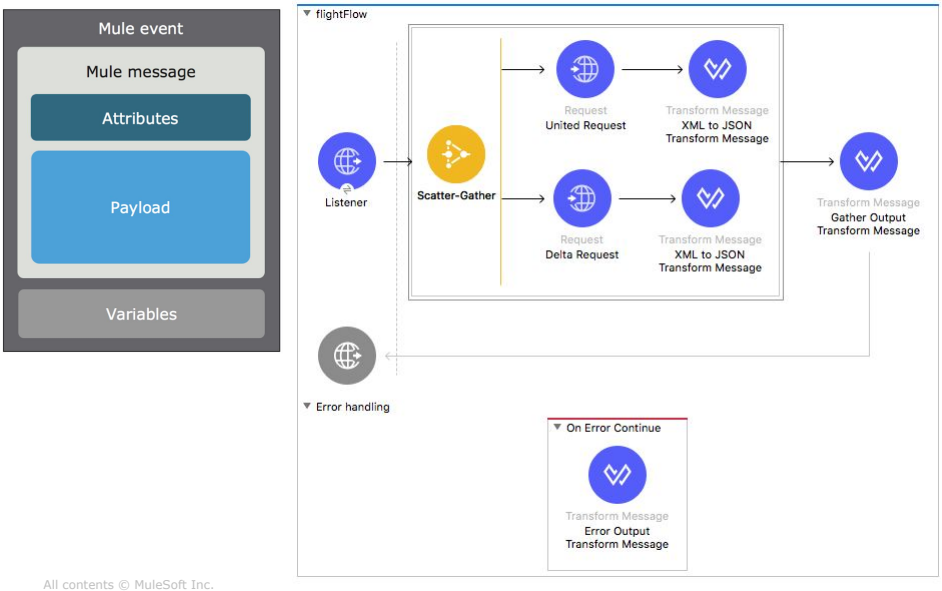
Module 3: Designing Integration Solutions with Mule Application Components

Goal



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

MuleSoft

- Identify how Mule application components are typically used to implement integration solutions
- Identify how class loading isolation is implemented in Mule runtimes
- Apply Mule application components and capabilities to the course case study

Specifying detailed Mule application designs

Identifying components and capabilities of Mule runtimes and MuleSoft toolsets

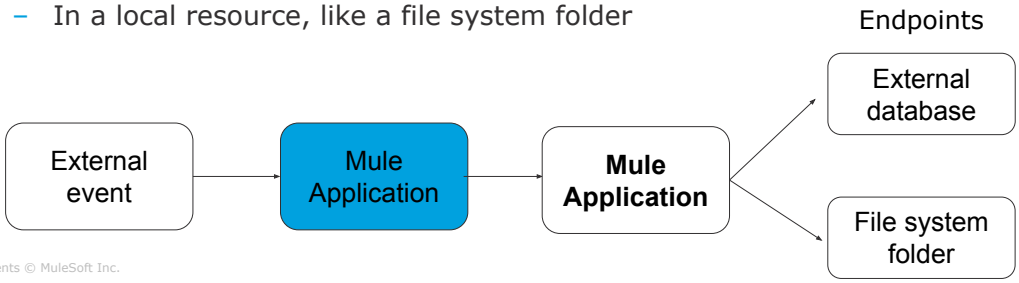
- The last module identified at a high level how to leverage the unified Anypoint Platform framework and services to achieve use case requirements

- This module focuses on more specific details related to designing the implementations of specific use cases
  - Identify components that make up typical Mule application flows and their behaviors to
    - **Exchange data with external resources using Mule 4 connectors**
    - **Transform data using DataWeave 2**
    - **Control the processing flow between event processors**
    - **Handle errors generated in flows**
    - **Troubleshoot Mule 4 class loading issues**

Identifying typical Mule Application components

What is a Mule application?

- A **Mule application** runs in a Java virtual machine (JVM) and is **triggered** by internal or external **events**, **processes** each event, and **routes** events to other **components** or **endpoints**
- The endpoints may be
  - In another Mule application
  - In another external system or resource
  - In a local resource, like a file system folder

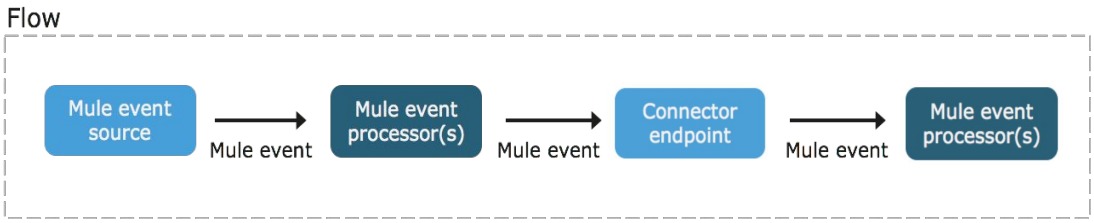


How Mule applications process events

- Mule applications accept and process **events** through a series of **Mule event processors** (also called **components**) linked together in one or more **flows**

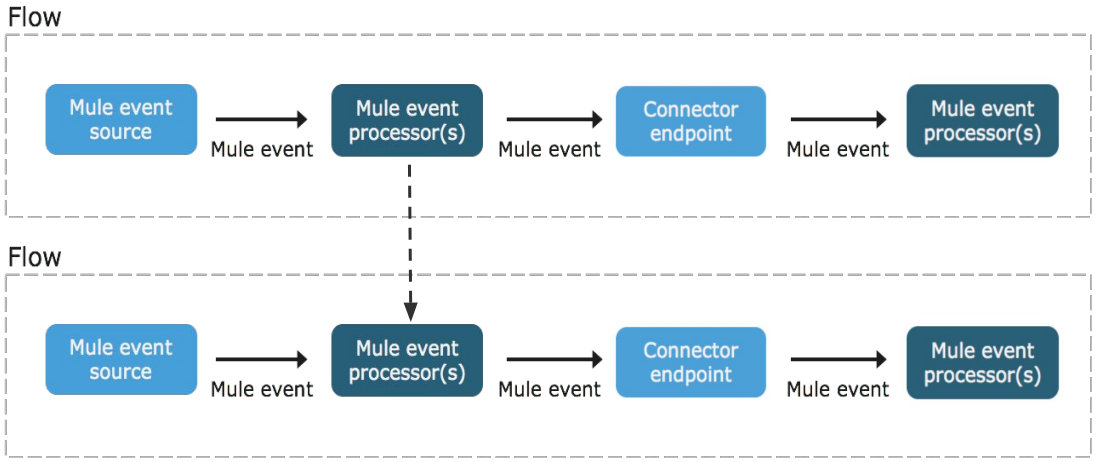
  Note: In Mule 3, components were called elements

- A flow is a linked sequence of event processors
  - Think of each **event processor** as a Lego block, while a **flow** is something you construct by linking the blocks together in a certain order



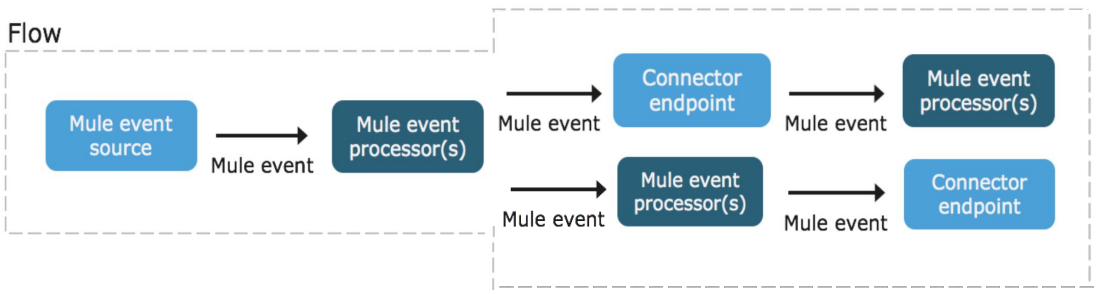
A Mule application consists of one or more flows

- Each flow can be independent of the other flows, or may invoke other flows in the Mule application



How Mule application flows are built and executed

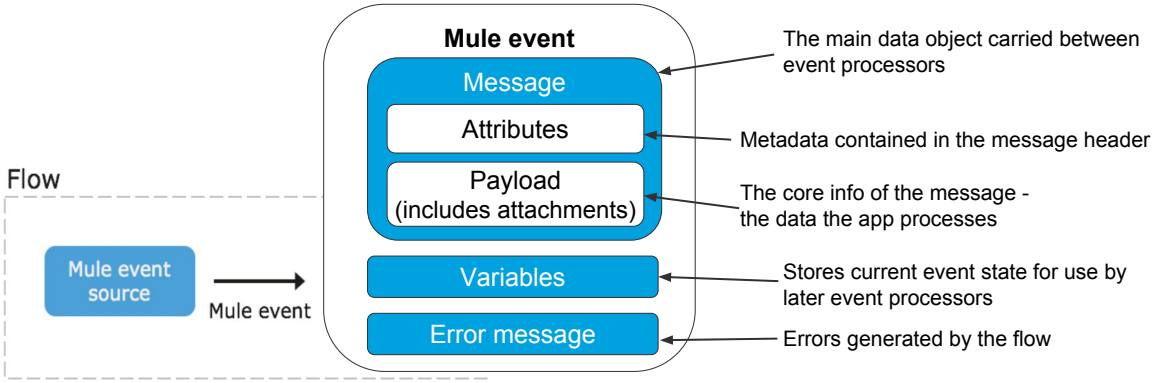
- A flow can coordinate services within one or more flows
  - A Mule 4 flow will often execute one event in multiple different threads
  - Mule 4 optimizes asynchronous event processing to avoid some unnecessary thread switches
  - Can execute concurrent tasks in parallel



Event sources create Mule 4 events at the start of a flow

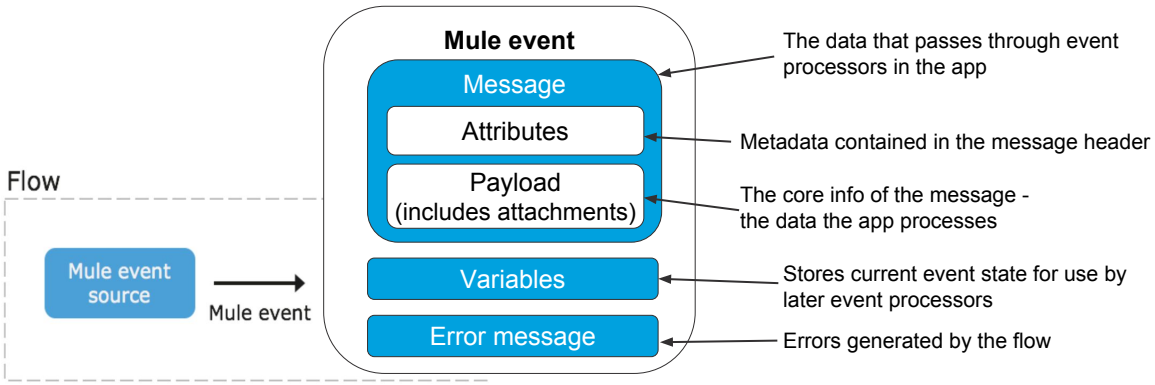
- A flow can start with an **event source**
  - Accepts inbound events and converts them to **Mule events**
  - The Mule event is passed or copied between event processors in the flow

Note: In Mule 3, event sources were called message sources

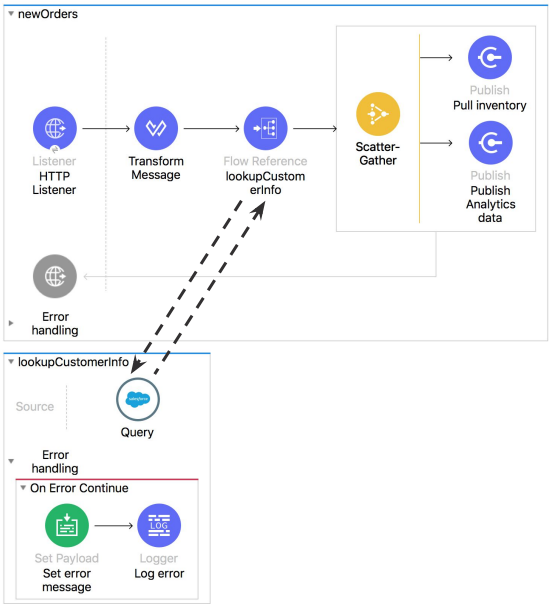


Comparing the structure of Mule 4 attributes vs. Mule 3 properties

- Mule 3 messages had inbound and outbound properties
- In Mule 4, there are only attributes, and each connector is now responsible for setting all outbound properties
  - The types of properties set in connectors is mainly the same

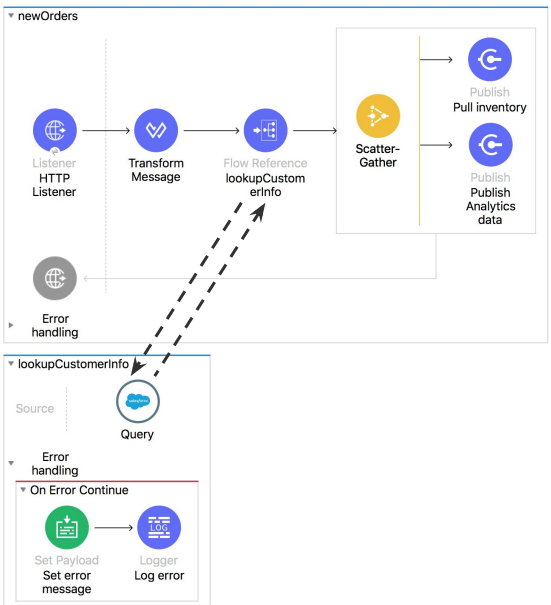


The structure of flows



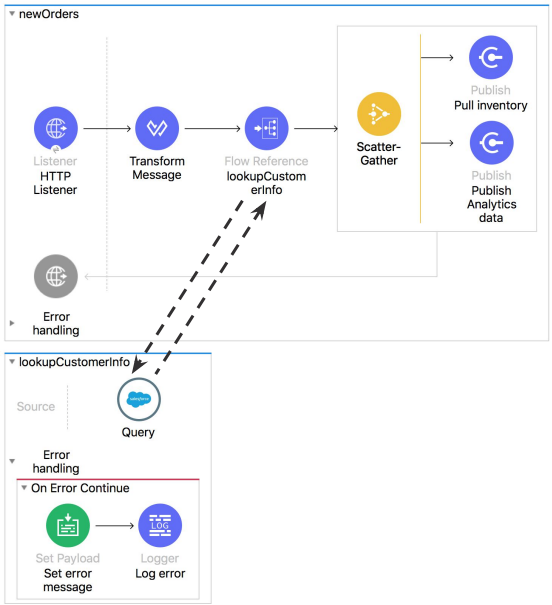
- Each flow
  - Can have at most one event source
  - Must have at least one event processor
  - Can have an error handler
  - Can invoke other flows

How Mule 4 events progress through event processors in a Mule application



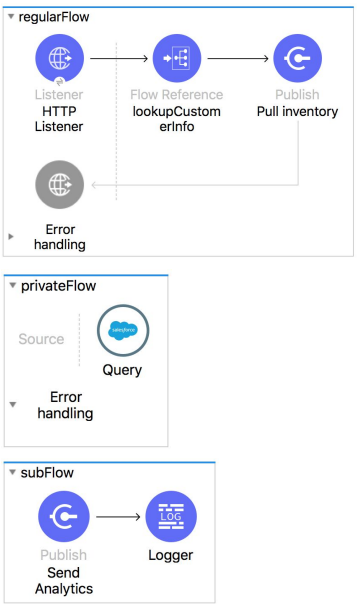
- Flows always function synchronously
  - Each event processor always outputs a Mule event, which might be the same as the input Mule event
  - Some connector operations (such as async) copy the Mule event for concurrent processing (on a new thread)
  - To call a sequence of one or more event processors **asynchronously**, for concurrent processing, wrap them in an **async scope**

Mule 4 vs. Mule 3 flows



- In Mule 3, flows could define a processing strategy to decide if the flow processing was synchronous or asynchronous
- Flows used various Staged Event Driven Architecture (SEDA) queues
- In Mule 4, asynchronous processing strategies and thread pools are automatically selected by the self-tuning Mule 4 runtime
  - More on this later in the course

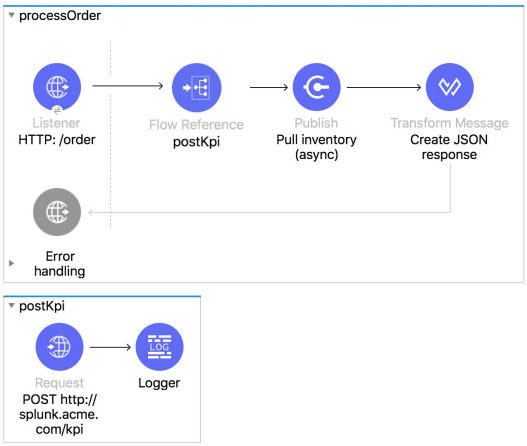
Three types of flows



- Regular flow
  - Starts with a message source
  - Can have its own error handler
- Private flow
  - Identical to a regular flow, but no message source
  - Can only be triggered from within the Mule application
    - Such as through a flow reference or a DataWeave lookup function
- Sub flows
  - No message source, like a private flow
  - No error handling
    - Errors bubble up into the parent flow

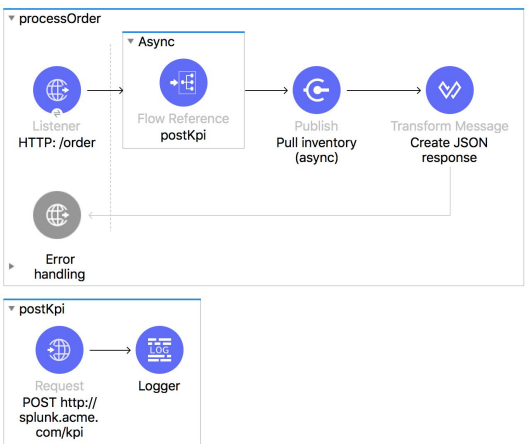
Flows can call other flows

- Flows can synchronously call other flows or subflows via a flow reference
  - In that case, the called flow's message source is skipped synchronously



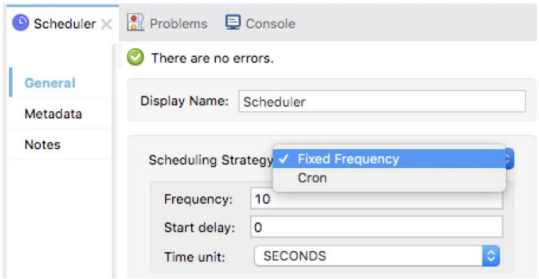
Calling flows asynchronously

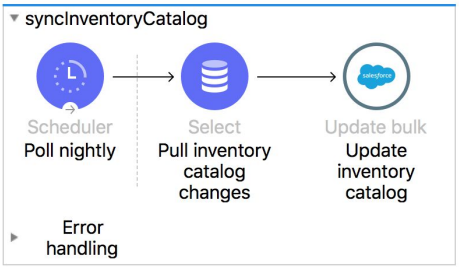
- All types of flows can be triggered asynchronously from a parent flow using an Async scope



Internally triggering a Mule 4 flow using a scheduler

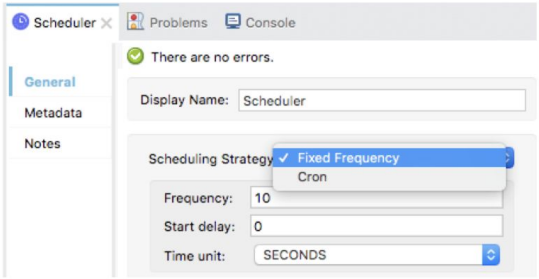
- The **Scheduler** component can trigger a flow at a specific **time**
  - **Fixed frequency**
    - The default is to poll every 1000 milliseconds
  - **Cron**
    - A standard for describing time and date information
    - Can specify an event to occur just once at a certain t specific dates or at some frequency

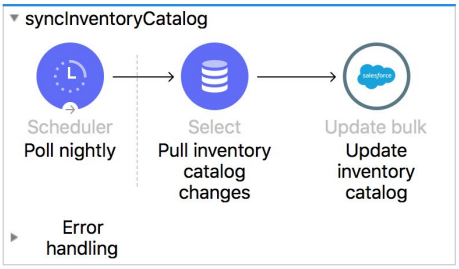




Mule 3 uses polling scopes instead of schedulers

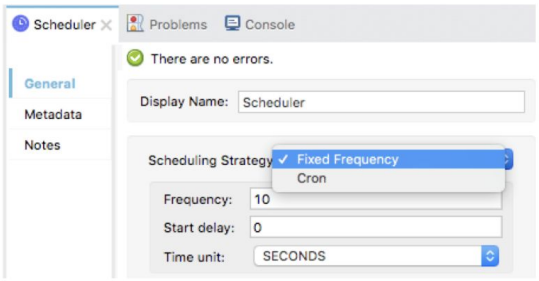
- Mule 3 flows could wrap message sources in a polling scope
- The behavior of a scheduler and a polling scope are similar
- In Mule 4, some connectors themselves provide operations to handle the polling and watermarks

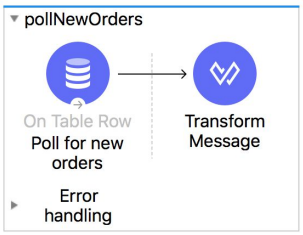




Configuring schedulers inside some message sources

- Some connector operations use an internal and implicit scheduling strategy to
  trigger a flow, such as
  - Database: On Table Row
  - File, FTP, SFTP: On New or Updated File

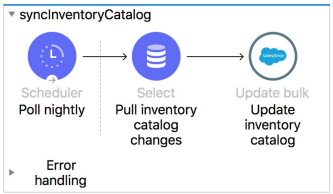




Exchanging data with external resources using Anypoint Connectors

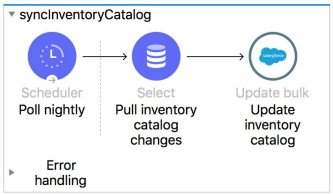
Reviewing Anypoint Connectors

- Anypoint Connectors **abstract and simplify** connections to third party resources and other protocols
  - Built using a common Mule SDK
    - Helps to build new custom connectors in a consistent way that can be shared in Anypoint Exchange
  - For a REST API, an Anypoint connector can be automatically generated in Anypoint Exchange from a RAML file, using the Anypoint REST Connect feature
- In Anypoint Exchange, connectors are divided up into various categories based on support and distribution rights



How to invoke REST or SOAP services

- An HTTP or Web Service Consumer connector can be used
  - But a **custom connector** can explicitly expose the operations of the web service and simplify some operations, such as authorization and authentication
  - A connector automatically generated by Anypoint REST Connect and stored in Anypoint Exchange is also faster and easier to use vs. an HTTP connector

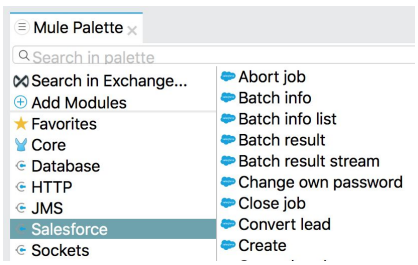


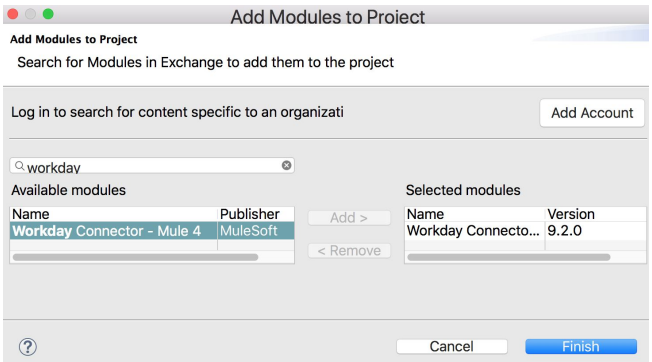
Advantages of using Anypoint Connectors

- **Facilitate integration** of Mule applications with **third-party systems** such as Salesforce, Google Apps, Google Cloud, MongoDB, ServiceNow, Workday, Facebook, or Twitter
- No need to study and code to underlying **protocol** or **security** used by third party systems
- **Speed up** development and deployment of Mule applications
- Make mule applications are easier to **maintain**

Adding Anypoint Connectors to a Mule application

- Anypoint Studio can log in to Anypoint Exchange as a particular Anypoint Platform organization user
  - Search for and install a particular versions of an Anypoint Connector
  - Might include connectors published to the organization's private Anypoint Exchange



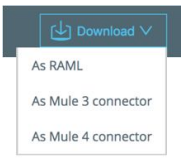


Implementing REST APIs with APIkit

- APIkit can auto-generate flows from a RAML file
  - These flows can proxy access to other external services
  - Inside the generated API service flows, external services can be accessed with other Anypoint Connectors
- Once the APIkit flows are implemented, other web clients can connect to the API endpoints using the same RAML specification

Invoking REST APIs using REST Connect

- Other Mule applications can also act as web clients to APIkit-generated flows
  - First upload a RAML or OAS REST API file to Anypoint Exchange
  - Anypoint Exchange then uses its internal Anypoint REST Connect service to auto-generate two complete Anypoint connectors
    - One for Mule 3 and one for Mule 4
  - The connectors are then available for download from the API asset in Anypoint Exchange



Why REST Connect generated connectors are helpful to Mule application developers

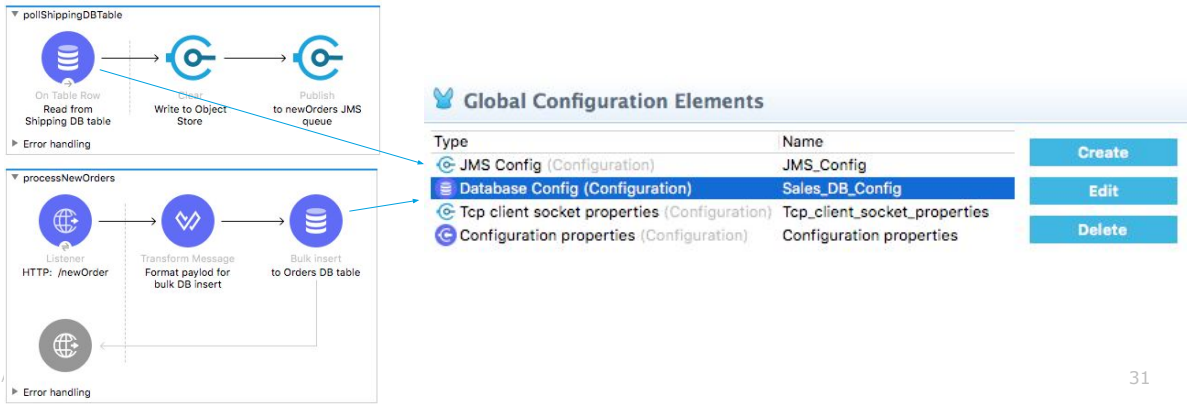
- The connector reflects the RESTful API with its resources and methods
  - If invalid parameters or data is input to the connector, specific context sensitive **error messages** are generated
  - Defines **specific metadata** to help with data transformation
    - Such as when using a Transform Message component to visually map DataWeave code
  - This helps Mule applications **fail fast and helps to quickly identify issues** between distributed API endpoints

Sharing connections and configurations

Sharing connections and other configurations

MuleSoft

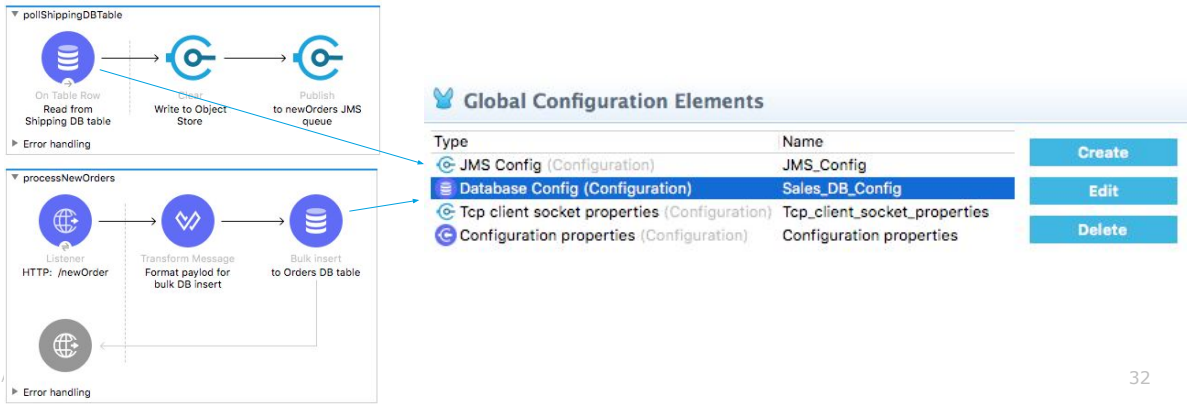
- **Global elements** are connectors, caches, and other configurations that can be shared between Mule components
  - Global elements can be **shared** among event processors of the same type
  - For example, a database connection shared by several database operations



Benefits of using global elements

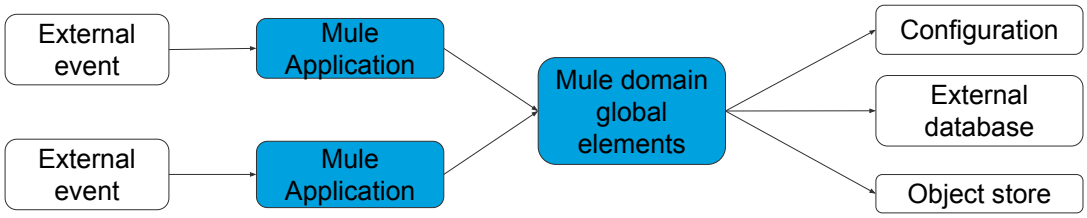
MuleSoft

- Global elements simplify the operations and maintenance of the Mule application
  - Change once in the global element, then changes apply to all Mule flow elements that reference the global element



Mule domains can share global elements across Mule applications

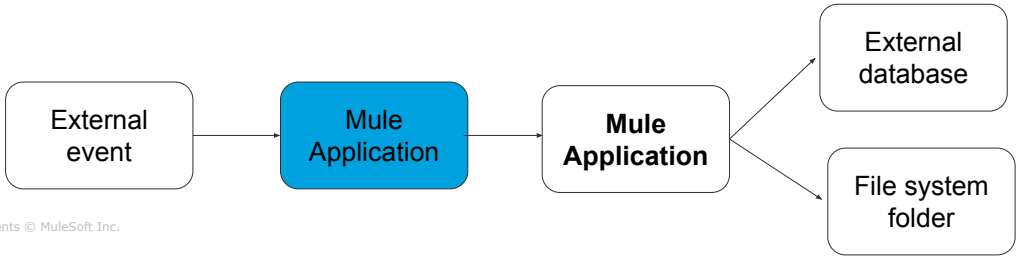
- Global elements can also be added to a Mule domain project
  - One or more Mule applications can be associated with the same Mule domain
  - A Mule domain can be deployed to Mule runtimes along with Mule applications
  - So now Mule applications deployed to the same Mule domain can now share any connection type or other global element
    - Simpler operations and maintenance
  - Mule domains are only supported in **customer-hosted** runtime planes



Comparing Mule 4 and Mule 3 applications

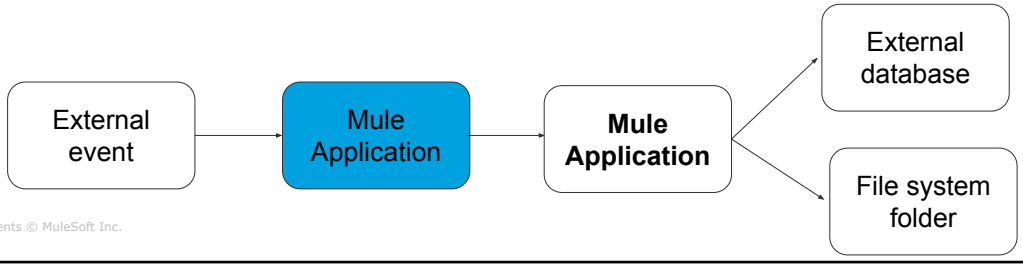
What changed in Mule 4?

- Mule 4 and Mule 3 applications are very similar
- Messages are now called events, and the structure changed
- Inbound and outbound properties are called attributes
- Each Mule 4 connector explicitly sets its own outbound properties
- Session/record variables are removed, and instead each connector passes data over transports as Payload, headers, attachments, etc.
- Variables are now stored and passed around with the Mule event



How flows changed in Mule 4

- The Mule 4 runtime is self-tuning so SEDA queues are removed
- Flows do not need to set processing strategies

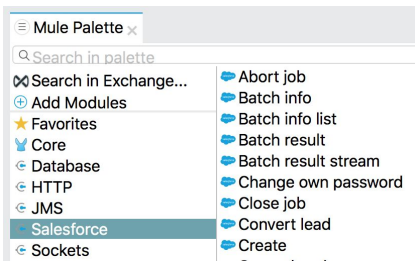


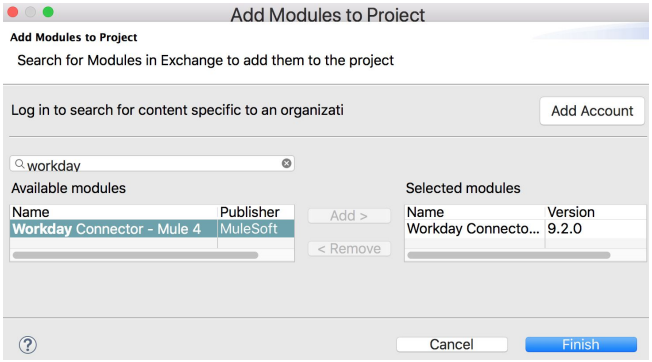
Reusing global elements in Mule applications

- Create reusable configuration details for connectors, caches, other components, and other shared configurations
- Can be shared by multiple components in the Mule application or Mule domain
  - Share a global configuration among processors of the same type
- Maintainability
  - Change once in the global element, and the changes apply to all Mule flow elements that reference the global element

Mule 4 vs. Mule 3 connectors

- Unlike Mule 3, Mule 4 applications are **always** Maven projects
- Anypoint Exchange access is integrated into Anypoint Studio
  - New connectors and modules can be added directly into Anypoint Studio from Anypoint Exchange, within a Mule project





Mule 4 offloads connectors and other components to separate **modules**

- In Mule 3, connectors were bundled with the Mule 3 runtime
  - This created brittle, tightly coupled deployments
  - To upgrade or patch a connector or other component, you had to wait for a new Mule runtime release
  - For customer-hosted Mule runtimes, you also had to download and reinstall the new Mule runtime version
- In Mule 4, each connector and many other components are separated out into independent extension libraries, called **modules**
  - This avoids version conflicts between libraries from different modules

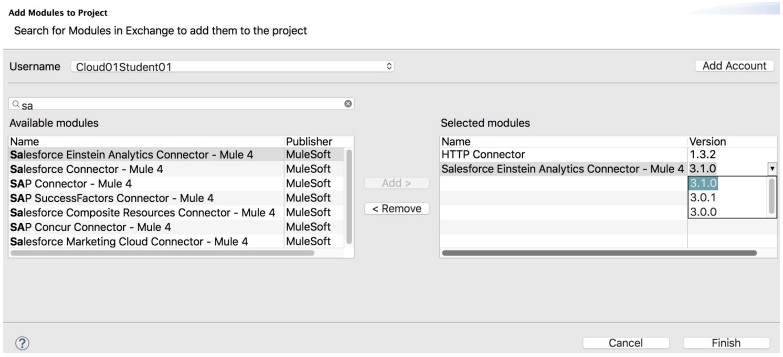
How modules are configured in a Mule application

- Each **module is configured as a Maven dependency** in the Mule application's pom.xml file
  - New module versions can be independently downloaded from Maven repositories without needing to update the Mule runtime or any other modules

```
<dependencies>
    <dependency>
        <groupId>org.mule.connectors</groupId>
        <artifactId>mule-http-connector</artifactId>
        <version>1.3.2</version>
        <classifier>mule-plugin</classifier>
    </dependency>
    <dependency>
        <groupId>org.mule.connectors</groupId>
        <artifactId>mule-sockets-connector</artifactId>
        <version>1.1.2</version>
        <classifier>mule-plugin</classifier>
    </dependency>
    <dependency>
        <groupId>org.mule.connectors</groupId>
        <artifactId>mule-objectstore-connector</artifactId>
        <version>1.1.1</version>
        <classifier>mule-plugin</classifier>
    </dependency>
```

The classloading lifecycle of Mule 4 modules

- Each Mule 4 module now has an independent and isolated class loading lifecycle
  - These are supported by a more complex connected graph of classloaders
  - This is the mechanism to support different versions of the same Java libraries (with the same classes and interfaces) in the same Mule application without conflicts



Reviewing the behavior of common connectors

Common Mule 4 connectors

- Anypoint Studio ships with some common connectors
- New connectors can be easily downloaded and added from Anypoint Exchange, directly inside Anypoint Studio
- Common connectors include HTTP, Database, File, and FTP

The Mule HTTP connector

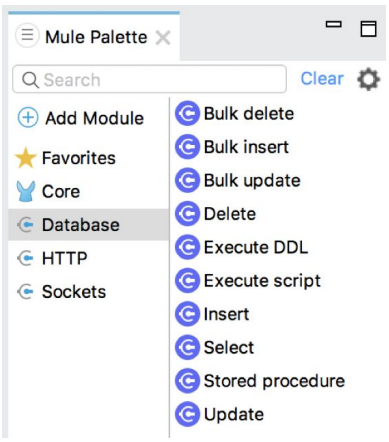
- Supports both client and server operations
  - Listen for inbound requests over HTTP/HTTPS to trigger a flow
  - Send an HTTP/HTTPS request in the middle of a flow
  - HTTP 1.0 and 1.1 (HTTP 2.0 is not supported)

The Mule HTTP connector uses the Grizzly libraries internally

- Grizzly is a standard open source library supporting the HTTP/S protocols
- Uses **selector** thread pools
  - Selector threads check the state of the NIO channels and creates and dispatches events when they arrive
  - Listener selectors poll only for **request events**
  - Requester selectors poll only for **response events**
- HTTP Listener has a **shared selector pool** for all Mule applications deployed to a particular Mule runtime
- HTTP Requester has a **dedicated selector pool** for each Mule application

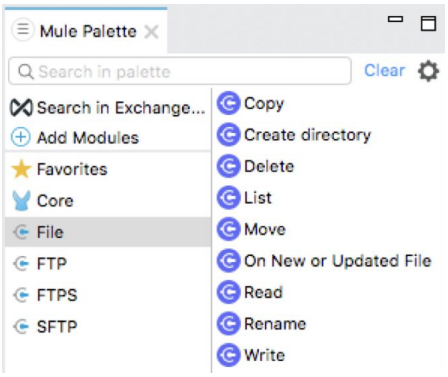
Database connectors

- Can call out to almost any JDBC relational database
  - Any database engine for which you have a JDBC driver
- Support for
  - DDL
  - DML
  - Stored procedure
  - DB script
  - Bulk operation



File and FTP connectors

- These four connectors can work with files and folders
  - File (for locally mounted file system)
  - FTP
  - FTPS
  - SFTP
- All have the same set of operations and they behave almost identically
- Support for
  - File matching
  - Overwriting, appending, and generating new files

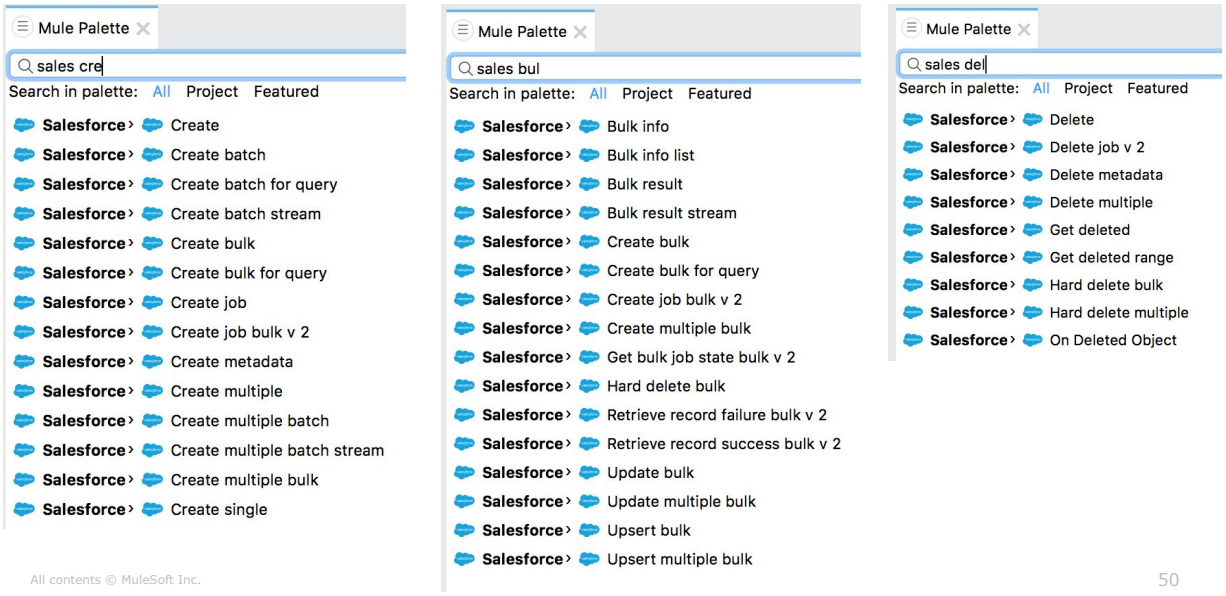


SaaS connectors

- There are many dedicated connectors such as for SaaS systems
  - Available from Anypoint Exchange
  - Import the connector into Anypoint Studio or flow designer and start integrating

IBM AS/400
Oracle E-Business Suite
Twilio
Microsoft Azure Storage
Salesforce
Netsuite
Amazon SNS
BMC Remedy
Amazon S3
MongoDB
HL7
Marketo
Temenos T24
X12 EDI
Zuora
Marketing Cloud
SAP
Microsoft Sharepoint
Workday
Google DoubleClick
Amazon SQS
Einstein Analytics
SAP Hybris
ServiceNow

SaaS connectors often support many specialized and optimized operations



Building custom connectors with the Mule 4 SDK

- The Mule SDK for Java supports building **custom** connectors for Mule 4
  - Custom connectors are used to integrate existing business logic, algorithms, etc.
- You should first prefer OOTB components or web services (REST or SOAP) over custom components
- Build custom connectors only when an API or OOTB connector does **not** support the use case
  - First, look in Anypoint Exchange for a specialized pre-built connector (for example SFDC)
  - If you don't find one, see if you can connect to the external system with a web service (REST or SOAP)
  - Finally, evaluate if it is worth creating a custom connector

Migrating Mule 3 DevKit custom connectors to Mule 4

- The Anypoint Connector DevKit is not compatible with Mule 4
- A DevKit migration tool will support migration of custom Mule 3 connectors build with the Anypoint Connector DevKit
  - Will typically also involve some custom coding

Types of Anypoint Connectors and levels of support

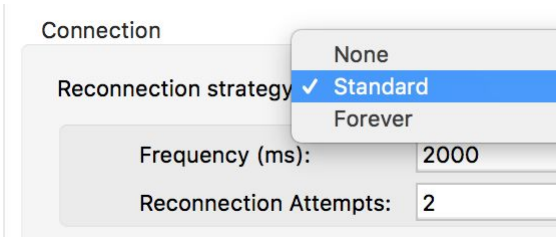
Connector type	Description
Community https://anypoint.mulesoft.com/exchange/?search=community	• MuleSoft or members of the MuleSoft community write and maintain the community connectors • No license
MuleSoft Certified https://www.anypoint.mulesoft.com/exchange/?search=mulesoft-certified	• Developed by MuleSoft's partners and developer community and are reviewed and certified by MuleSoft • Contact the MuleSoft partner that created the MuleSoft Certified connector for support
Select https://anypoint.mulesoft.com/exchange/?search=select	• MuleSoft maintains Select Connectors • Connectors included in the open source Mule distribution can be used by everyone, however support is only included in an Anypoint Platform subscription • To use all other Select Connectors and access support, you must have an active Anypoint Platform subscription.
Premium https://anypoint.mulesoft.com/exchange/?search=premium	• MuleSoft maintains Premium Connectors • You must have an active CloudHub Premium plan or an Enterprise subscription with an entitlement for the specific connector you wish to use.

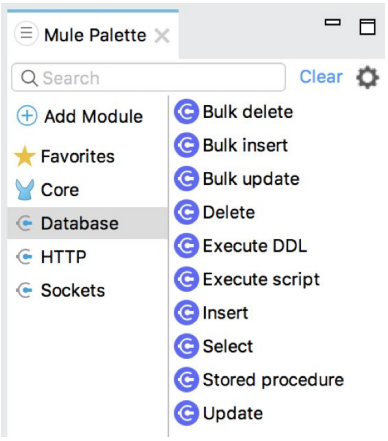
Configuring reconnection strategies for connectors

Many connectors can configure a reconnection strategy

MuleSoft

- Connectivity tests run when the Mule application starts, then periodically while the Mule application runs
- The reconnection strategy dictates what to do when connectivity fails





How connector reconnection strategies work

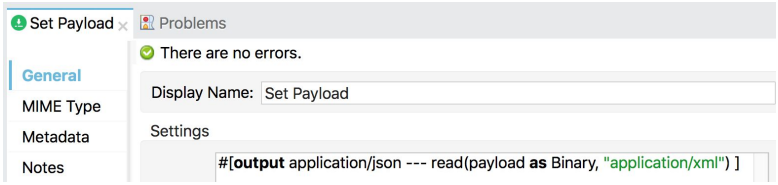
- By default, a failed connectivity test is just logged and the Mule application **starts anyway** or **continues to run** without **trying to reconnect**
- However, a **reconnection strategy** can be configured on some connector operations to instead repeatedly try to connect
- A **failsDeploy** attribute can be set to **true** to **throw an exception** if the reconnection attempts fail, which **prevents the Mule application** from **starting**

```
<http:request-config name="HTTP_Request_Config" doc:name="HTTP Request Config">
  <http:request-connection host="https://jsonplaceholder.typicode.com/posts" port="80" >
    <reconnection failsDeployment="true">
      <reconnect frequency="4000" count="5" />
    </reconnection>
  </http:request-connection>
</http:request-config>
```


Transforming data using DataWeave

Transform data structures using DataWeave 2

- DataWeave is a functional programming language to convert input data to new and restructured output data
- DataWeave 2 is the main **expression language** used everywhere in Mule 4 to access, query, and transform data
  - Mule 3 data format transformers like object-to-string are no longer needed
  - In Mule 4, data formats can be converted directly in-place in the event processor attribute's value



DataWeave is an any-to-any transformation language

- All input data types (such as XML, JSON, Java, CSV files) are represented by a common data-type-neutral data model
- All DataWeave expressions are applied to and execute against the DataWeave representation of input data

The **header** contains directives that apply to the body expression

Input	Transform	Output
{ "firstname": "Max", "lastname": "Mule" }	%dw 2.0 output application/xml --- { user: { fname: payload.firstname, lName: payload.lastname } }	<?xml version="1.0" encoding="UTF-8"?> <user> <fname>Max</fname> <lname>Mule</lname> </user>

Delimiter to separate header and body

The **body** contains a DataWeave expression that generates the output structure

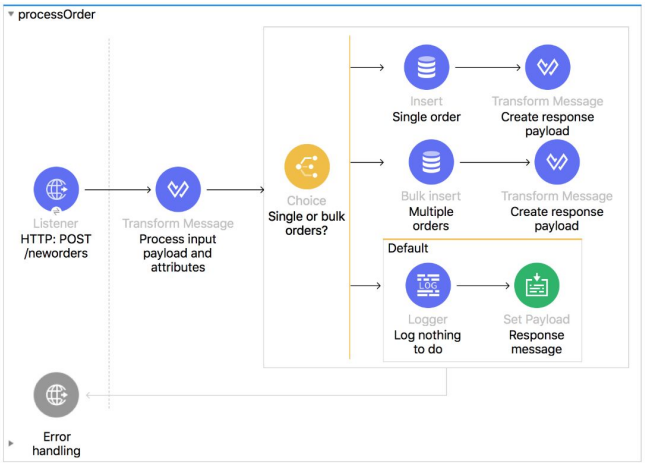
DataWeave expressions always evaluate to a DataWeave type

- DataWeave expressions always evaluate to a valid DataWeave type
  - **Objects**: Represented as collection of key-value pairs
  - **Arrays**: Represented as a sequence of comma separated values
  - **Simple literals**: String, Number, Date, Time, DateTime, Boolean
- The output type of a DataWeave expression is declared in a header
  - Such as XML, JSON, Java, CSV

Routing Mule events between event processors

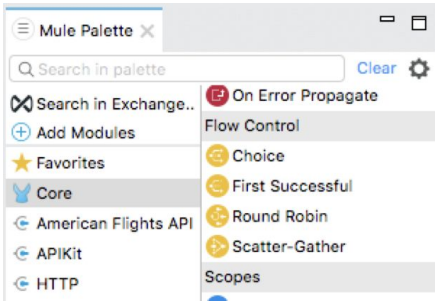
Controlling and routing Mule events in flows using **routers**

- A router uses particular logic to send events to one or more **sequences of event processors** (also called **routes**)



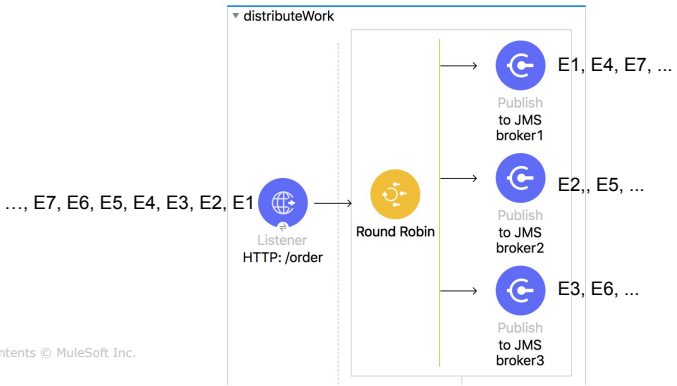
Types of Mule routers

- Choice
  - One route is executed based on conditional logic
  - The logic is a DataWeave expression usually based on the inbound Mule event
- Scatter-Gather
  - All routes are executed in parallel with copies of the same inbound Mule event
- First Successful
  - Each route is executed sequentially until one is successfully executed
- Round Robin
  - One route is executed each time
  - The next route is selected by iterating through a list maintained across executions
- Error handling
  - Re-routes event processing to try to handle various errors



Round-robin Mule event processing

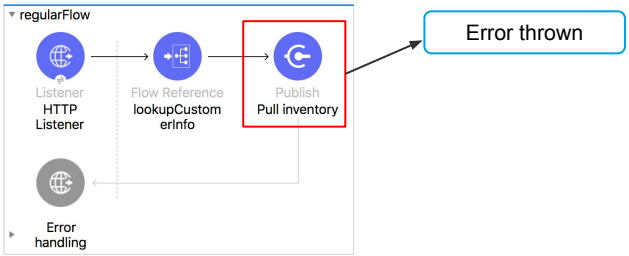
- Routes the incoming Mule event to one of its routes
- A different route is selected each time in a looping round-robin manner
- Each invocation of the Round Robin router is synchronous



Reviewing how errors are generated by flows

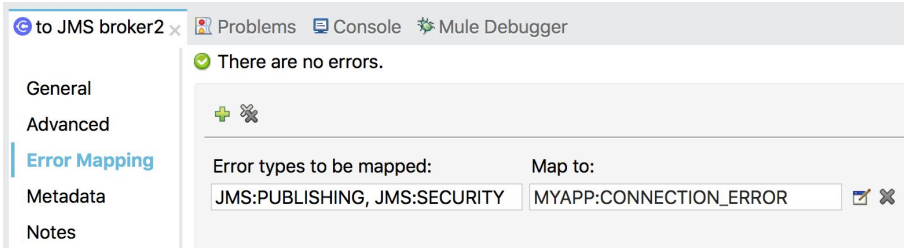
How errors are generated in flows

- Each event processor might throw an error
- The error might originate from a Java library in the event processor's source code, or from a DataWeave expression, or other source
- If an error is not handled, the error message is logged and the flow processing stops
  - If the flow has an event source, a response error message is usually returned



Mapping error types in Mule components

- Many Mule components allow mapping errors to a new type
- The new type is a custom namespace and error type
  - Allows the error to abstract and apply context to underlying failures
  - Cannot use an existing error namespace from another Mule component in the Mule application
  - For example, cannot map FILE:FILE_NOT_FOUND to HTTP:UNAUTHORIZED



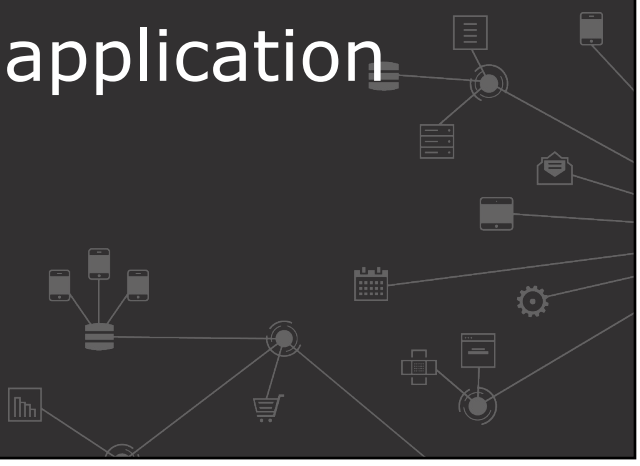
Throwing custom errors

- The Raise Error component can throw a standard or custom error type

```
<raise-error type="ACCOUNT:INSUFFICIENT_FUNDS"
description="#['Cannot transfer $(payload.amount) since only
$(vars.balance) are available.']"/>
```

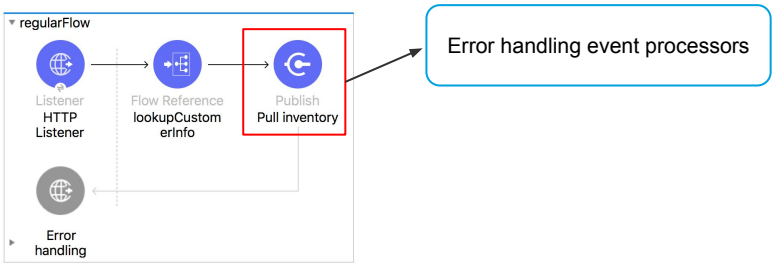
- The description is the message in the thrown error
- The error type must be
  - A core MULE error type (with no namespace)
    - Example: SECURITY (note MULE:SECURITY)
  - A custom namespace
    - Example: MYAPP:BAD_DATE

Reviewing Mule application error handling



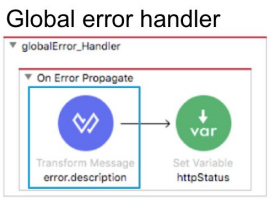
How errors generated in a Mule 4 flow are handled

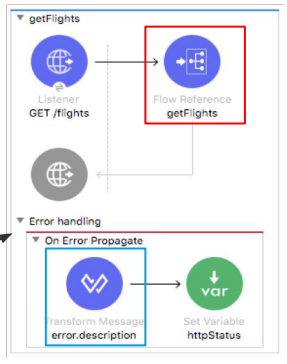
- When an event is being processed through a Mule flow that throws an error
  - Normal flow **execution stops**
  - The Mule event is passed to the first processor in an **error handler**
  - Depending on the type of error handler(s) configured, the error is either caught and handled, or is thrown up the flow invocation chain

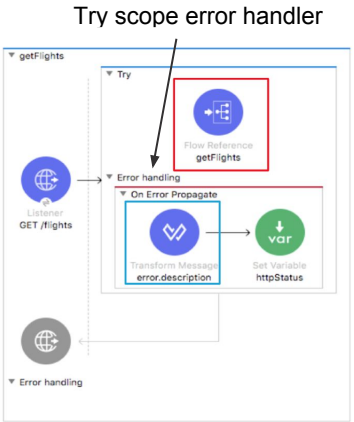


Where an error handler can be added in a Mule application

- An error handler can only be added to
  - An global error handler (outside of any flows)
  - A flow
  - A Try scope
    - A sequence of one or more Mule event
- Not to subflows





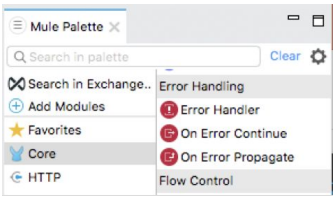


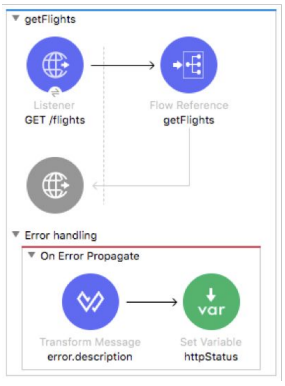
How default error handling works

- If there is no error handler defined for a flow or try scope, a **Mule default error handler** is used
- The Mule application can configure a global error handler as its default error handler
- Otherwise the Mule runtime provides its own default behavior
  - Implicitly and globally handles all messaging errors thrown in Mule applications
  - Stops execution of the flow and logs information about the error
  - Automatically rethrows the error
  - Cannot be configured

Error handling

- Each error handler can contain one or more error scopes
  - On Error Continue scope
  - On Error Propagate scope
- Each error scope can contain any number of event processors



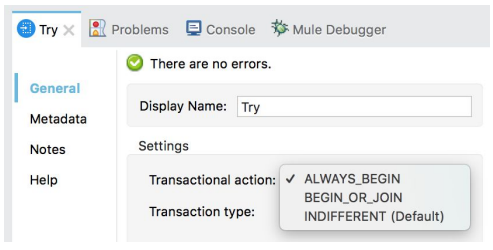


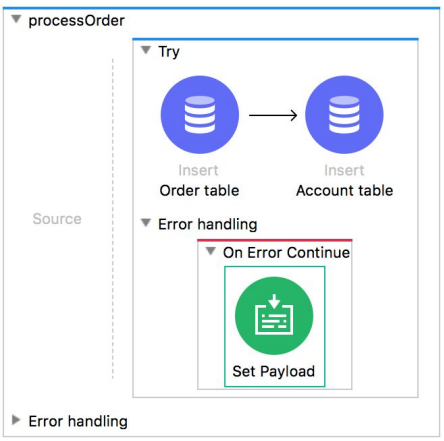
Comparing how the two types of error scopes behave

- **On Error Propagate** scope
  - 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 re-thrown up to the next level and handled there*
  - If the erroring flow starts with an HTTP Listener, it returns an **error** (5xx) response
- **On Error Continue** scope
  - 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 result of the error handler scope is passed up to the next level as if the flow execution had completed successfully*
  - If the erroring flow starts with an HTTP Listener, it returns a **success** (2xx) response

Error handling within a transaction

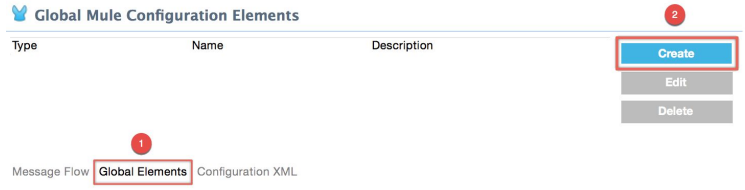
- On Error Propagate scope
  - The error is thrown up the call stack (propagated), so the current open transaction is rolled back
- On Error Continue scope
  - The error is handled, so the current open transaction is committed

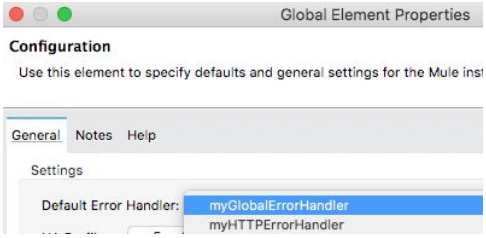




Setting global error handlers for Mule applications

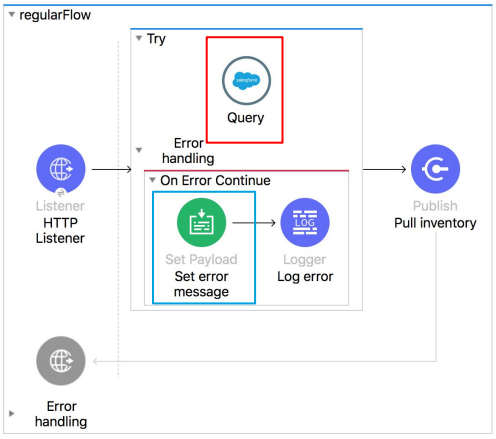
- A global error handler can be configured in a Configuration global element for the entire Mule application
  - This error handler is then used by any flow or Try scope that does not configure an error handler
- A Configuration global element can also be added to a Mule domain
  - Then every Mule application in the Mule domain uses the same global error handler instance





Grouping sequences of event processors with a **try scope**

- Acts like a private flow inside the parent flow
  - Can have its own error handling
  - Unhandled errors bubble up (out) to the parent flow



When and how the system error handler is used

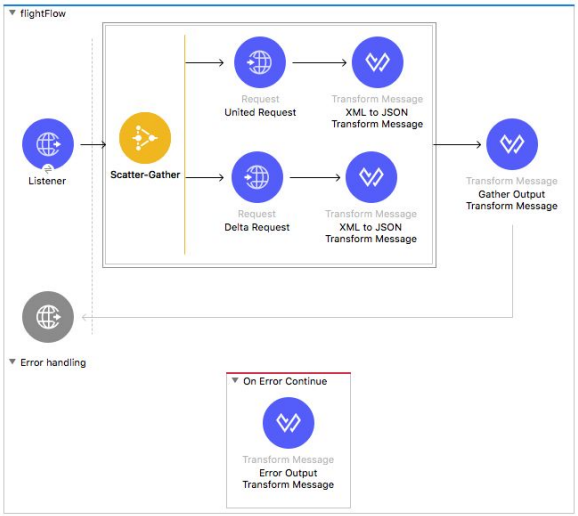
MuleSoft

- When an error does not involve a Mule event, the system error handler is invoked
  - Examples
    - The Mule runtime has errors while starting up, like running out of memory
    - If the connection for a Database connector goes offline or the network disconnects
- The system error handler
  - Logs the error
  - If the error was caused by a connection failure, the connector's reconnection strategy is executed
- These system errors are **not** caught by any flow or Try scope's error scopes, nor by any configured global error handlers

Applying Mule application components to the course case study

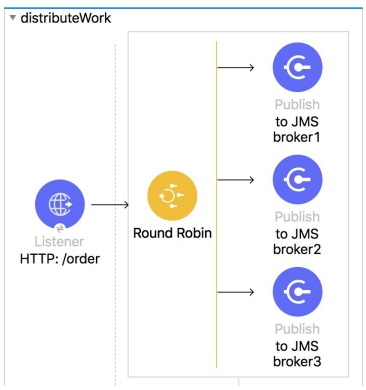
Design Mule applications to consume and process several APIs

- You have now seen the basic building blocks of common Mule applications
  - Connectors
  - DataWeave transformations
  - Flow control
  - Error handlers
- Now you will design solutions for common use cases
- Start with designing how to consume and process two APIs



Exercise 3-1: Design an integration application that consumes and combines two external APIs

- Design Mule flows to retrieve data from two separate APIs
- Design Mule flows to combine two API XML responses together
- Design Mule flows to translate an XML API response to JSON



Exercise context

- GetAways needs to integrate flight information into their travel bundles
- This project is similar to the Development Fundamentals use case
- In this case though, the United and Delta airline APIs expose flight information as a REST service with XML formatted responses

Exercise goals

- You need to design an integration solution that fulfills these functional requirements
  - Call the United and Delta airline REST APIs
  - Process each flight API's result information
  - Combine the airline API XML responses together
  - Return the combined response as JSON
- For this exercise, do not worry about particular SLAs relating to latency or data throughput
  - You will address SLAs and other NFRs later in the course

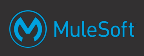
Exercise steps required to build the solution

- Identify flows to fulfill all the integration requirements
- Identify Mule components for each flow, and their interactions across flows
- Identify how to transform and combine data
- Identify error handling for the solution
- Confirm your design meets the integration requirements
  - Call the United and Delta airline REST APIs
  - Retrieve flights information from each airline's API
  - Combine the airline API XML responses together
  - Return the combined response as JSON

Exercise steps: Design the initial data gathering stages

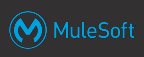
- Identify components to use in the integration flows
  - How should the entire integration process be kicked off, and what are the tradeoffs of each option?
  - What components should retrieve data from the two REST APIs?
  - What components can be used to orchestrate the integration of the responses from the REST APIs?
  - Implement a POC of the integration solution using flow designer or Anypoint Studio
    - Do not worry about connecting to any real data source

Exercise steps: Design the initial data gathering requirements



- Integration requirements
  - Call the United and Delta airline REST APIs
  - Retrieve flights information from each airline's API
  - Combine the airline API XML responses together
  - Return the combined response as JSON

Exercise steps: Design the data aggregation and post-processing stage



- How does the data from each REST API need to be transformed, if at all?
- What are options for combining the flights data together, and what are the tradeoffs of these options?
- What Mule components or custom code are needed for these options?
- Pick the most idiomatic (used for its intended purposes) options for this use case
- What are the best options to transform the XML responses to JSON

Exercise steps

- Decide on error handling options
- Analyze the relative merits of each error handling option
  - Add error handling to your POC flows
- Analyze how you could reuse error handling logic between flows
- Document the tradeoffs of different types of error scopes in your POC
- If an external client triggers the integration process, how are error statuses returned to the client?

Exercise steps

- Identify scopes to use in the integration flows
  - Identify interdependencies between components in the Mule application
  - Can we process independent components in a different construct?
  - Analyze the impact of these optimizations

Exercise worksheet

MuleSoft

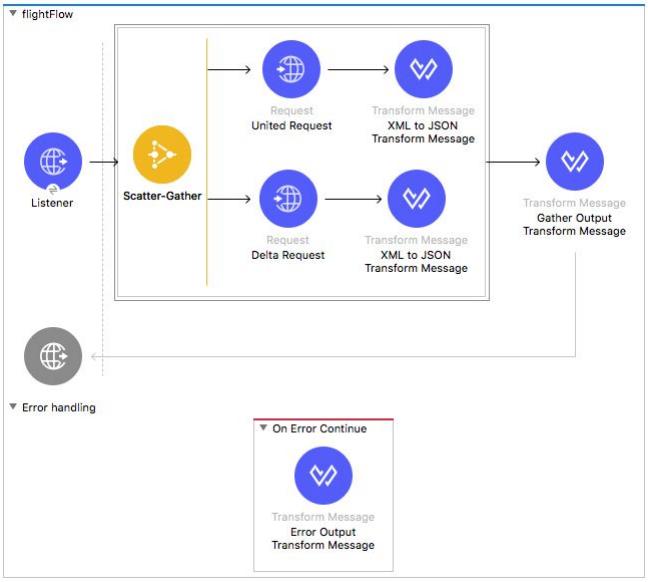
Question	Current solution	Ways to improve
How does the data from each REST API need to be transformed, if at all?		
What are options for combining the flights data together, and what are the tradeoffs of these options?		
What Mule components or custom code are needed for these options?		
Pick the most idiomatic (used for its intended purposes) options for this use case What are the best options to transform the XML responses to JSON		

Exercise worksheet

MuleSoft

Related components	Independent components	Ways to improve the design	Impact of the design change

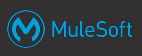
Exercise solution



Exercise solution

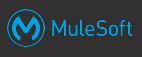
- The flight request is split through a Scatter-Gather
  - The United and Delta web services are called in parallel
  - Each response is transformed to a common Flights JSON schema
  - After both transformed web service responses complete, the result is automatically gathered into a single Mule event
- After the Scatter-Gather, the gathered responses are further transformed to flatten each flight result into one common object
- Errors are handled inside the flow
- The On Error Continue allows this flow to handle its own errors
  - This will hide the error if this flow is called from a parent flow

Exercise reflection: Analyze your solution choices against other options



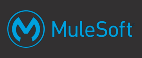
- Did you use a generic HTTP or REST connector, or is there a more targeted connector available in Anypoint Exchange?
  - Document your reason for your connector choices
- Did you use a Scatter-Gather component to perform the two API requests in parallel?
  - Document your reason for this
  - If you used a Scatter-Gather component, how does this affect the error handling?
- How are the two API responses transformed to JSON?
  - Did you pre-process each API response first, then combine to JSON, or did you transform them all at once in a single transformation?
  - Document the tradeoffs for your data transformation choices

Exercise reflection: Analyze error handling choices and tradeoffs



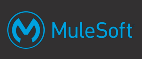
- How did you handle errors?
  - Document the tradeoffs to using a common global error handler vs. more localized error handling (such as with Try scopes)
- Did you use any error mappings?
  - Document your reasons to use or not to use error mappings
  - Document your reasons for your choices of return statuses

(Optional) Exercise 3-2: Design an application that one-time-loads data into a database from files



- Design a Mule application to load data from a file into a database.

Exercise 3-3: Design an integration application that synchronizes a database and SaaS system



- Design a Mule application to load data from a file into a database
- Design a Mule application to poll a database for changes
- Design a Mule application to only transform new database entries
- Design Mule flows to translate a database response to Salesforce operations
- Design Mule flows to call Salesforce operations in bulk

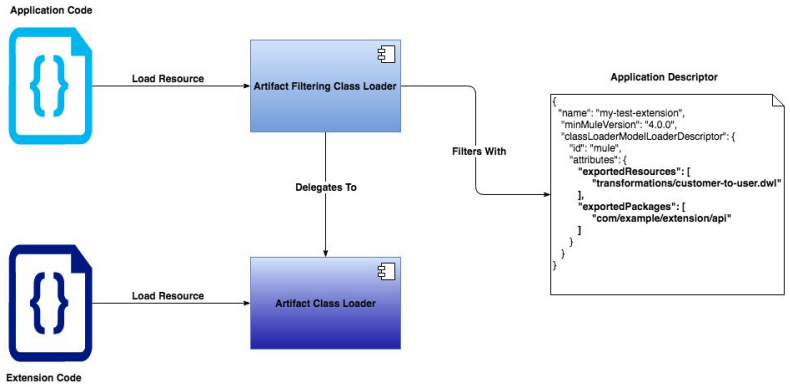
Exercise step

- Continue to design Mule applications to meet the course case study
- Perform a similar analysis as Exercise 3-1 for the DB to SFDC synchronization use case
  - Identify where Mule application components can help integrate between the database and the Salesforce systems
  - Identify which connectors can be used
  - Identify where schedulers or batch jobs might be applied
  - Identify the types of data transformation that may required, and how DataWeave can help
  - Identify how errors are handled

How Mule 4 isolates class loading

Mule 4 class loading isolation

- Mule 4 uses separate class loaders to isolate the Mule runtime, Mule applications, and modules from each other
- Each class loader specifies the packages (rather than classes and resources) to export as part of the module's interface



Class loading isolation - Mule 4

- Artifact class loader
  - A regular Java class loader pointing to the JAR files included in the module
  - This class loader will load all files and classes of the module
- Artifact filtering class loader
  - A wrapper created over the Artifact class loader, which will enforce the access restrictions to the module's code for foreign artifacts (the app or other plugins)
  - It uses the content of the mule-artifact.json descriptor to determine what is public
- Module code
  - Plugin, modules and connectors
  - Uses Artifact class loader (which does not have any restriction), and it is only able to locate resources of the plugin itself
- Application code
  - Mule app
  - Uses the Artifact Filtering class loader of the module to prevent the app from accessing restricted code or resources.

Summary

Summary

MuleSoft

- Mule applications are built from various event processors
- Mule events include the message and variables
- The event's message includes a payload (which includes attachments) and attributes
- Event processors include connectors and Transform Message components
- Event processors can be grouped into various scopes for flow control, async processing, batch processing
- Flows can configure various hierarchies of event handlers