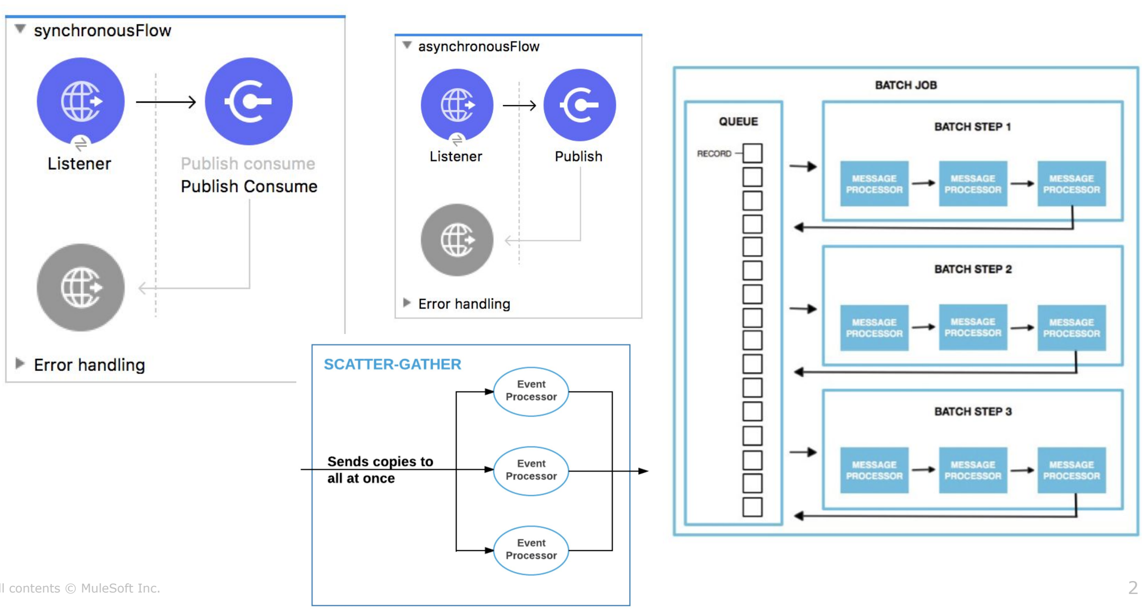
MuleSoft

Module 4: Choosing Appropriate Mule 4 Event Processing Models

Goal

MuleSoft



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

- Identify and distinguish between Mule 4 event processing models
- Identify what event processing models are used in various Mule 4 scopes and components
- Identify Mule 4 streaming options and behaviors
- Select appropriate processing models for a particular use case

Introducing Mule 4 event processing models and options

Abstracting event processing options into processing models

- **Processing models** collect together options and behaviors related to Mule event processing by Mule application components and flows within a Mule runtime
- Event processing by **Mule scopes** can be modelled to describe
  - How Mule components can process a copy of the same Mule event in parallel, then combine the results
  - How Mule components process a Mule event that contains a collection of records, sequentially or in parallel
- Event processing by **Mule connectors** can be modelled to describe
  - The behavior of inbound Mule event sources at the beginning of a flow
  - How message queuing connectors process Mule events

Modelling Mule 4 event processing

- These models describe the **event processing** behavior of
  - **Non-blocking** and **concurrent** Mule event processing by the Mule 4 runtime
  - **Synchronous** Mule event processing by Mule 4 components and flows
  - **Asynchronous** Mule event processing by Mule 4 components and flows
  - **Parallel processing** of Mule events by Scatter-Gather components
  - **Streaming processing** of larger-than-memory Mule events
  - **Iterative processing** of Mule events containing collections of records
  - **Real-time** and **scheduled** event processing of Mule events

Describing non-blocking and reactive event processing by Mule 4 runtimes

What is **reactive programming**?

- A declarative programming paradigm that combines **concurrency** with **event-based** and **asynchronous** systems
- Popular with web-based distributed systems

References:

https://en.wikipedia.org/wiki/Reactive_programming

https://www.reactivemanifesto.org/

Features of reactive programming and reactive streaming

- Uses the best ideas from the Observer pattern, the Iterator pattern, and functional programming
- Deals with an ongoing and building **event stream**
  - Rather than one complete, static, all in-memory data collection
- Is an **asynchronous**, **non-blocking**, and **declarative** programming style
  - Avoids the "callback hell" brought about by Java's imperative programming approach
- Incorporates handling of **back-pressure**
  - Automatically slows down Mule event producers if event consumers are being overwhelmed by the rate of events

The Mule 4 runtime uses a non-blocking and reactive processing model

- **Non blocking** is a central theme in Reactive principles
- Non blocking is the norm in Mule 4
  - Every flow has top level support for **non-blocking** operations
  - Although many connector operations are blocking, the Mule 4 runtime makes it easy to develop non-blocking operations
    - For example, the HTTP connector is non-blocking
  - **Threads do not block** waiting for IO intensive operations to complete
  - Unlike Mule 3, the developer does **not** need to assign a **processing strategy** to each flow

The Mule 4 reactive streams use common global thread pools

MuleSoft

- In Mule 3, each flow had its own thread pools, SEDA queues, etc.
- Flows had to specify a processing strategy
- In Mule 4, there are three global executors (schedulers) that run all tasks in the Mule runtime
- The Mule runtime's attempts to automatically infer the execution type of each event processor
  - BLOCKING - Operation requires a connection and is blocking
  - CPU_INTENSIVE - Operation requires a connection and is non-blocking
  - CPU_LITE - Otherwise
- The developer of a connector can also explicitly annotate the execution type

The three possible execution types that can be set in an Anypoint connector's source code

MuleSoft

- These are set as annotations in the Java classes used to build connectors using the Mule SDK
  - IO_INTENSIVE (BLOCKING)
    - Blocking processing that performs blocking IO operations, all blocking **IO-intensive connectors (e.g. DB via JDBC), Transaction scopes, Lock.lock(),** or any other technique that blocks the current thread during processing without exercising the CPU
  - CPU_INTENSIVE
    - Intensive processing, such as complex **time-consuming calculations, scripting, or DataWeave transformations**
    - This processing type is never automatically inferred by the Mule runtime
  - CPU_LITE
    - Processing that neither blocks nor is CPU intensive, such as **message passing, filtering**, **routing**, or non-blocking IO

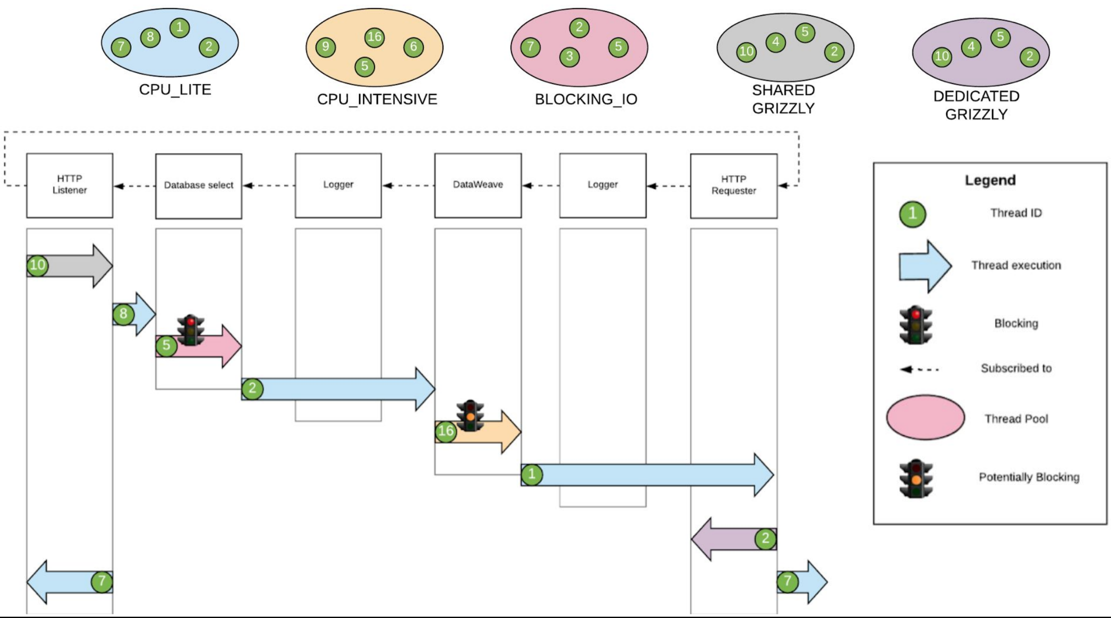
Advantages of the non-blocking Mule 4 runtime

- The Mule 4 runtime leverages **non-blocking IO calls** to avoid performance problems
  - Removes complex tuning of thread pools and their sizes requirement to achieve optimum performance

How the Mule 4 runtime self tunes Mule application performance

- The Mule 4 runtime uses a **scheduler execution** service to automatically optimize performance
  - The Mule 4 runtime provides three types of thread pools for use by the event processors in a Mule application's flows
  - The scheduler service assigns each event processor to one of these three thread pools

Thread switching between event processors in a Mule 4 application



Example: Introducing the HTTP connector processing model

- Mule HTTP connectors use Grizzly under the covers
- Grizzly uses **selector** thread pools
  - Selector threads check the state of the non-blocking IO (NIO) channels and creates and dispatches events when they arrive
  - HTTP Listener selectors poll for **request events only**
  - HTTP Requester selectors poll for **response events only**
- The **HTTP Listener** uses a **shared selector pool** provided by the Mule runtime for **all** Mule applications
- The **HTTP Requester** has a **dedicated selector pool** local to **the** Mule application

Reflection questions

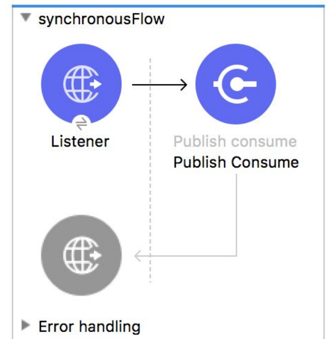
- Give some examples of components you feel should be CPU_INTENSIVE, and why?
- Give some examples of components you feel should be CPU_LITE, and why?
- Give some examples of components you feel should be IO_INTENSTIVE (non-blocking IO), and why?
- What would need to be done if the Mule runtime could not self-tune these component execution profiles?

Describing synchronous Mule event processing

Features of synchronous single-threaded processing of Mule flows

MuleSoft

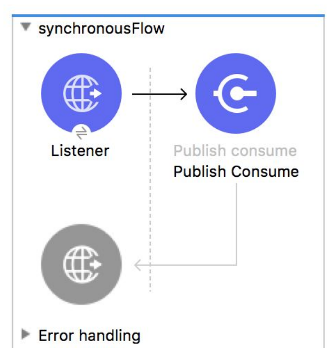
- All processing is performed in a **single thread**
- The main thread **blocks waiting** for the called worker thread to complete execution
- The entire request and response is handled synchronously in **one thread**



How transactions are processed

MuleSoft

- **Transactions** are **processed** in **synchronous** mode
  - A transaction scope can be for an entire flow, a Try scope, or an individual connector
  - If a Connector listener operation starts a transaction, the transactional scope is for the entire flow

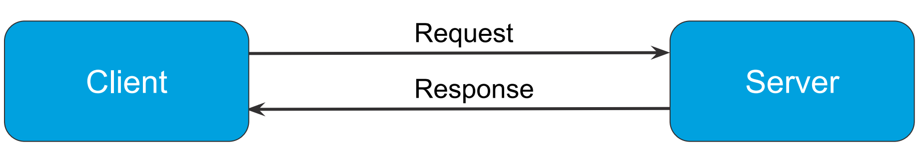


Describing synchronous Mule event processing of web services

Understanding the Hypertext Transfer Protocol (HTTP)

MuleSoft

- A **synchronous** and **stateless** communication protocol
- Can transfer **hypermedia** type data
  - Hypermedia is structured text with links to other documents, details, or media
- Built on top of the TCP **client/server** communication protocol
- An HTTP request **message** is submitted from a **client** to a **server**, then the server returns a response back to the client



The structure of an HTTP request and response

MuleSoft

- The response **status line** with the protocol version, status code, and status message
- HTTP **response headers**
- A **response body** (optional)

- The response **status line** with the protocol version, status code, and status message
- HTTP **response headers**
- A **response body** (optional)

Request

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

status line

start-line

HTTP headers

empty line

body

Response

```
HTTP/1.1 403 Forbidden
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Date: Wed, 10 Aug 2016 09:23:25 GMT
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
Age: 3464
Date: Wed, 10 Aug 2016 09:46:25 GMT
X-Cache-Info: caching
Content-Length: 220

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
(more data)
```


Understanding the Representational State Transfer (REST) protocol

MuleSoft

- An **architectural style**
  - To provide **interoperability** between computer systems over the internet
  - consists of a coordinated set of components, connectors, and data elements within a **distributed hypermedia system**
  - Does not specify any underlying communication protocol to use
    - Although in practice RESTful systems **always communicate over HTTP**

Architectural constraints that make a system RESTful

MuleSoft

- Addressable resource
  - A resource is any information or concept that can be **named**
  - A resource identifier should **represent only one resource**
- Stateless
  - The server does **not store application state**
  - Application state is to be managed by the client
- Connectedness
  - The server guides the client to change state
  - This is done through **Hypermedia**, and also through headers in the case of HTTP
- Uniform interface
  - **Resources share interface characteristics**
- Idempotency
  - The same **HTTP request** should always create the same server state

In practice, RESTful services always use HTTP as a uniform interface

MuleSoft

- Resources are operated on using standard HTTP methods
  - Unlike SOAP, this standardizes **C.R.U.D** operation names in every API
- HTTP methods can support RESTful operations
  - In practice, many REST API implementations violate some RESTful principles
  - For example, not using Hypermedia (links) to guide clients across requests

HTTP method	Intended CRUD operation	Comment	Typical response status codes
POST	**C**reate		201 Created / 405 Method Not Allowed
GET	**R**ead	Idempotent	200 OK / 404 Not Found
PUT	**U**pdate/Replace	Idempotent	200 OK / 204 No Content
DELETE	**D**elete	Idempotent	200 OK / 204 No Content
PATCH	Partial update/modify		200 OK / 204 No Content

Reference: https://www.w3.org/Protocols/rfc2616/rfc2616.txt

Understanding how the SOAP messaging protocol compares with REST

- A specification for **exchanging structured information (messages) via web services** in computer networks
  - Focuses on exposing pieces of application logic (not data) as services
  - Typically used for communication between applications
- SOAP is based on the XML and W3C standard
- Both SOAP and REST support SSL

How REST methods compare with SOAP operations

- SOAP focuses on exposing and accessing **named operations**, while REST focuses on accessing resources via standard HTTP methods
  - Unlike REST, operations are not tied to the HTTP method, and can have any arbitrary name
    - All SOAP operations are usually performed via HTTP POST methods
  - Unlike REST, operation names must be invented and typically differ for every SOAP API
  - Each operation usually describes the implementation of some business logic, so the context changes between different SOAP APIs

Popular SOAP extensions

- Many SOAP extensions begin with the prefix WS-, and are collectively referred to as WS-* standards
- **WS-Security**
  - Adds integrity and confidentiality to SOAP messages using, respectively, **XML Signature** and **XML Encryption**
- **WS-AtomicTransaction**
  - Supports **ACID** Transactions over a service
- **WS-ReliableMessaging**
  - Provides **asynchronous processing** and a **guaranteed level of reliability** by retrying in case of communication failures
  - Provides various levels of **qualities of service** (**QoS**) for the delivery assurance of messages

Reflection questions

- Why would you develop a new SOAP service instead of a REST service?
- What are the tradeoffs of using SOAP vs. REST?
- How are data schemas validated and enforced in SOAP vs. REST?
- Does REST support XML payloads?
- What types of data formats are supported by SOAP vs. REST?
- What are some other common synchronous data processing models, and what are the tradeoffs compared with REST or SOAP?
- Compare how errors are communicated in REST vs. SOAP vs. other event processing models?
- How are error status messages returned to clients in SOAP vs. REST, and what are the tradeoffs between these two styles?

Streaming larger-than-memory Mule events

Introducing streaming processing models

- **Streams** are data structures that provide efficient processing of large data objects such as files, documents, and records, by processing data **continuously as it arrives**
  - This is **different** behavior compared with other in memory data structures that **hold all values in memory** before computing
  - Allows large datasets to be processed without **running out of memory**
- The Mule 4 runtime **automatically** streams large data payloads without any special configuration
- The default streaming option varies for the Mule EE and Community Edition

How connectors support repeatable streams

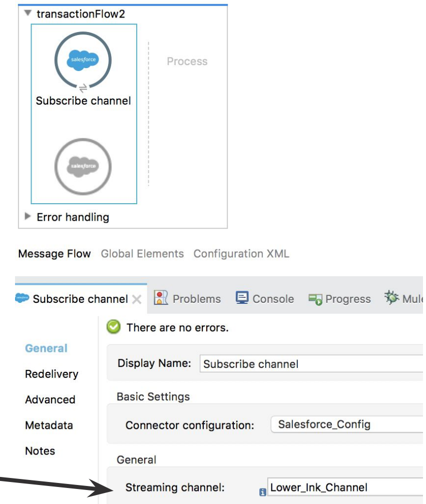
- Consumers receive individual cursors
  - Provides repeatable and concurrent random access to repeatable streams
  - Only for Mule 4
- Every component in Mule 4.0 that returns an InputStream or a Streamable collection supports repeatable streams
- Some connectors that support repeatable streams include
  - File
  - FTP
  - DB
  - HTTP
  - Sockets
  - Salesforce

Identifying streaming options for supported Mule 4 connectors

- **File stored repeatable** streams
  - **Default for Mule EE**, and only available in Mule EE
  - By default stores 500 objects in its in-memory buffer
    - This number can be configured
  - Excess objects are serialized using a Kyro serializer that writes to disk
- **In-memory repeatable** streams
  - **Default for Mule Community Edition**
  - Uses a default max buffer size of **500 objects**
  - The buffer is expanded from an initial buffer size (default 100), in a default increment size of 100 objects, **until** the **max buffer size** is reached
  - If streams exceed the max buffer size, then the application fails
- **Non Repeatable** streams
  - The input stream is only **read once**
  - No extra memory or performance overhead compared with repeatable streams

Describing the Salesforce streaming processing model

- Salesforce provides a **Streaming API**
  - To receive notifications from Salesforce on a PushChannel about modified Salesforce data
    - A PushChannel is defined and customized with a valid Salesforce Object Query Language (SOQL) query
  - Reduces the number of requests that return no data from the Salesforce server
    - By pushing rather than polling for changes
- Mule applications can listen to Salesforce notifications by
  - Subscribing to a Salesforce streaming channel
    - The subscription is handled through a Salesforce connector's **streaming channel** operation configuration



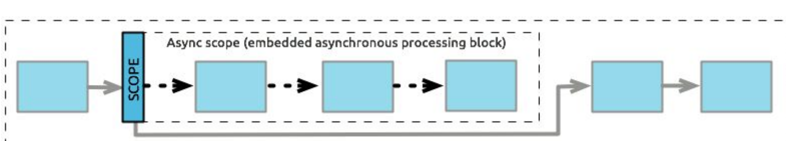
Reflection questions

- What are the tradeoffs between using a file stored repeatable stream (that uses Kryo) vs. an in-memory repeatable stream?
- What type of use cases would benefit from repeatable streams?
- When would a non-repeatable stream be helpful or necessary?

Describing asynchronous Mule event processing

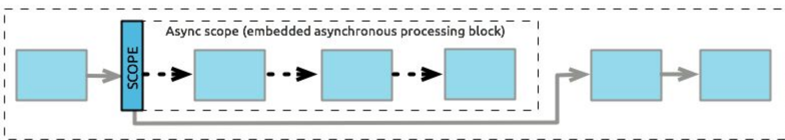
Asynchronous processing models

- The main thread **does not wait** for its worker threads to complete execution
- Uses branch processing that executes separate worker threads simultaneously with the main flow's thread(s)
- **Failure** in one of the branches does **not** impact the main flow
- **Responses** from branch processing are **not** available to the main flow



Distinguishing between consumable and non-consumable event payloads

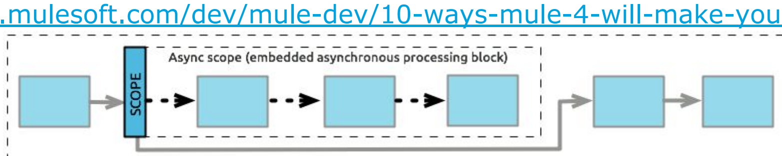
- Mule events can contain consumable or non-consumable payloads
- A consumable payload **cannot be re-read**, so there is never contention between asynchronous processing of the same Mule event
- Non-consumable payloads **can be re-read** between threads, so there could be a race condition between threads or flow routes



Mule 4 simplifies side effects from non-consumable payload

- In Mule 4, The Mule runtime's repeatable streaming hides all the complexity of **sharing Mule events between threads and routes**
- Unlike Mule 3, you don't have to worry about
  - Which components are streaming and which are not?
  - Is the stream consumed at a particular point?
  - At which position is the stream at anyway?
  - What does streaming even mean?
- For rare corner cases, repeatable streaming can be disabled
  - In which case you will need to again worry about these issues

https://blogs.mulesoft.com/dev/mule-dev/10-ways-mule-4-will-make-your-life-easier/

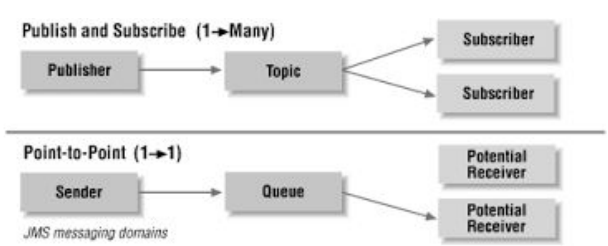


Describing asynchronous Mule event processing using JMS

Understanding the **Java Message Service** (JMS) message processing model

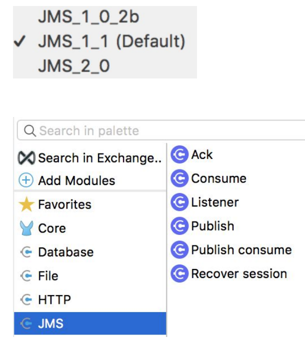
MuleSoft

- JMS is a Java standard
  - Specifies how to exchange messages between **producers** and **consumers** through an intermediary **message broker (also called the JMS provider)**
- Supports **one-to-one and one-to-many exchanges o**f messages
- Messages can be stored and forwarded in the message **broker**
  - To support **asynchronous** message exchange patterns



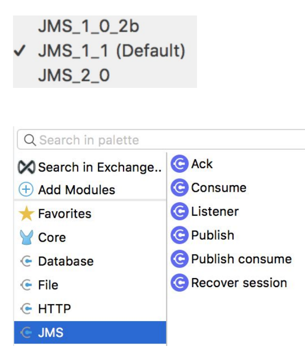
Using a JMS connector in a Mule application

- Connects to an external JMS provider (also called a message broker)
  - The JMS connector relies on a JMS client library for a particular JMS provider
  - This library is added to the Mule application as a Maven dependency
  - Simplifies Mule application development by providing a standard interface for creating **destinations** and exchanging messages through destinations



Types of message exchanges supported by an Anypoint JMS connectors

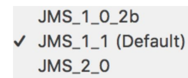
- Supports message exchanges with both types of JMS destinations
  - **Queues** (Point to point messaging model)
  - **Topics** (one to many publish/subscribe messaging model)

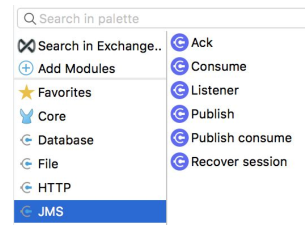


Types of message exchanges supported by an Anypoint JMS connectors

MuleSoft

- Supports both **synchronous** and **asynchronous** communication
  - Publish sends a message for asynchronous delivery
    - Immediately moves forward to the next event processor and never gets a response back from the JMS provider
  - Consume operation performs a blocking receipt of a message, anywhere in a flow
  - Listen operation is an Event source that triggers the flow upon receipt of a message
  - Publish consume operation is synchronous
    - Blocks the flow until a response is returned from the JMS provider or a timeout expires
    - Combines publish and immediate (blocking) consume operations

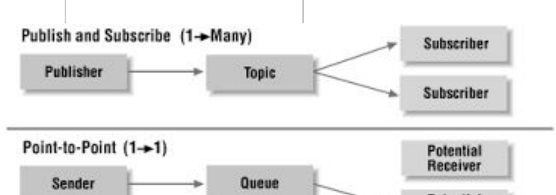




Distinguishing between how JMS queues and topics handle messages in asynchronous communication

MuleSoft

	Queue	Topic
Typical use cases	Work distribution	One-to-many broadcasts
Number of connected consumers allowed	Zero or more	Zero or more
Who receives message	One receiver	All active subscribers
If no active subscribers	Message persists	Message dropped if subscriber is not durable

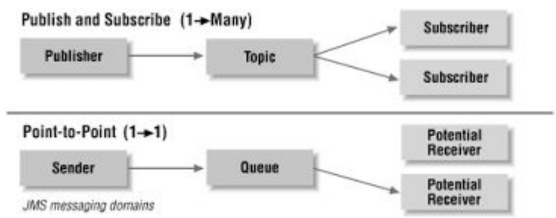


JMS messaging domains

The order that messages are delivered to consumers from a JMS queue

MuleSoft

- The JMS provider queues messages **in the order** they are received
- For a **single consumer**, the JMS provider dispatches messages to the connected consumer, **in order**
- If there are multiple message consumer instances consuming from the same queue (whether in the same JVM or not) then
  - Messages are no longer guaranteed to be processed in order
  - This is because messages will be processed concurrently in different threads or different processes

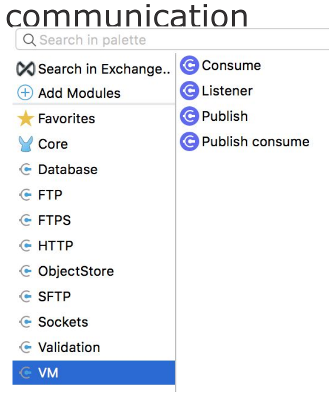


JMS messaging domains

Describing asynchronous Mule event processing using VM queues

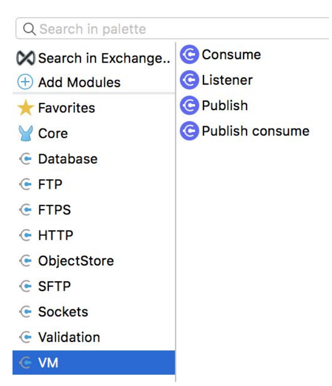
How VM connectors work

- Unlike JMS, VM queues do not use any intermediate message broker
- Creates and communicates with virtual machine (VM) queues using a publish/consume messaging model
- Supports intra-app and inter-app (in Mule domain) communication through queues that can be transient or persistent
- Supports sync or async communication



How persistent VM queues are implemented

- On a single customer-hosted Mule runtime instance, persistent VM queues work by serializing and storing the contents on the disk
- In a cluster of customer-hosted Mule runtimes, persistent queues are backed by the Hazelcast distributed data grid
  - Can be configured to be only in-memory or also persisted to disk
- In CloudHub, VM queues are stored in a CloudHub service
  - Persistent queues retain data after the Mule application restarts



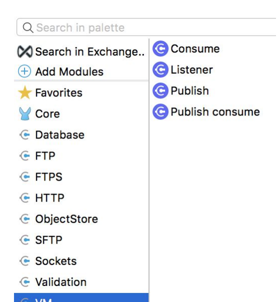
How Mule 4 serializes objects

- Mule 4 EE supports a Kryo based alternative to default Java serialization to serialize objects to improve performance in particular use case
  - Read/write from a persistent ObjectStore
  - Read/write from a persistent VM or JMS queue
  - Replicating an object across a Mule cluster
  - Read/write an object from a file
- Note: Kryo is only used by some components (Batch, repeatable streaming) by default, but can be set as the default Serializer throughout the Mule runtime"

https://blogs.mulesoft.com/dev/tech-ramblings/need-77-performance-boost-no-problem-with-mule-3-7-using-kryo/

Comparing VM queues with other messaging solutions

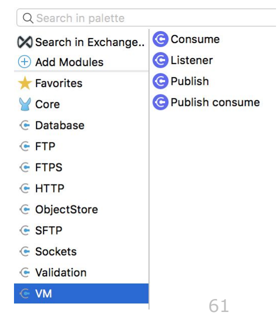
- Many Mule applications use an external messaging system like Anypoint MQ or a JMS provider
- With VM queues, reliability and quality of service is limited
- VM queues can be used for specific use cases
  - To distribute messages (work) across a cluster of Mule runtimes
  - For high-performance async communication within a Mule application
  - In CloudHub, to distribute work within the same Mule application across multiple CloudHub workers, perhaps with lower latency compared to other messaging solutions
  - When investment in a JMS broker or other messaging system is not justified or supported



Comparing VM queue load balancing compared to other messaging solutions

MuleSoft

- When a Mule application is deployed to a customer-hosted cluster or multiple CloudHub workers, messages are consumed in a fast, non-deterministic way
  - Not strictly round-robin
  - One node (worker) might get a sequence of messages
  - The load balancing algorithm cannot be changed or tuned
- Full messaging solutions are often more flexible
  - May allow the load balancing algorithm to be changed or tuned
  - Often allow messages (work) to be distributed to different applications, even non-mule applications



Mule 4 event processing models for asynchronous messaging

MuleSoft

- Implemented using specific messaging connectors
  - Such as the connectors for **Java Message Service** (**JMS**), **Anypoint MQ**, or **VM**
  - Each of these **external messaging systems** has their own concept of a message and message structure
  - In Mule flows, the messaging structure is transformed to and from a **Mule event**
- Depending on the messaging connector type, messages might be exchanged
  - Through an **intermediate message broker**
  - Or directly in Mule application memory (such as with VM queues)
- **Consumers don't need** to be connected at the time the message is produced

JMS extensions and guarantees for clusters of consumers

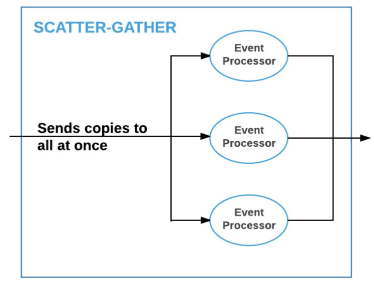
- Runtime behaviour varies by JMS provider, but is transparent to the Mule JMS Connector
  - Some JMS providers allow a queue to be configured as an **exclusive queue**
  - An exclusive queues guarantees each message is delivered at least ONCE to one of the cluster's consumers
  - Then message processing is automatically bound to one connected consumer, called the **exclusive consumer**
    - This one consumer processes all the queue messages, in order
  - In case of failure of the primary node, another node will be elected as the exclusive consumer and will take over processing the rest of the messages in the exclusive queue

Describing parallel Mule event processing

Describing the Scatter-Gather processing model

MuleSoft

- The Scatter-Gather component is a routing event processor
  - Processes the Mule event **in parallel in multiple routes**
  - Parallel execution of routes may **improve performance**
  - Mule 4 repeatable streaming eliminates any possible contention of the Mule event as it is processed through each route
  - Just like in any Mule flow, if an event processor in any route consumes any part of the Mule event, a new Mule event is created for that route
- The Scatter-Gather completes after every route completes, or after a configured timeout expires



Iteratively processing collections of records

Processing collections all at once using a connector's batch operation

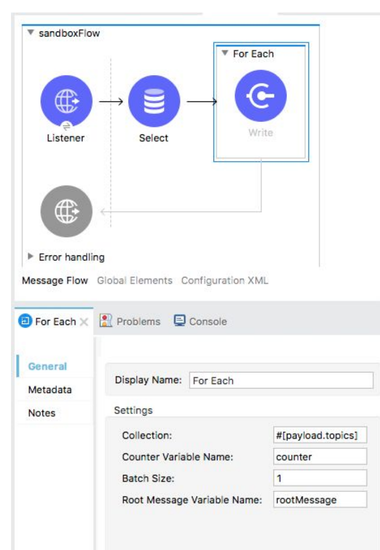
MuleSoft

- Some connectors support batch operations
  - Collections of items can be batched together and sent as a single event to the connector

Using For Each scopes to process collections of items

MuleSoft

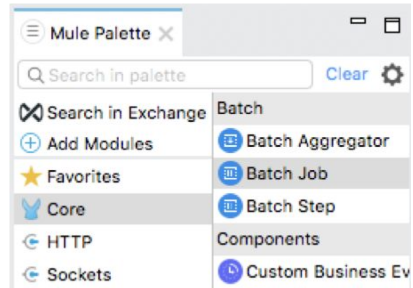
- The For Each scope is configured with a Collection object
  - Synchronously processes one item at a time from the Collection
  - Cannot stream the payload
- The payload after the For Each scope returns to the unprocessed payload before the For Each scope



Process batch jobs asynchronously using a Batch Job scope

MuleSoft

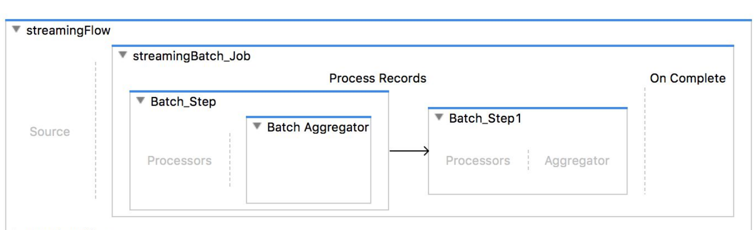
- Provides ability to split large messages into records that are processed **asynchronously** in a batch job
- Created especially for processing data sets
  - Splits large or streamed messages into individual records
  - Performs actions upon each record
  - Handles record level failures that occur so batch job is not aborted
  - Reports on the results
  - Potentially pushes the processed output to other systems or queues
- **Enterprise edition only**



Describing the Batch Job processing model

MuleSoft

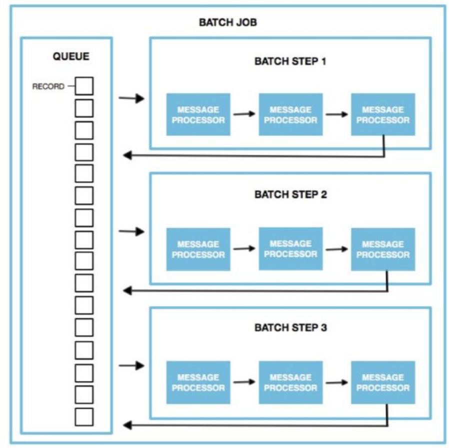
- Only available with Mule Enterprise Edition (EE) runtimes
- Processes individual records from a collection across one or more batch steps, one step at a time
- Provides the ability to split large messages in a "splittable" format such as a collection or an array, streaming or not, into records that are processed asynchronously in a batch job
  - A certain number of records can be aggregated together and sent all together to other systems or queues



How Batch Jobs process records

MuleSoft

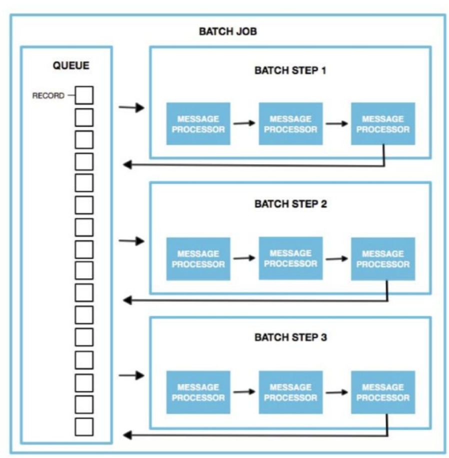
- From the input payload, a **fixed block of records** (a batch) is sent through the batch job for processing by all the batch steps
- All the batch records are first queued
- Records are taken from the top of the queue, one at a time, and sent to the first batch step
- Several threads may process multiple records in parallel



How records can jump ahead to later batch steps

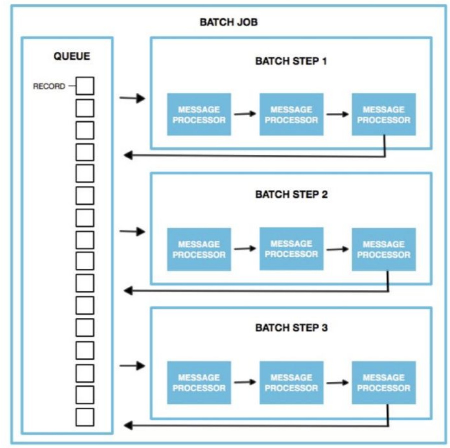
MuleSoft

- After completing the batch step, the record is put on the **bottom** of the queue
- Records are always processed, in order, by one particular batch step
- But later records can **jump ahead** to the next batch step while earlier records are still being processed by another thread in the previous batch step
- In this way, some later records in the original record queue may complete traversing through the entire batch job before earlier slower records finish



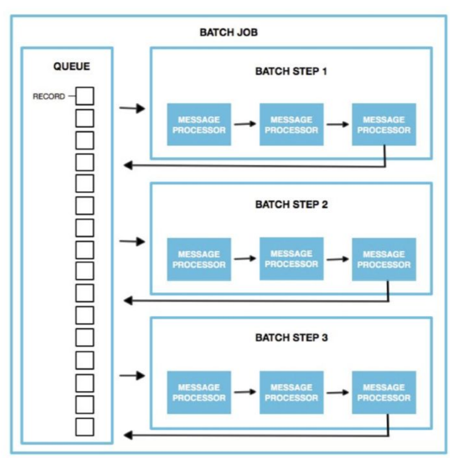
Filtering which batch steps a record uses

- Batch step filters with **accept Expressions** are used to process records that didn't fail during the previous batch step
- Other filters can configure a batch step to handle previously failed records



How a batch job completes

- Successful execution of a batch job creates a **BatchJobResult** object
  - Contains a summary report about the records processed for the particular batch job instance
  - Does not include any of the actual processed records or data
  - The payload after a Batch Job is the original payload before entering theBatch Job



Reflection questions

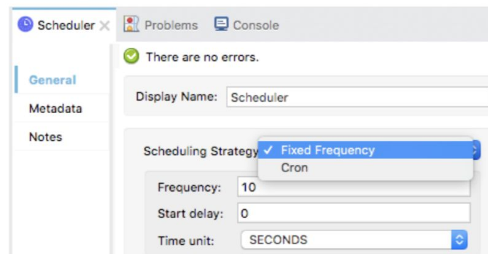
- What is the difference between using a Batch Job vs. calling a For Each scope that contains an Async scope?
- Can a record in a Batch Job complete a later Batch Step before finishing a previous Batch Step?
- Can a later record in a Batch Job complete before an earlier record completes?

Synchronizing data with real-time and scheduled (batch) Mule event processing

How Scheduler components processing Mule events

MuleSoft

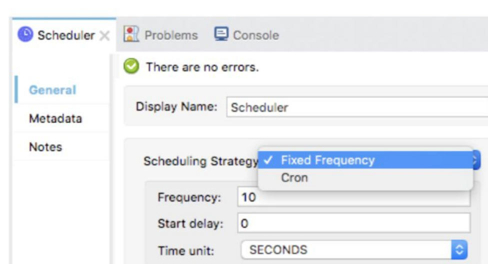
- To trigger any flow **at any time**, use the **Scheduler** component
- In a Mule runtime cluster or multi-worker CloudHub deployment, the Scheduler only runs (or triggers) on one **single** Mule runtime and it is **not guaranteed to always run on the same instance** of that Mule runtime
- Max Concurrency of 1 set on the scheduler flow does not ensure only one instance of the Scheduler runs
  - If the scheduler interval is shorter than the flow execution time



Selecting and configuring a scheduling strategy

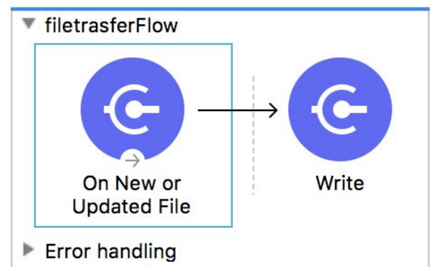
MuleSoft

- Fixed frequency
  - Set the time unit and frequency
- Cron
  - A standard for describing time and date information
  - Can specify either
    - An event to occur just once at a certain time
    - A recurring event on some frequency
- A time zone used in a Scheduler corresponds to the region of the one CloudHub worker running the scheduler



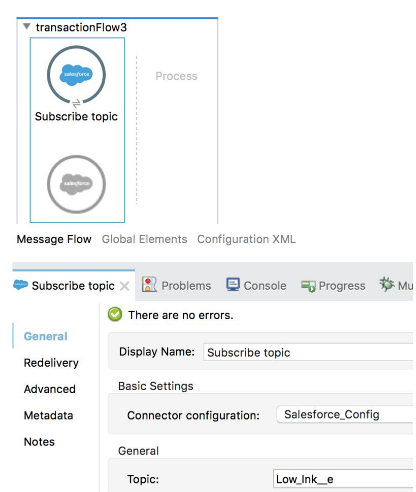
Describing the File Transfer Protocol (FTP) processing model

- The FTP connector provides access to files and folders on an FTP/SFTP server
- Supports common FTP operations
  - Listen for new or modified files
  - Read files
  - List directory contents
  - Create directory
  - Copy, move, rename and delete files
  - Write to file
  - Lock files
  - File matching
  - Watermark Enabled



Describing the Salesforce connector event processing model

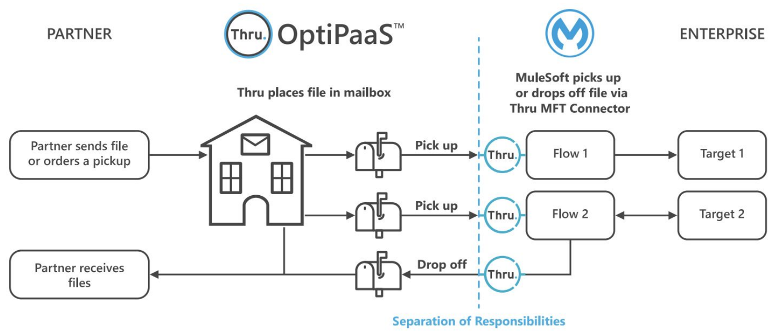
- Salesforce applications send events (or notifications) that Mule applications consume to take further action
- Publishers and subscribers communicate with each other through events
- One or more subscribers can listen to the same event and carry out actions
- Mule applications can listen to event messages by subscribing to a topic
- The Salesforce connector supports event processing by subscribing to a topic



Describing the **Thru Managed File Transfer** processing model

MuleSoft

- **Thru MFT** is a MuleSoft certified third party connector
  - Free with a Mule EE subscription
  - Download from Anypoint Exchange
  - Provides file management via an Thru's OptiPaaS file management server
  - Provides audits, alerts, and replay for file transfers
  - Include processing dashboards to monitor and control all file transfers



How Thru MFT works

MuleSoft

- Thru **OptiPaaS™ (MFTaaS)** is the **post office** where partners and the enterprise set up their endpoints to send and receive files for each process
- The same endpoints can be used in multiple processes and hence referred to as **reusable** endpoints
- The process uses a connector (Thru MFT Connector for MuleSoft, in this example) **to pick up files and possibly drop off files** with delivery instructions
- The post office places files of a specific process in a single **Pick Up Box**
- This means the consuming process (the flow) is decoupled from file sources and does not change when file sources change
- The platform takes the files from the "Drop Off Box" and delivers them to the targets via the organization endpoints

Deciding between Mule event processing options

Abstracting event processing options into processing models

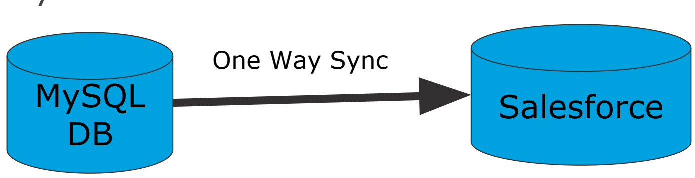
- **Processing models** collect together options and behaviors related to Mule event processing by Mule application components
- Event processing by **Mule scopes** can be modelled to describe
  - How Mule components can process a copy of the same Mule event in parallel, then combine the results
  - How Mule components process a Mule event that contains a collection of records, sequentially or in parallel
- Event processing by **Mule connectors** can be modelled to describe
  - The behavior of inbound Mule event sources at the beginning of a flow
  - How message queuing connectors process Mule events

Deciding between processing models for the use case

- Selecting the best processing model for a use case involves various factors
  - Throughput of data (dataset)
  - Memory or CPU limitation
  - Latency/Response Time
  - Message size
  - Performance requirement
  - Cluster or load balanced deployment
  - Request, response exchange pattern
  - Processing - parallel, sequential
  - Error handling and error response
  - Concurrency requirement
  - Advance capability of connectors or supported connectors
  - Real time, near real time, schedule, periodic processing of data
  - Synchronization of resource
  - Atomicity
  - Idempotency of system

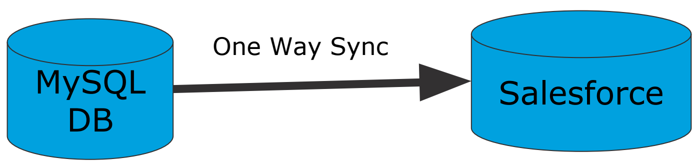
Exercise 4-1: Appropriately model Mule event processing for a data synchronization use case

- Choose the most appropriate integration style to design integration solutions that meets the organization's current requirement
- Design an integration solution to synchronize data between two databases or other systems of record
- Modify the integration style (Mule event processing models) based on data throughput vs. data processing latency requirements
- Identify the impact of competing requirements on the selected integration style



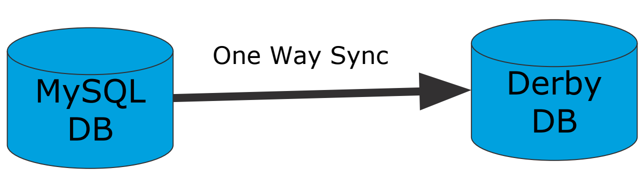
Exercise context

- Use case: One way syncing of data between a **source** MySQL database and a **target** Salesforce CRM system or other target database
- SLAs include
  - Average and maximum throughput rates of 200 records per minute



Exercise steps

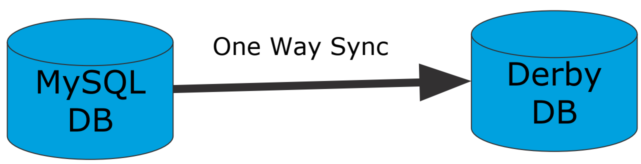
- Decide and document your assumptions of the data processing requirements (SLAs) for the use case
  - Average and maximum throughput rates
  - Processing latency SLAs
    - Must data be processed in real time or near real time?
    - If not, at what rate is data processing scheduled?
    - Will schedule data processing be at a fixed rate, or on specific dates (cron jobs)?
- Document a proposed processing model for the data synchronization use case based on the data processing SLAs



Exercise steps

MuleSoft

- Analyze data processing requirements and tradeoffs for the use case
  - Decide how the processing model can change based on changes to these competing data processing SLAs
    - Average and maximum throughput rates
    - Processing latency SLAs
- Explain how these factors impact your processing model



Exercise steps

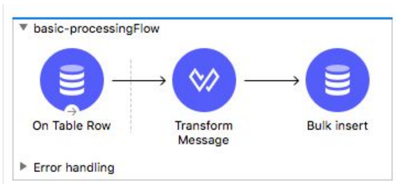
MuleSoft

- Answer these additional questions to decide and verify your proposed event processing model
  - How is data read from the DB?
  - How is flow processing triggered?
  - Must the DB reading option change to support real time or near real time latency SLAs?
  - Does the number or size of records read impact how records are processed?
  - How can the data synchronization between the system be limited to only new or modified records?
- What MuleSoft tools can help you validate your proposed processing model?

Exercise solution

MuleSoft

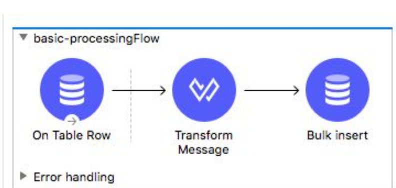
- One solution is proposed in a sketched Mule flow
  - On Table Row as a message source
  - On Table Row message source in first flow allows to perform **real time** processing from the source MySQL DB
  - Use Bulk insert to improve performance

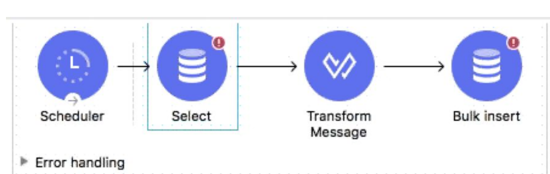


Exercise solution

MuleSoft

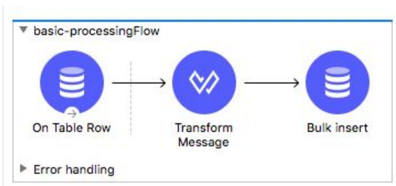
- What is another way to poll the source database records periodically?
- What are the tradeoffs to using Bulk insert vs. individual inserts?
- What is the tradeoff of using On Table Row vs. a Scheduler and a Select operation?
  - Which one of these triggers can set streaming data?

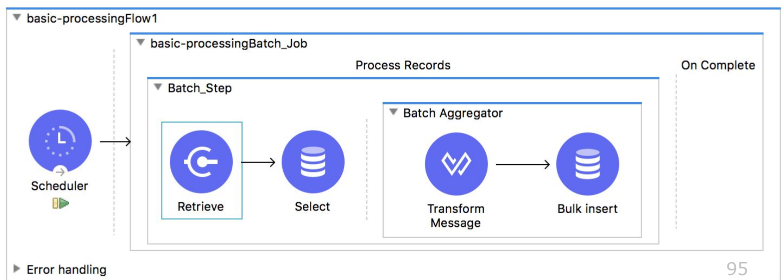




Exercise solution

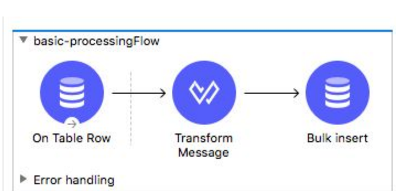
- A second proposed solution is also sketched in a second Mule flow
  - The second flow uses a Scheduler as the message source
  - The scheduler performs **scheduled** processing from the source MySQL DB
  - The Batch Step includes a Batch Aggregator scope to collect records together that are sent all together to the Bulk insert operation
    - What are the tradeoffs to using bulk operations vs. processing individual records?

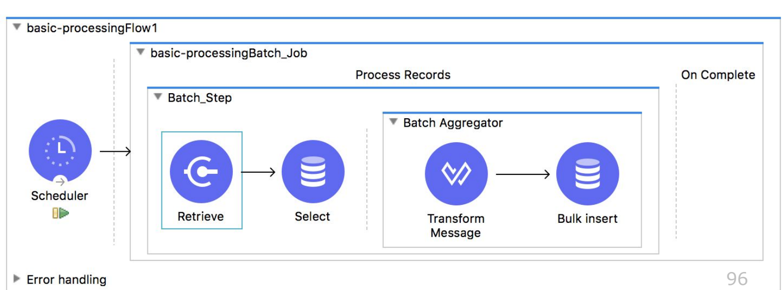




Exercise solution

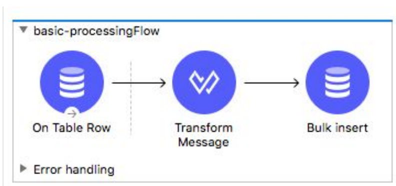
- Which components should have streaming configured?
- What are the tradeoffs to enabling streaming?
- What are the a design and performance tradeoffs between
  - Using streaming (in-memory or in a file) and no Batch Aggregator or Bulk operation
  - Not using streaming, with a Batch Aggregator and a Bulk operation

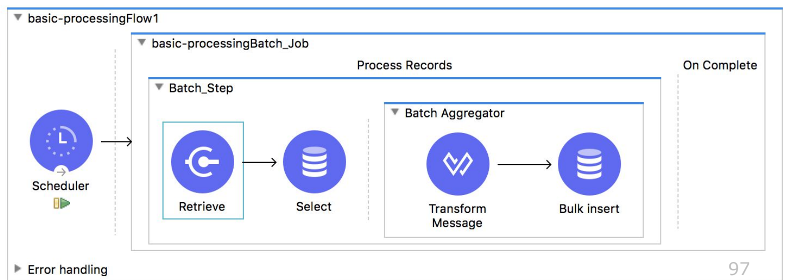




Exercise solution

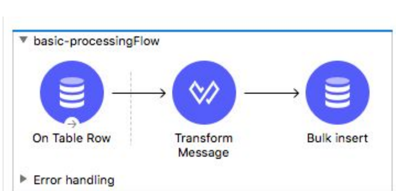
- Comparing data throughput in both processing models
  - Throughput is comparatively larger in the second flow due to periodic polling without a watermark
  - Throughput is initially the same in both flows the first time data is retrieved from the database, but varies for subsequent selects from the database

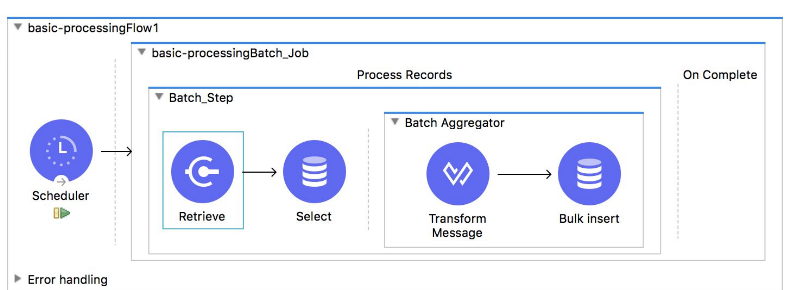




Exercise solution

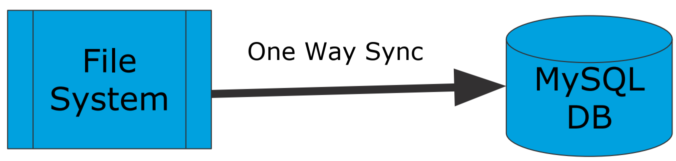
- Comparing configurations of both the flows
  - Watermarking (last record processed) is managed by On Table Row while a Scheduler is used to maintain the watermark
  - Bulk insert is performed in both the flow to commit data in batch





Exercise 4-2: Appropriately design Mule event processing for a file transfer use case

- Identify and recommend the Mule event processing models for a file transfer use case
- Identify and explain every factor that helps evaluate the best processing model
- Justify each Mule event processing model decision based on the identified factors
- Document flows that can implement the selected Mule event processing models



Exercise context: File transfer use case requirements and constraints

- Randomly, **a few times a day**, a **new source file** containing financial data for customers are **uploaded** to a specific directory on the **file server**
- The Mule application must quickly **process the file** and then send results to a **target MySQL database**
- Each file has about **1 million records** of customer data
- **Auditing** and **traceability** of each record is critical
- **Human intervention** must be allowed to **post process** any **failed records**
- The Mule application has been budgeted to deploy to **one 0.2 vCore CloudHub worker** (with 1 GB of heap memory)

Exercise steps

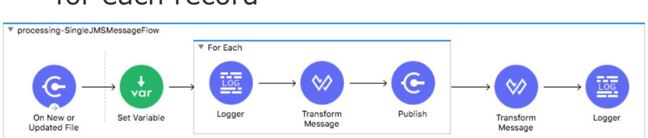
MuleSoft

- Define options to process the file
  - What are the options to read in a file?
  - After reading in a file, how are records in files processed?
  - Should records be processed sequentially, parallelly, or in batch?
  - How can the memory footprint be reduced while processing?
  - How are failed records managed?

Exercise solution

MuleSoft

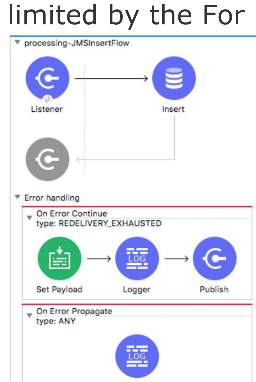
- A proposed processing model using a For Each scope
  - Processing is sequential, even though records are published to a JMS destination
  - For Each does not support buffering so the entire file is loaded into memory
  - Throughput of the process is determined and limited by the For Each scope
  - A separate insert into the DB is performed for each record



Display Name: For Each

Settings

Collection:	vars.customerRecordsFromFile
Counter Variable Name:	counter
Batch Size:	1
Root Message Variable Name:	rootMessage



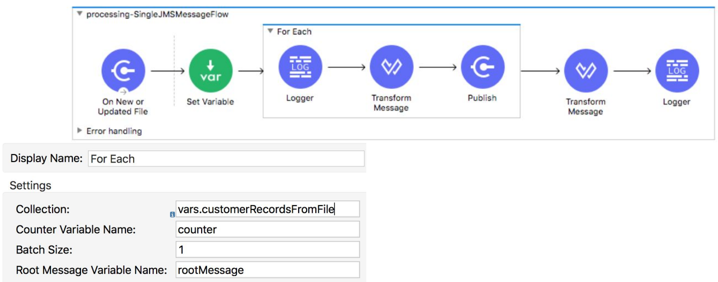
Exercise solution: Processing tradeoffs when using a For Each scope

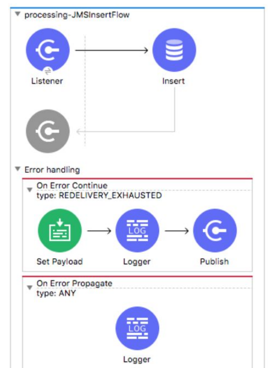
Pros

- Audit and tracing per record
- Auditing and tracing on records is easier and more open with a JMS topic
- Easier and isolated error handling of failed inserts to the target DB

Cons

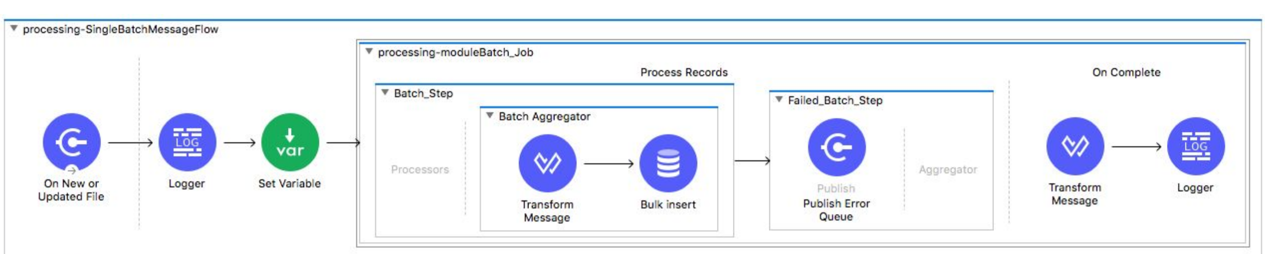
- Throughput limited by the For Each scope
- Difficult to identify when file processing completes





Exercise solution

- A different proposed processing model using a Batch Job scope
  - File is read as a stream in a Batch Job scope
  - The stream is transformed in a Batch Aggregator and then multiple records are inserted into the target DB in a single Bulk Insert operation
  - Failed records are sent to a JMS server to a dead letter queue (DLQ)



Exercise solution: Processing tradeoffs when using a Batch Job scope

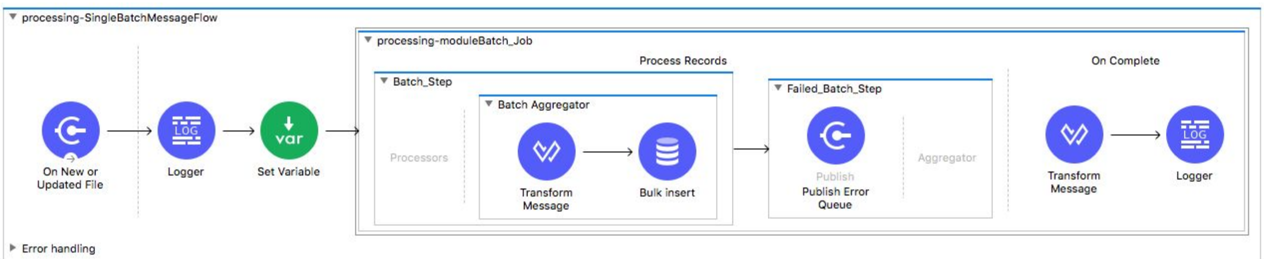
MuleSoft

Pros

- Audit and tracing per record
- ~~Auditing and tracing on records is easier and more open with a JMS topic~~
- ~~Easier and isolated error handling of failed inserts to the target DB~~

Cons

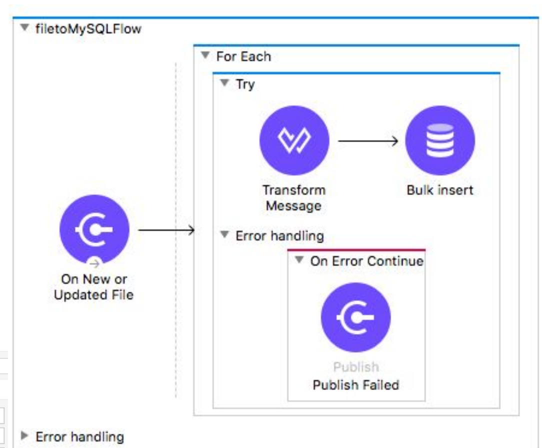
- Uses internal queues
  - May cause out of memory errors with large payloads and high throughput

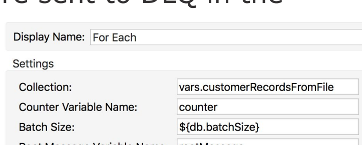


Exercise solution

MuleSoft

- A proposed processing model using a For Each scope configured to process the source file as streaming data
  - File is read as a stream in the flow
  - The stream is transformed using DataWeave inside a Try Catch block
  - Then records are inserted into the target DB in batch commit using the batch size of For Each
  - The batch size can be tuned by Ops as a property placeholder
  - Failed records are sent to DLQ in the On Error scope





Exercise solution: Processing tradeoffs when using a For Each scope with streaming data

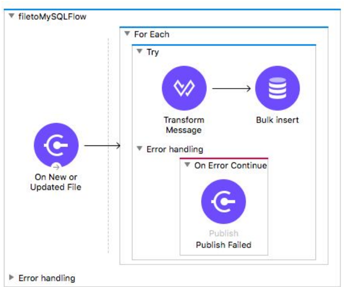
MuleSoft

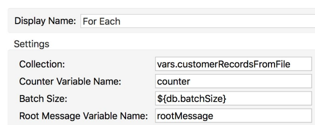
Pros

- Audit and tracing per record
- Audit and tracing on records is easier with a JMS topic
- Easier and isolated error handling of failed inserts to the target DB
- Database impact is tunable

Cons

- ~~May cause out of memory errors with large payloads and high throughput~~
- ~~Throughput limited by the For Each scope~~
- ~~Difficult to identify when file processing completes~~





Exercise solution

MuleSoft

- Decide the best processing model for the file transfer to database synchronization use case

Factors	Optimum processing model
Large payload	• Payload has 1 million records • Streaming is the best option for effective utilization of memory
Memory/CPU	• Limited size of vCore requires the processing model should work with smaller memory and CPU footprints • Streaming is the best option

- Documented flow

https://docs.google.com/document/d/1C9Xd75l4gO8yGYXlW8sFshSuujHiZoY19q2iN2Tmypc/edit

Summary

Summary

- Selecting the best processing model for use cases involve various factors
- Mule provides different options to process large vs. small datasets, streaming or not, sequentially or in parallel order
- A Scatter-Gather component provides parallel processing of events
- Batch processing is exclusive to Mule EE runtimes
- Mule 4 runtimes automatically configure event processors to automatically switch to streaming to handle large datasets

Additional references

- Salesforce event processing
  - https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_intro.htm
- WS-ReliableMessaging
  - https://en.wikipedia.org/wiki/WS-ReliableMessaging
- XML Signature
  - https://en.wikipedia.org/wiki/XML_Signature
- XML Encryption
  - https://en.wikipedia.org/wiki/XML_Encryption