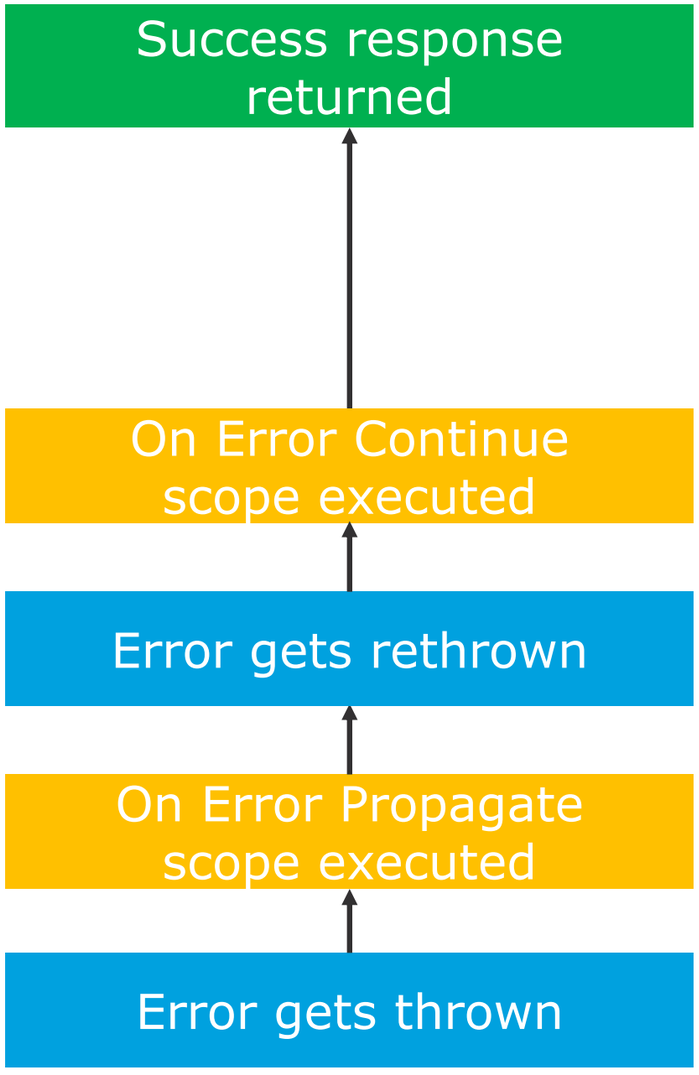
Error handling scenario 4



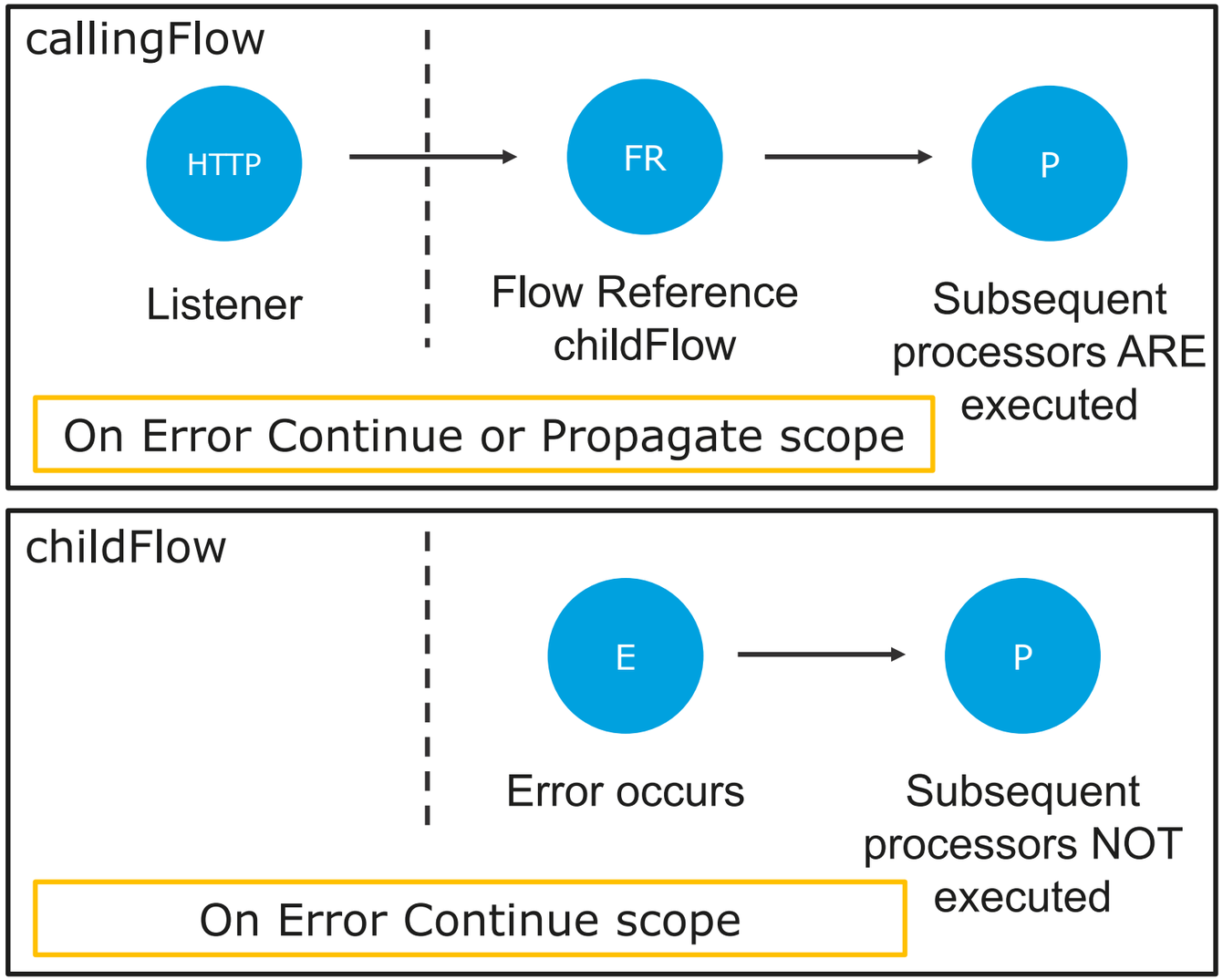


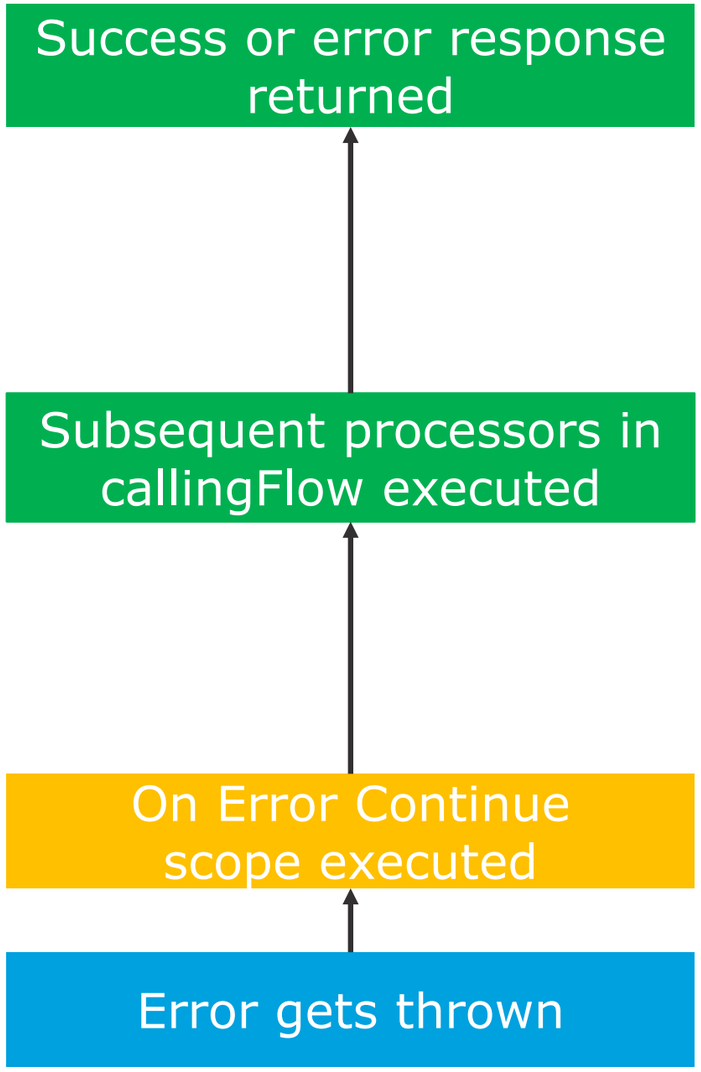
Error handling scenario 5





Error handling scenario 6

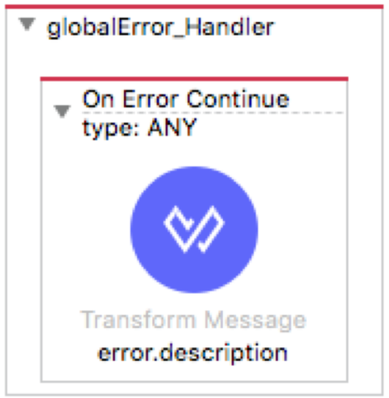




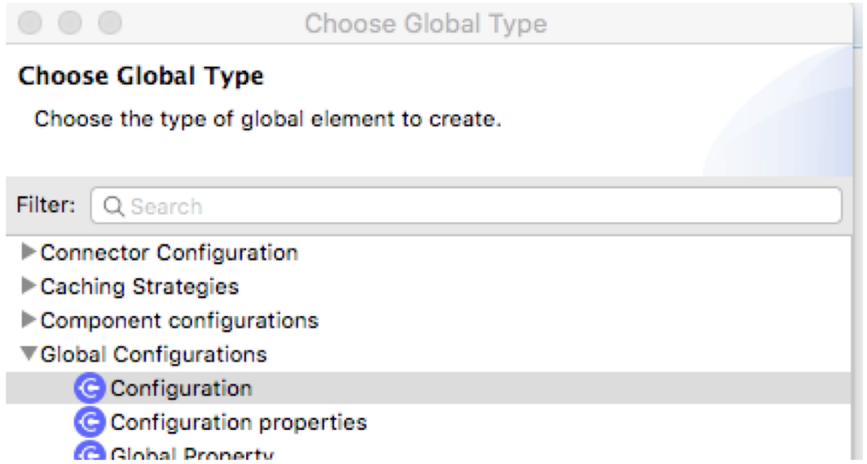
Handling errors at the application level

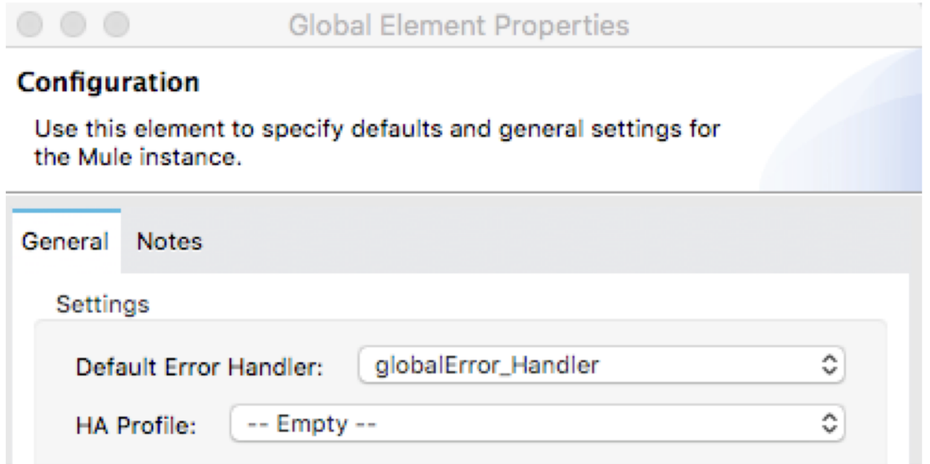
Defining a default error handler for an application

- Add an error handler outside a flow
  - Typically, put it in the global configuration file



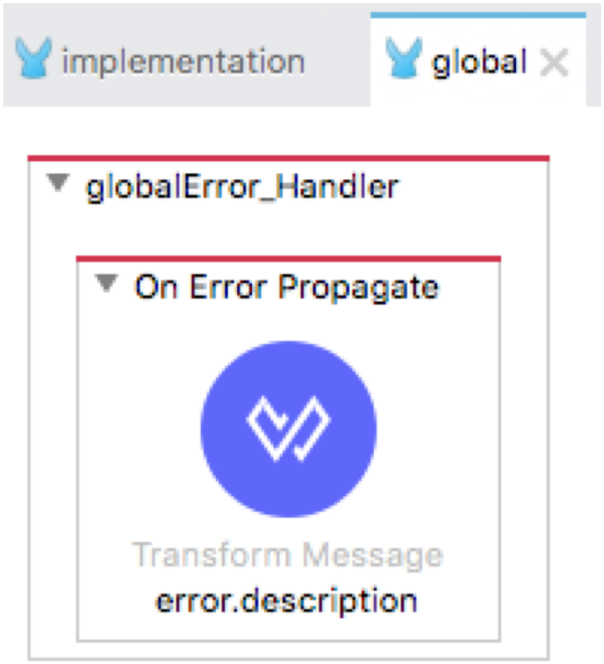
- Specify this handler to be the application's default error handler





Walkthrough 10-2: Handle errors at the application level

- Create a global error handler in an application
- Configure an application to use a global default error handler
- Explore the differences between the On Error Continue and On Error Propagate scopes
- Modify the default error response settings for an HTTP Listener

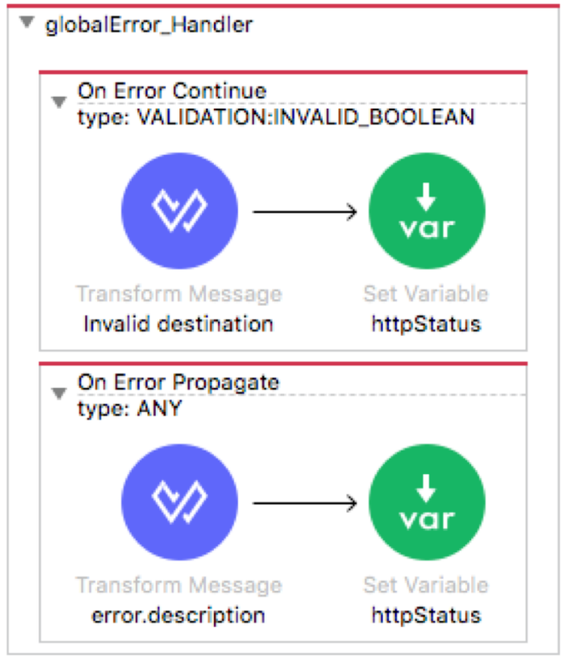


Handling specific types of errors

Adding multiple error handler scopes

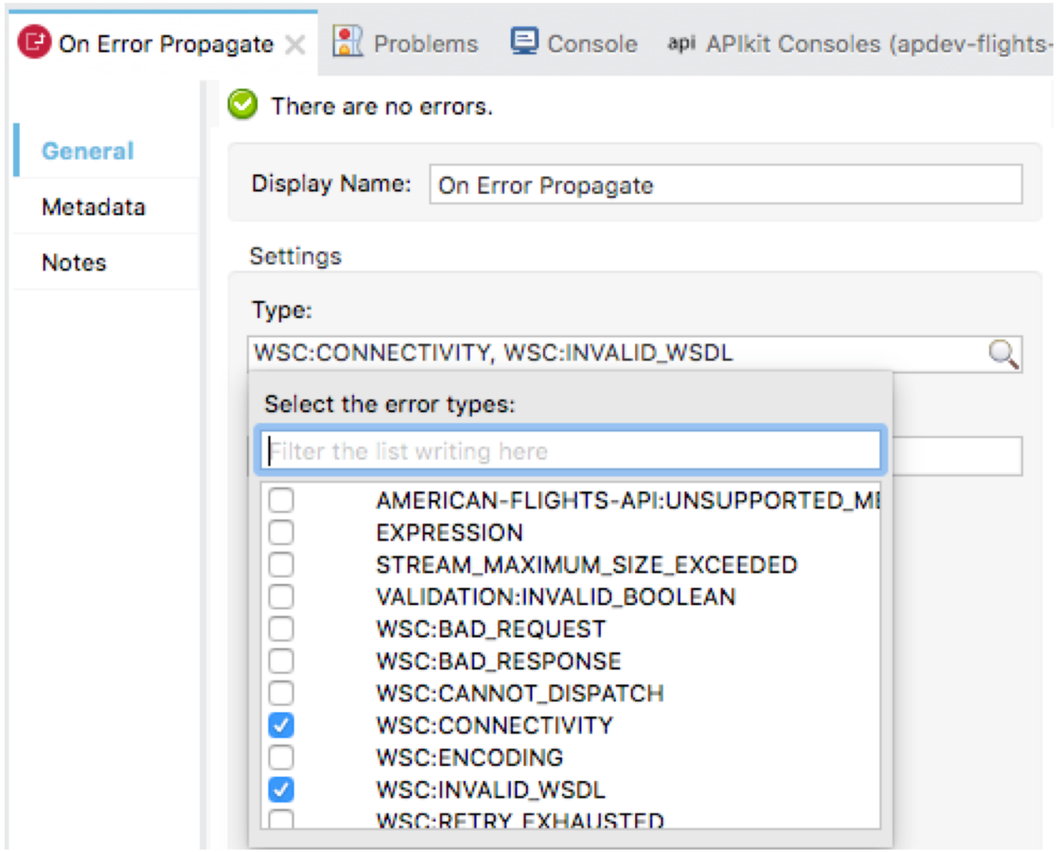
- Each error handler can contain one or more error handler scopes
  - Any number of On Error Continue and/or On Error Propagate

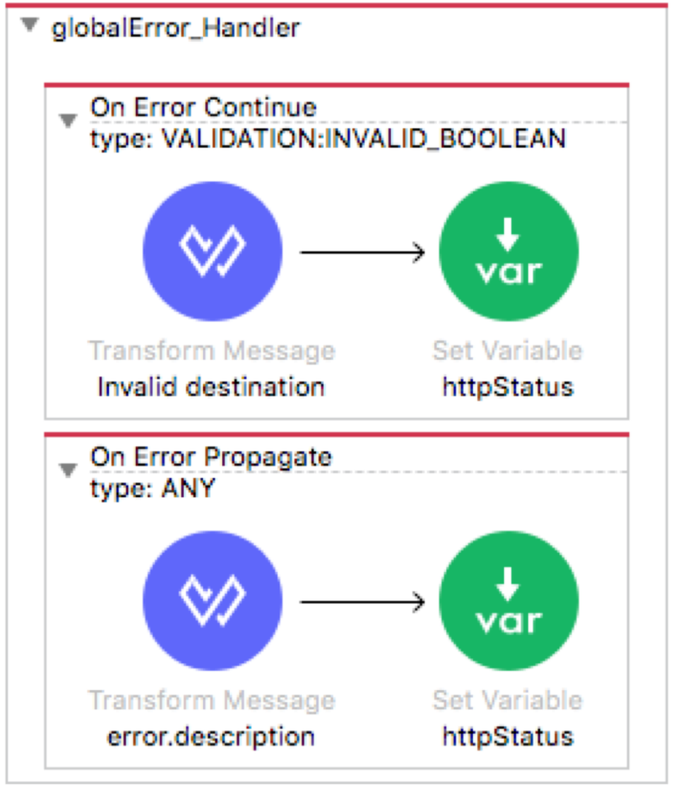
- Each error handler scope specifies when it should be executed
  - The error is handled by the *first* error scope whose condition evaluates to true



Specifying scope execution for specific error types

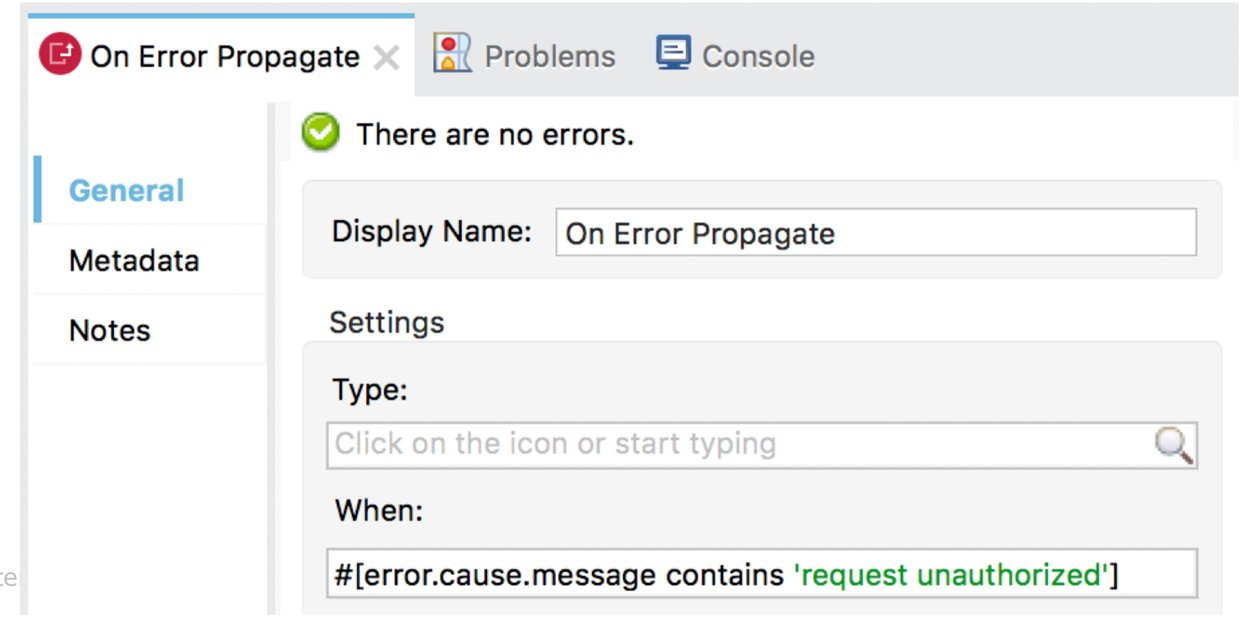
- Set the **type** to ANY (the default) or one or more types or errors





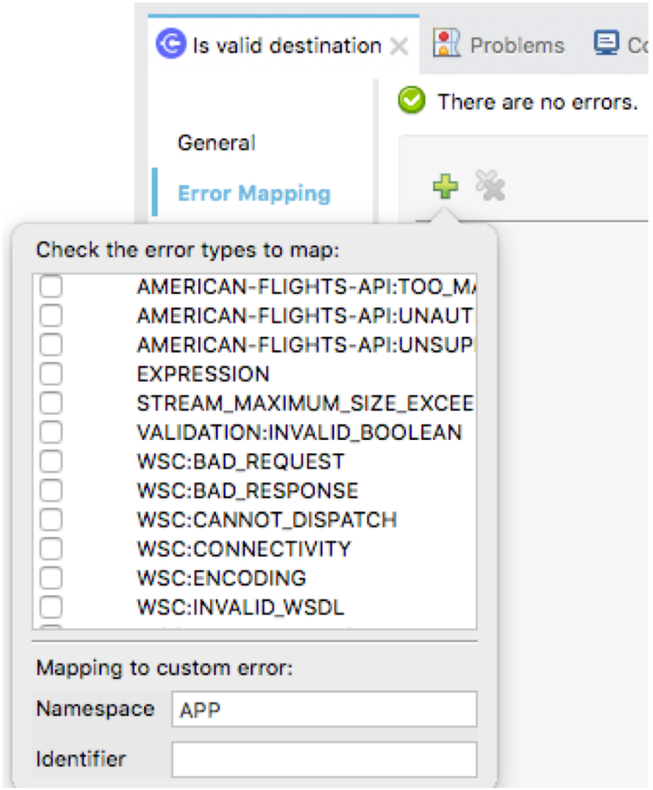
Specifying scope execution upon a specific condition

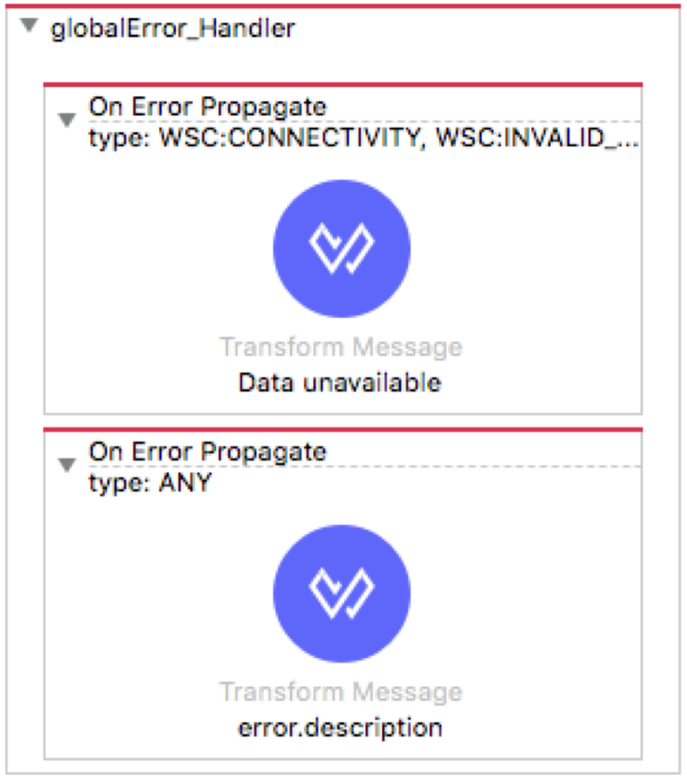
- Set the **when** condition to a Boolean DataWeave expression
  - HTTP:UNAUTHORIZED
  - error.errorType.namespace == 'HTTP'
  - error.errorType.identifier == 'UNAUTHORIZED'
  - error.cause.message contains 'request unauthorized'
  - error.cause.class contains 'http'



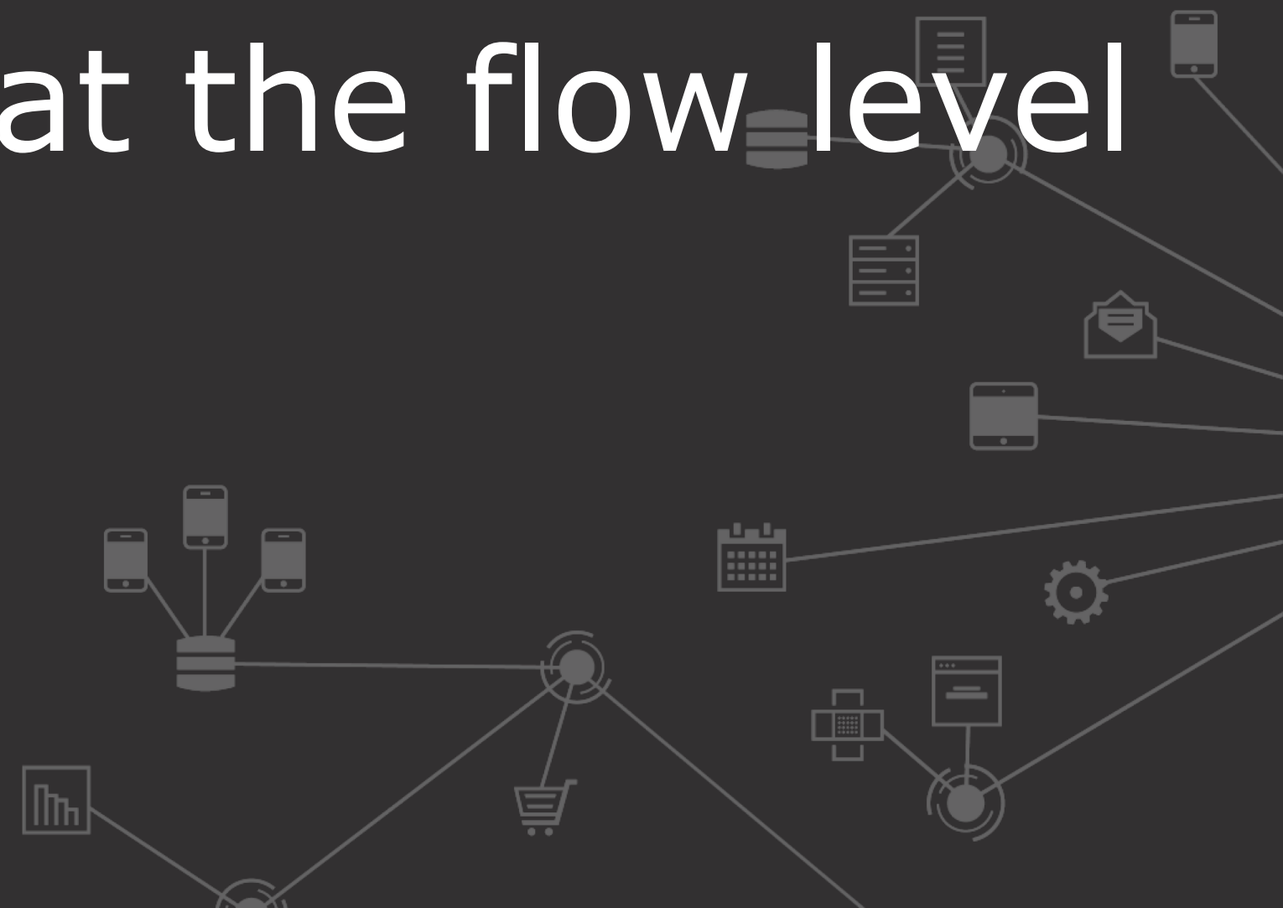
Walkthrough 10-3: Handle specific types of errors

- Review the possible types of errors thrown by different processors
- Create error handler scopes to handle different error types



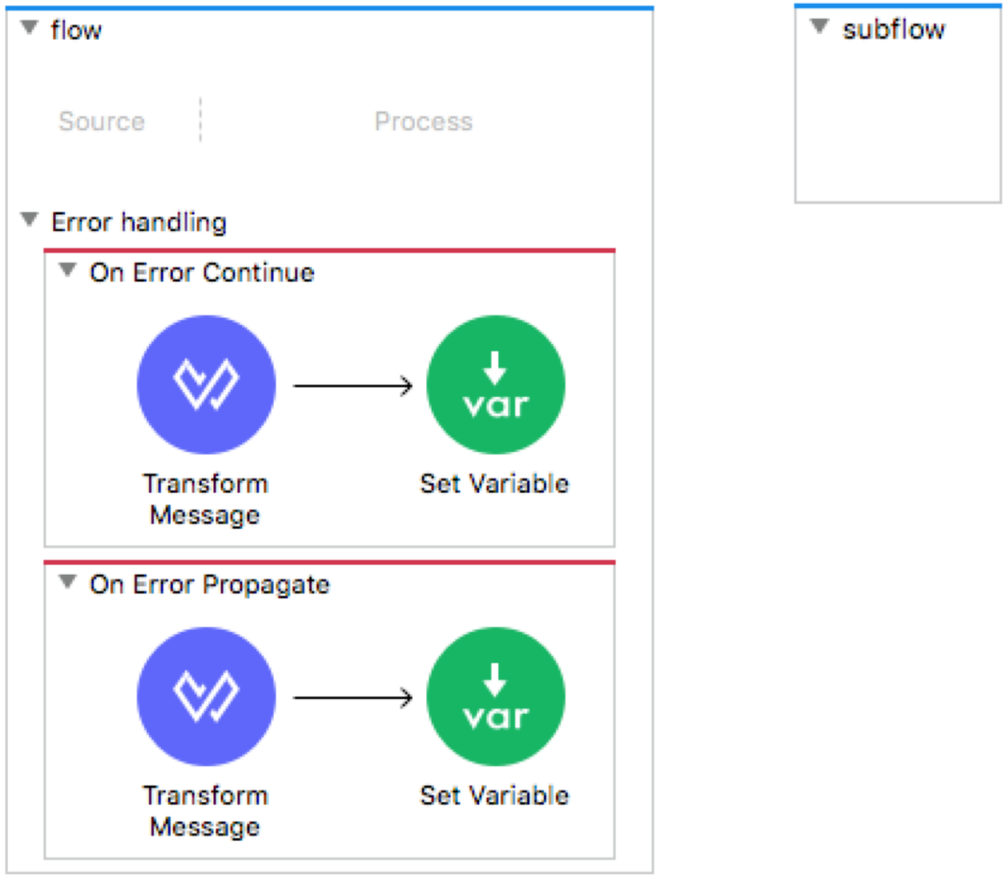


Handling errors at the flow level



Defining error handlers in flows

- All flows (except subflows) can have their own error handlers
- Any number of error scopes can be added to a flow's error handler

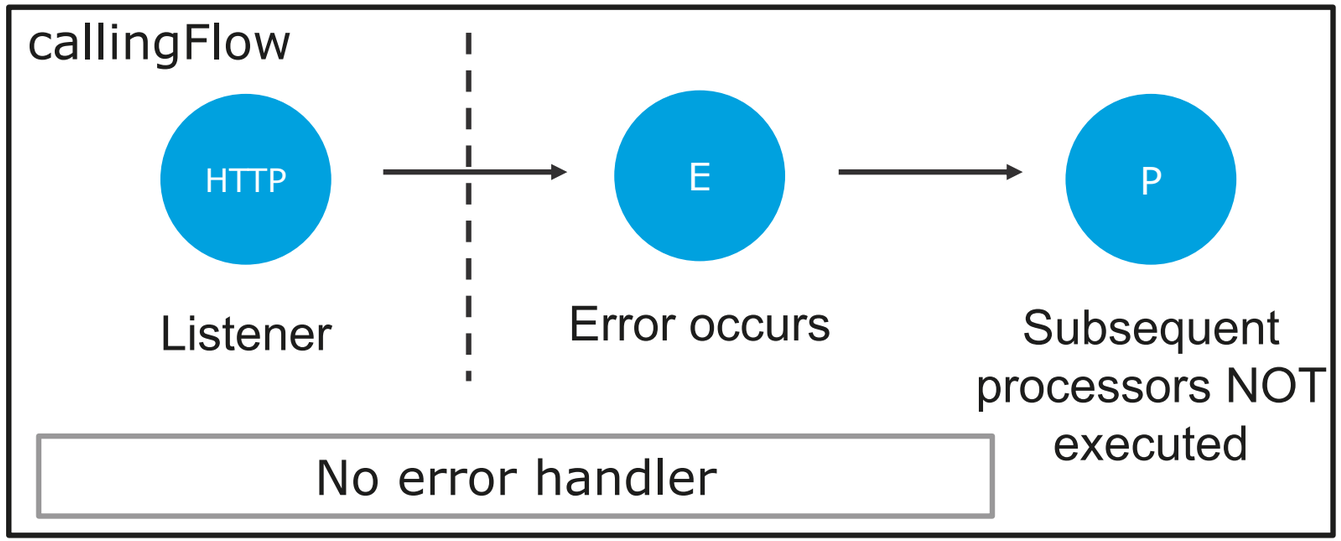


Which error scope handles an error?

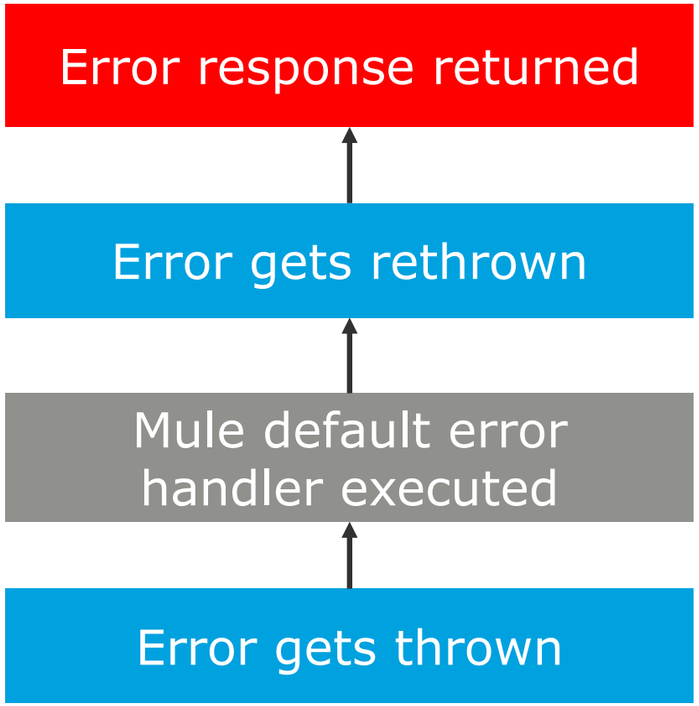
- If a flow *has* an error handler
  - The error is handled by the *first* error scope whose condition evaluates to true
  - **If *no* scope conditions are true,** the error is handled by the **Mule default error handler** NOT any scope in an application's global default handler
    - The Mule default error handler propagates the error up the execution chain where there may or may not be handlers

- If a flow *does not have* an error handler
  - The error is handled by a scope in an **application's default error handler** (the first whose scope condition is true, which may propagate or continue) otherwise it is handled by the Mule default error handler

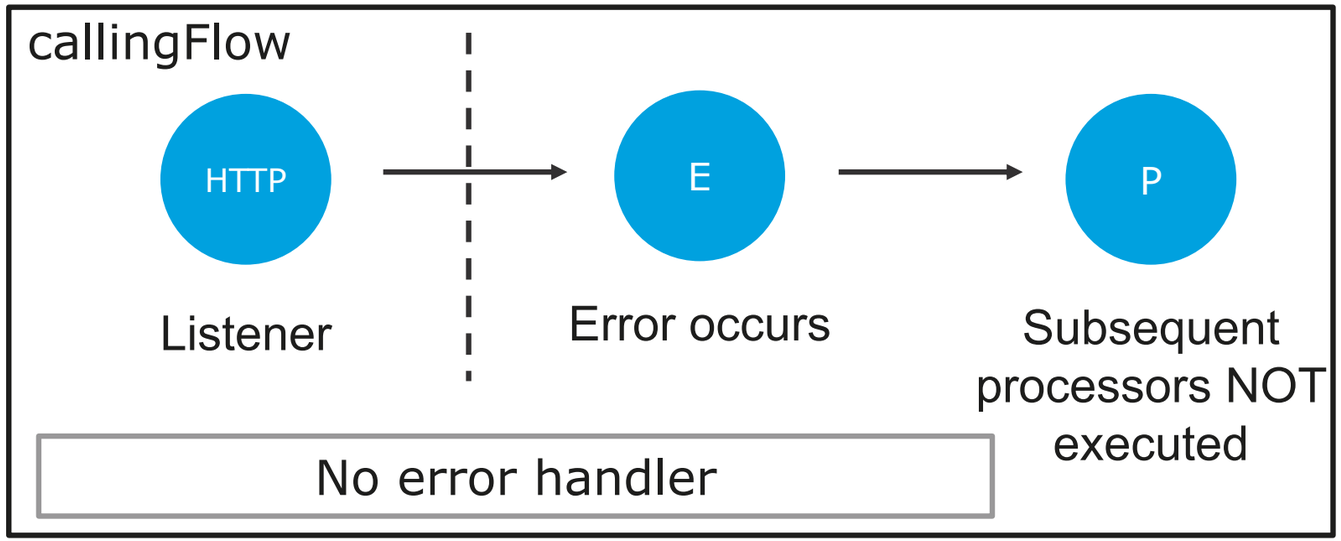
Error handling scenario 1



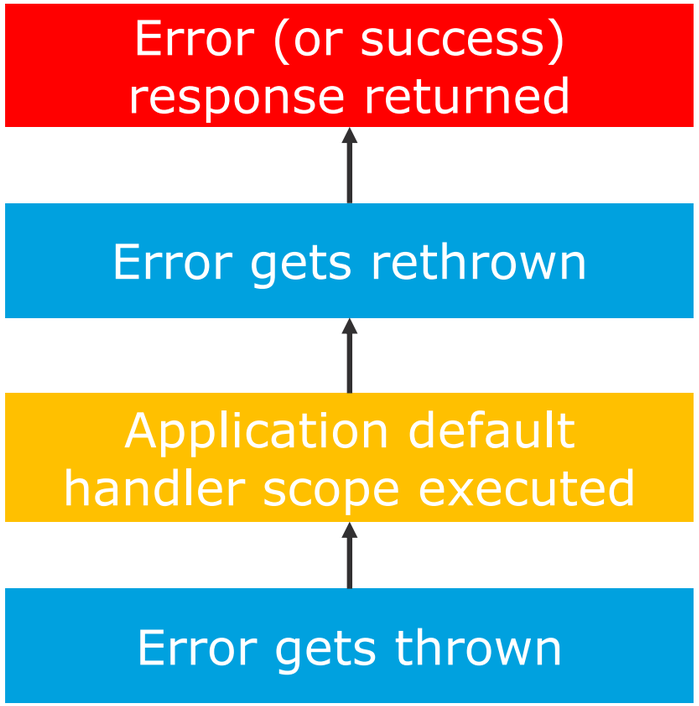
Application default error handler – ***with no matching scope*** *condition*



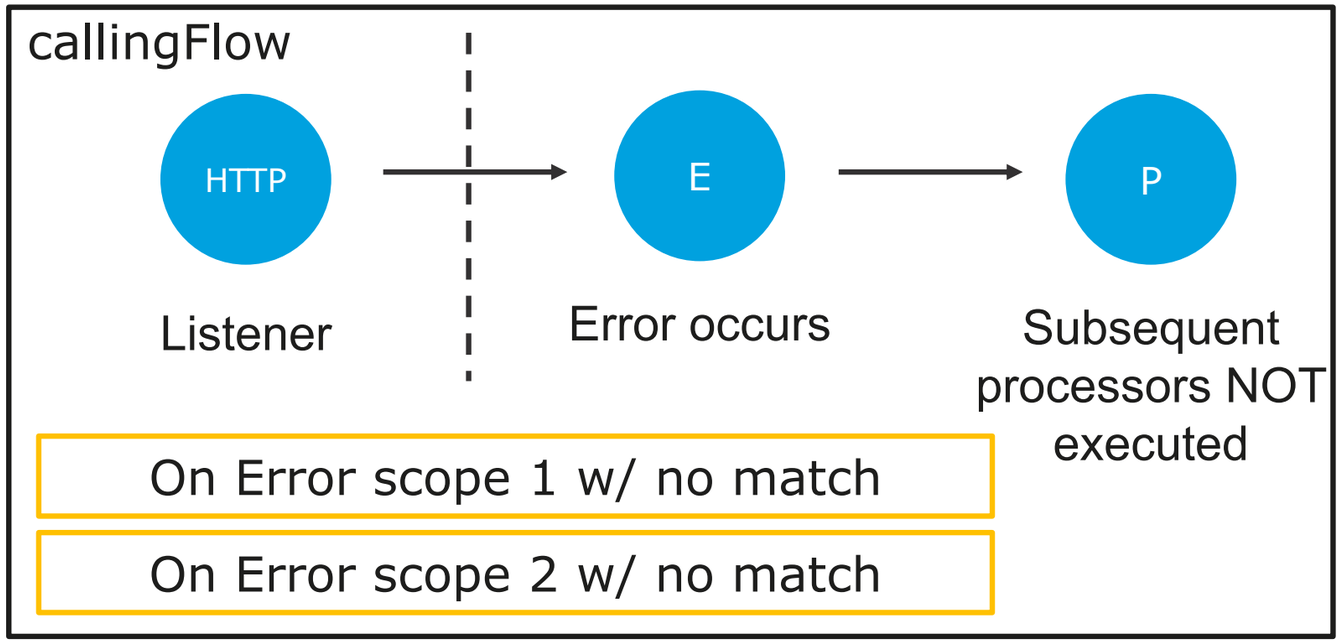
Error handling scenario 2



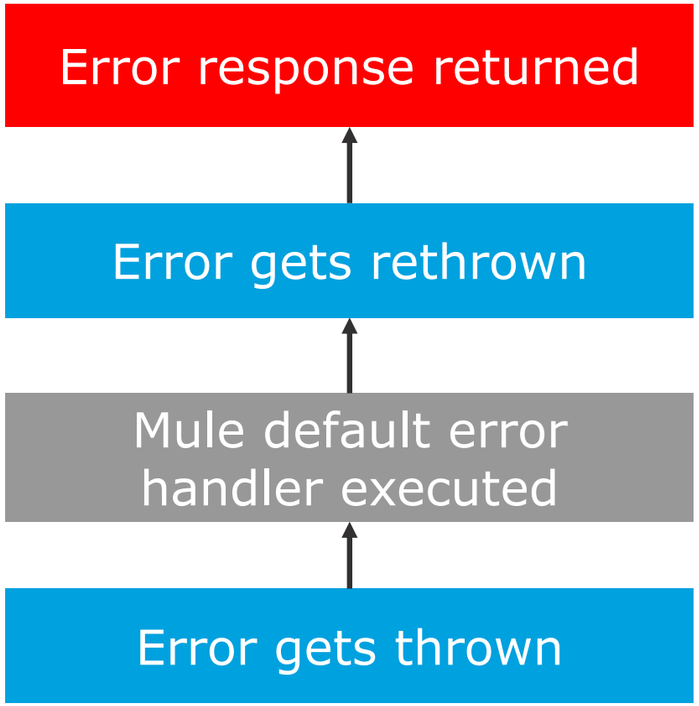
Application default error handler – ***with a matching scope condition***



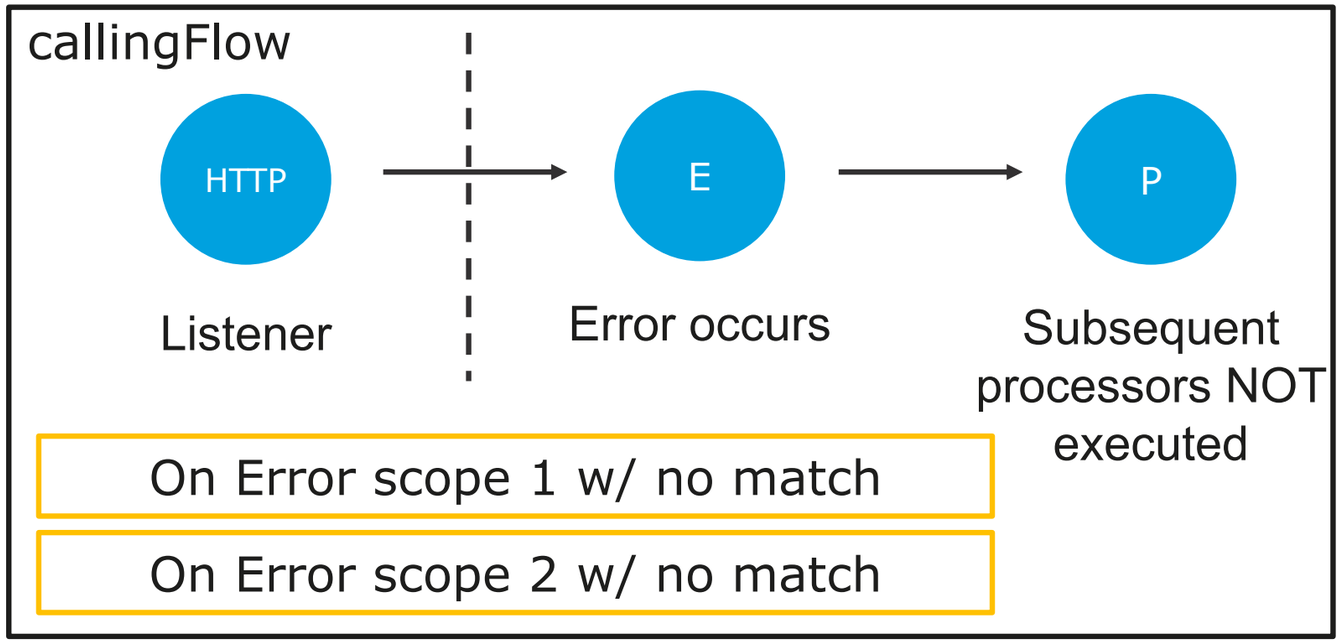
Error handling scenario 3



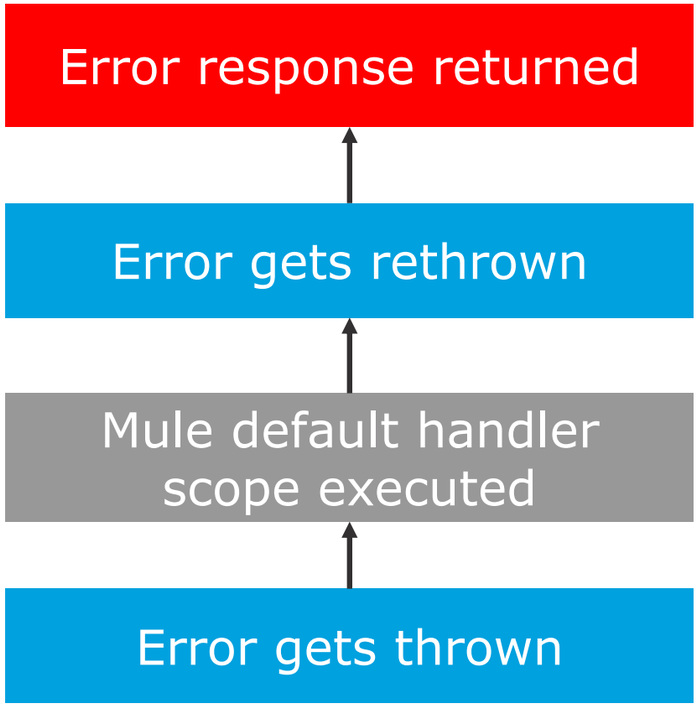
Application default error handler – ***with no matching scope*** *condition*



Error handling scenario 4

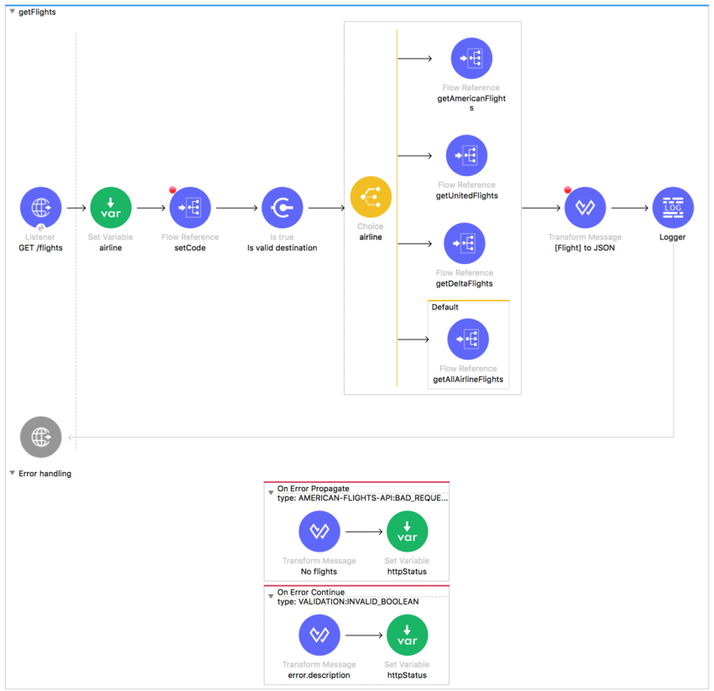


Application default error handler –
with a matching scope condition



Walkthrough 10-4: Handle errors at the flow level

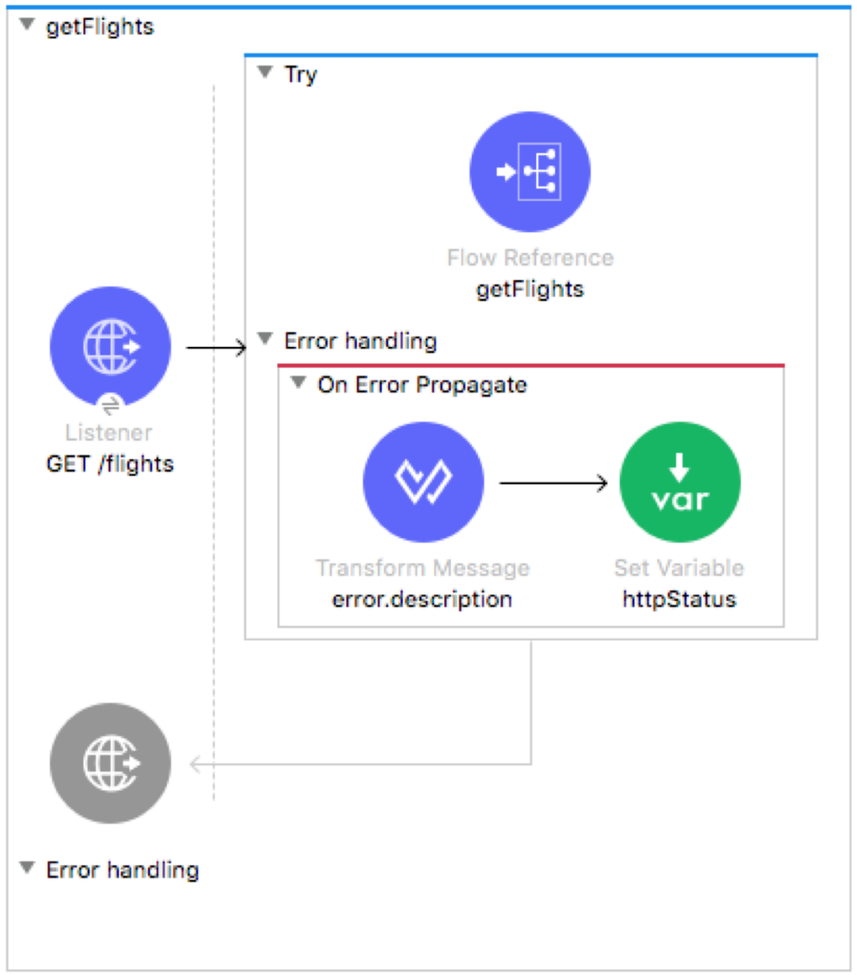
- Add error handlers to a flow
- Test the behavior of errors thrown in a flow and by a child flow
- Compare On Error Propagate and On Error Continue scopes in a flow
- Set an HTTP status code in an error handler and modify an HTTP Listener to return it



Handling errors at the processor level

Handling errors at the processor level

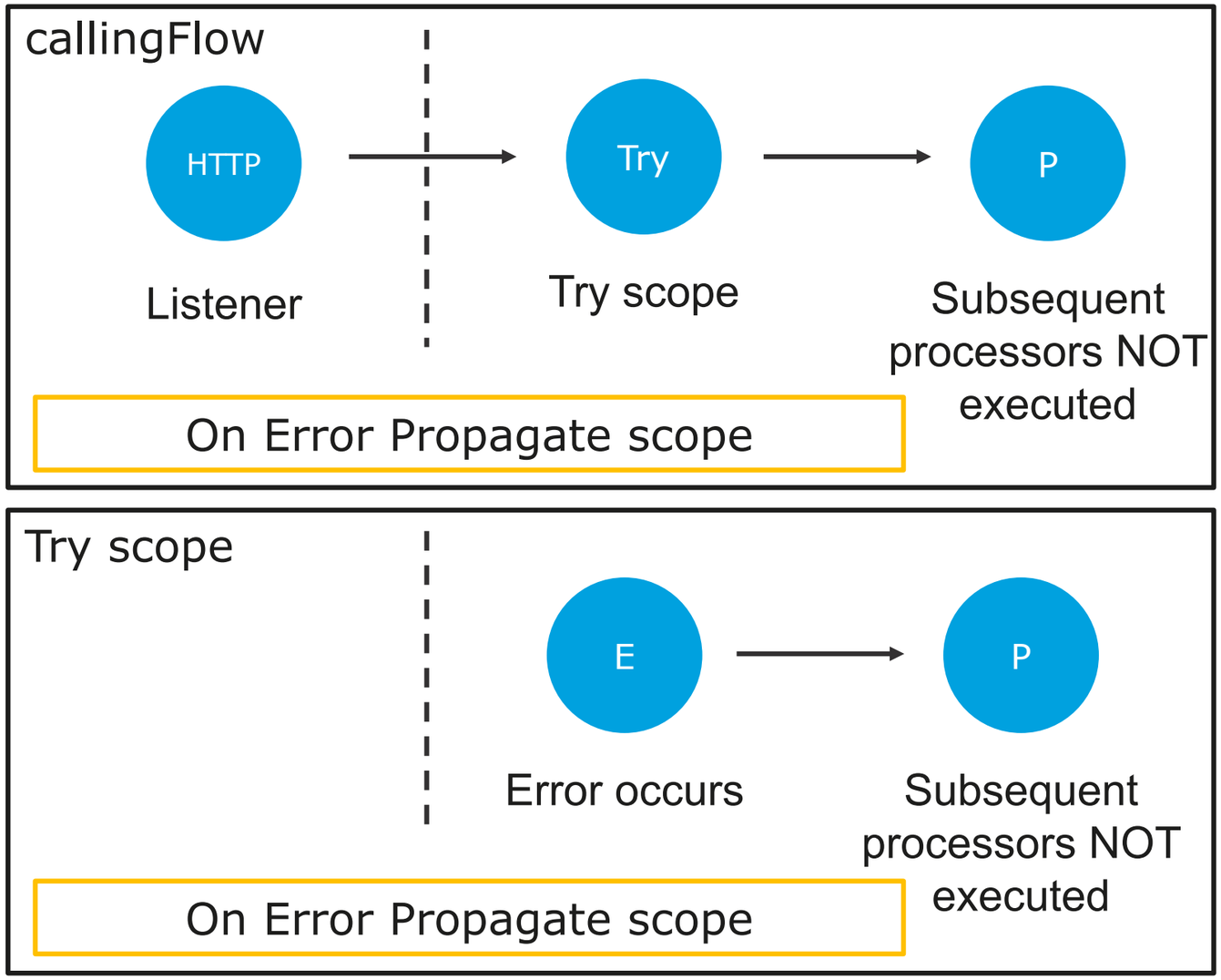
- For more fine grain error handling of elements within a flow, use the Try scope
- Any number of processors can be added to a Try scope
- The Try scope has its own error handling section to which one or or more error scopes can be added

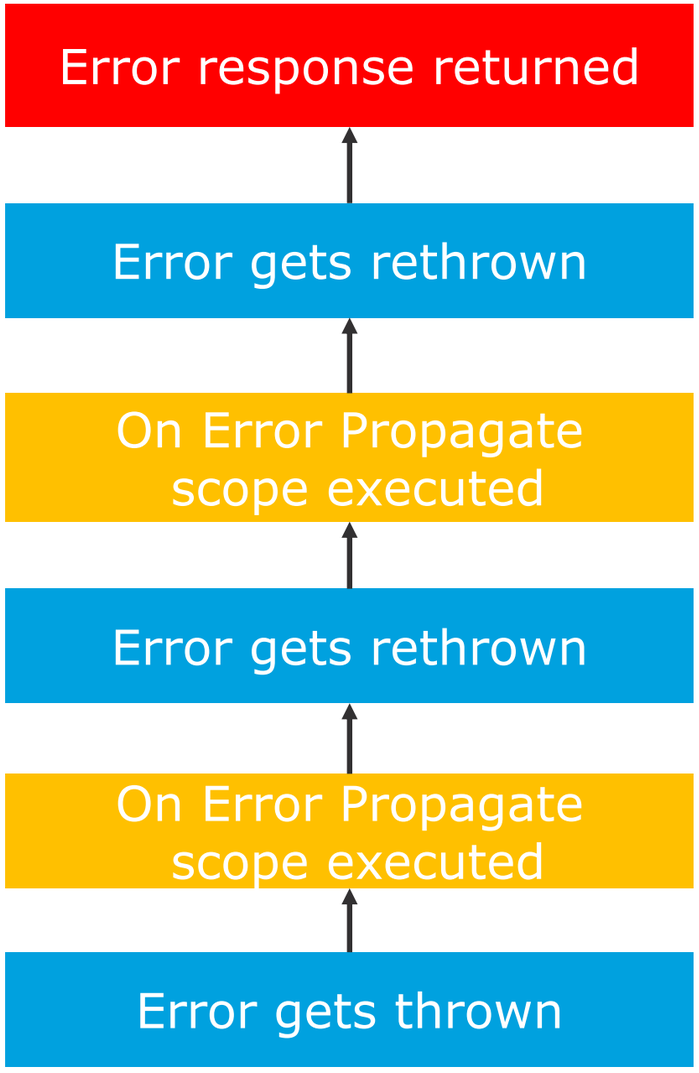


Error handling behavior in the Try scope

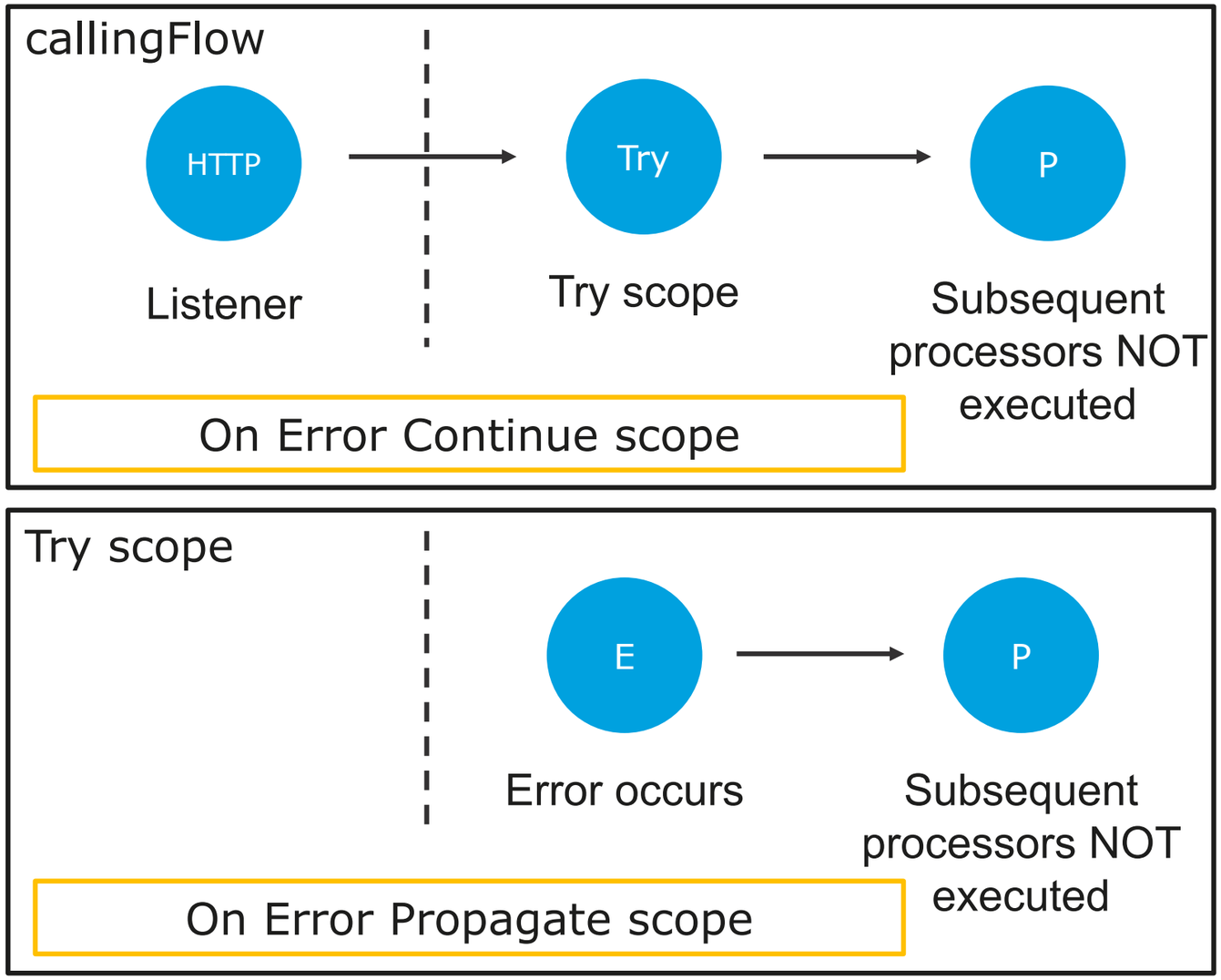
- **On Error Propagate**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the *Try scope* is not executed
    - If a transaction is begin handled, it is rolled back
    - The error is rethrown up the execution chain to the parent flow, which handles the error
  - An HTTP Listener returns an *error* response
- **On Error Continue**
  - All processors in the error handling scope are executed
  - At the end of the scope
    - The rest of the *Try scope* is not executed
    - If a transaction is begin handled, it is committed
    - The event is passed up to the parent flow, which continues execution
  - An HTTP Listener returns a *successful* response

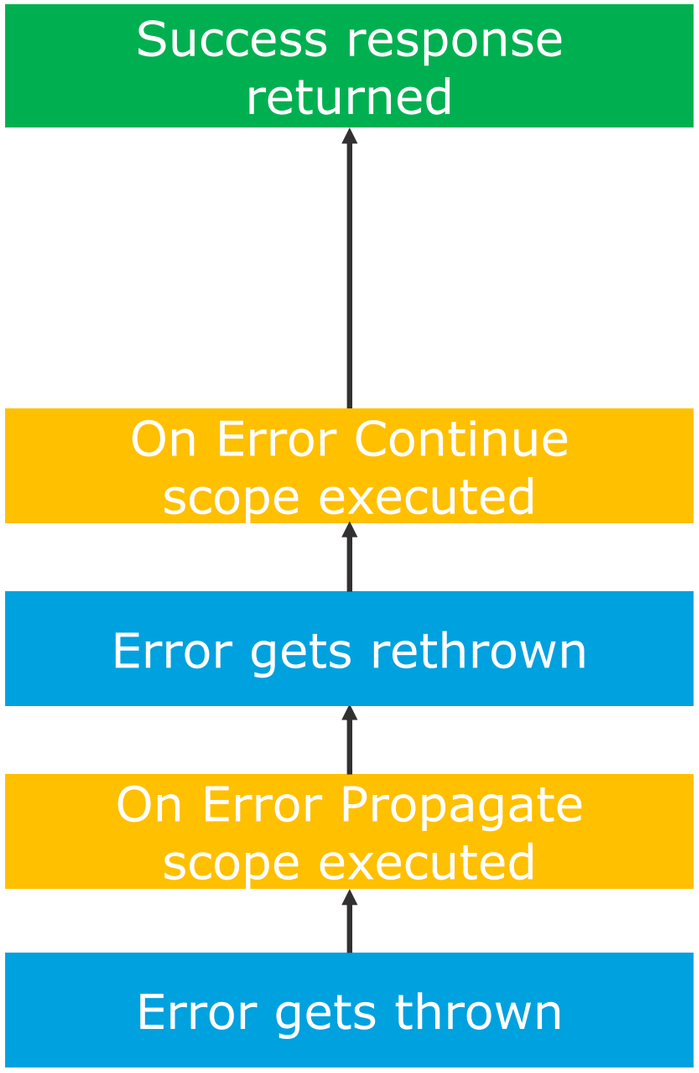
Try scope: Error handling scenario 1



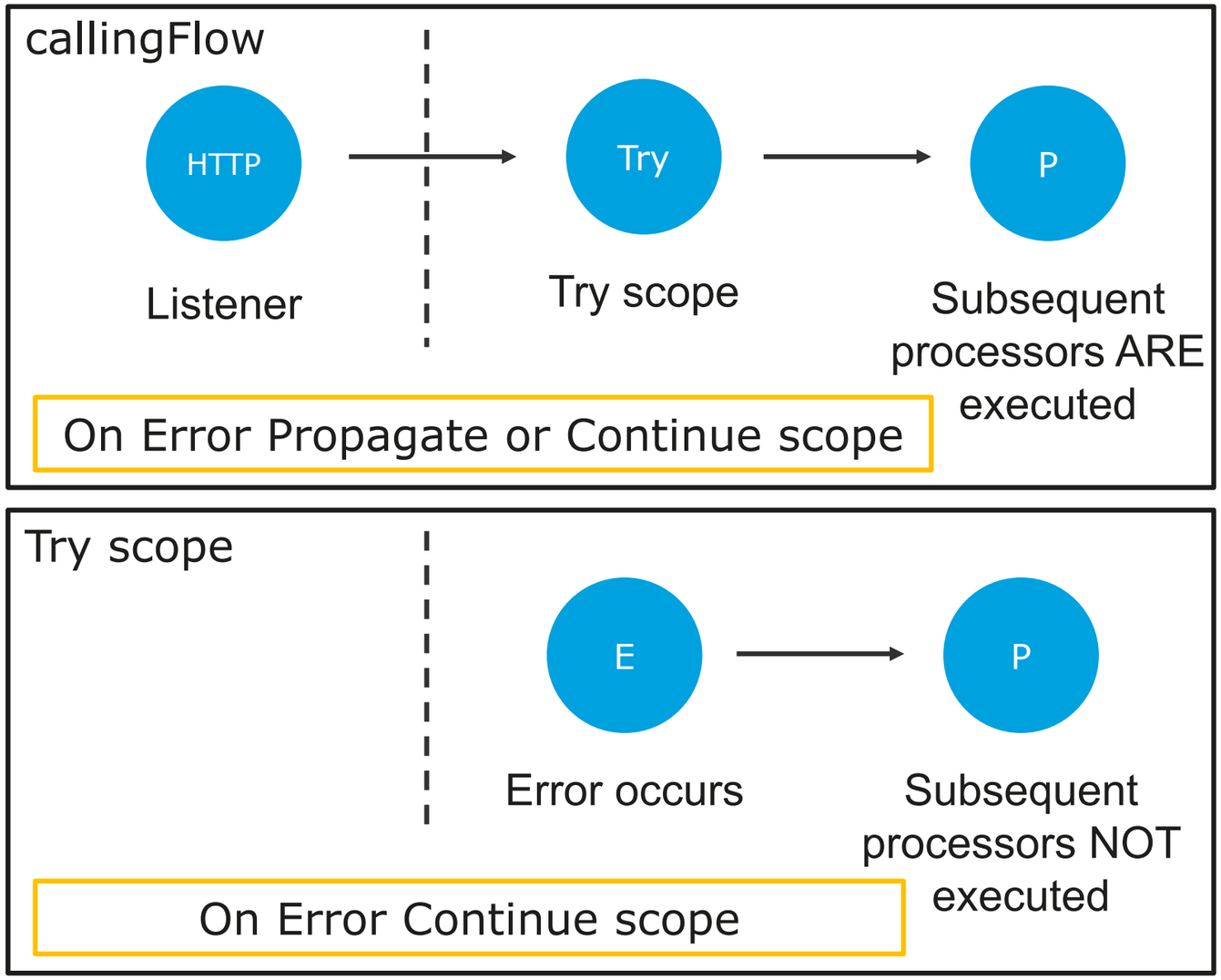


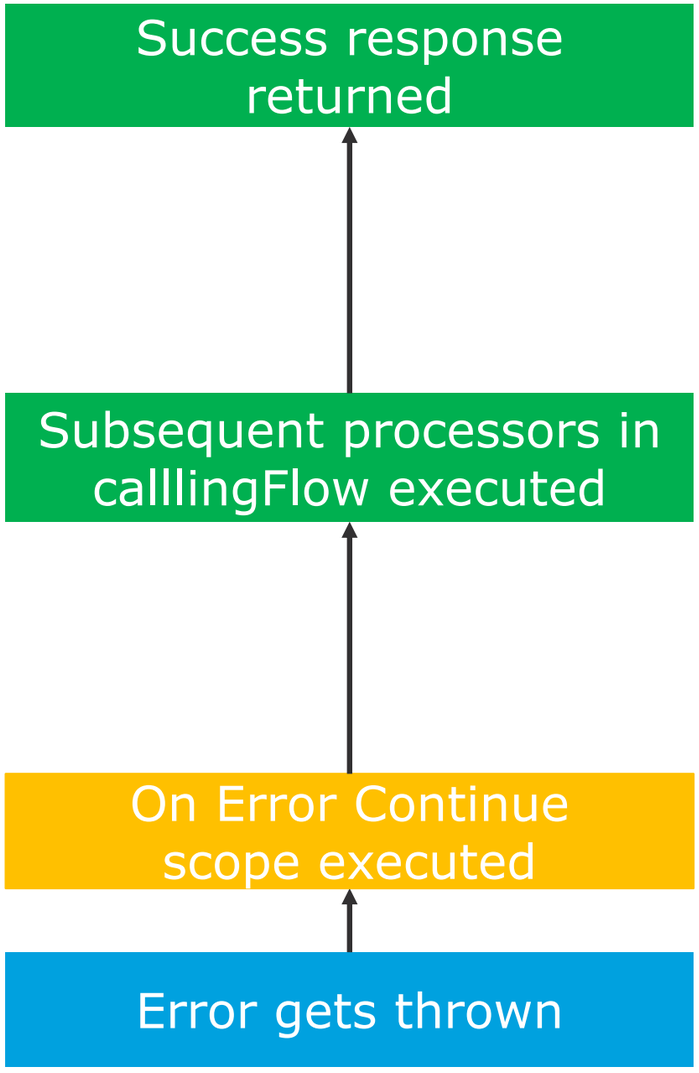
Try scope: Error handling scenario 2





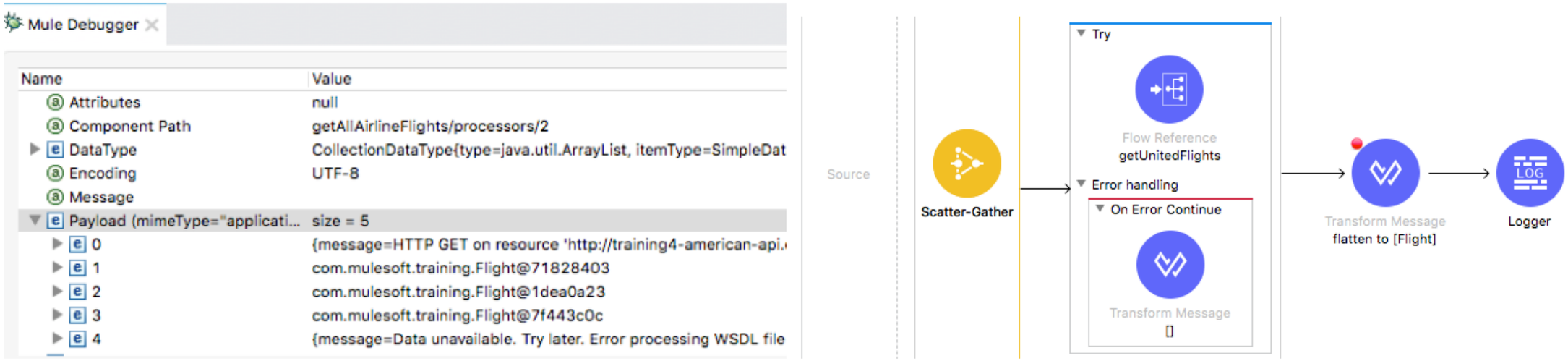
Try scope: Error handling scenario 3





Walkthrough 10-5: Handle errors at the processor level

- Wrap the Flow Reference in each branch of a Scatter-Gather in a Try scope
- Use an On Error Continue scope in each Try scope error handler to provide a valid response so flow execution can continue



Mapping errors to custom error types

Mapping errors for more granular error handling

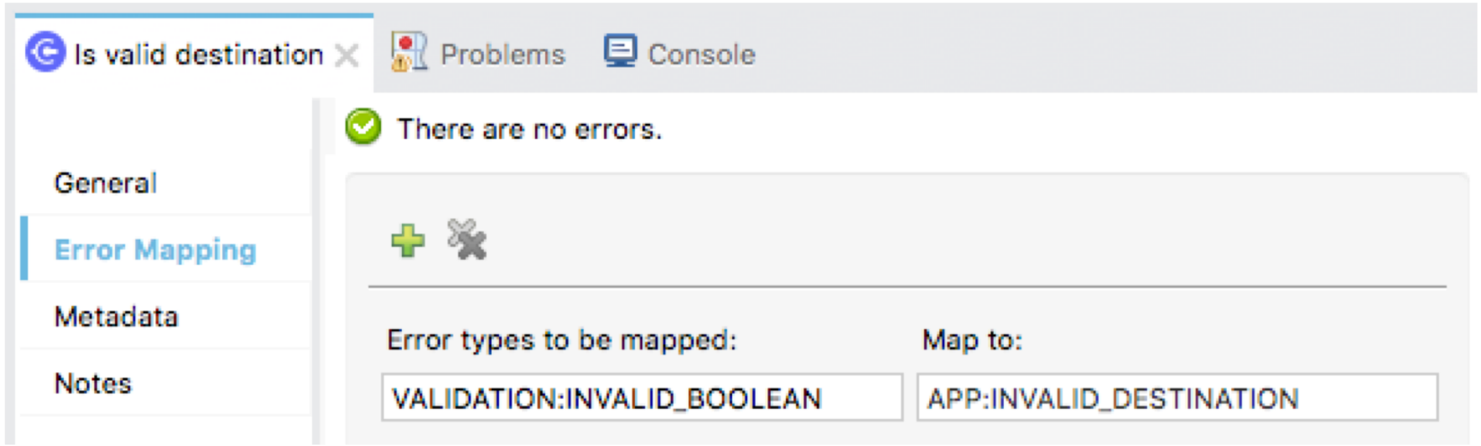
- If an app has two HTTP Request operations that call different REST services, a connectivity failure on either produces the same error
  - Makes if difficult to identify the source of the error in the Mule application logs
- You can map each connectivity error to different custom error types
- These custom error types enable you to differentiate exactly where an error occurred

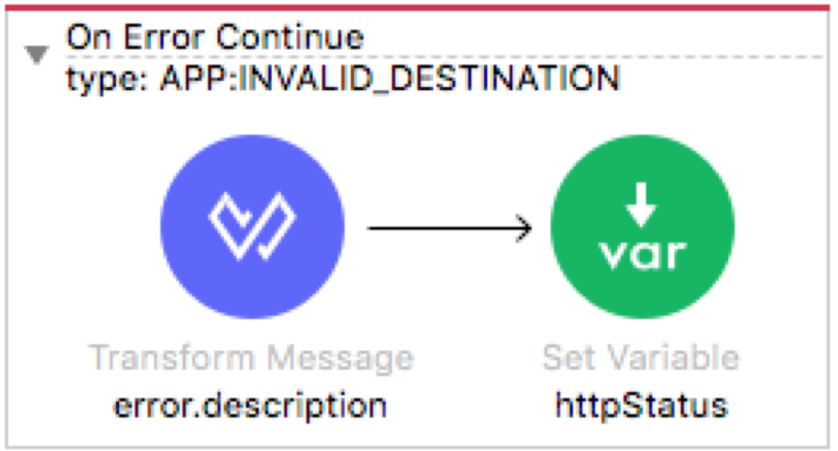
Mapping errors to custom error types

- For each module operation in a flow, each possible error type can be mapped to a custom error type
- You assign a custom namespace and identifier to distinguish them from other existing types within an application
  - Define namespaces related to the particular Mule application name or context
    - CUSTOMER namespace for errors with a customer aggregation API
    - ORDER namespace for errors with an order processing API
  - Do not use existing module namespaces

Walkthrough 10-6: Map an error to a custom error type

- Map a module error to a custom error type for an application
- Create an event handler for the custom error type



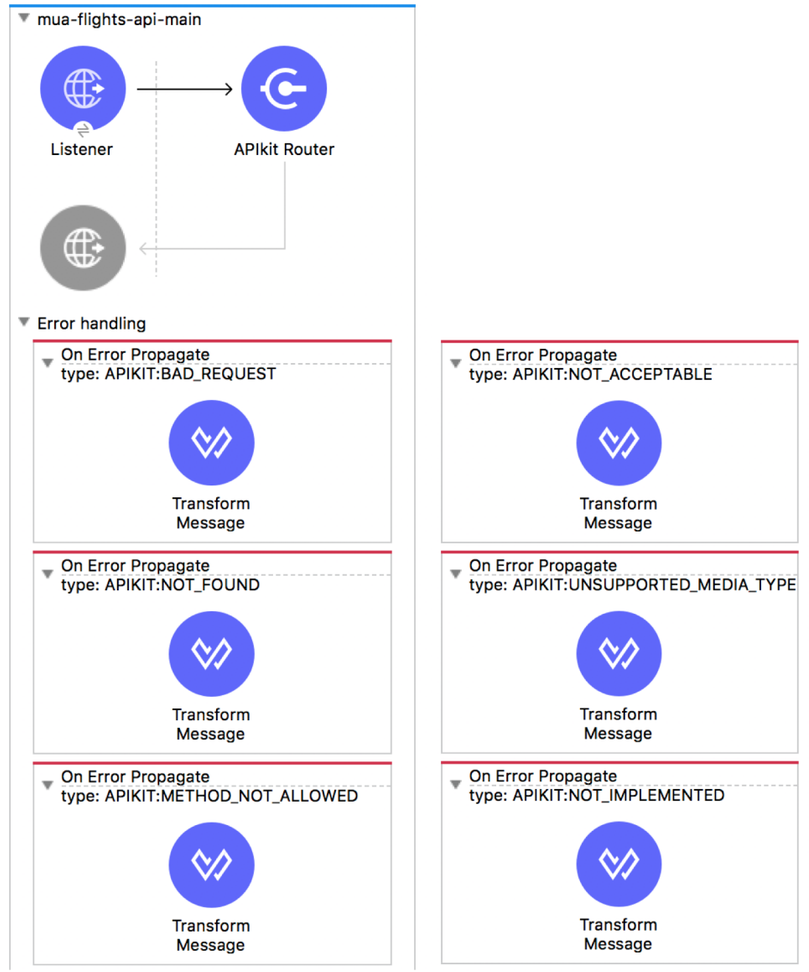


Reviewing and integrating with APIkit error handling

Error handling generated by APIkit

- By default, interfaces created with APIkit have error handlers with multiple On Error Propagate scopes that handle APIkit errors
  - The error scopes set HTTP status codes and response messages

- The main routing flow has six error scopes
  - APIKIT:BAD_REQUEST > 400
  - APIKIT:NOT_FOUND > 404
  - APIKIT:METHOD_NOT_ALLOWED > 405
  - APIKIT:NOT_ACCEPTABLE > 406
  - APIKIT:UNSUPPORTED_MEDIA_TPYE > 415
  - APIKIT:NOT_IMPLEMENTED > 501



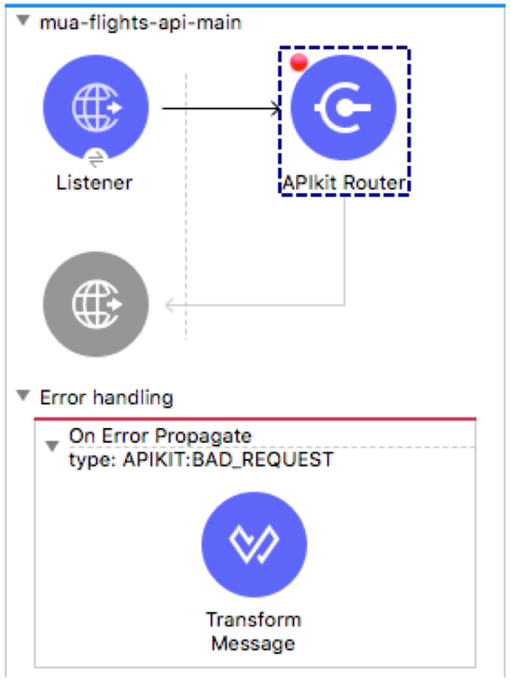
Integrating with APIkit error handling

- You can modify the APIkit error scopes and add additional scopes

- You also need to make sure the error handling in the application works as expected with the new interface router
  - **On Error Continue**
    - Event in implementation is not passed back to main router flow
  - **On Error Propagate**
    - Error is implementation is propagated to main router flow
      - Lose payload and variables

Walkthrough 10-7: Review and integrate with APIkit error handlers

- Review the error handlers generated by APIkit
- Review settings for the APIkit Router and HTTP Listener in the APIkit generated interface
- Connect the implementation to the interface and test the error handler behavior
- Modify implementation error scopes so they work with the APIkit generated interface



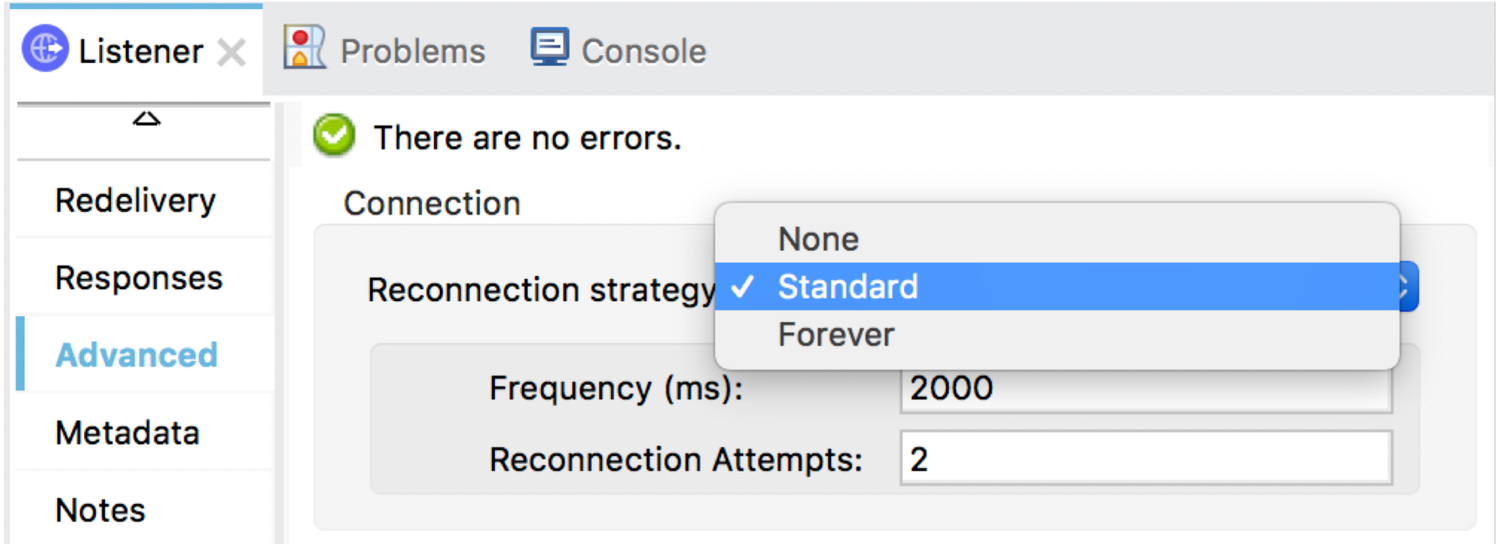
Handling system errors

Applications can have two types of errors

- Messaging errors
  - Thrown within a flow whenever a Mule event is involved
- System errors
  - Thrown at the system-level when *no* Mule event is involved
  - Errors that occur
    - During application start-up
    - When a connection to an external system fails
  - Handled by a system error handling strategy
    - Non configurable
    - Logs the error and for connection failures, executes the reconnection strategy

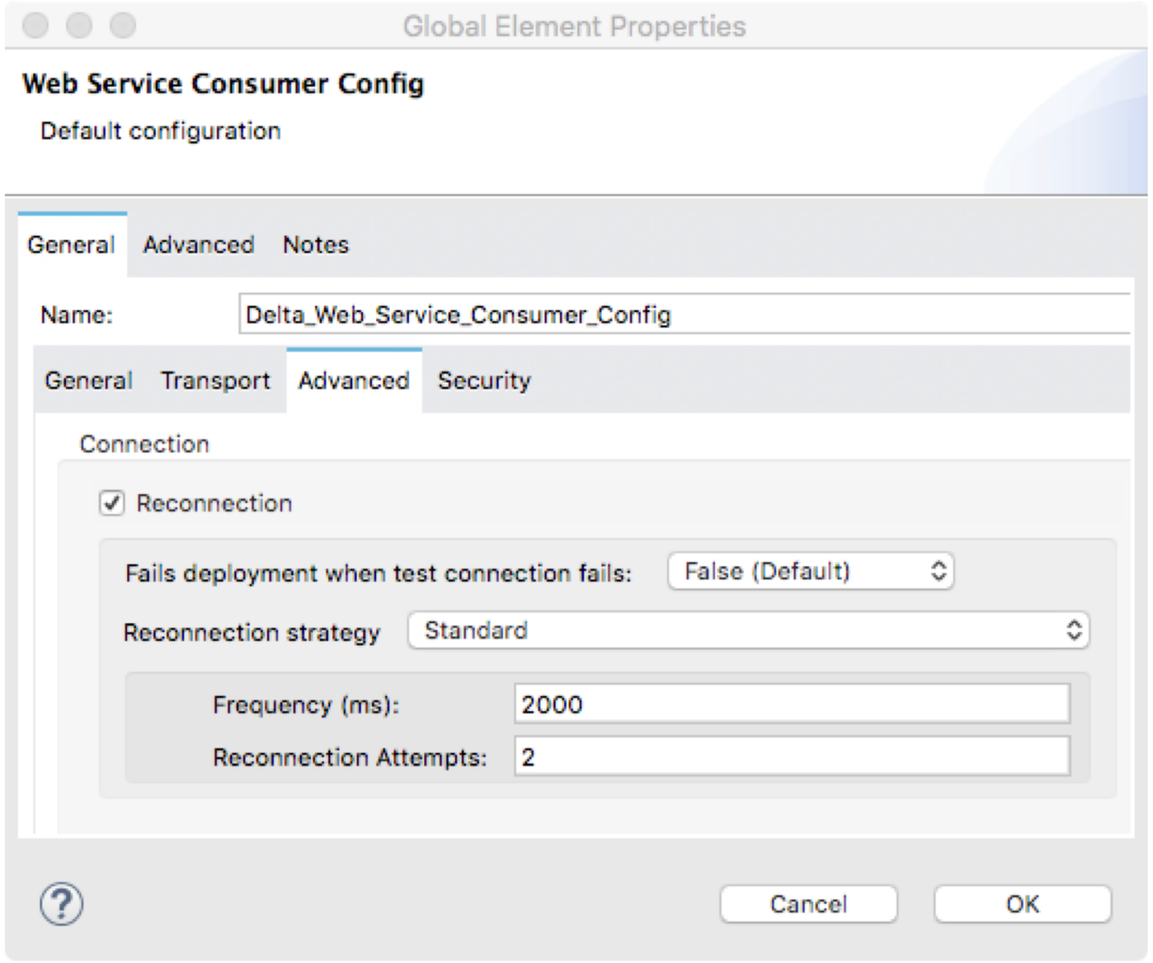
Reconnection strategies

- Set for a connector (in Global Elements Properties) or for a specific connector operation (in Properties view)



Walkthrough 10-8: Set a reconnection strategy for a connector

- Set a reconnection strategy for the Web Service Consumer connector



Summary

Summary

- An application can have system or messaging errors
- **System errors** are thrown at the system level and involve no event
  - Occur during application start-up or when a connection to an external system fails
  - Non-configurable, but logs the error and for connections, executes any reconnection strategy
- **Messaging errors** are thrown when a problem occurs within a flow
  - Normal flow execution stops and the event is passed to an error handler (if one is defined)
  - By default, unhandled errors are logged and propagated
  - HTTP Listeners return success or error responses depending upon how the error is handled
  - Subflows cannot have their own error handlers

Summary

- Messaging errors can be handled at various levels
  - For an **application**, by defining an error handler outside any flow and then configuring the application to use it as the default error handler
  - For a **flow**, by adding error scopes to the error handling section
  - For one or more **processors**, by encapsulating them in a Try scope that has its own error handling section
- Each error handler can have one or more error scopes
  - Each specifies for what error type or condition for which it should be executed
- An error is handled by the first error scope with a matching condition
  - **On Error Propagate** rethrows the error up the execution chain
  - **On Error Continue** handles the error and then continues execution of the parent flow

Summary

- Error types for module operations can be mapped to **custom error types**
  - You assign a custom namespace and identifier to distinguish them from other existing types within an application
  - Enables you to differentiate exactly where an error occurred, which is especially useful when examining logs
- By default, interfaces created with **APIkit** have error handlers with multiple On Error Propagate scopes that handle APIkit errors
  - The error scopes set HTTP status codes and response messages
  - You can modify these error scopes and add additional scopes
  - Use On Error Continue in implementation to not pass event back to main router
  - Use On Error Propagate in implementation to propagate error to main router flow