

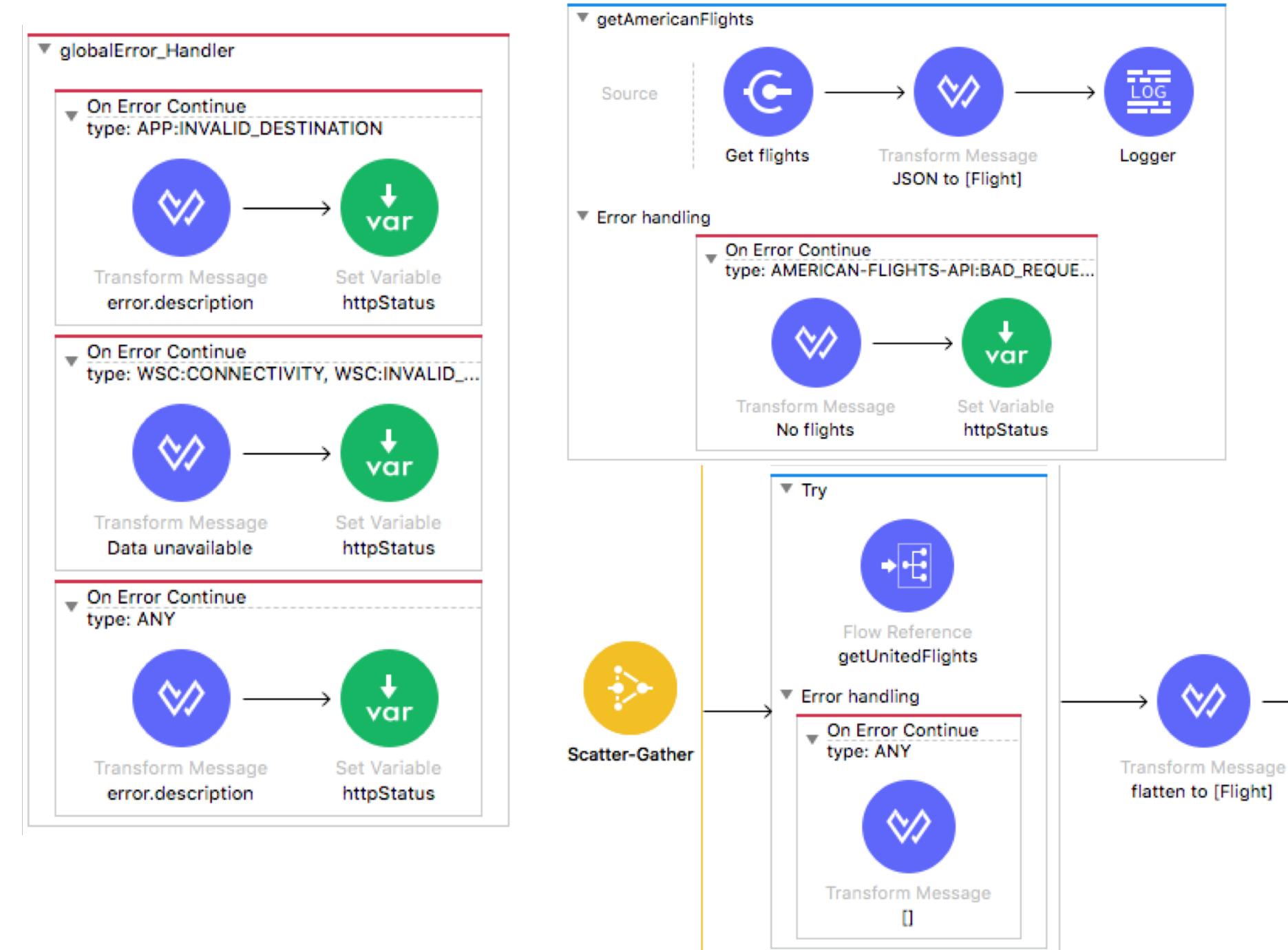


MuleSoft®

# Module 10: Handling Errors



# Goal



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



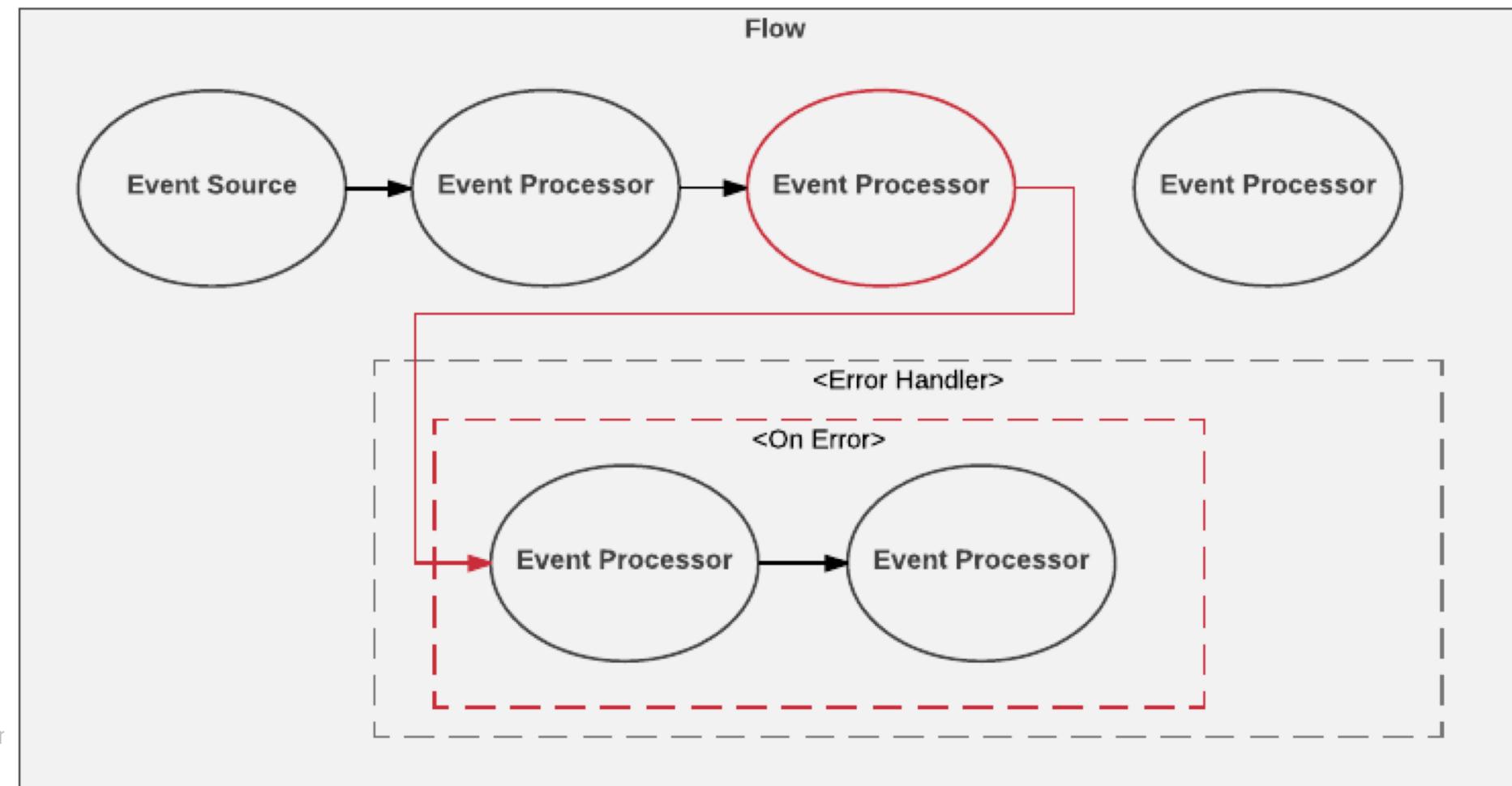
- Handle messaging errors at the application, flow, and processor level
- Handle different types of errors, including custom errors
- Use different error scopes to either handle an error and continue execution of the parent flow or propagate an error to the parent flow
- Set the success and error response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

# Reviewing the default handling of messaging errors

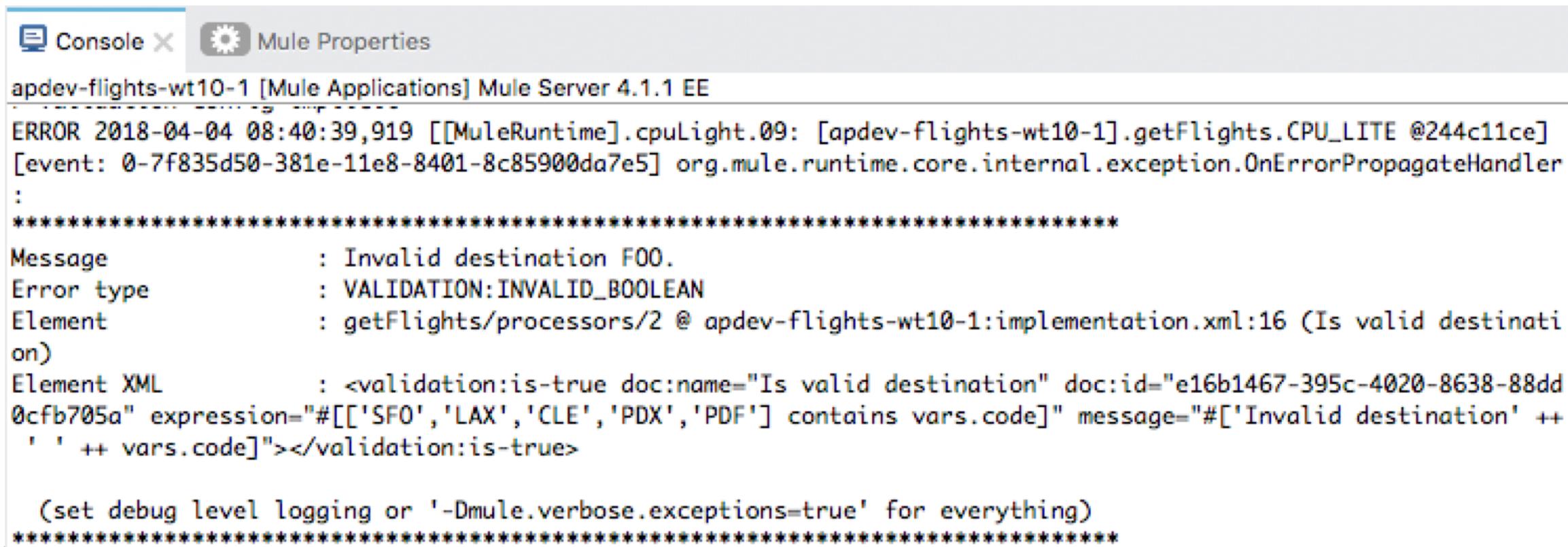


# Handling messaging errors

- 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



- 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



The screenshot shows the Mule Studio interface with the 'Console' tab selected. The title bar indicates 'apdev-flights-wt10-1 [Mule Applications] Mule Server 4.1.1 EE'. The console output displays an error message:

```
ERROR 2018-04-04 08:40:39,919 [[MuleRuntime].cpuLight.09: [apdev-flights-wt10-1].getFlights.CPU_LITE @244c11ce]
[event: 0-7f835d50-381e-11e8-8401-8c85900da7e5] org.mule.runtime.core.internal.exception.OnErrorPropagateHandler
:
*****
Message          : Invalid destination FOO.
Error type       : VALIDATION:INVALID_BOOLEAN
Element          : getFlights/processors/2 @ apdev-flights-wt10-1:implementation.xml:16 (Is valid destination)
Element XML      : <validation:is=true doc:name="Is valid destination" doc:id="e16b1467-395c-4020-8638-88dd
0cfb705a" expression="#[['SFO','LAX','CLE','PDX','PDF']] contains vars.code]" message="#['Invalid destination' ++
' ' ++ vars.code]"></validation:is=true>

(set debug level logging or '-Dmule.verbose.exceptions=true' for everything)
*****
```

# 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

Name	Value
EncodingException	UTF-8
Exception	
description	Invalid destination FOO
detailedDescription	Invalid destination FOO
errors	[]
errorType	VALIDATION:INVALID_BOOLEAN
exception	org.mule.extension.validation.api.Va
muleMessage	
Message	

# Error types follow a hierarchy



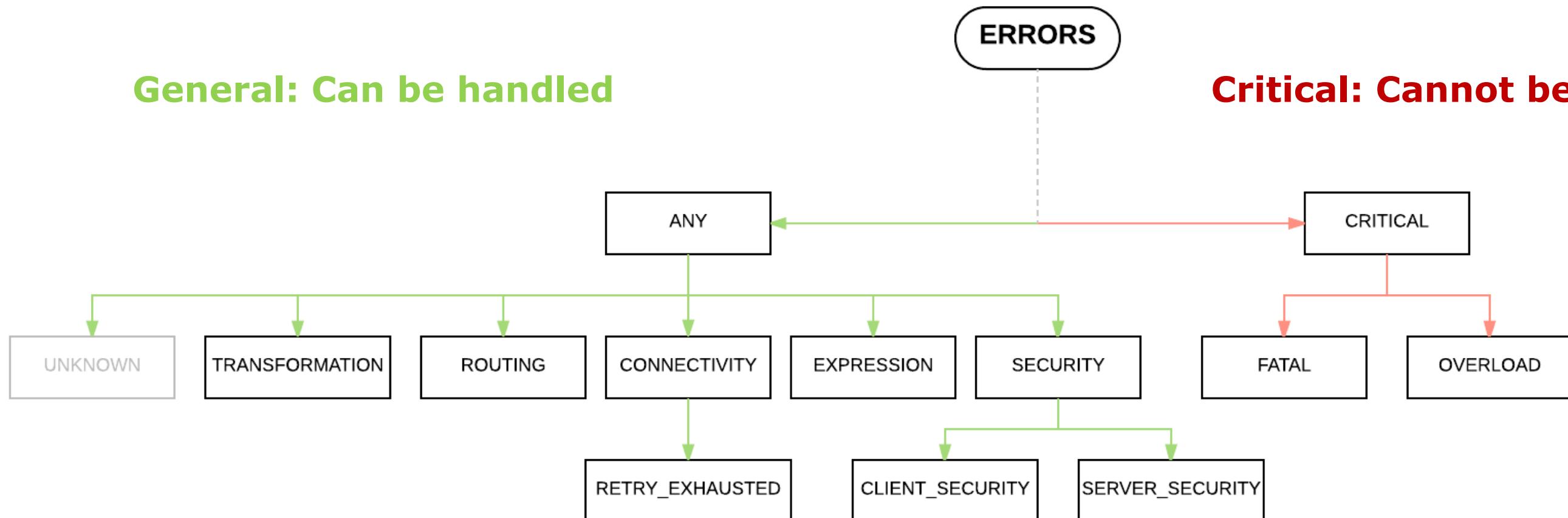
- Each error type has a parent
  - HTTP:UNAUTHORIZED has MULE:CLIENT\_SECURITY as the parent, which has MULE:SECURITY as the parent
  - VALIDATION:INVALID\_BOOLEAN has VALIDATION:VALIDATION as the parent, which has MULE:VALIDATION as the parent
- The error type ANY is the most general parent

```
{  
  "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

**General: Can be handled**

**Critical: Cannot be handled**



# 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

A screenshot of the MuleSoft Anypoint Studio interface. The top bar shows a GET /flights request and a Problems tab indicating no errors. The left sidebar has tabs for General, MIME Type, Redelivery, Responses (which is selected), Advanced, Metadata, and Notes. The main area is titled 'Error Response' and contains sections for 'Body' with the code '#[output text/plain --- error.description]', 'Headers' with a table, and fields for 'Status code:' and 'Reason phrase:'.

# 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

The screenshot displays two Mule Studio panes illustrating error handling.

**Mule Debugger:** Shows the stack trace for an exception. The exception type is `WSC:CONNECTIVITY`, and the cause is `org.mule.runtime.api.conne`.

Name	Value
description	Error processing WSDL file
detailedDescription	Error processing WSDL file
errors	[]
errorType	WSC:CONNECTIVITY
exception	org.mule.runtime.api.conne
muleMessage	null

**APIkit Consoles:** Shows the configuration for an HTTP Listener at `GET /flights`. The error response is set to return a JSON object with the key `#[output application/json --- error.errorType]`. The status code is 400 Bad Request, and the reason phrase is "Bad Request".

**Body:**  
#[output application/json --- error.errorType]

**Headers:**

Name	Value
parentErrorType	{ "parentErrorType": { "parentErrorType": { "parentErrorType": null, "namespace": "MULE", "identifier": "ANY" }, "namespace": "MULE", "identifier": "CONNECTIVITY" }, "namespace": "WSC", "identifier": "CONNECTIVITY" }

**Status code:** 400  
**Reason phrase:** Bad Request

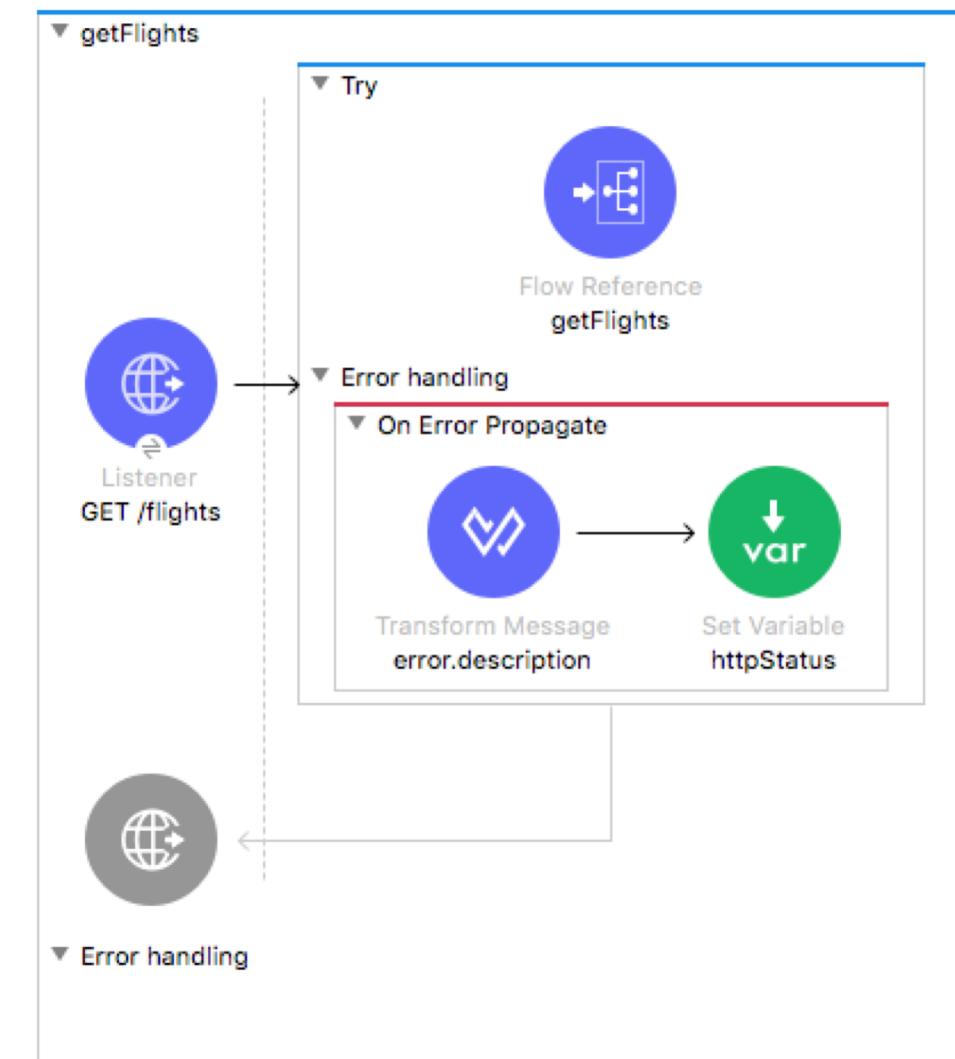
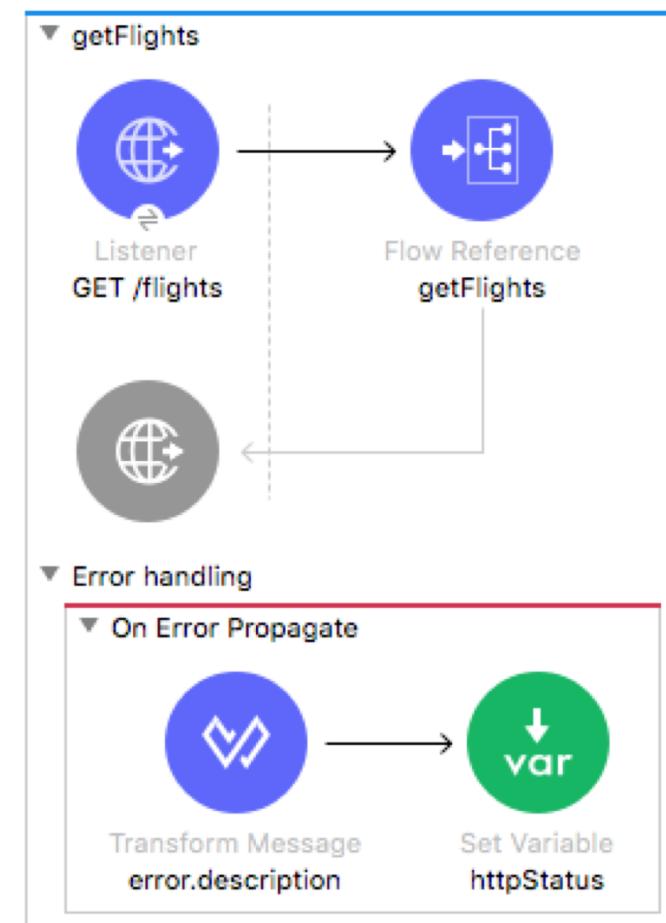
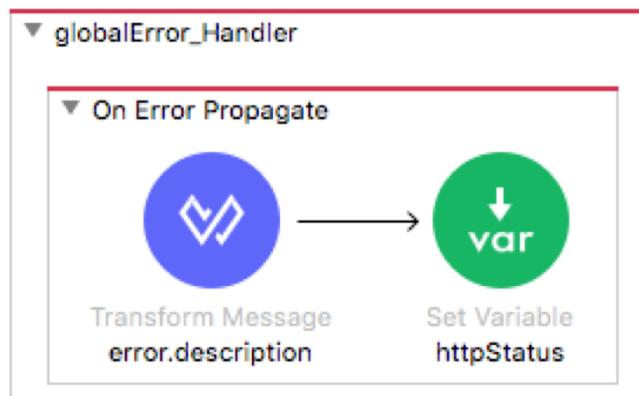
# Creating error handlers



# Creating error handlers



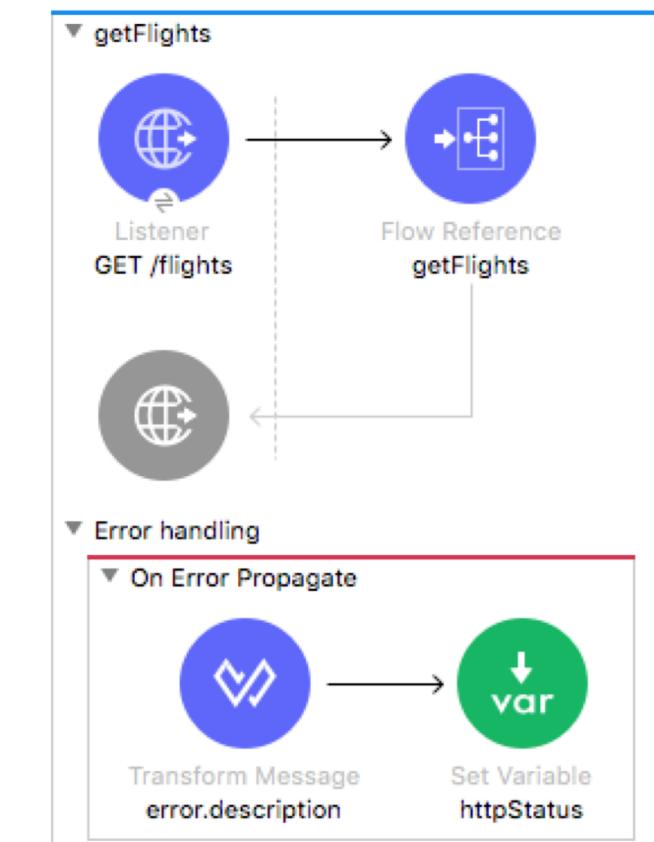
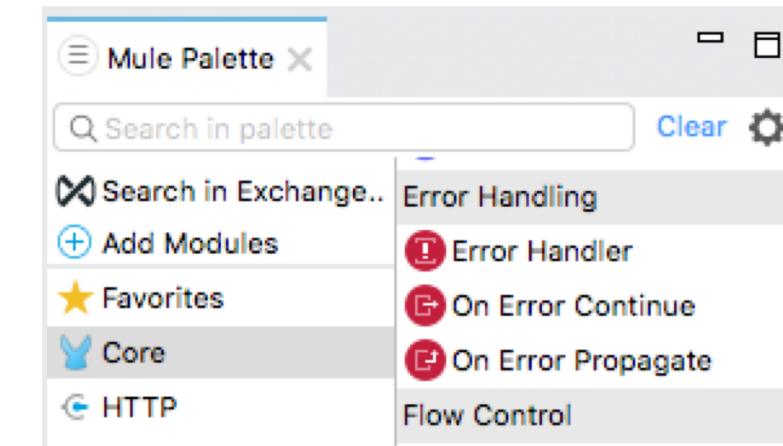
- Error handlers can be added to
  - An application (outside of any flows)
  - A flow
  - A selection of one or more event processors



# 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



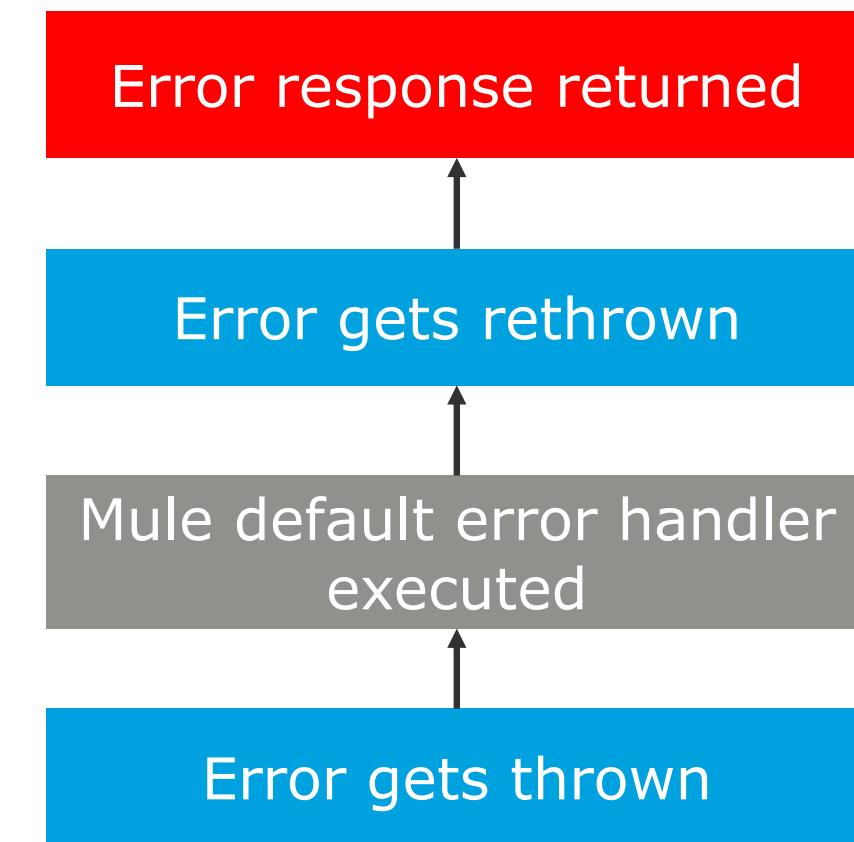
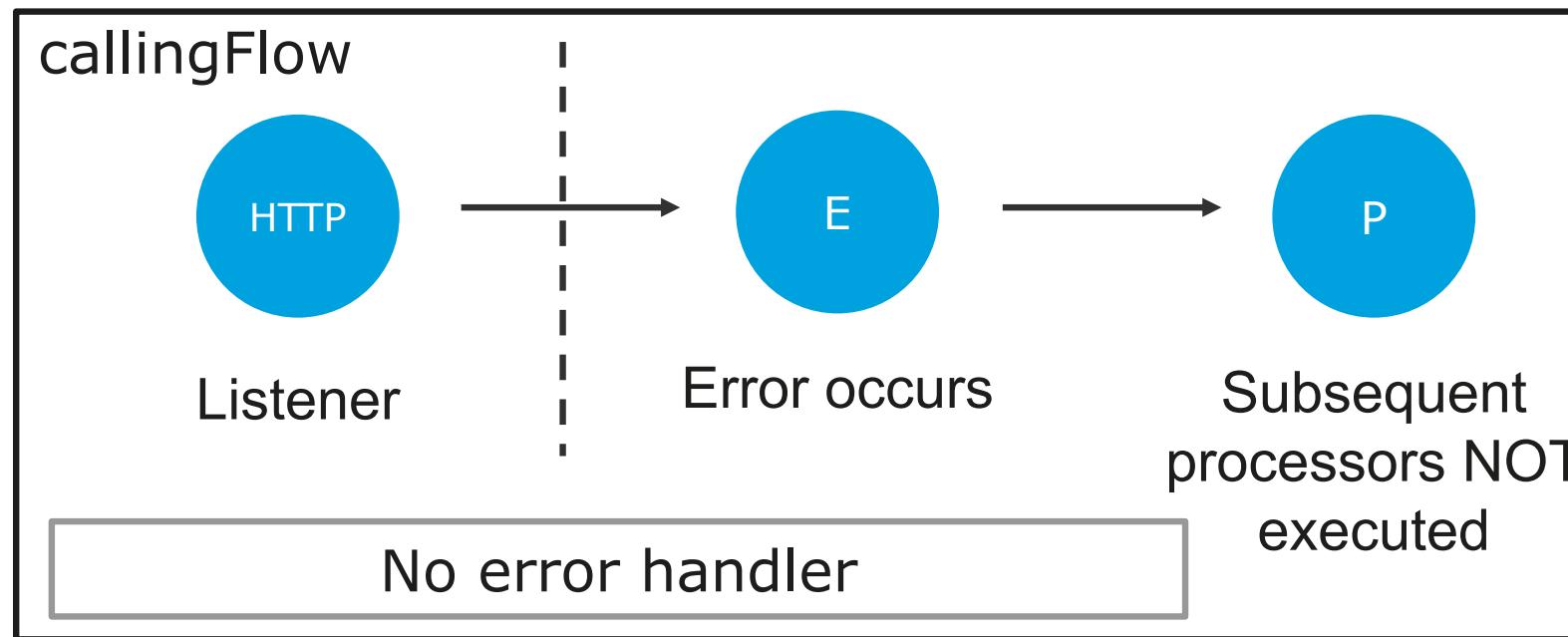
- **On Error Propagate**

- 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

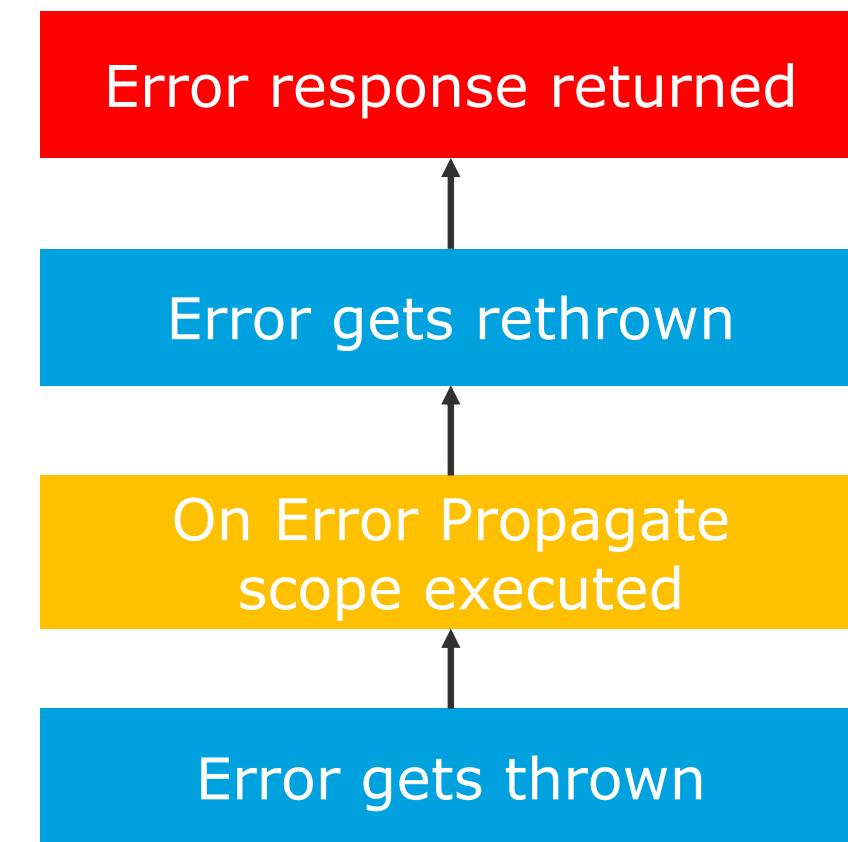
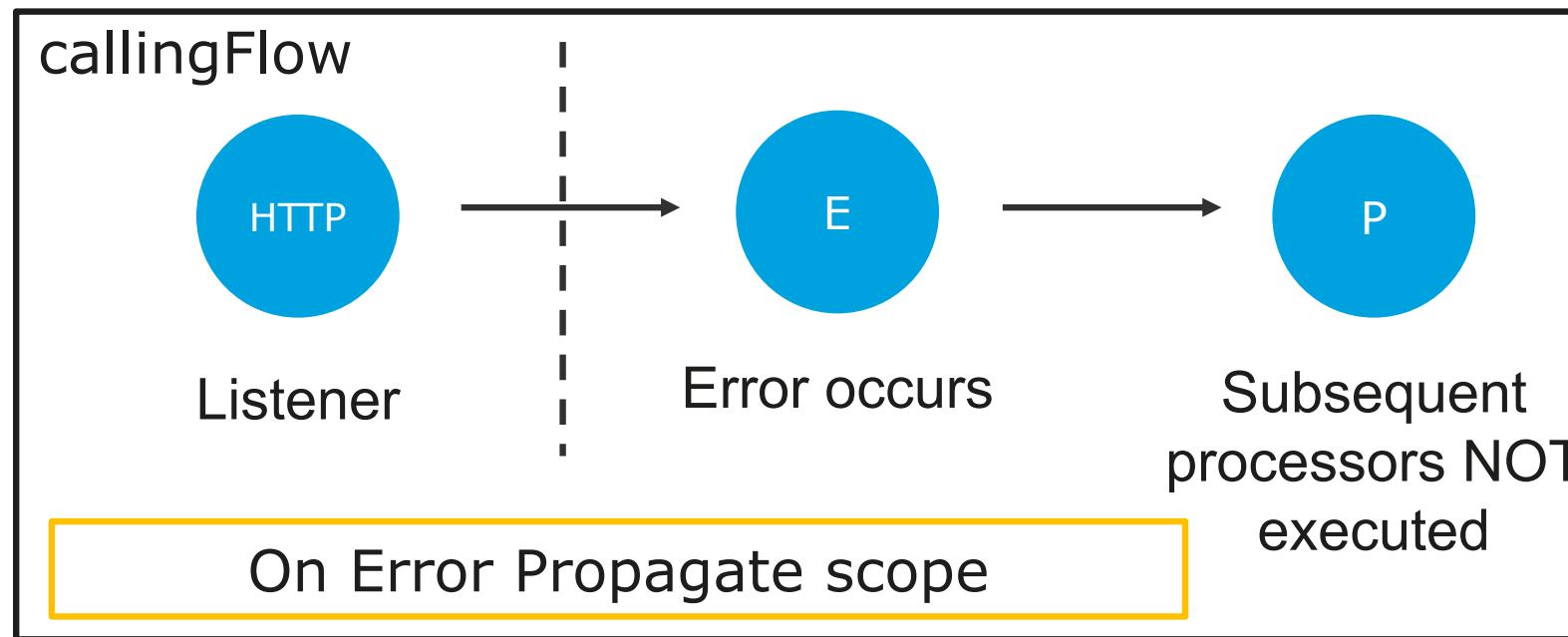
- **On Error Continue**

- 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

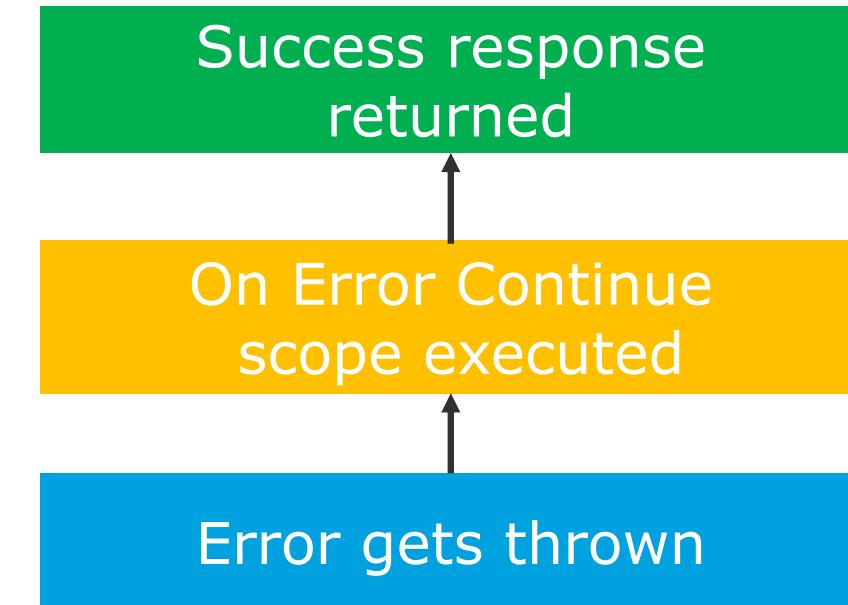
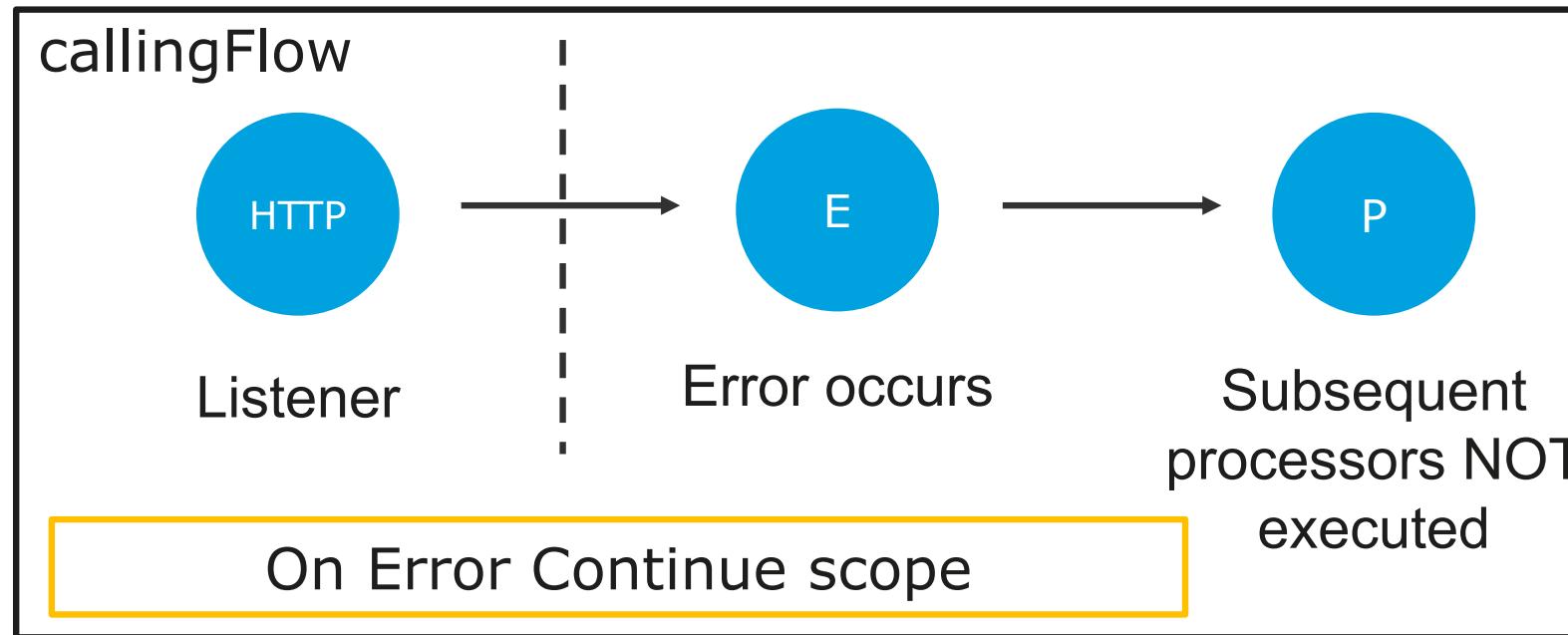
# Error handling scenario 1



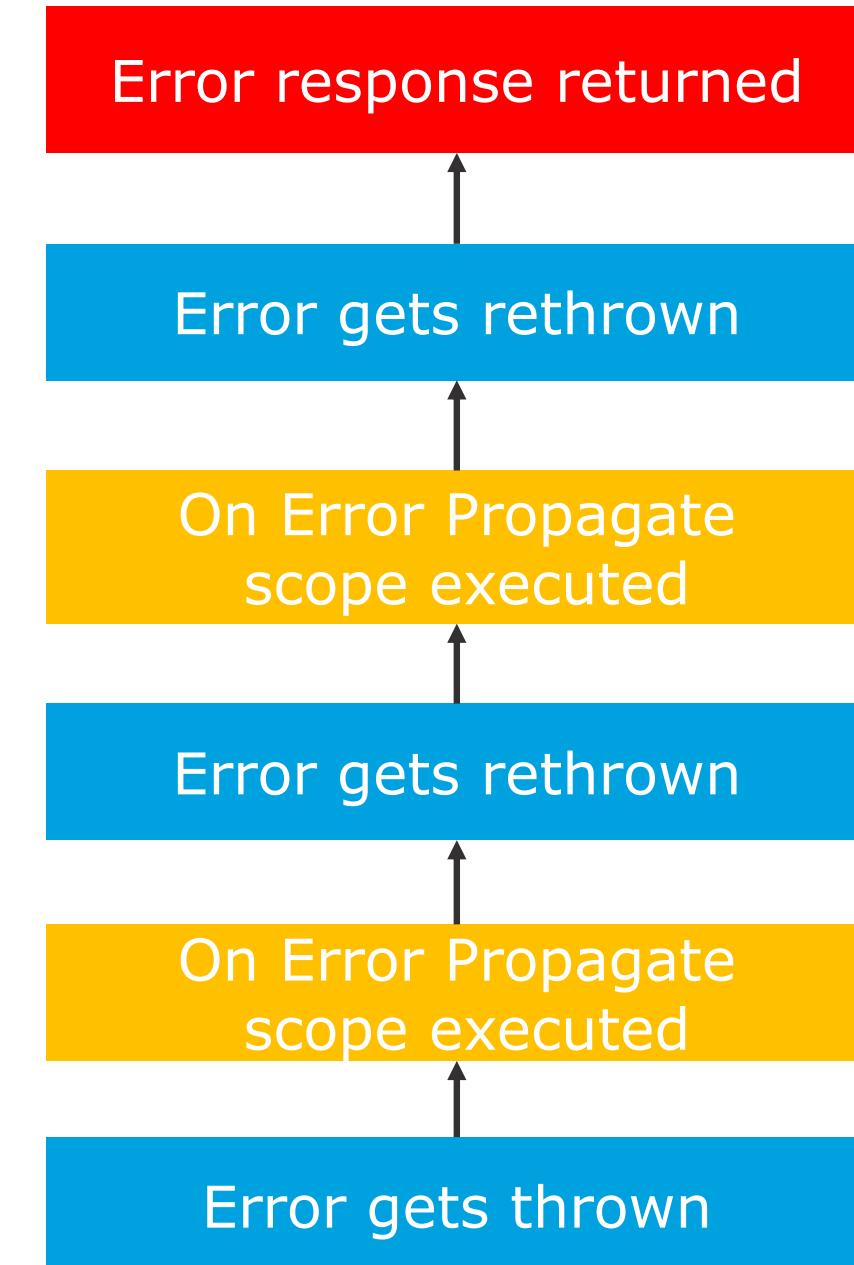
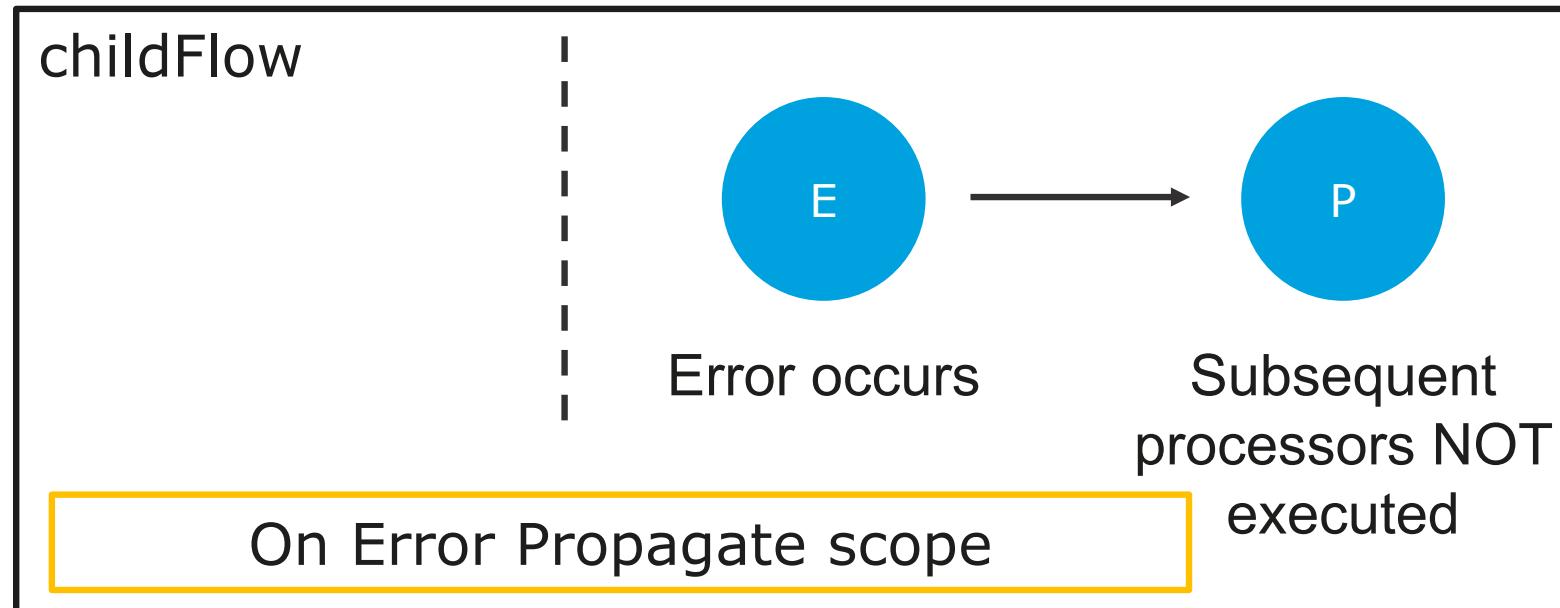
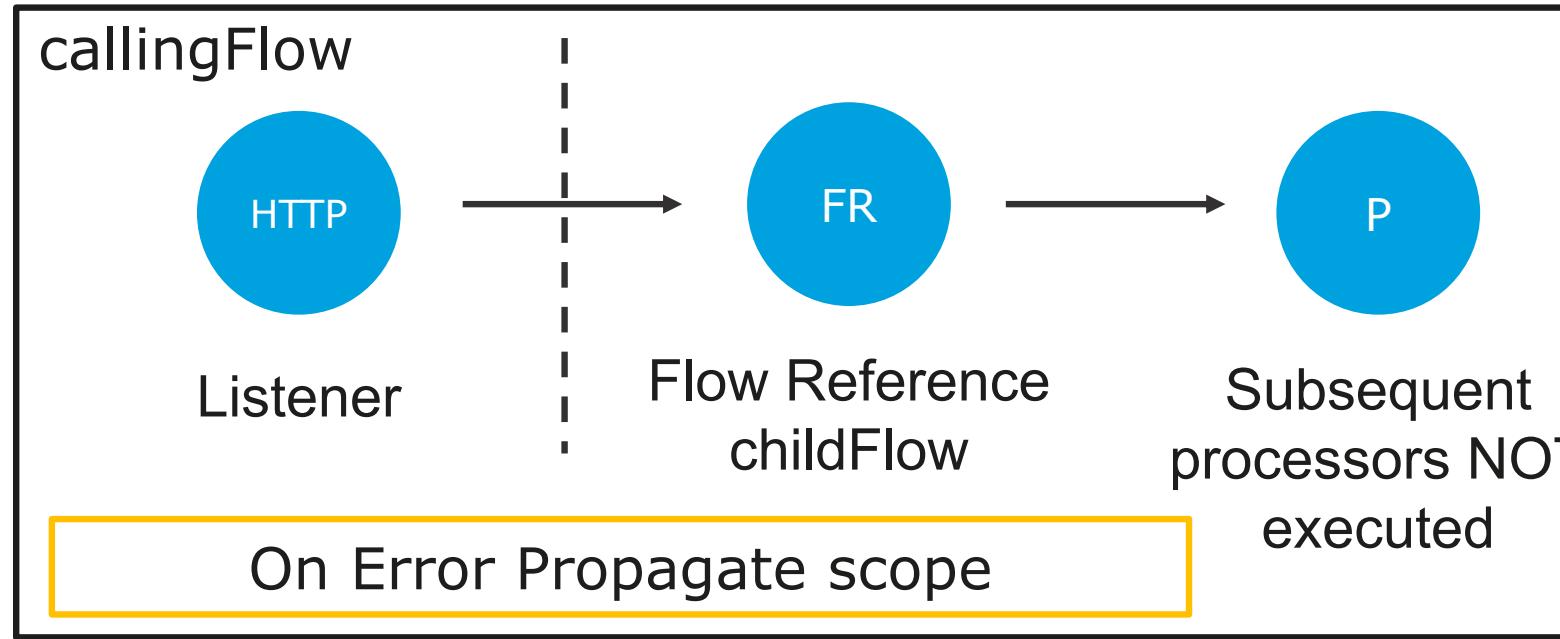
# Error handling scenario 2



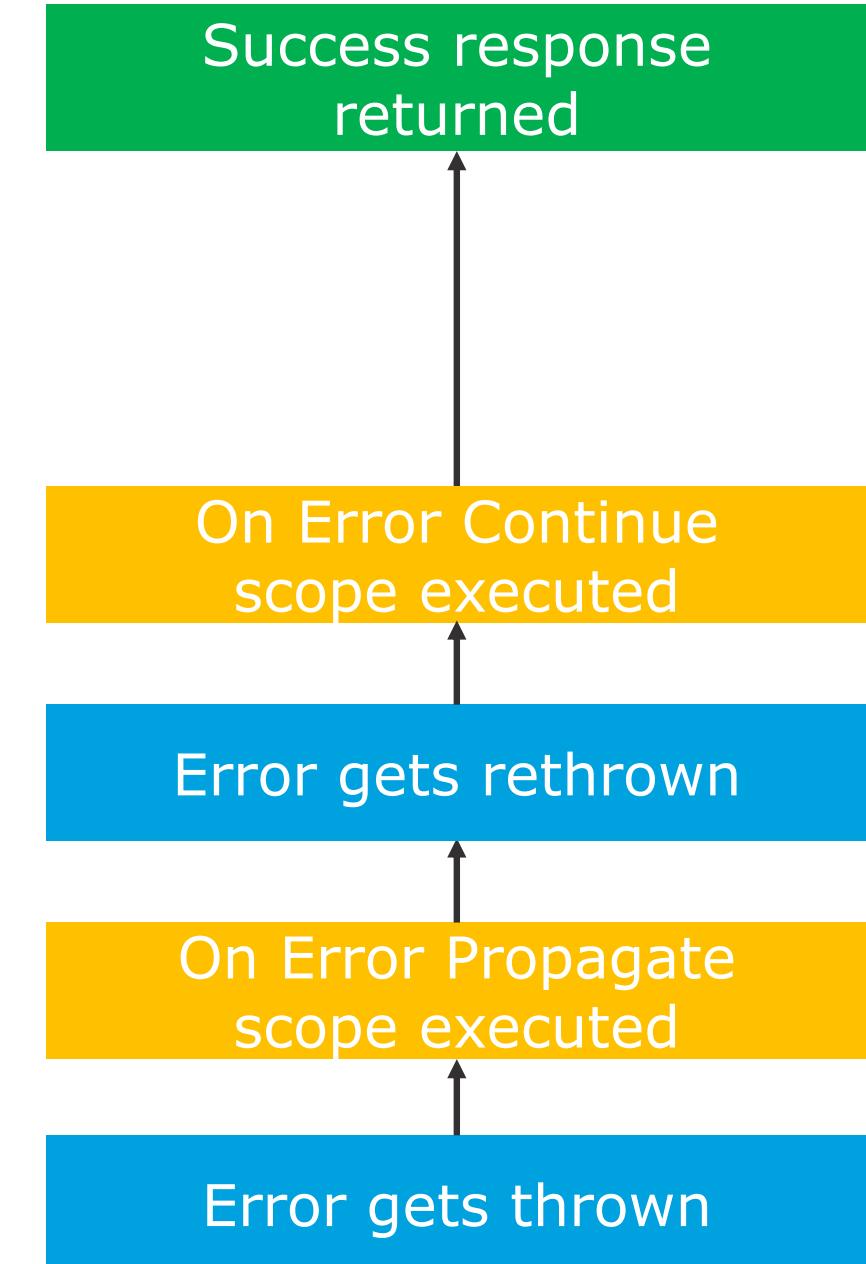
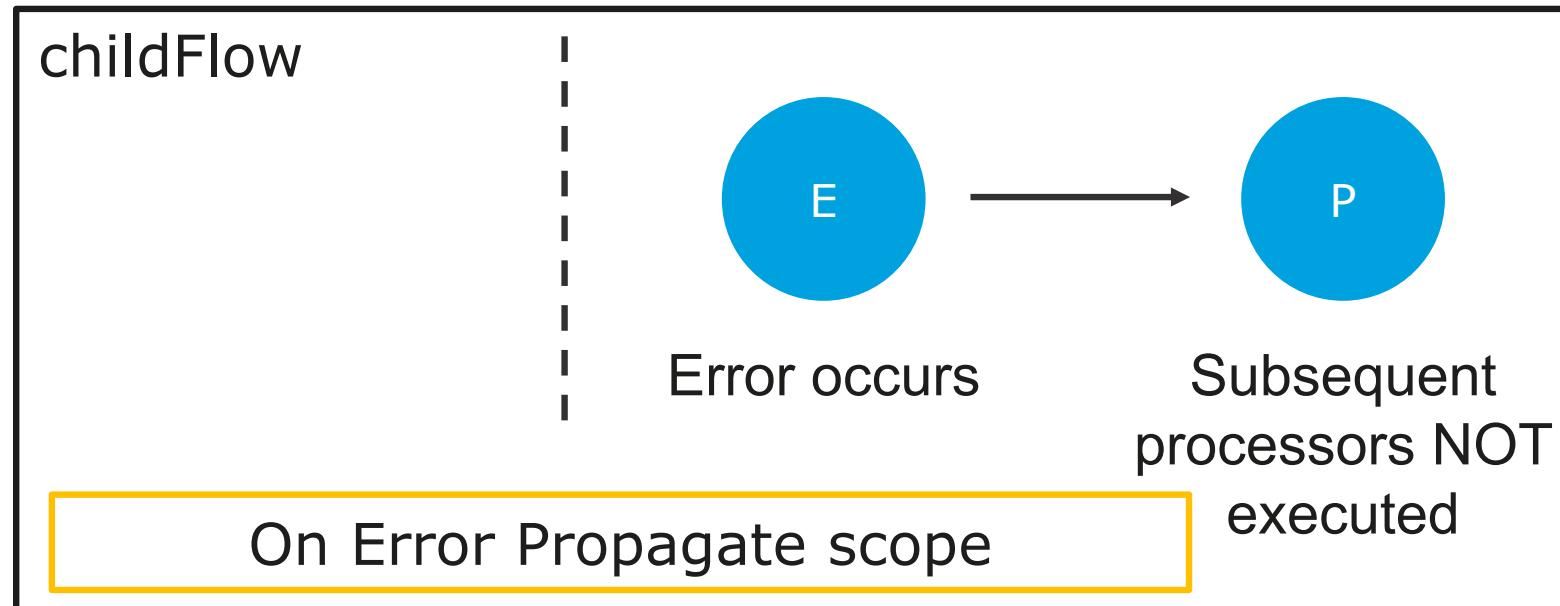
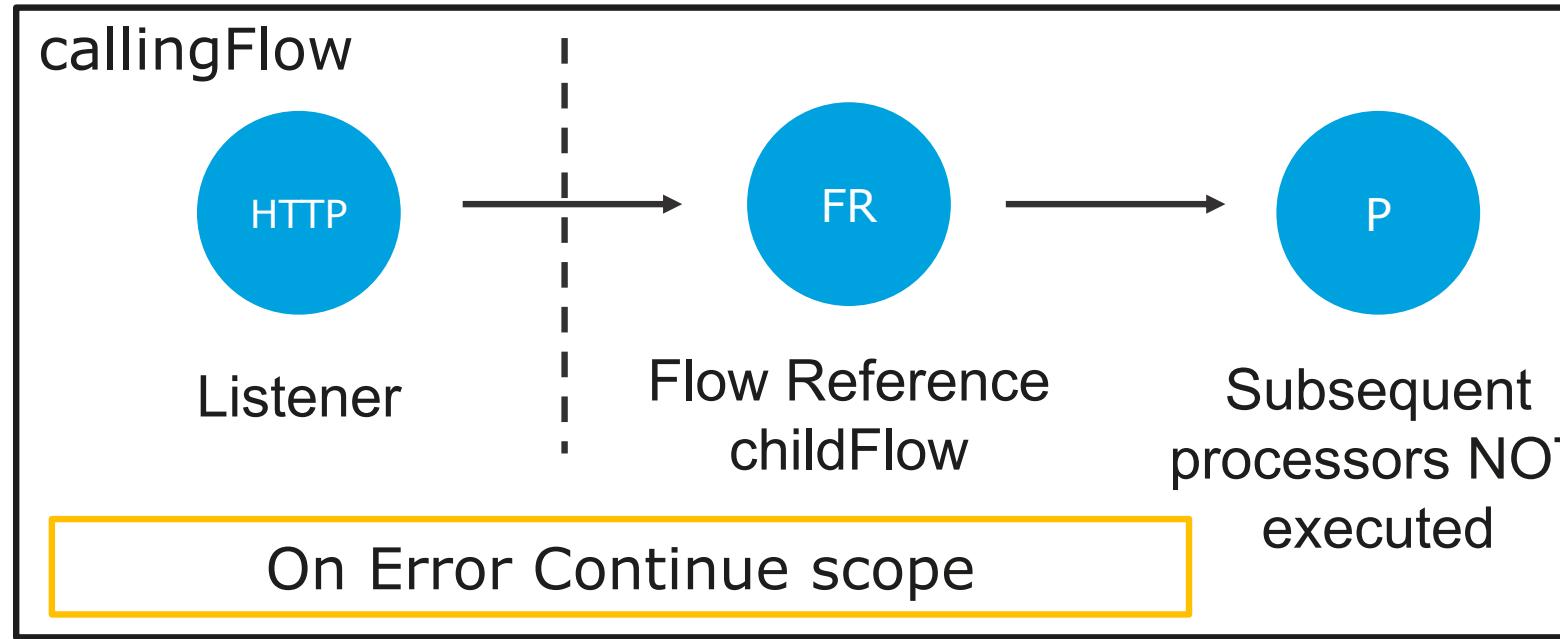
# Error handling scenario 3



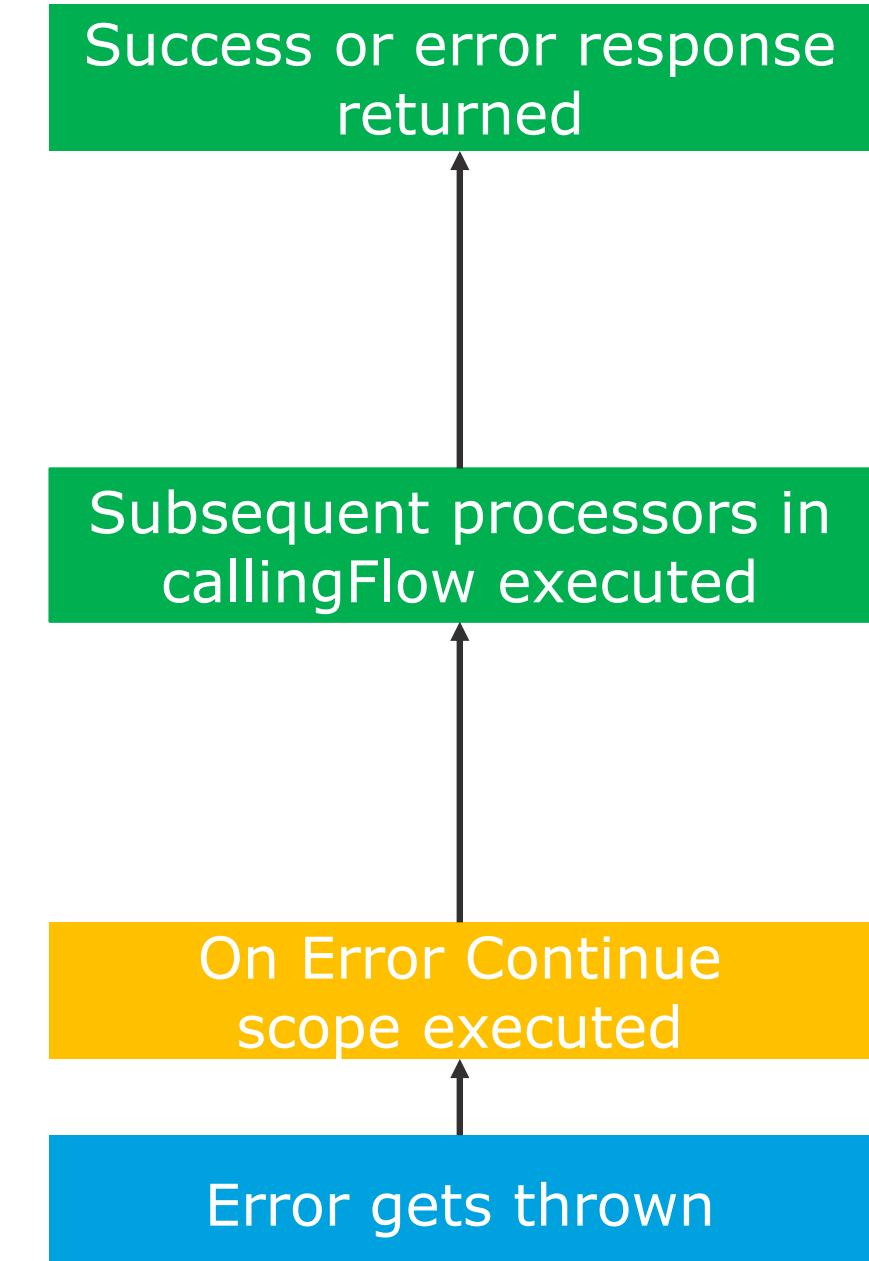
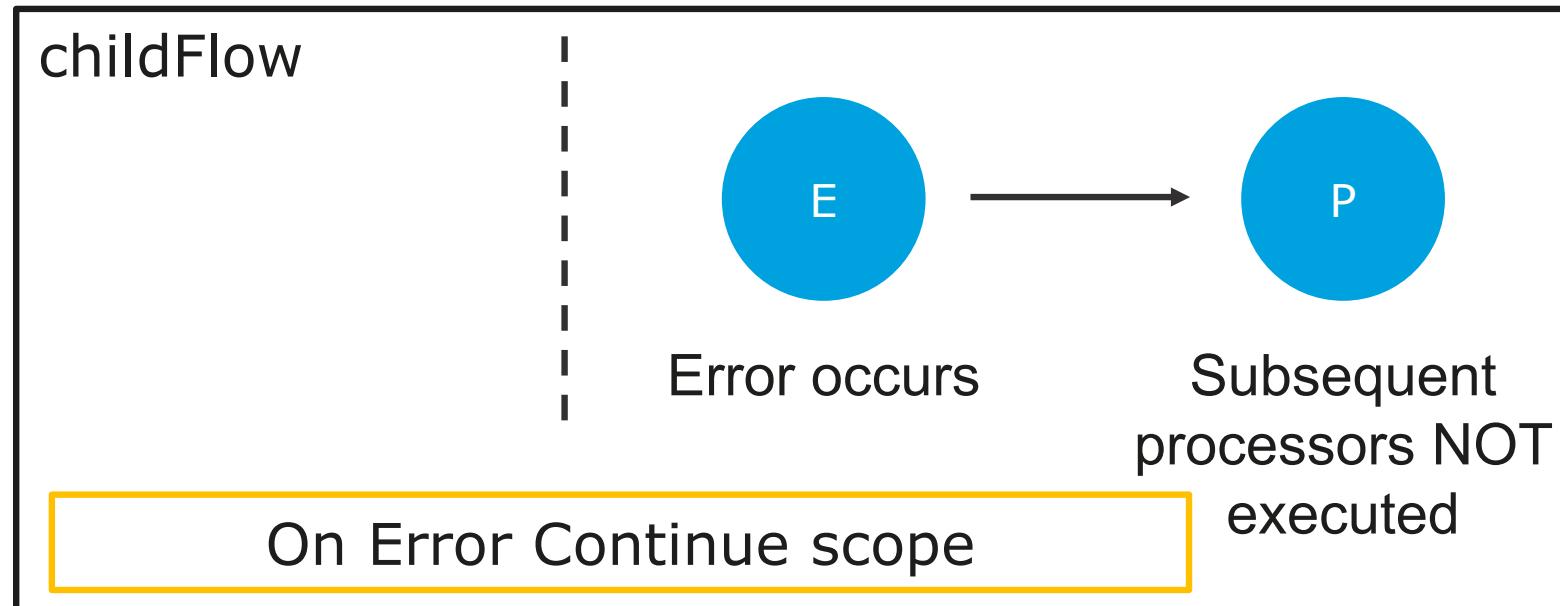
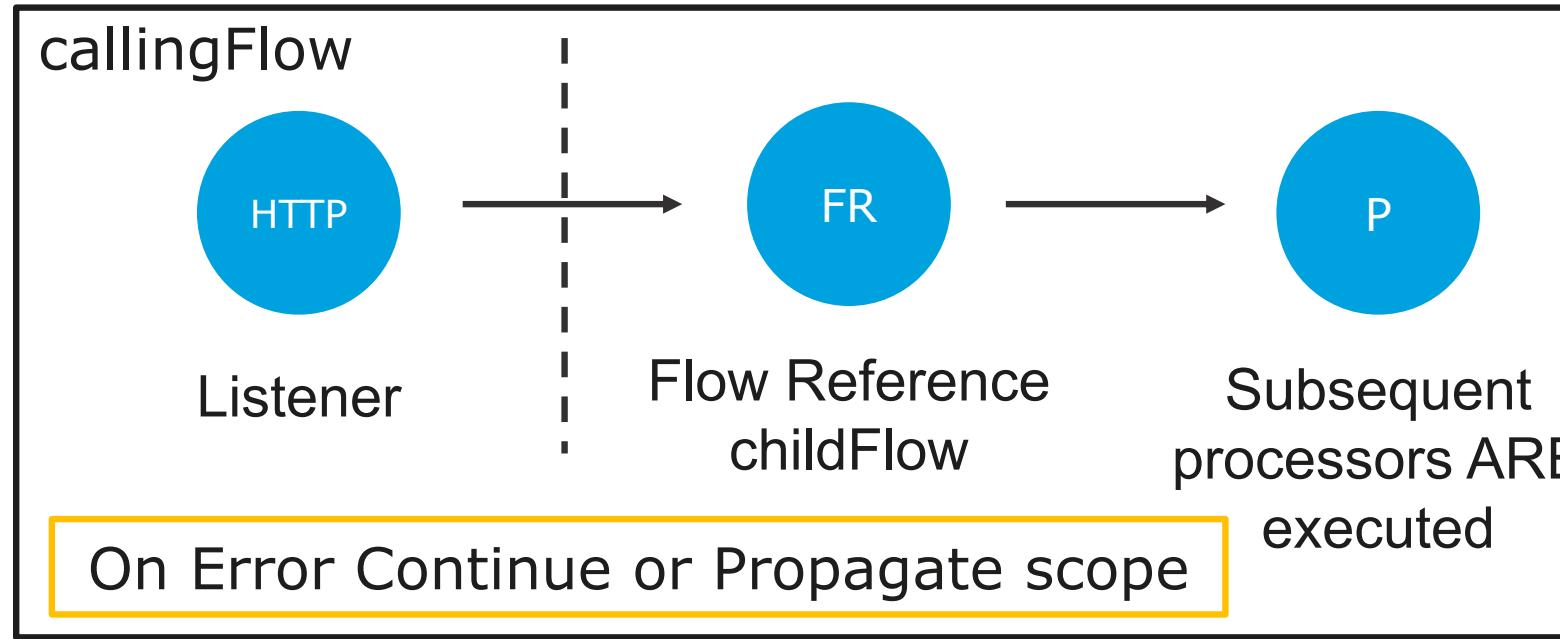
# Error handling scenario 4



# Error handling scenario 5



# Error handling scenario 6



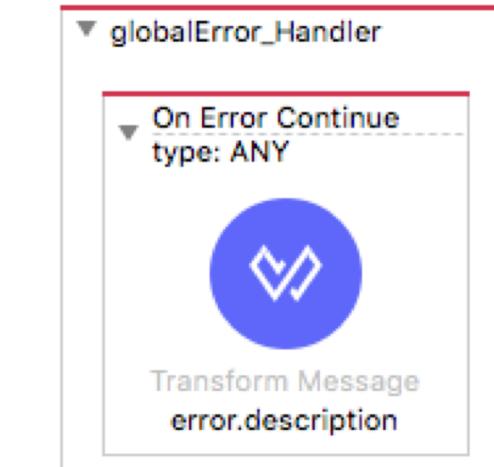
# Handling errors at the application level



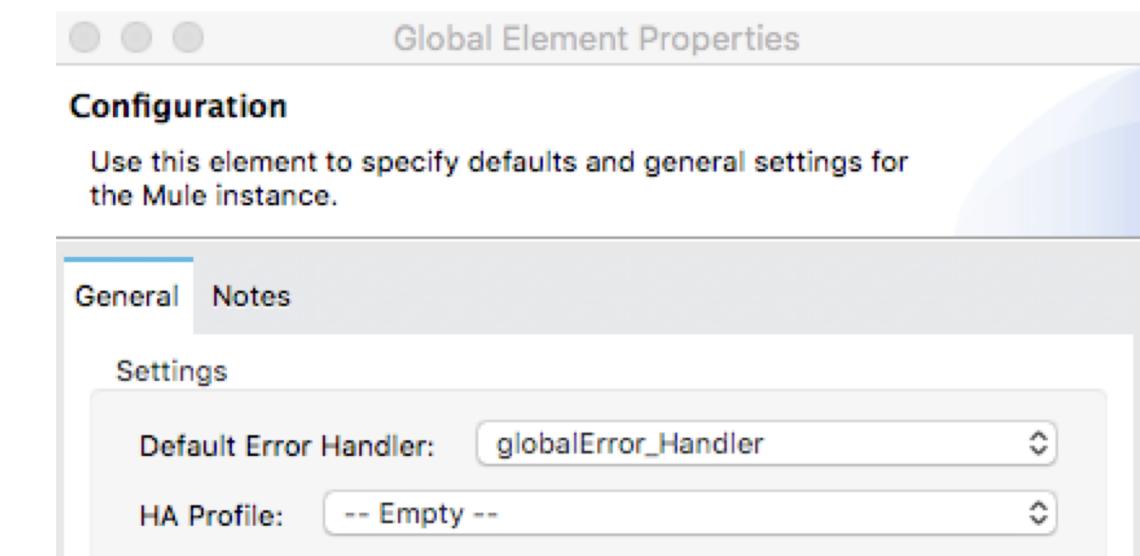
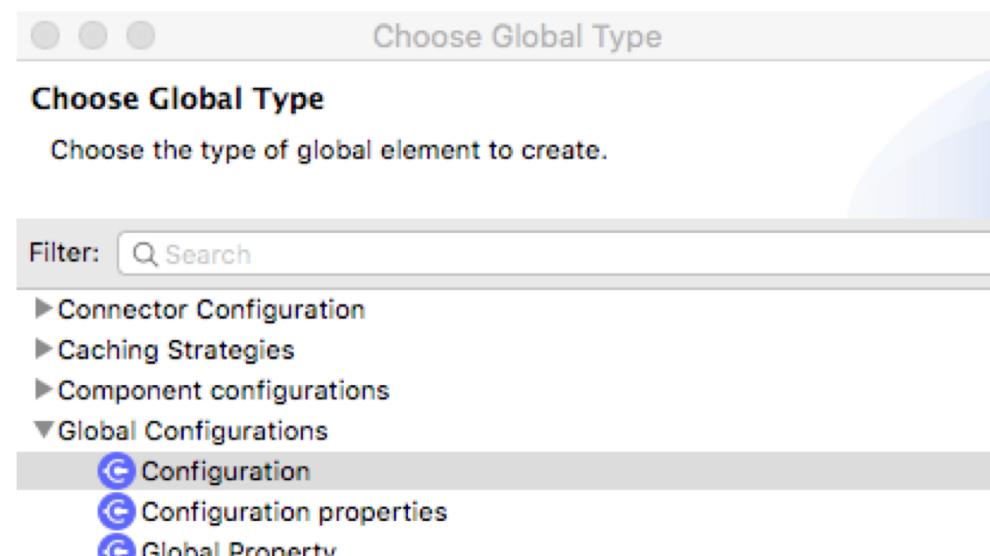
# Defining a default error handler for an application



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

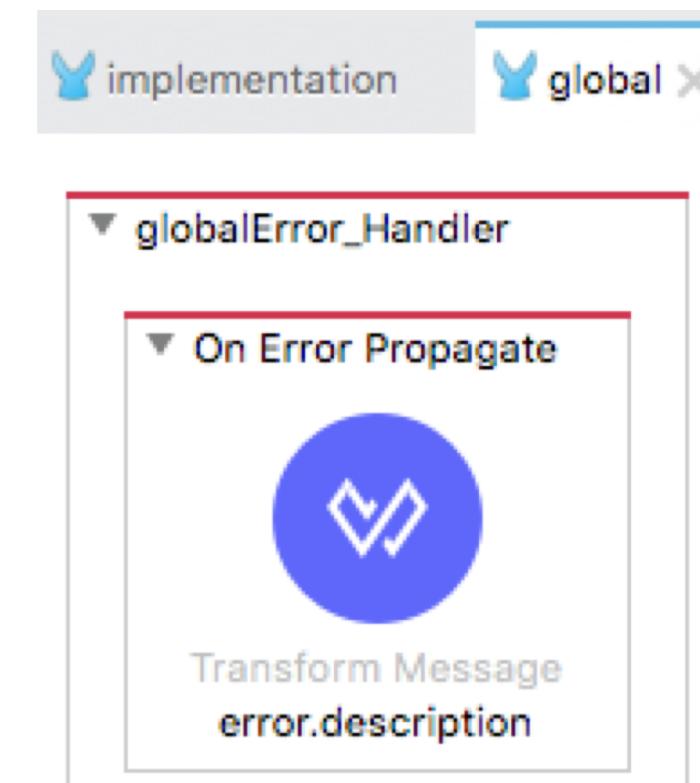


- Specify this handler to be the application's default error handler



# 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



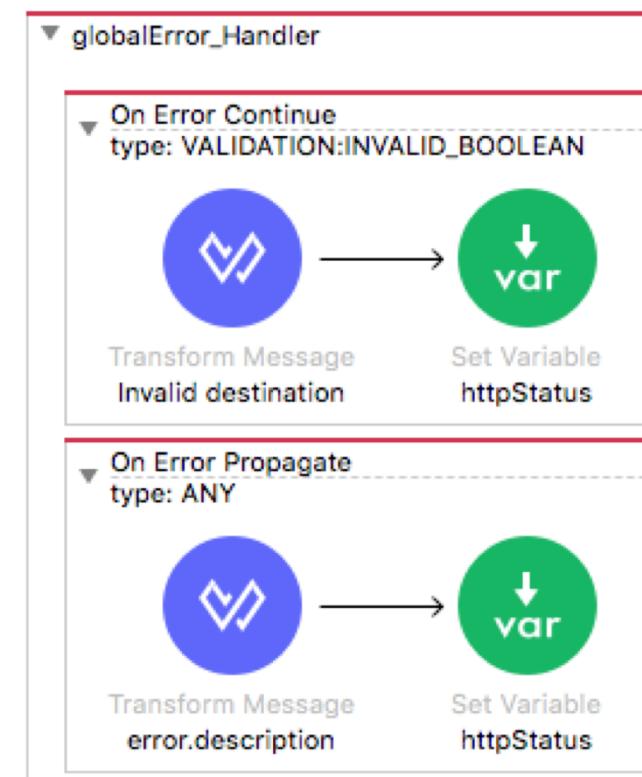
# Handling specific types of errors



# 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

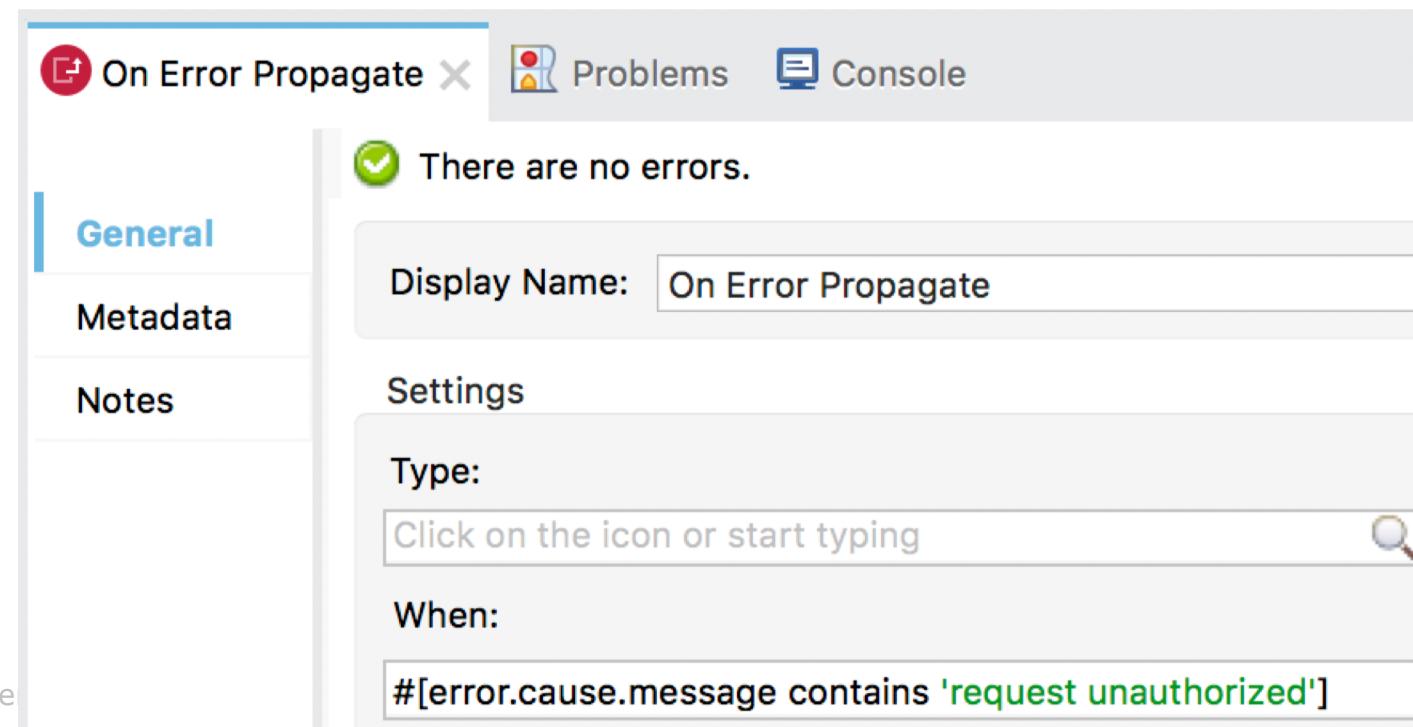
The screenshot shows the MuleSoft Anypoint Studio interface. On the left, the 'On Error Propagate' configuration screen is displayed. It has tabs for General, Metadata, and Notes. Under General, the 'Display Name' is set to 'On Error Propagate'. In the 'Type' section, 'WSC:CONNECTIVITY, WSC:INVALID\_WSDL' is selected. Below this, under 'Select the error types', several options are listed, with 'WSC:CONNECTIVITY' and 'WSC:INVALID\_WSDL' checked. On the right, the execution flow is shown in two sections: 'globalError\_Handler' and 'On Error Propagate'.  
  
In the 'globalError\_Handler' section:

- 'On Error Continue' type: VALIDATION:INVALID\_BOOLEAN
- Flow: Transform Message (Invalid destination) → Set Variable (httpStatus)

  
In the 'On Error Propagate' section:

- 'On Error Propagate' type: ANY
- Flow: Transform Message (error.description) → Set Variable (httpStatus)

- 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

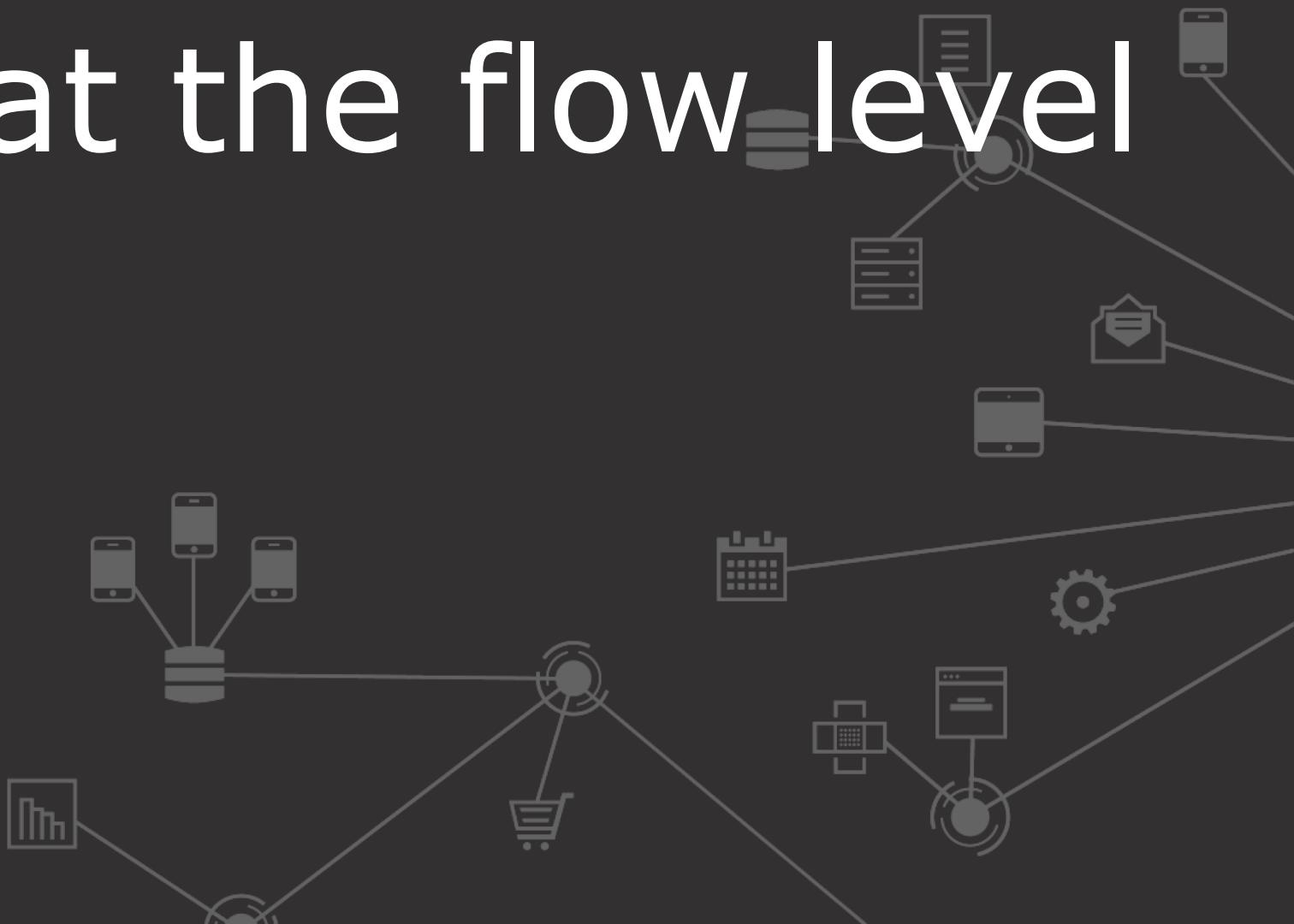


- Review the possible types of errors thrown by different processors
- Create error handler scopes to handle different error types

The screenshot shows the Mule Studio interface with two main panels:

- Error Mapping:** A dialog box titled "Check the error types to map:" lists various error types. Some are checked (e.g., AMERICAN-FLIGHTS-API:TOO\_MANY, WSC:BAD\_REQUEST) while others are not. Below this is a section for "Mapping to custom error:" with fields for "Namespace" (set to "APP") and "Identifier".
- globalError\_Handler:** A configuration panel for a global error handler. It contains two "On Error Propagate" sections:
  - The first section handles errors of type "WSC:CONNECTIVITY, WSC:INVALID\_WSDL" and contains a "Transform Message" step with the description "Data unavailable".
  - The second section handles errors of type "ANY" and contains a "Transform Message" step with the description "error.description".

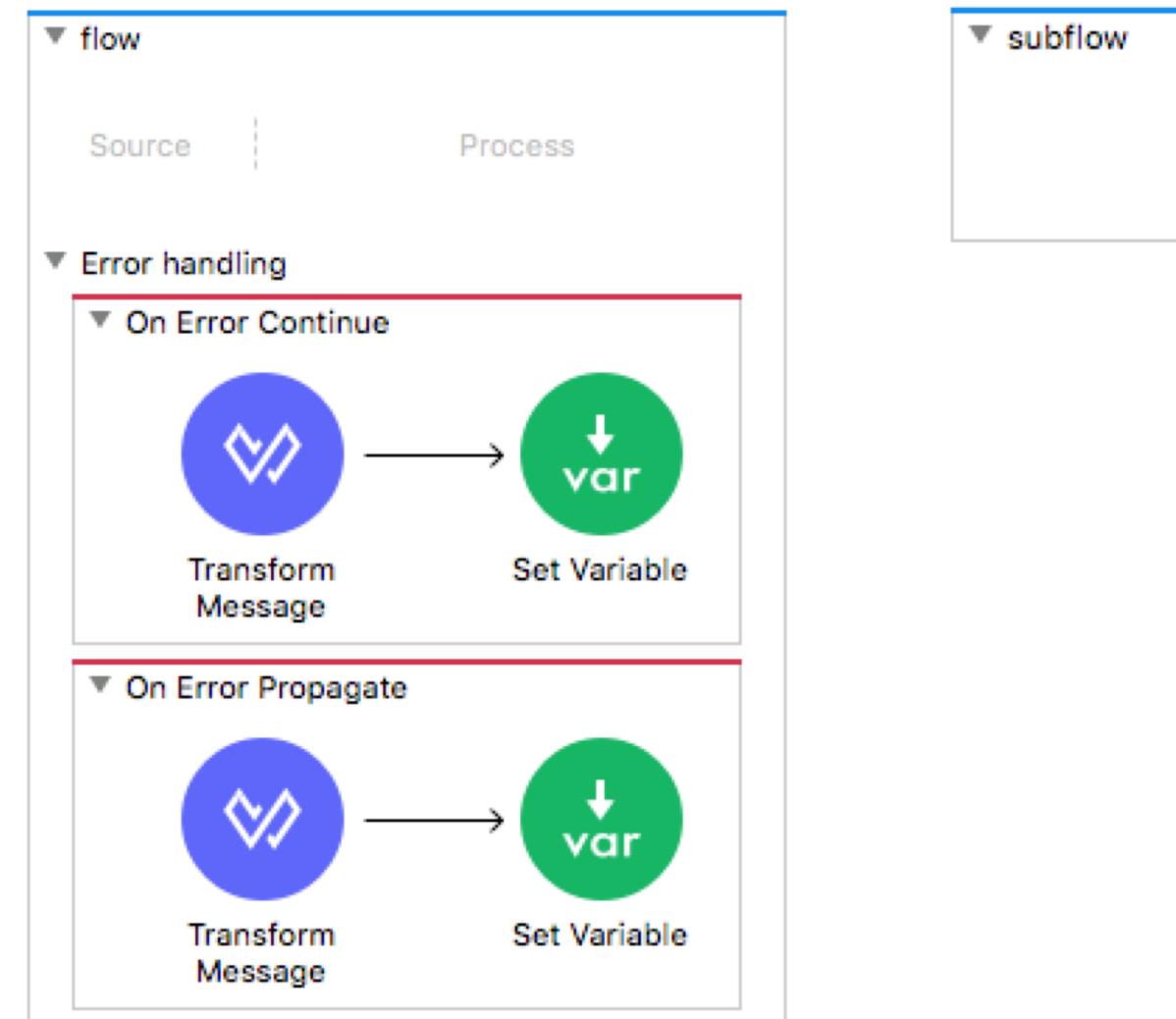
# Handling errors at the flow level



# 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

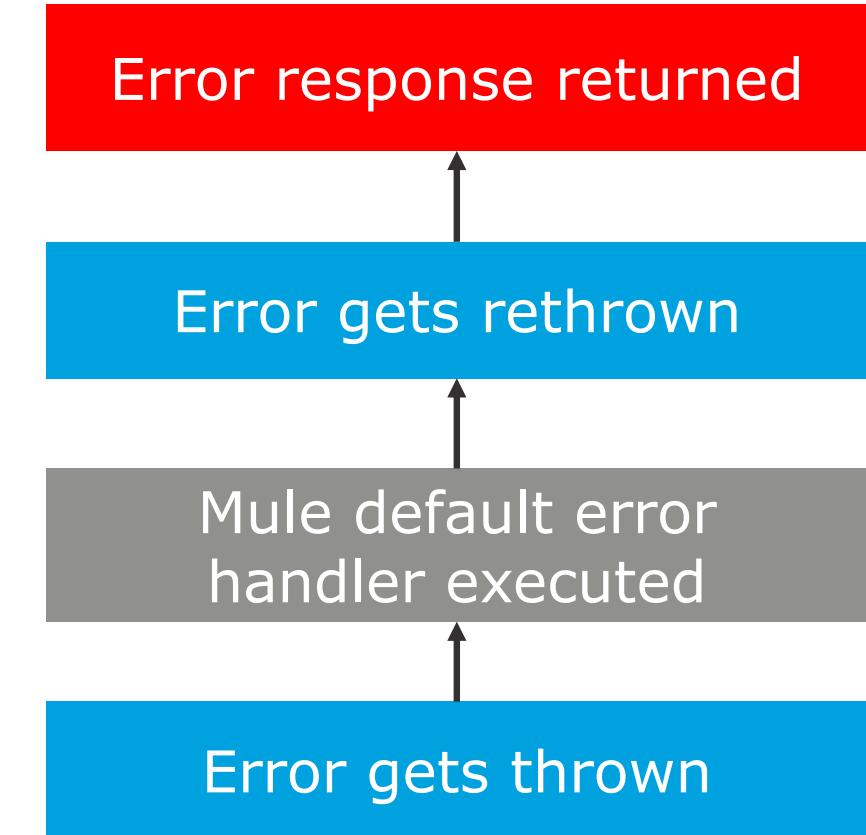
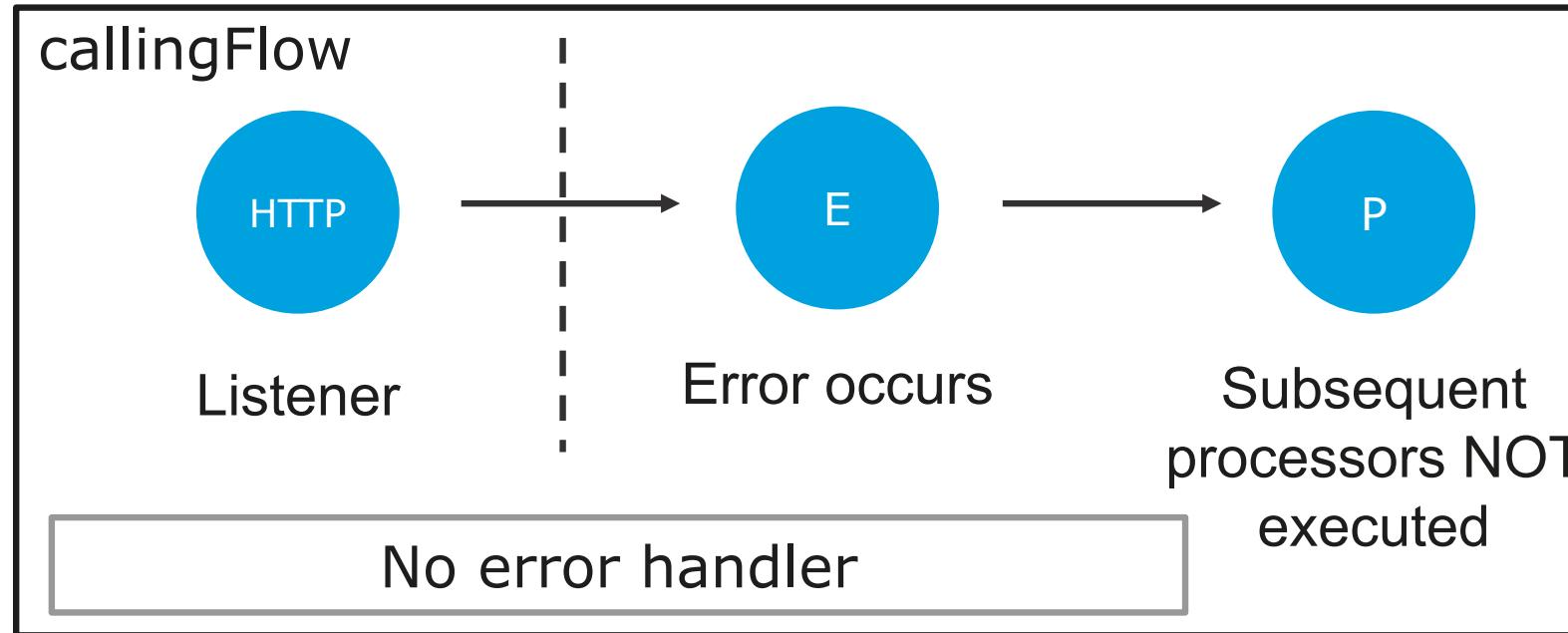


# Which error scope handles an error?



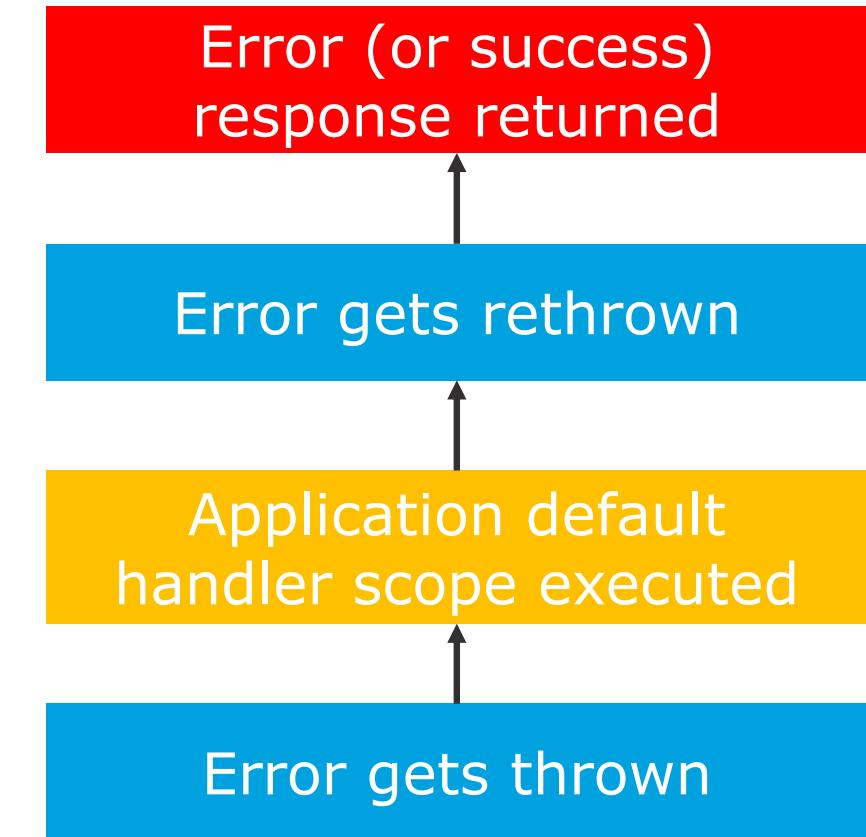
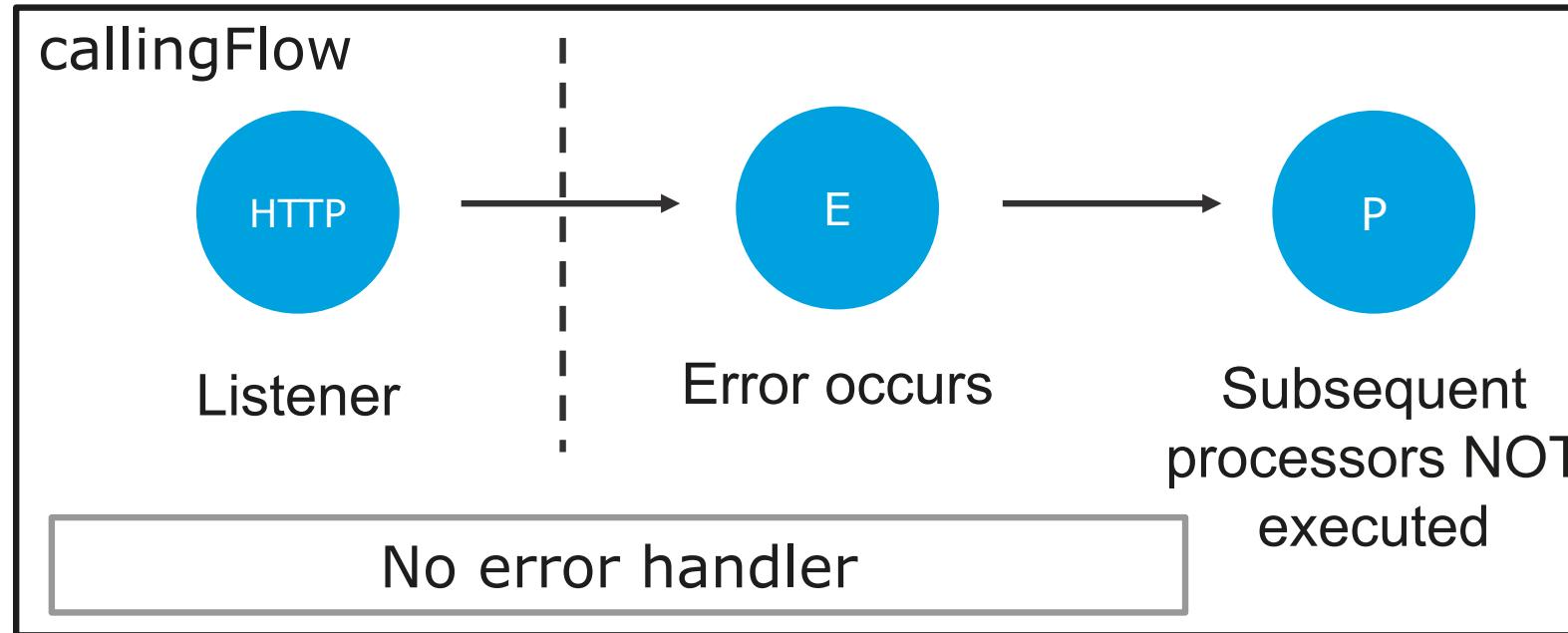
- If a flow *has* an error handler
  - The error is handled by the *first* error scope whose condition evaluates to true
  - **If no scope conditions are true**, the error is handled by the **Mule default error handler** **NOT** any scope in an **application's global default handler**
    - The Mule 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



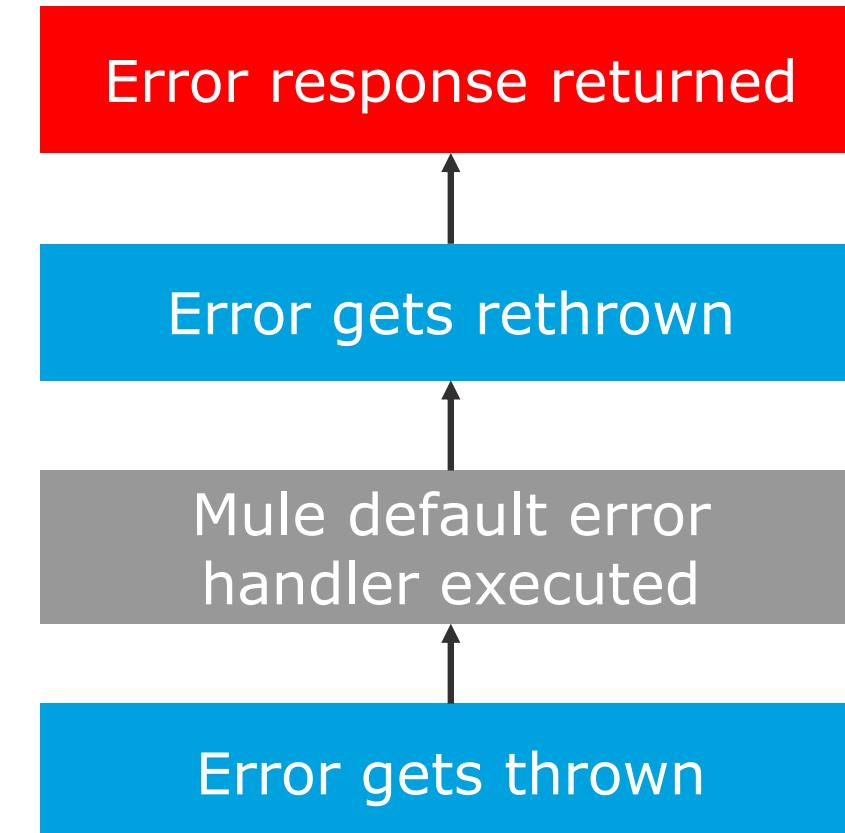
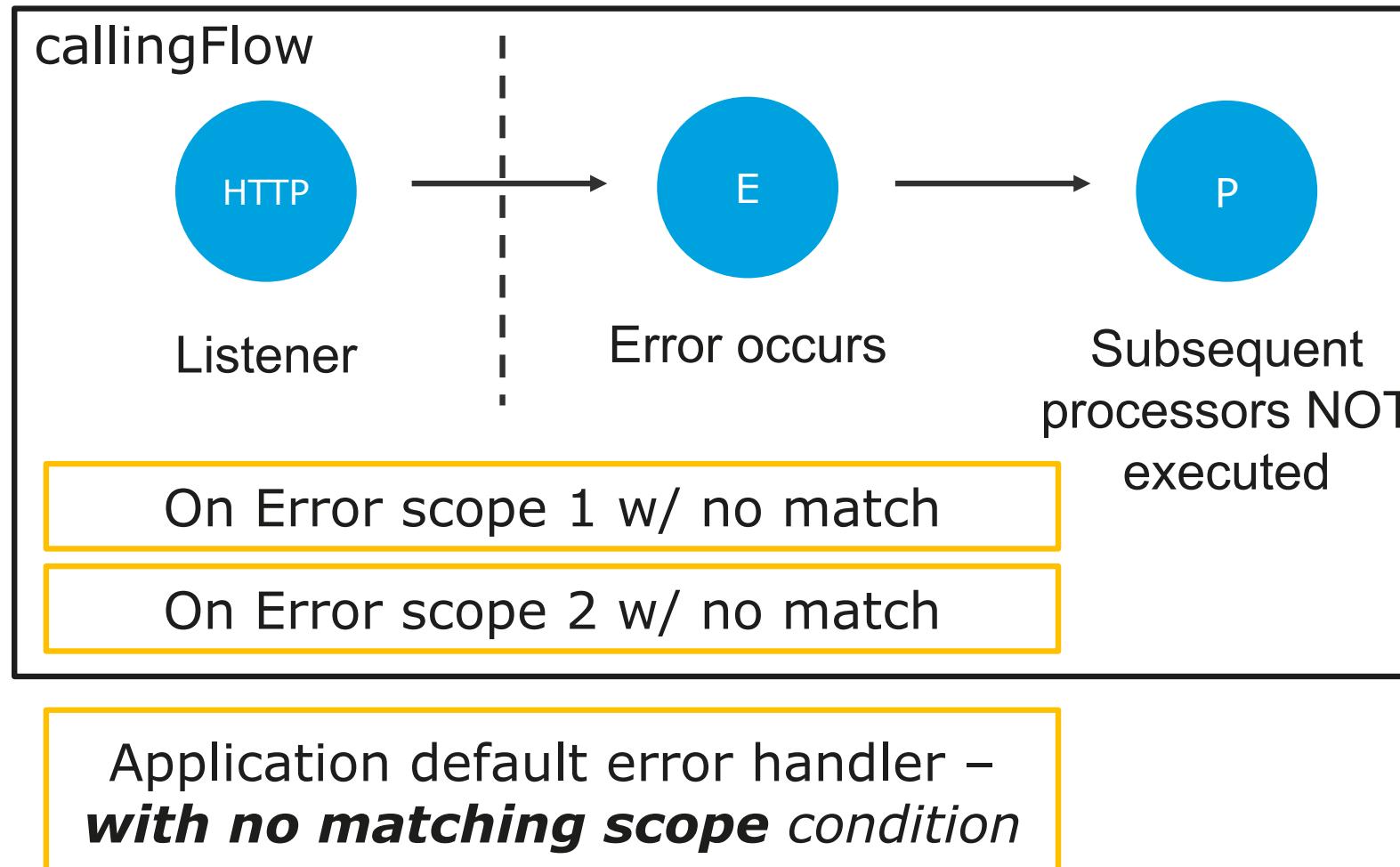
Application default error handler –  
***with no matching scope condition***

# Error handling scenario 2

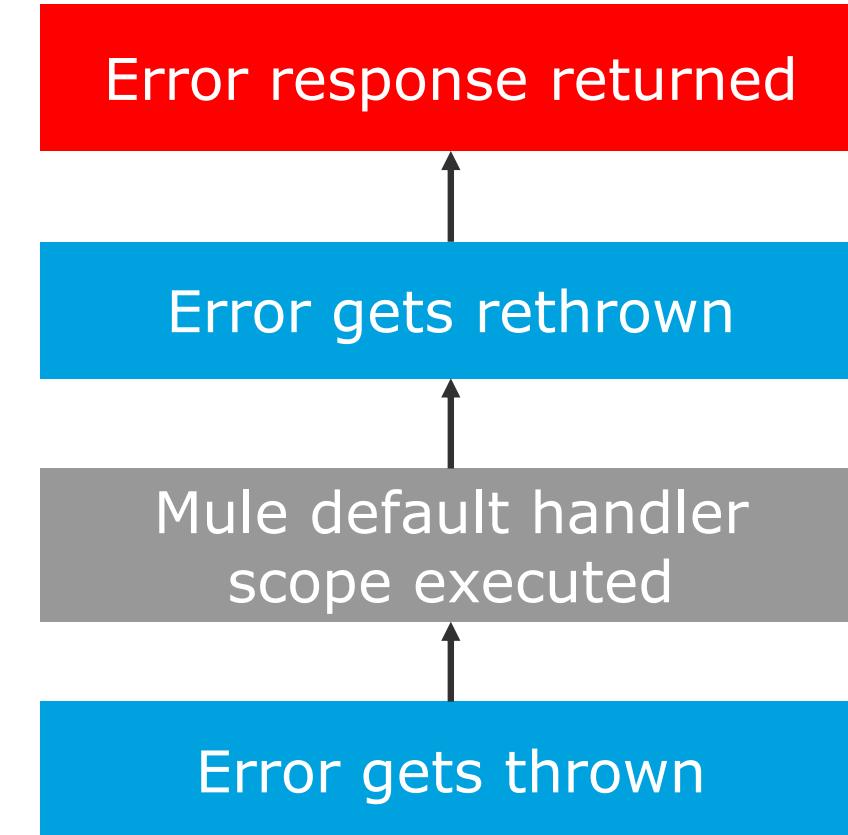
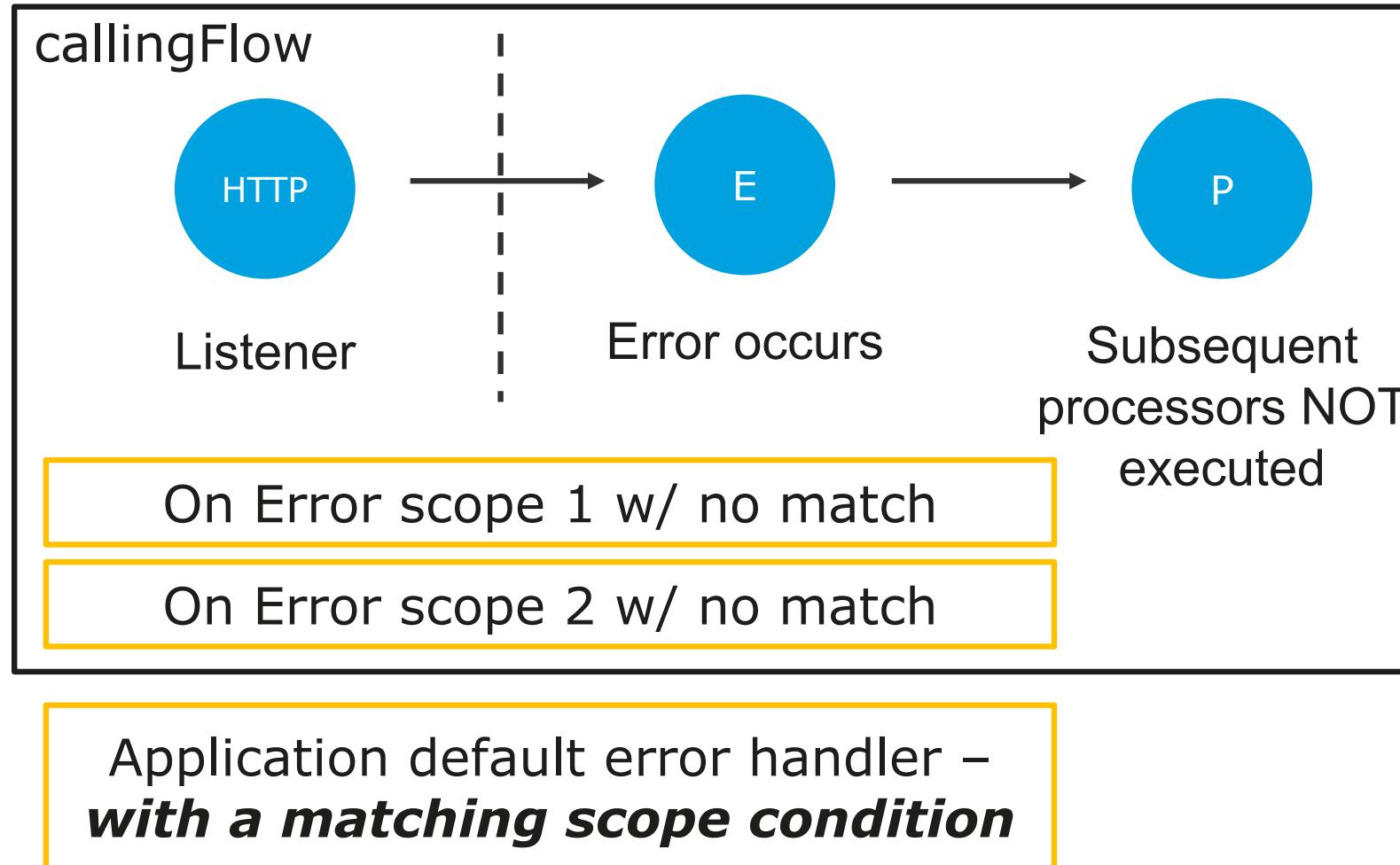


Application default error handler –  
***with a matching scope condition***

# Error handling scenario 3



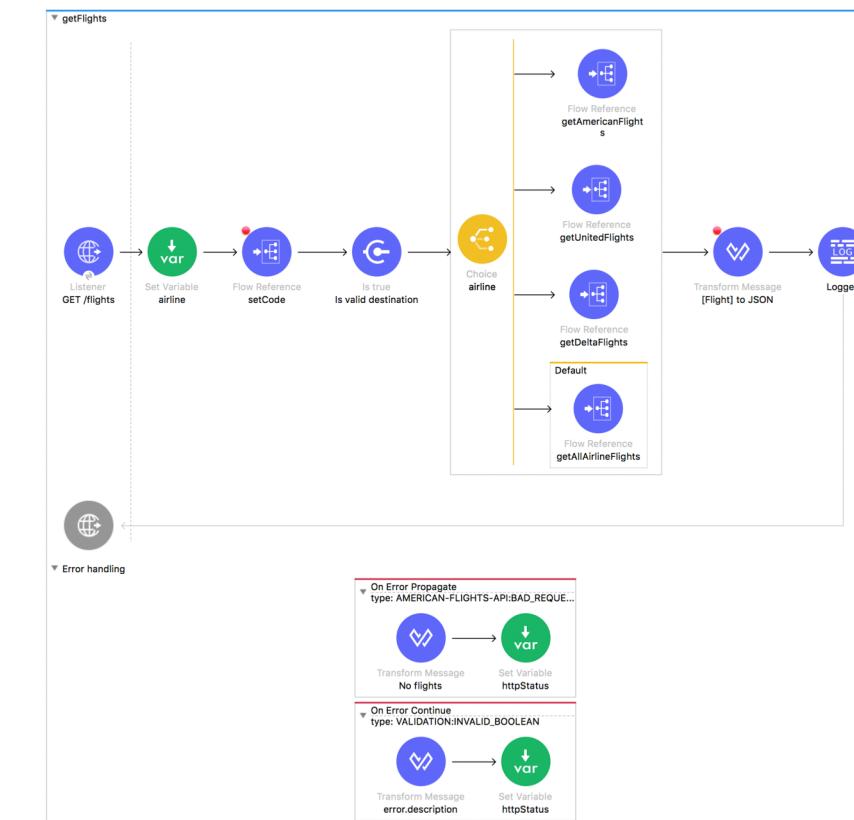
# Error handling scenario 4



# 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



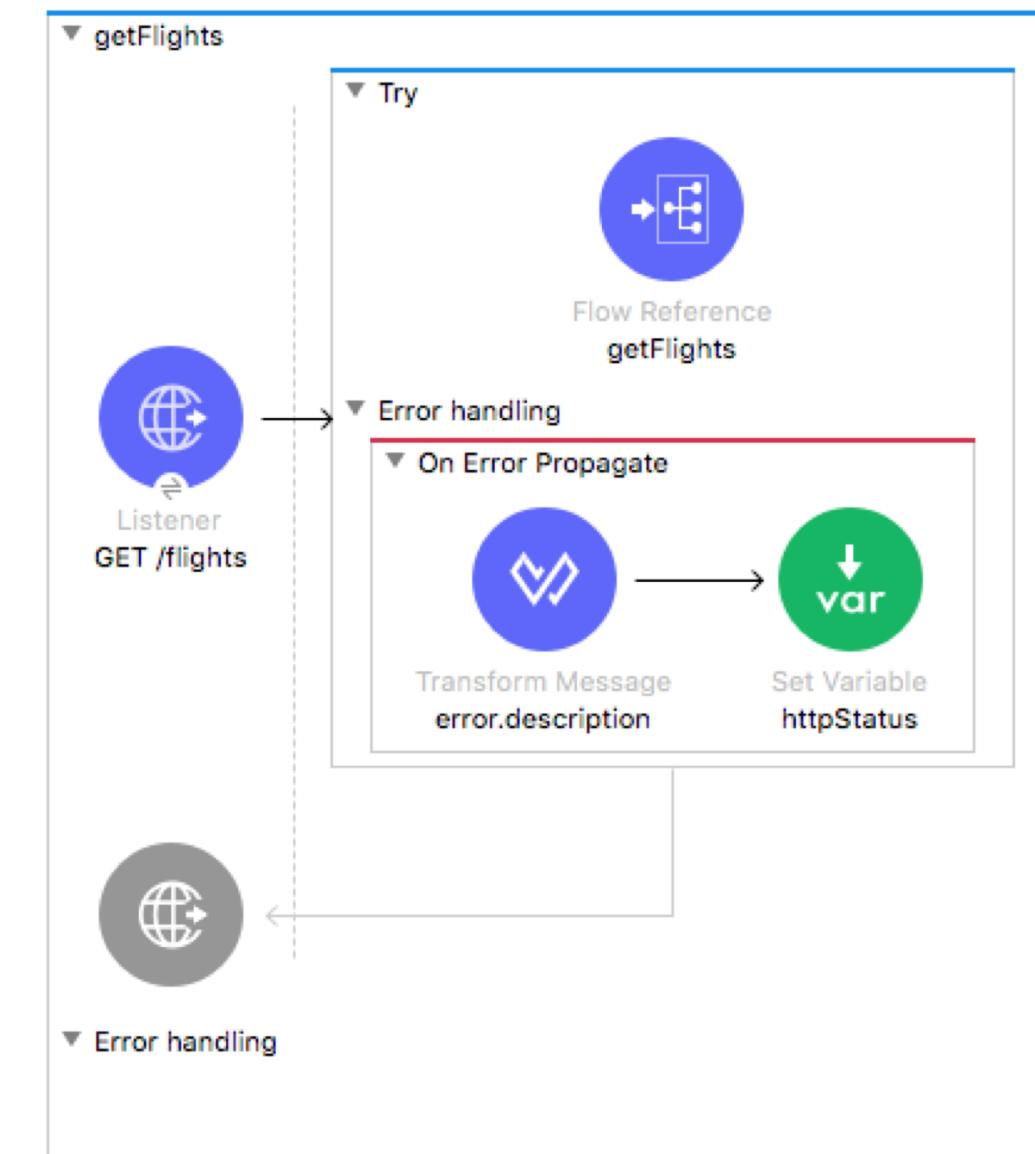
# Handling errors at the processor level



# 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 more error scopes can be added



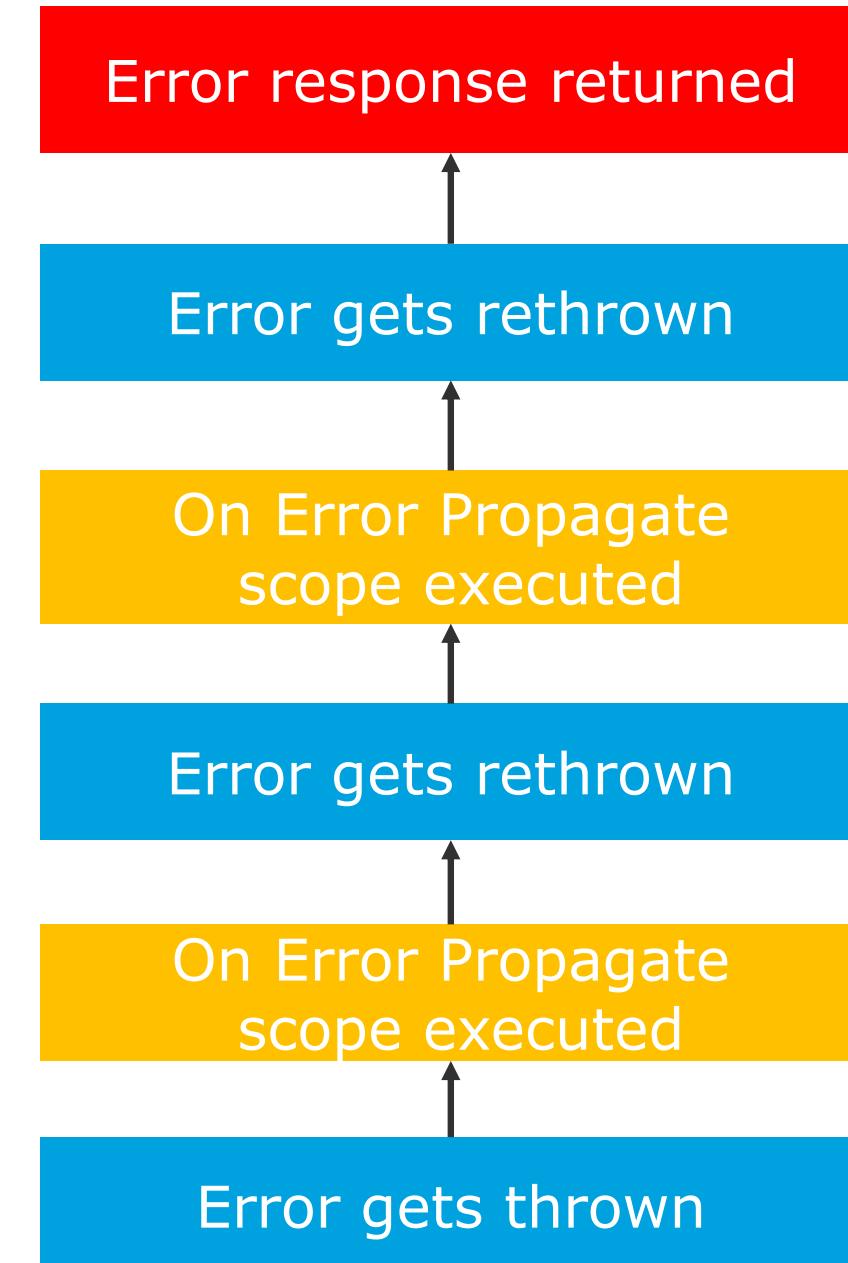
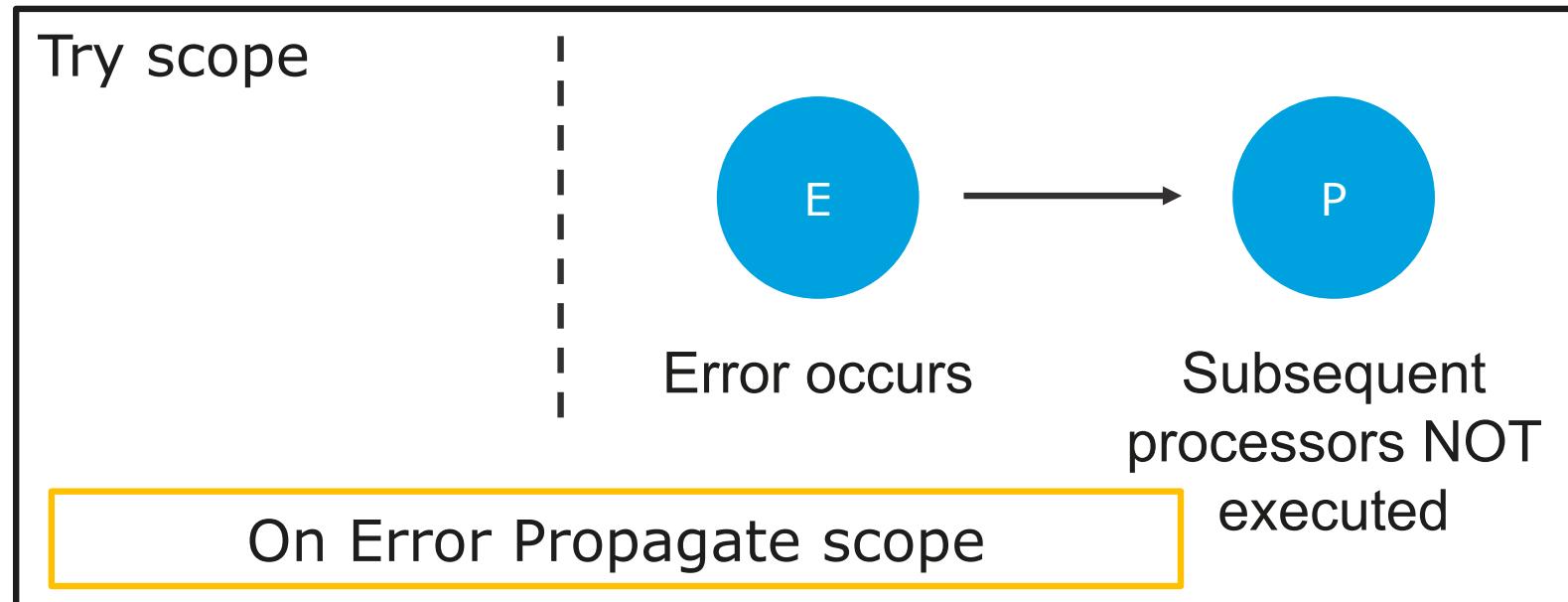
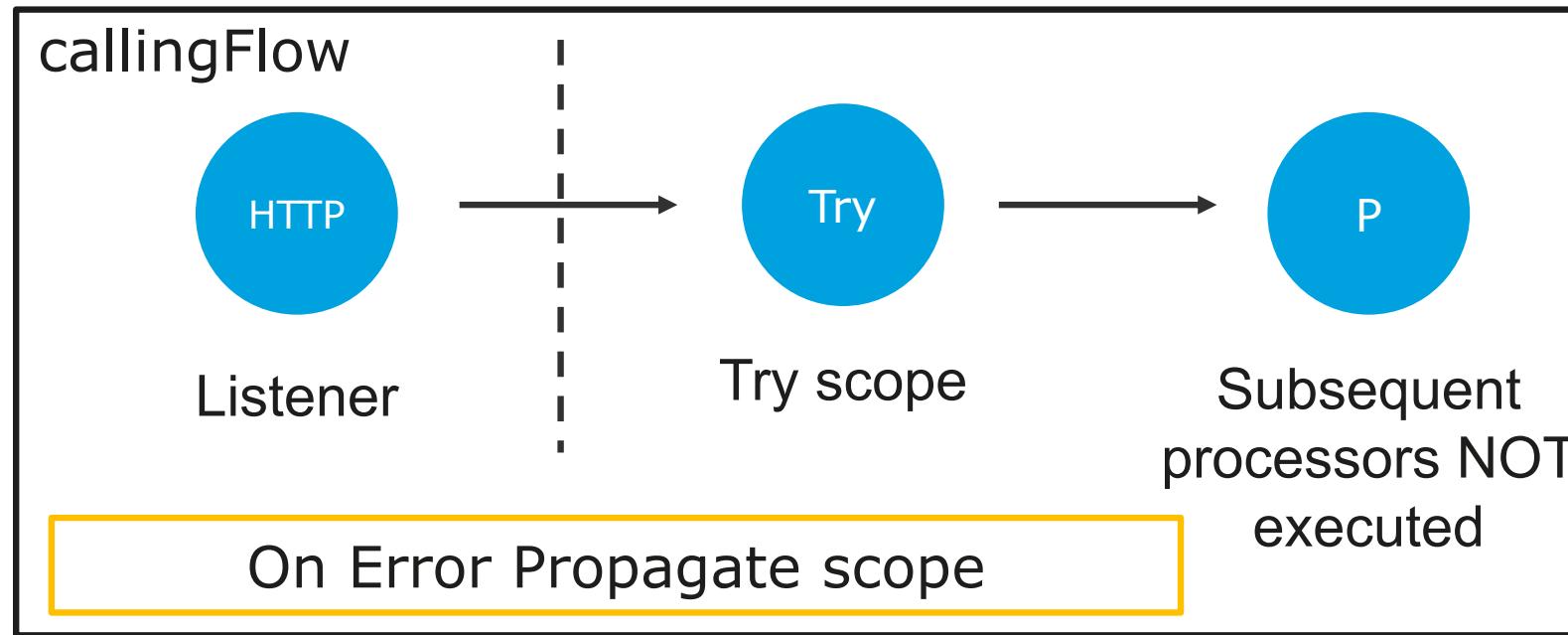
- **On Error Propagate**

- 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*

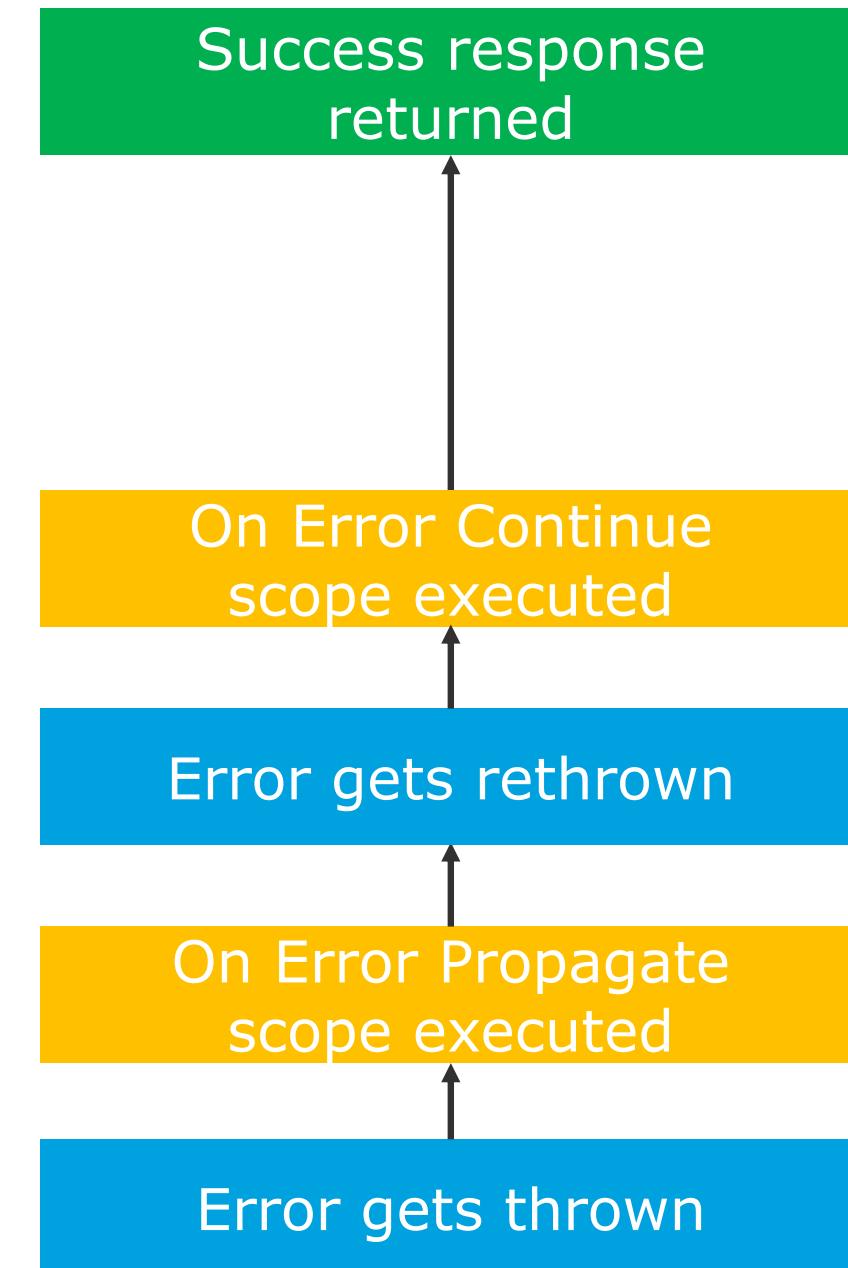
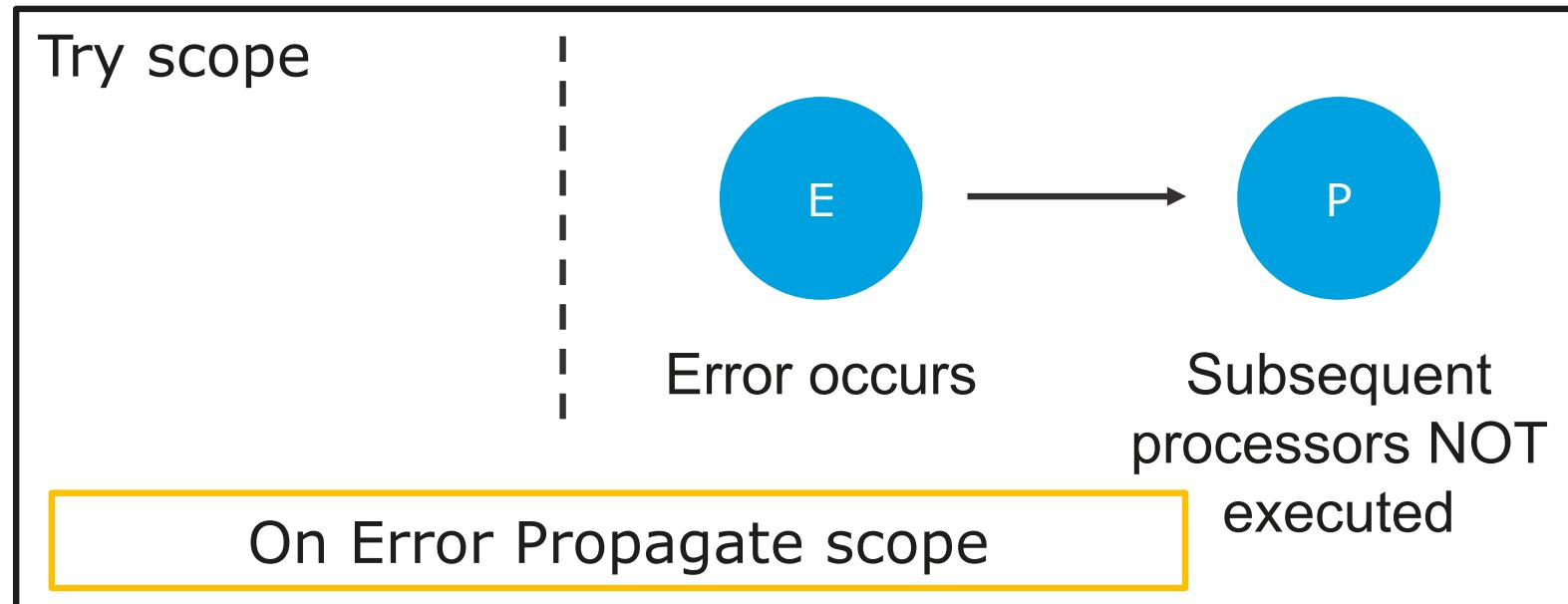
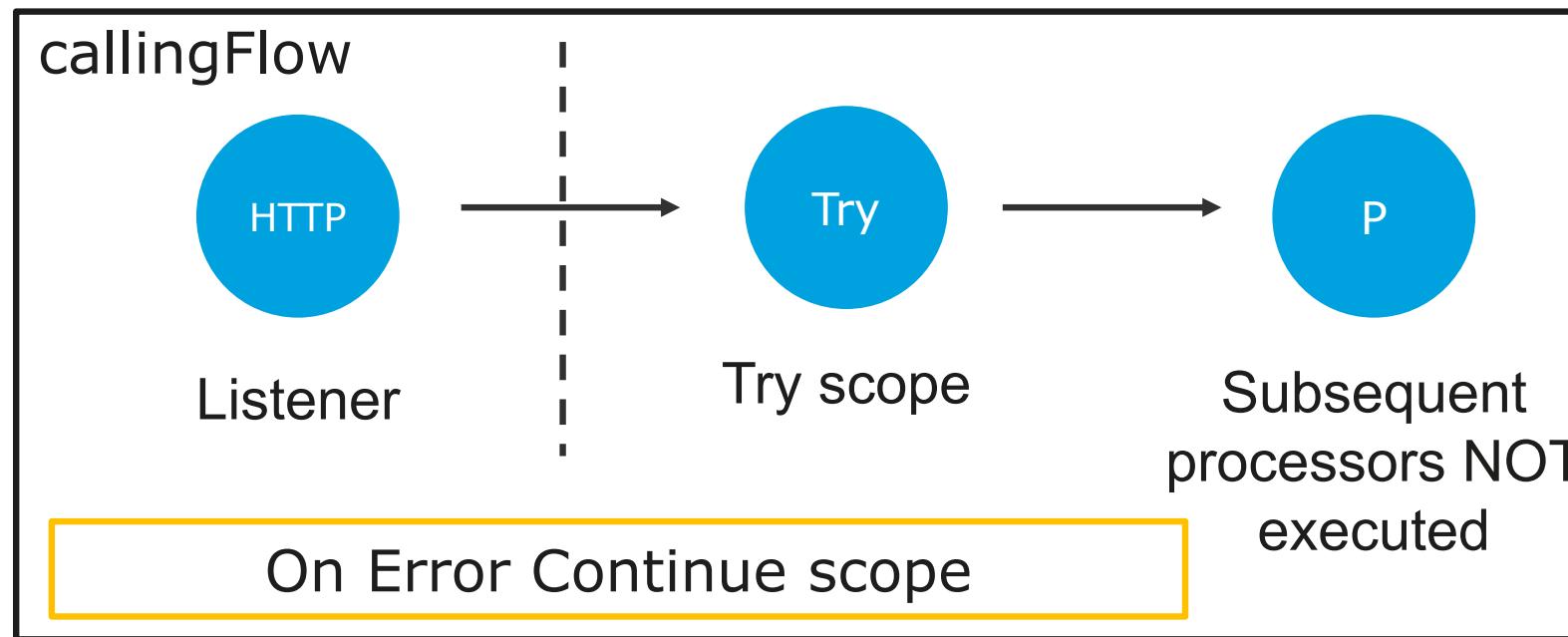
- **On Error Continue**

- 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*

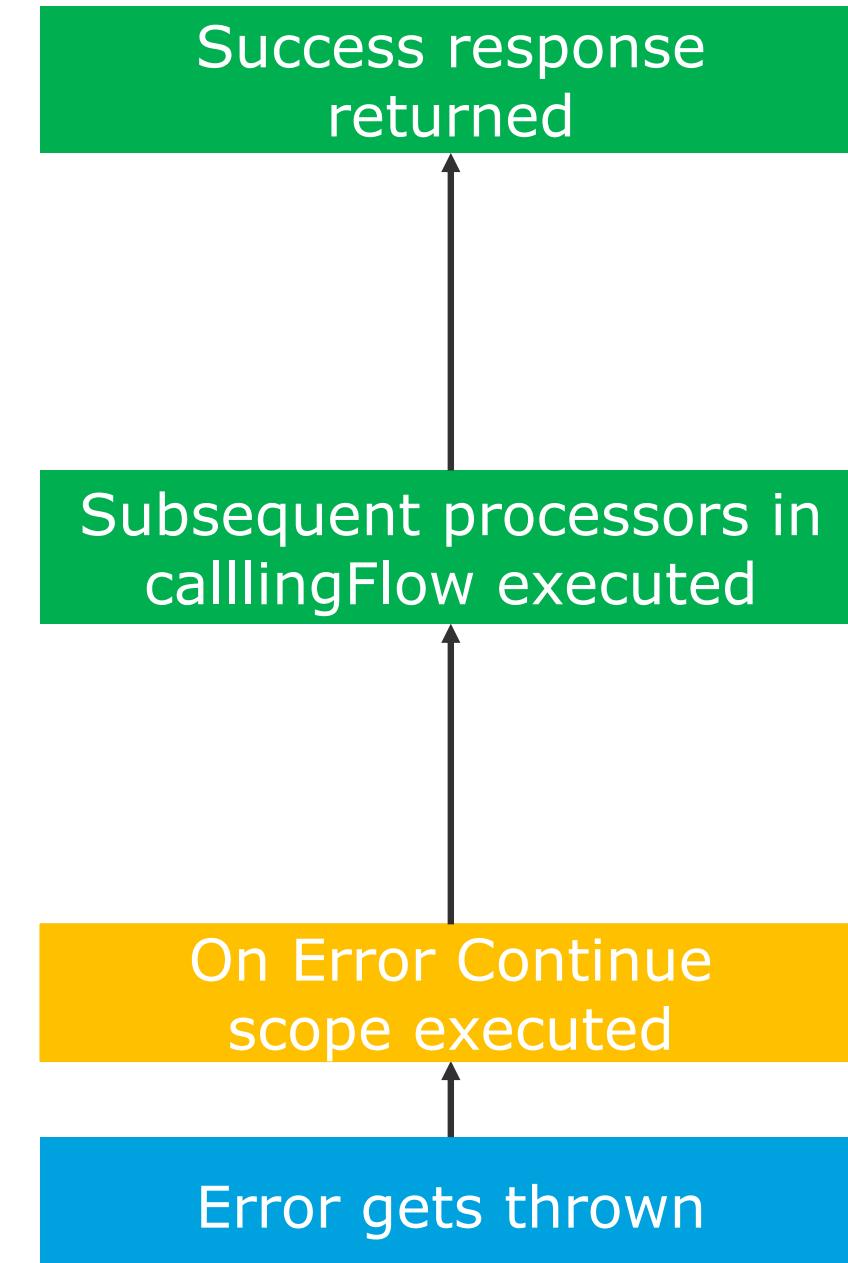
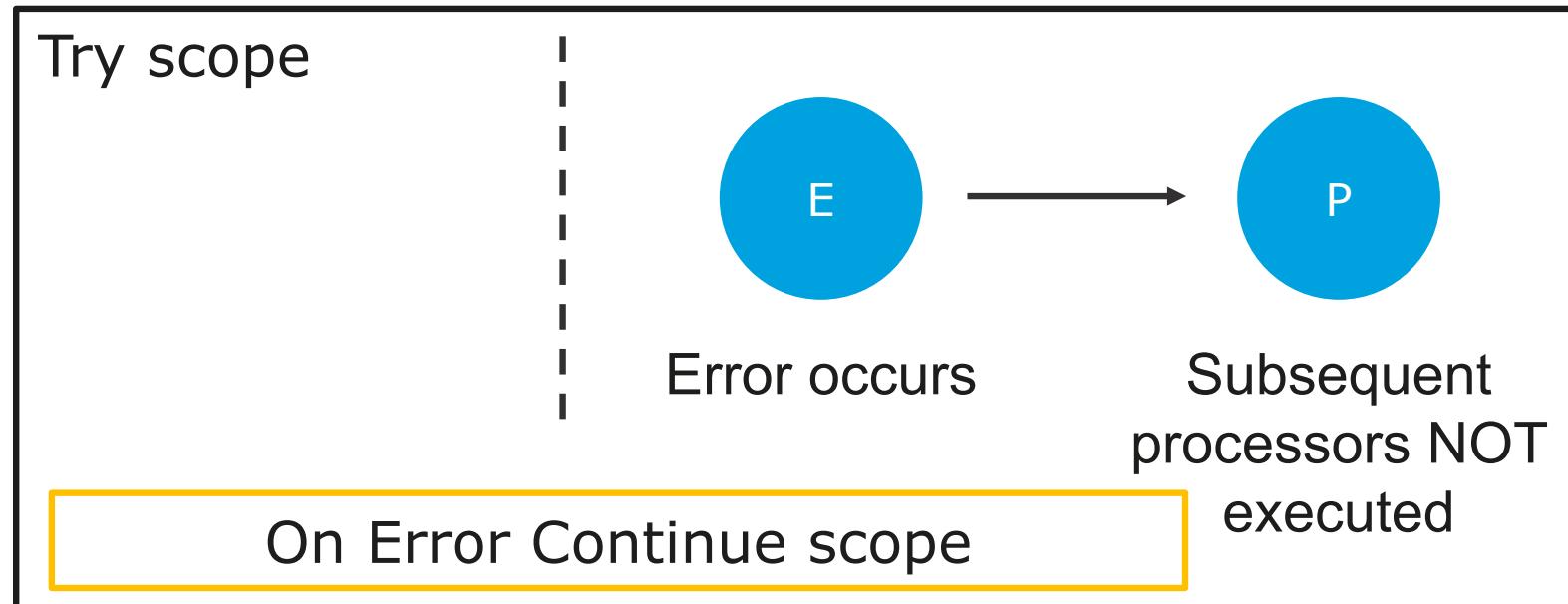
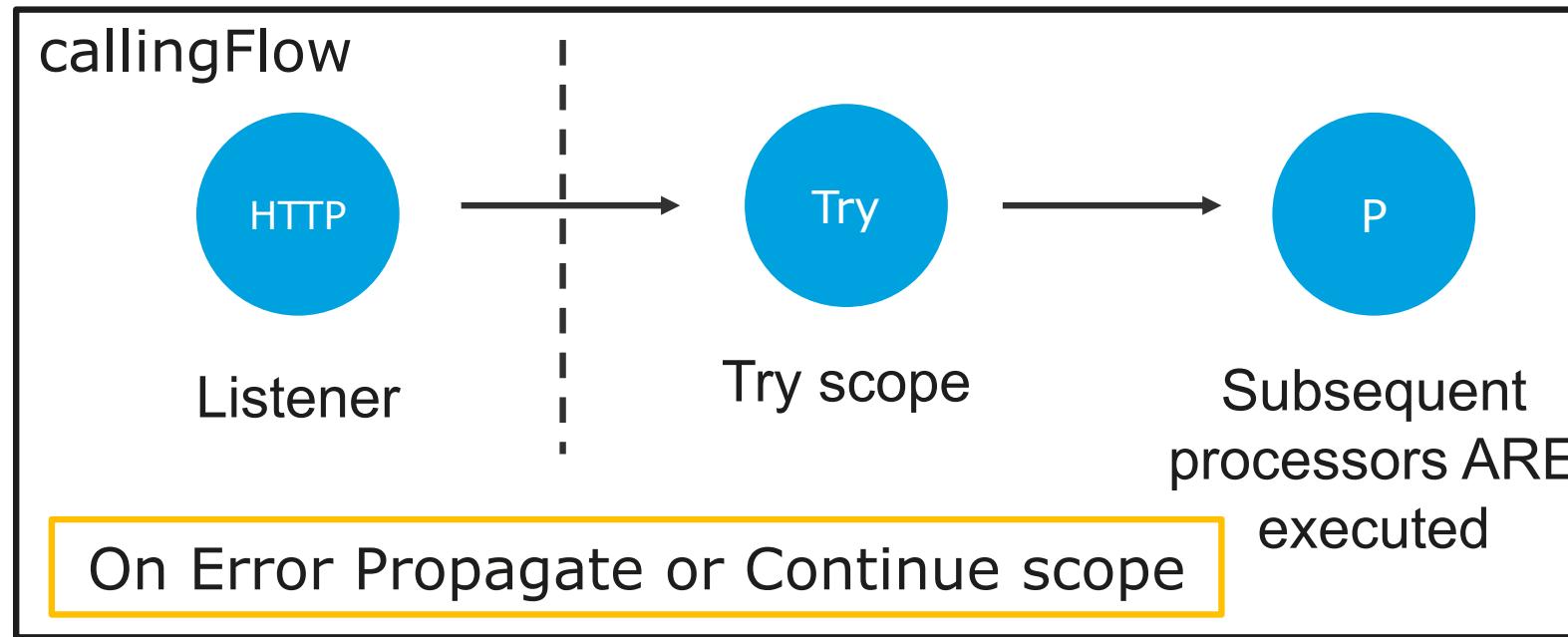
# Try scope: Error handling scenario 1



# Try scope: Error handling scenario 2

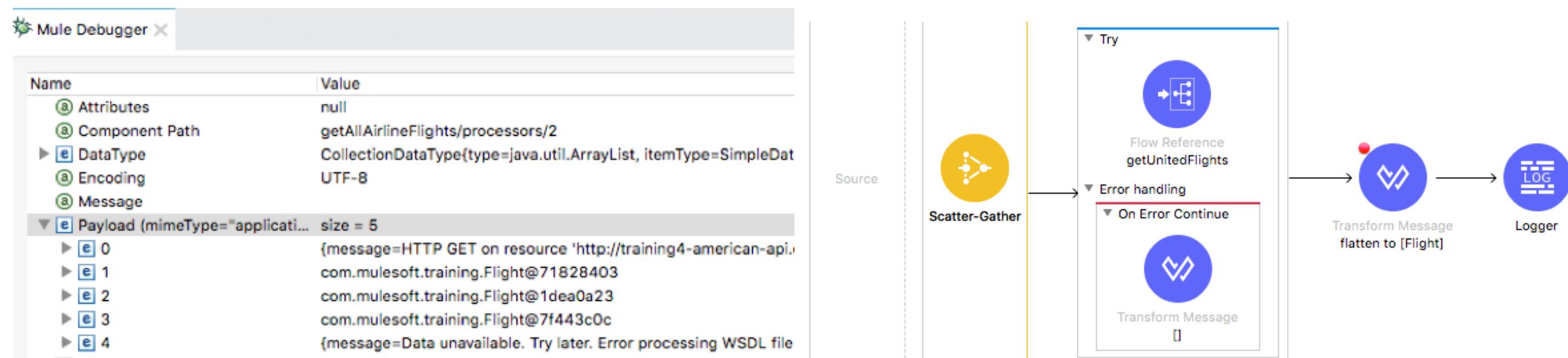


# Try scope: Error handling scenario 3



# 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 it 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

The screenshot shows the Mule Studio interface with two main panels. On the left, the 'Error Mapping' tab is selected in the General view. It displays a mapping from 'VALIDATION:INVALID\_BOOLEAN' to 'APP:INVALID\_DESTINATION'. On the right, the 'On Error Continue' event handler for the custom error type 'APP:INVALID\_DESTINATION' is configured. It consists of two steps: 'Transform Message' with the expression 'error.description' and 'Set Variable' with the variable 'httpStatus'.

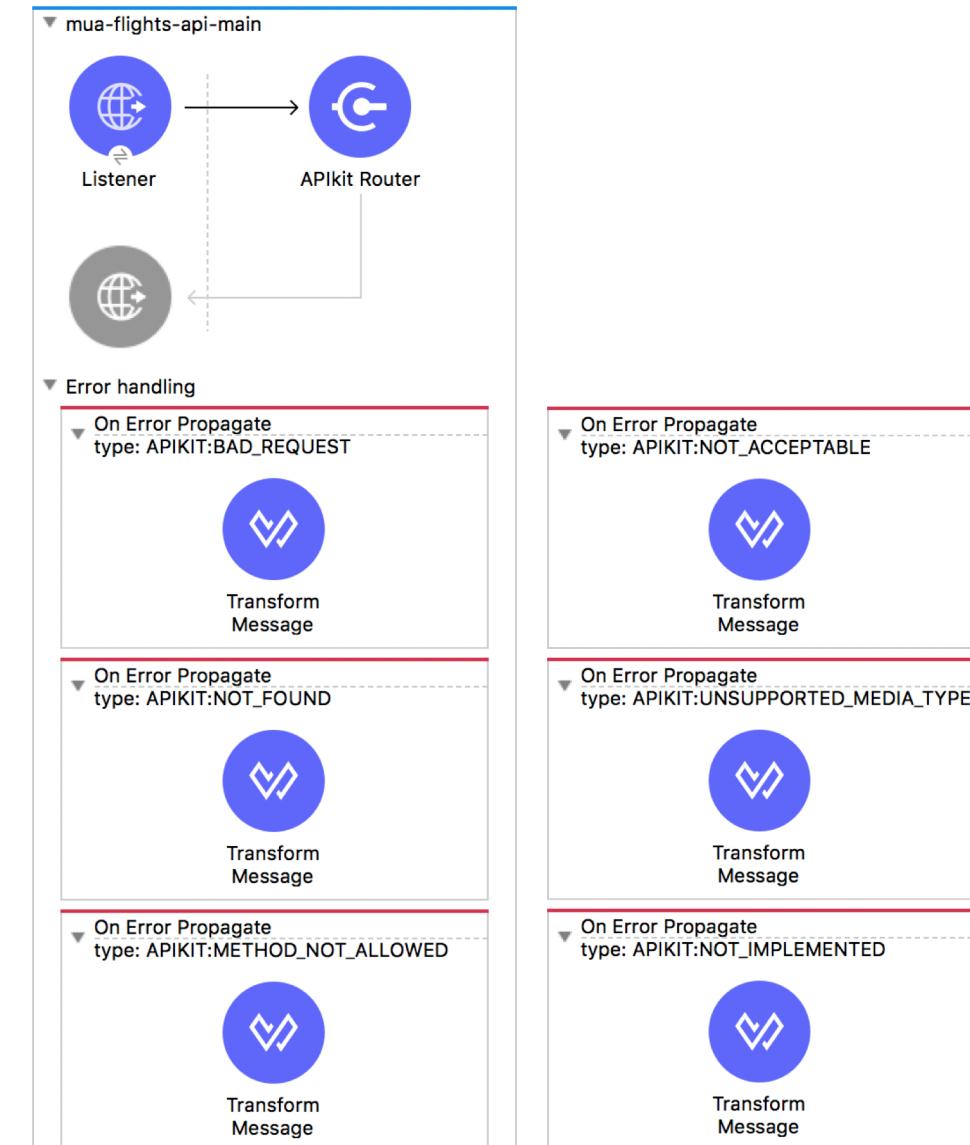
# Reviewing and integrating with APIkit error handling



# 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\_TYPE > 415
  - APIKIT:NOT\_IMPLEMENTED > 501

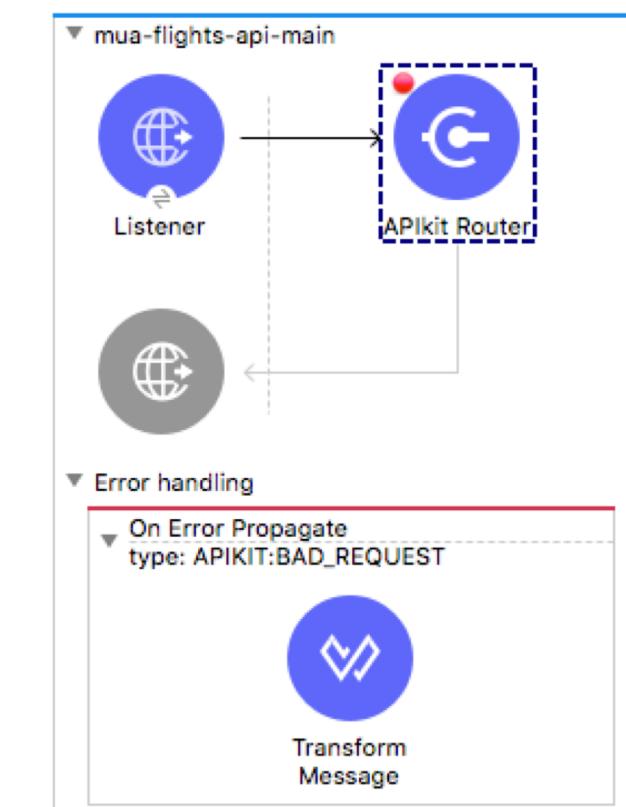


- You can modify the APIkit error scopes and add additional scopes
- You also need to make sure the error handling in the application works as expected with the new interface router
  - **On Error Continue**
    - Event in implementation is not passed back to main router flow
  - **On Error Propagate**
    - Error in 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



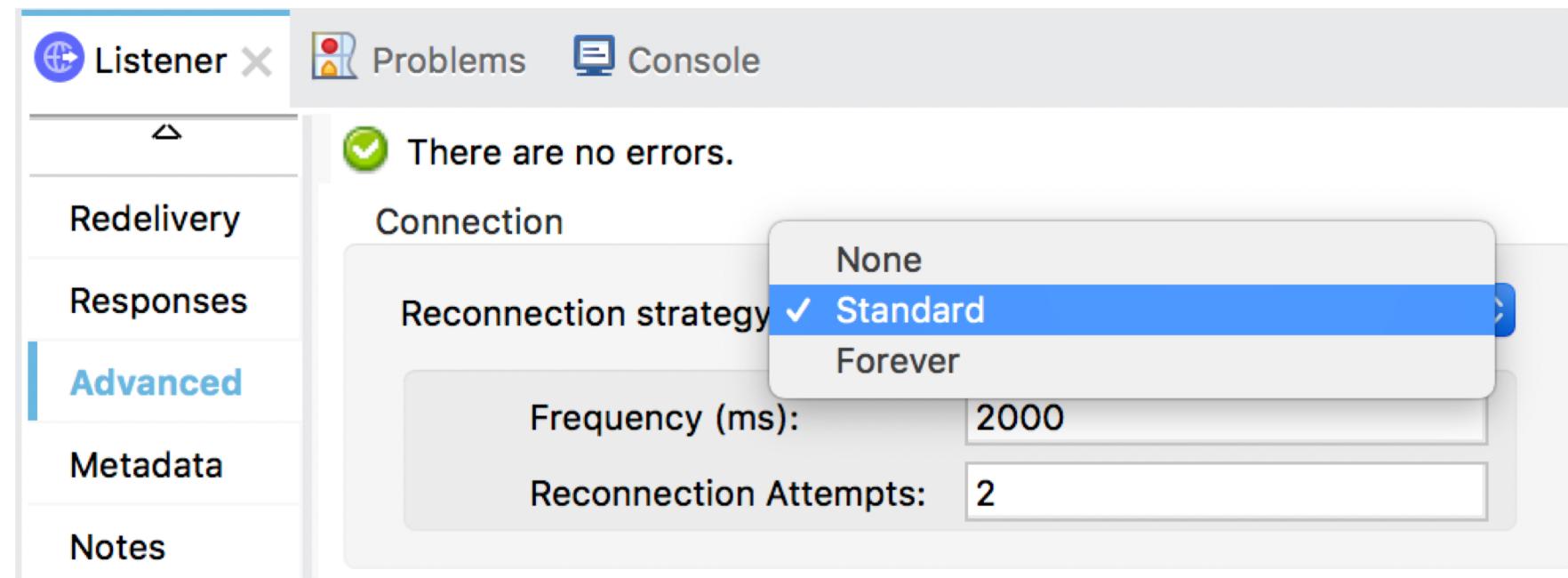
# Applications can have two types of errors



- **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

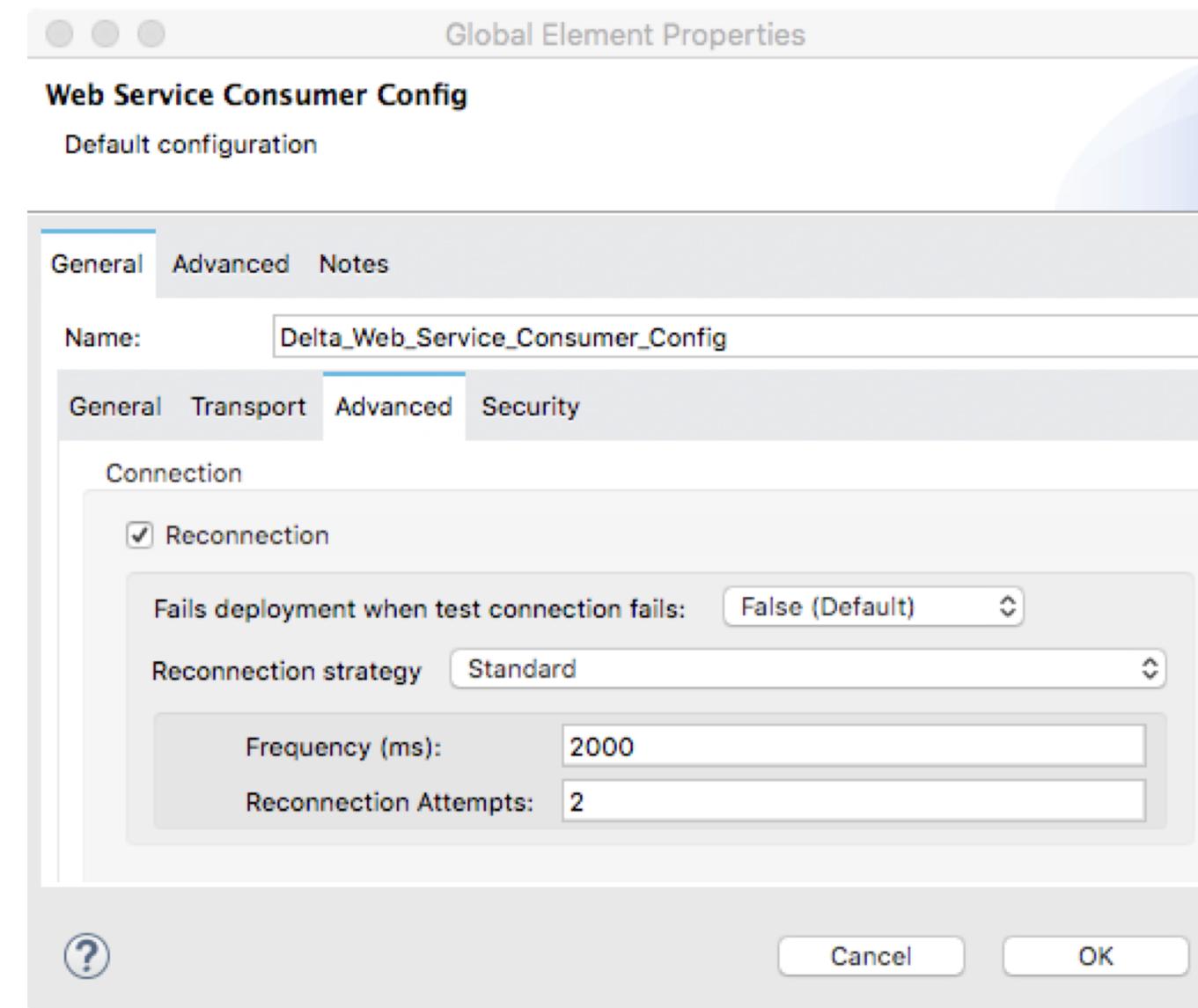
# Reconnection strategies

- 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



- 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

- 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

- 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