

**Mulesoft-SQS**

**Integration Documentation**

**Mule Accelerators**

**Table of Contents**

1. **Introduction**
2. **Integration Scenario**
3. **Solution Overview**
   1. **Scenario Summary/Solution**
   2. **Debug Snapshots**
   3. **Error Handling**
   4. **Rest Service Encapsulation**
   5. **Reprocessing Strategies**
   6. **Troubleshooting tips**
   7. **Performance Metrics/Regulations**
4. **Demo Setup**
5. **Live Demo**
   1. **Code Walkthrough**
   2. **Test Plans**
      1. **Happy Path**
      2. **Error Path**
      3. **Reprocessing Path**
      4. **Rest Service demo**
6. **Benefits and Value proposition**
7. **Use Case Examples**
8. **QA Session**
9. **Closing Remarks**
   1. **Future scopes**
   2. **Resources**
10. **Introduction:**
    * Provide a concise overview of the integration topic and its significance.
    * Introduce the specific integration solution to be demonstrated.

**Overview:**

Amazon SQS is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. It provides a durable, highly scalable, and secure way to exchange messages between different components of your application.

SQS offers asynchronous message communication as opposed to API calls.

Useable in variety of contexts:

* Data processing: processing mobile events.
* Real time events processing: sync operation to analytics dashboard.
* Delayed event processing.
* Ad hoc job queuing: scheduling Database snapshots at midnight.

**Why use SQS over API calls:**

* Backpressure control: consumers can choose rate of processing, inversely you can have many polling threads to have high concurrency.
* Fire and Forget: dependency reduction, decoupling of services.
* Eventual guaranteed processing: great for async or non-realtime apps.

Mule 4 provides an Amazon SQS module that allows you to easily integrate Amazon SQS into your Mule applications.

**Key Features of the Amazon SQS Module in Mule 4:**

* A variety of connectors for sending and receiving messages.
* Support for different message formats.
* Support for message batching and transactionality.
* Support for error handling.

**Fifo Queues:**

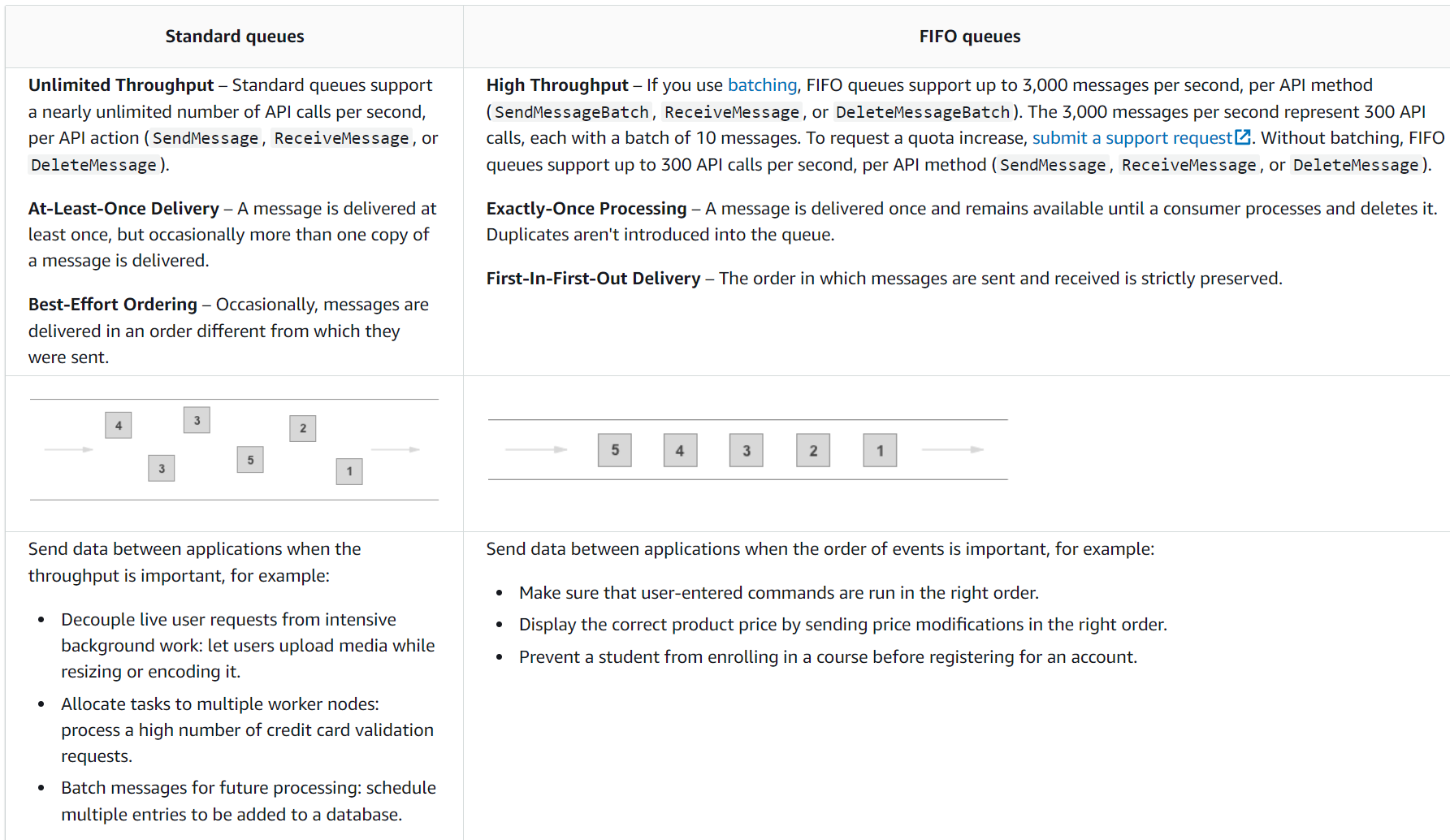
**DeduplicationScope** – Specifies whether message deduplication occurs at the message group or queue level. Valid values are **messageGroup** and **queue**.

**FifoThroughputLimit** – Specifies whether the FIFO queue throughput quota applies to the entire queue or per message group. Valid values are **perQueue** and **perMessageGroupId**. The **perMessageGroupId** value is allowed only when the value for **DeduplicationScope** is **messageGroup**.

To enable **high throughput for FIFO queues**, do the following:

Set DeduplicationScope to messageGroup.

Set FifoThroughputLimit to perMessageGroupId.



1. **Integration Scenario:**

* Describe a relevant real-world scenario or use case where integration is essential.
* Explain the challenges or inefficiencies that arise in the absence of integration.

**Scenario**

Imagine a business that operates an ecommerce retail platform. Purchase, Return and Sales orders are placed via BTB/DTC source system’s and the order processing system needs to handle these orders efficiently. To improve the productivity, scalability and reliability of the order processing system, the business decides to implement an asynchronous communication mechanism using Amazon SQS.

**Order Submission Flow:**

When an order is placed, the order information is received by the MuleSoft application.

Instead of processing the order synchronously, the application sends the order details to an Amazon SQS queue using the SQS connector.

**Asynchronous Order Processing:**

The order details are placed in an Amazon SQS queue, acting as a buffer.

A separate MuleSoft flow (consumer) is set up to continuously poll the SQS queue for new orders.

**Order Processing Flow:**

The consumer flow retrieves orders from the SQS queue and processes them asynchronously.

Processing may involve tasks such as inventory updates, customer fulfilment, item extension and shipping coordination.

**Scalability and Reliability:**

As the volume of orders increases, the asynchronous processing allows for better scalability.

The use of Amazon SQS ensures reliability, as messages are persisted in the queue, reducing the risk of data loss.

**Fault Tolerance:**

If any downstream system (e.g., inventory management) experiences temporary issues, the order details are retried using suitable error handling strategies and reprocessed using dead letter queue if needed.

Once the downstream system is back online, it can continue processing the queued orders.

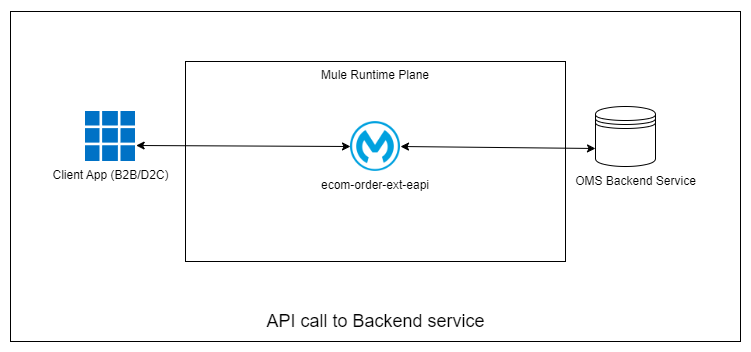


Fig 1.0 Sync Communication

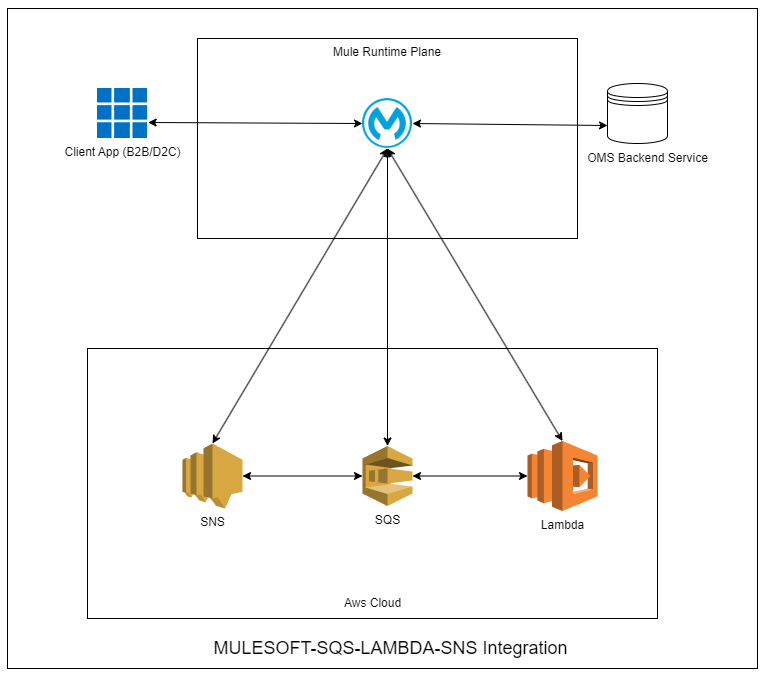


Fig 1.1 Async Communication

**Challenges/Inefficiencies in absence of proposed Integration**

* **Scalability:** Synchronous APIs may face scalability issues due to their sequential handling of requests. However, this can be mitigated by optimizing server resources and implementing efficient load-balancing strategies.
* **Increased server load:** When opting for synchronous APIs, it’s important to use modern server technologies and architectures that are designed to handle numerous simultaneous connections efficiently. Also, leveraging caching mechanisms and optimizing API endpoints can reduce the load significantly.
* **Risk of blocking operations:** A synchronous API waiting for a response can block other operations, especially in single-threaded environments. That’s why it’s important to use non-blocking I/O operations or multi-threading techniques to ensure the application remains responsive and efficient.
* **Dependency on external systems:** If synchronous APIs depend on external systems or services, timeout mechanisms and fallback strategies must be in place to mitigate this risk.
* Less efficient in resource usage, a potential limit in high-traffic scenarios.

1. **Solution Overview:**
   * Present a detailed overview of the integration solution being demonstrated.
   * Highlight its key features, components, and functionalities.

**Scenario**

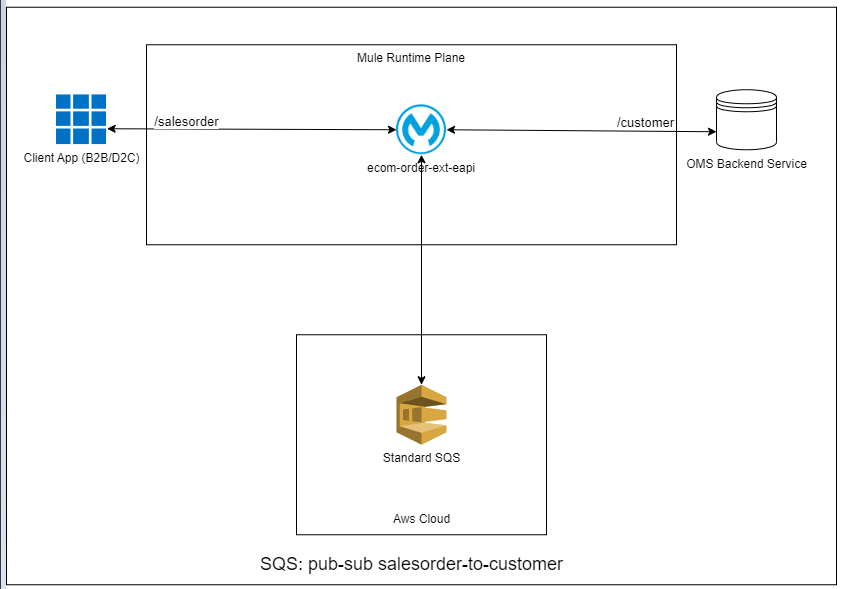
Imagine a business that operates an ecommerce retail platform. Purchase, Return and Sales orders are placed via BTB/DTC source system’s and the order processing system needs to handle these orders efficiently. To improve the productivity, scalability and reliability of the order processing system, the business decides to implement an asynchronous communication mechanism using Amazon SQS.

1. **Processing Sales Order to Customer Service**

**Requirement:**

* Best effort ordering, handle deduplication and large message payloads 500 transactions per second.
* No potential Data loss, Schema validation to be done, retry messages for x times on temporary outage.
* Log and report non retry able errors via SNOW incident registration.

**High Level Design:**



**Fig 1.0** sales order-to-customer message processing

**Solution:** Using Standard SQS queues will not guarantee deduplication or message ordering but it will provide unlimited throughput.

Handle message idempotency at consumer end.

* B2B client places sales order to fulfil to customer.
* Sales order received by Mule application and published to standard SQS queue,
* Mule application consumes order and send to oms backend service for customer fulfillment.

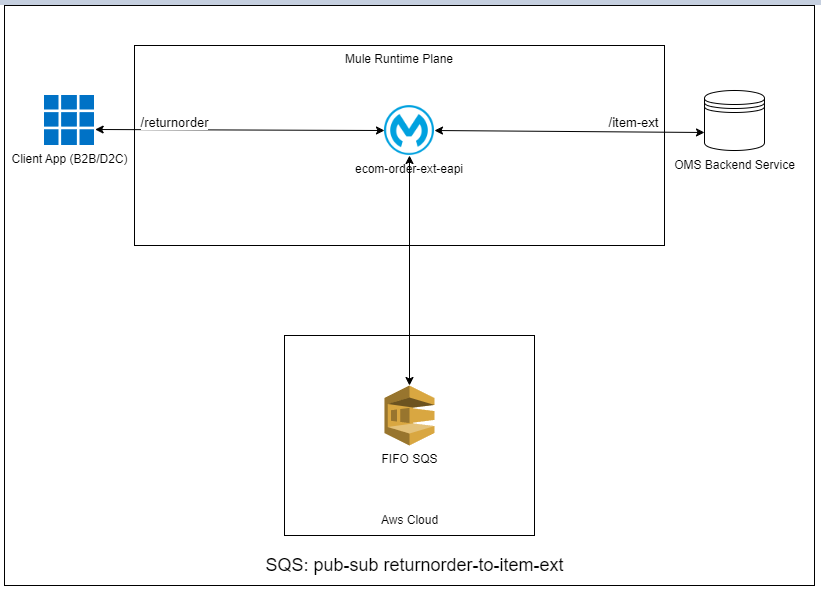
**Error Handling:** Manual Dlq Redirect Error Handler **(refer error handling section to know more)**

1. **Processing Return Order to Material Item extension Service**

**Requirement:**

* In-order processing, handle deduplication and expect return orders to come at customer level. Its important return orders are extended to item service at customer level and in the order as it arrived. Expect 300TPS
* No potential Data loss, Schema validation to be done, retry messages for x times on temporary outage.
* Log and report non retry able errors via SNOW incident registration

**High Level Design:**



**Fig 1.1:** return order to item extension message processing

**Solution:** Using FIFO SQS queues will guarantee deduplication within 5-minute interval at publisher end and message ordering with limited throughput. In order to maximize throughput consider setting same message group Id with return orders for same customer. This would introduce parallelism at consumer end.

* D2C client places return order to item extension service with message group id.
* Return order received by Mule application and published to FIFO SQS queue,
* Mule application consumes order and send to oms backend service for item extension for customer.

**Error Handling:** Auto Dlq Redirect Error Handler **(refer error handling section to know more)**

1. **Processing Purchase Order to Shipment and Product Inventory services.**

**Requirement:**

* fan-out write, best effort ordering, handle deduplication. Processing of order to shipment and product service should be independent of each other.
* No potential Data loss, Schema validation to be done, retry messages for x times on temporary outage.
* Log and report non retry able errors via SNOW incident registration.

**High Level Design:**

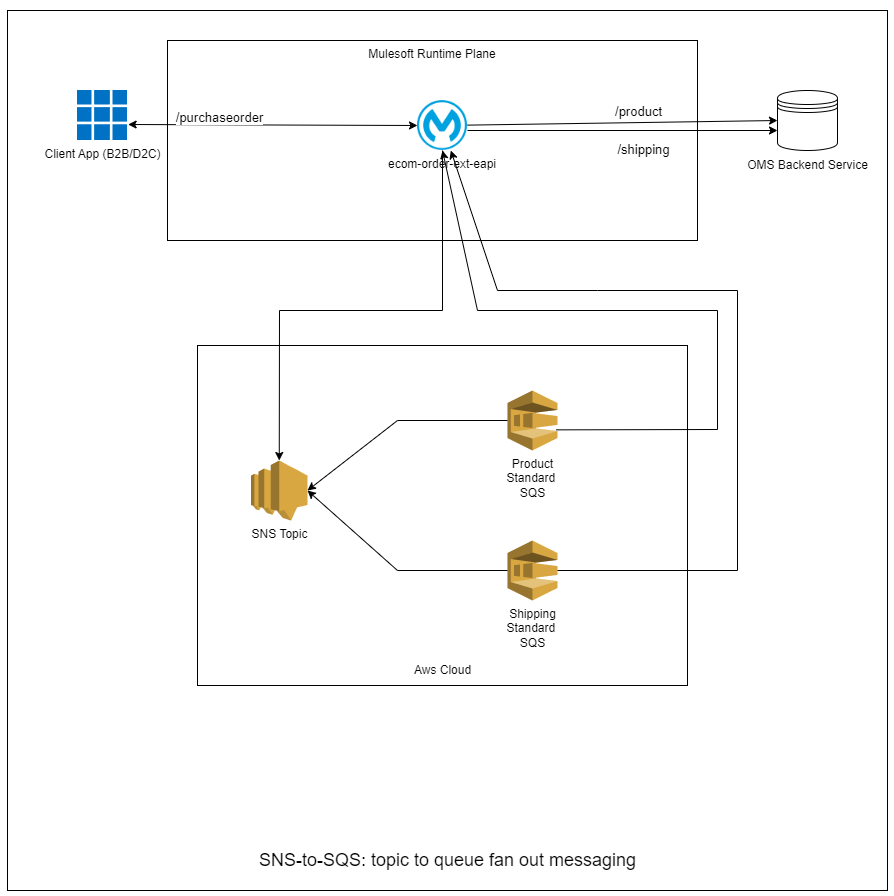


Fig: SNS-to-SQS messaging

**Solution:** Using Standard SQS queues is only option here since SNS topics don’t support fifo SQS queues endpoint subscriptions.

In order to have same copy of message sent to two different queue, we have to adopt SNS-SQS message publication pattern.

Two queues for products and shipping service bind to purchase order SNS topic.

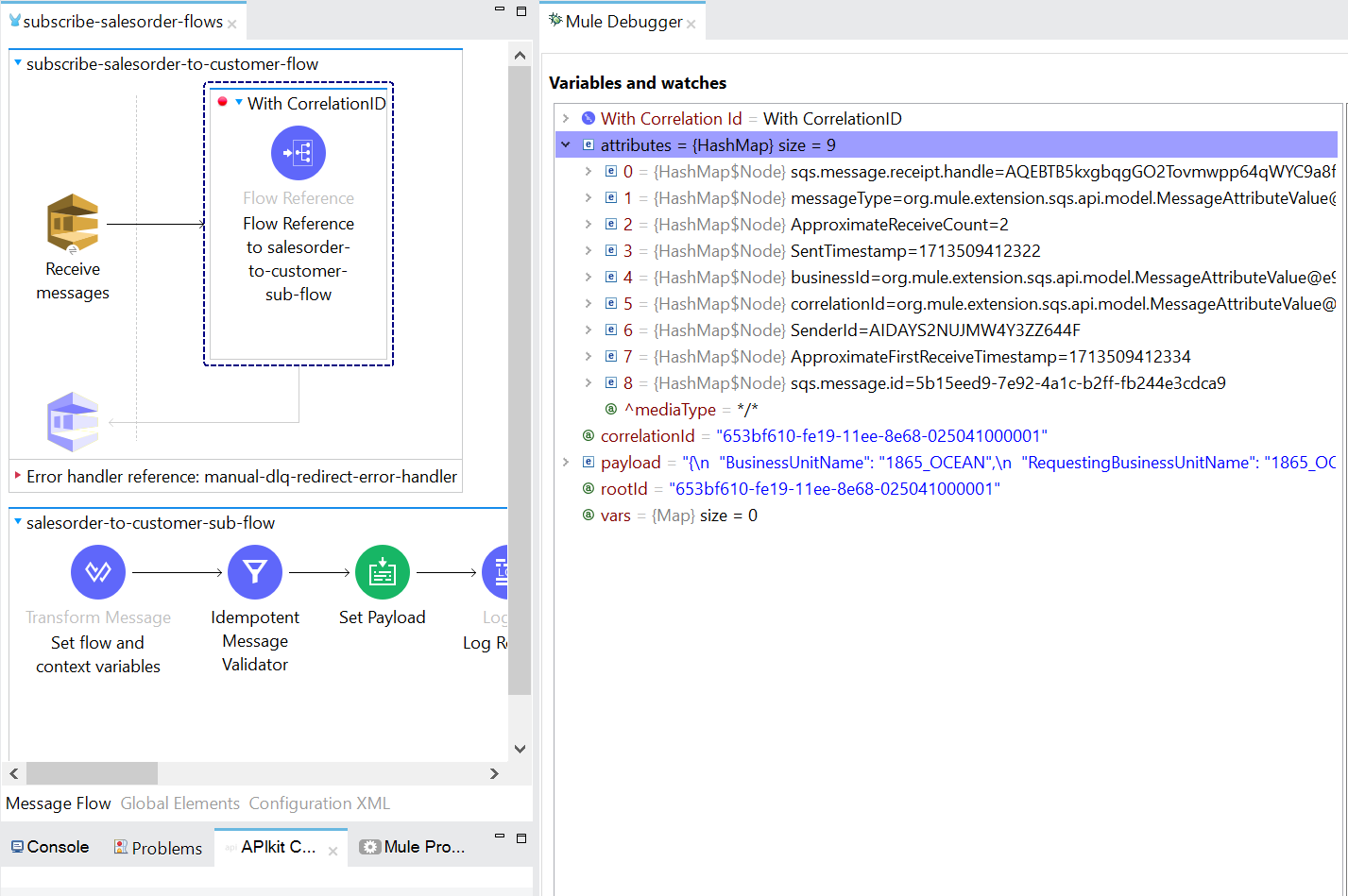
Handle message idempotency at consumer end.

* D2C client places purchase order to product and shipping service.
* Purchase order received by Mule application via SNS purchase order topic and published to standard SQS queues subscribed to topic,
* Mule application consumes purchase order from SQS queue and send to oms backend service for shipment and product inventory fulfillment.

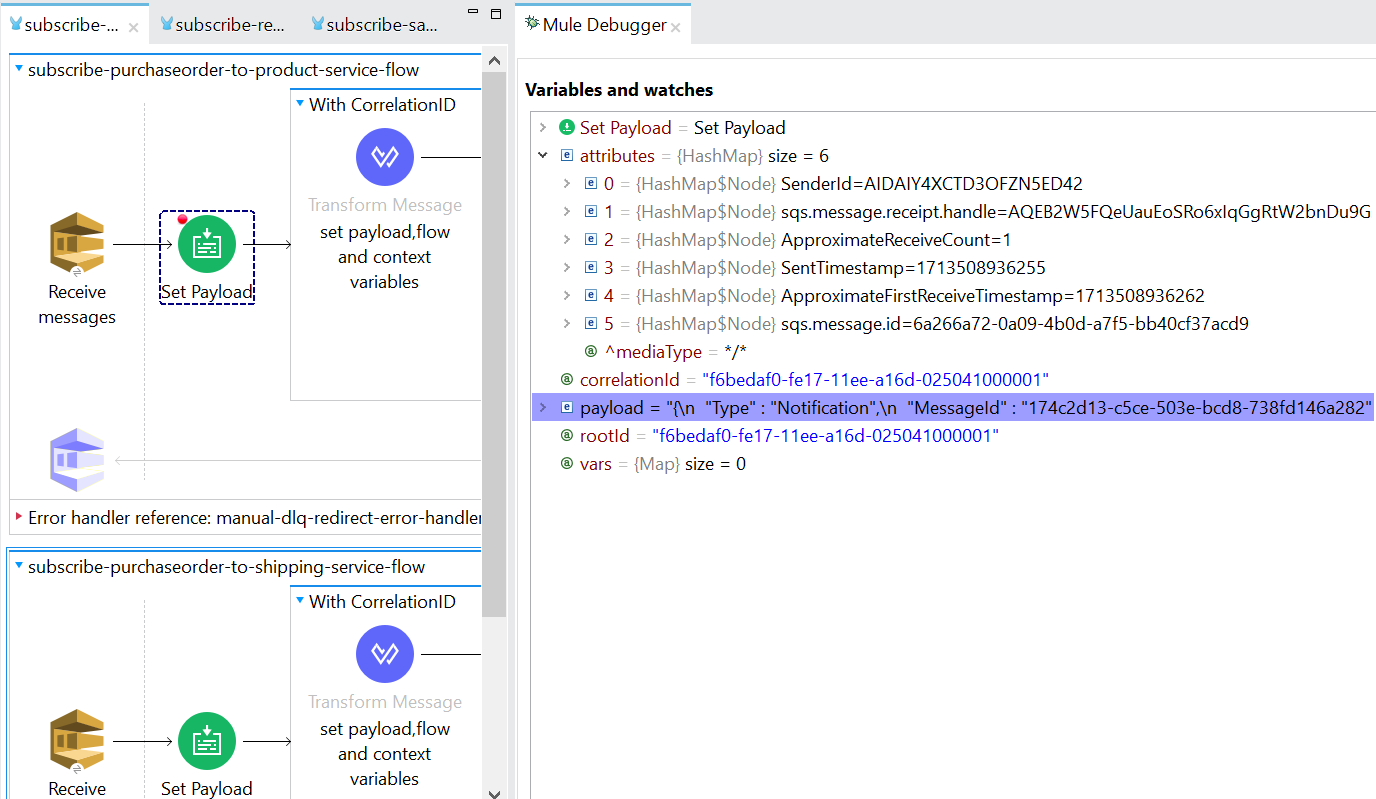
**Error Handling:** Manual Dlq Redirect Error Handler **(refer error handling section to know more)**

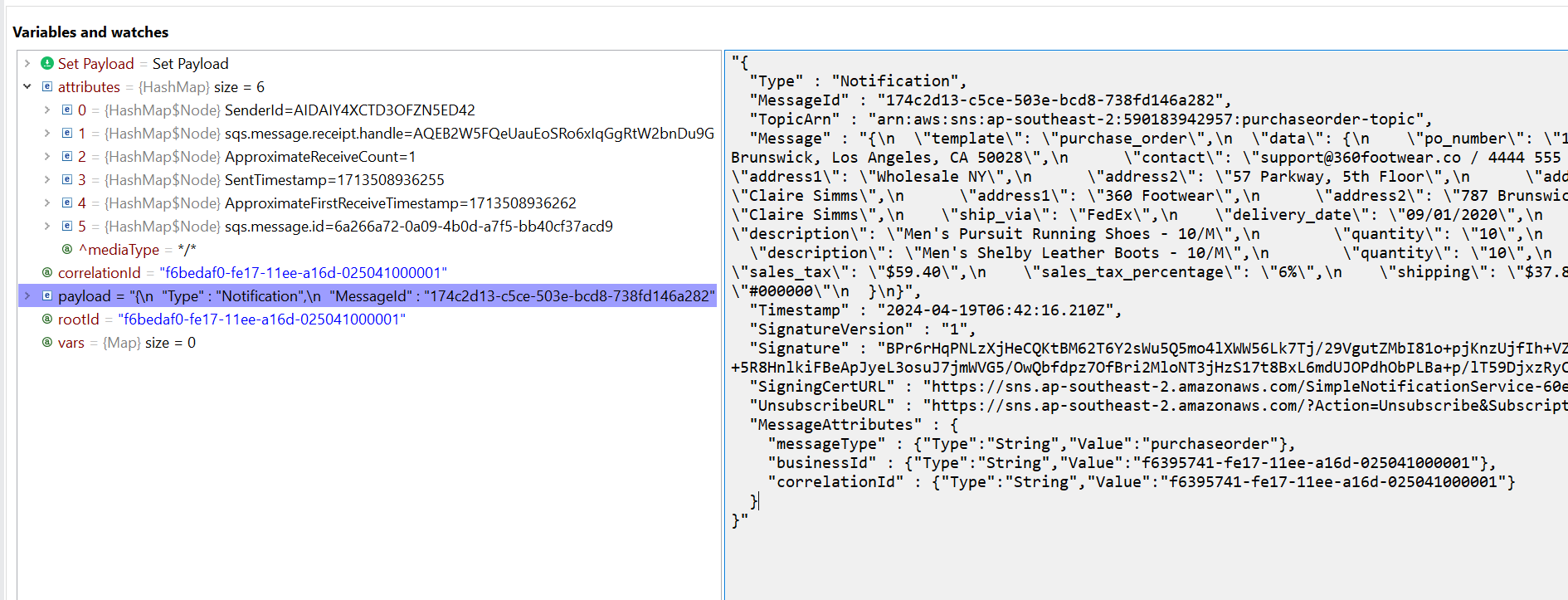
**Debug Snapshots**

* 1. **A message subscription from sqs standard queue.**

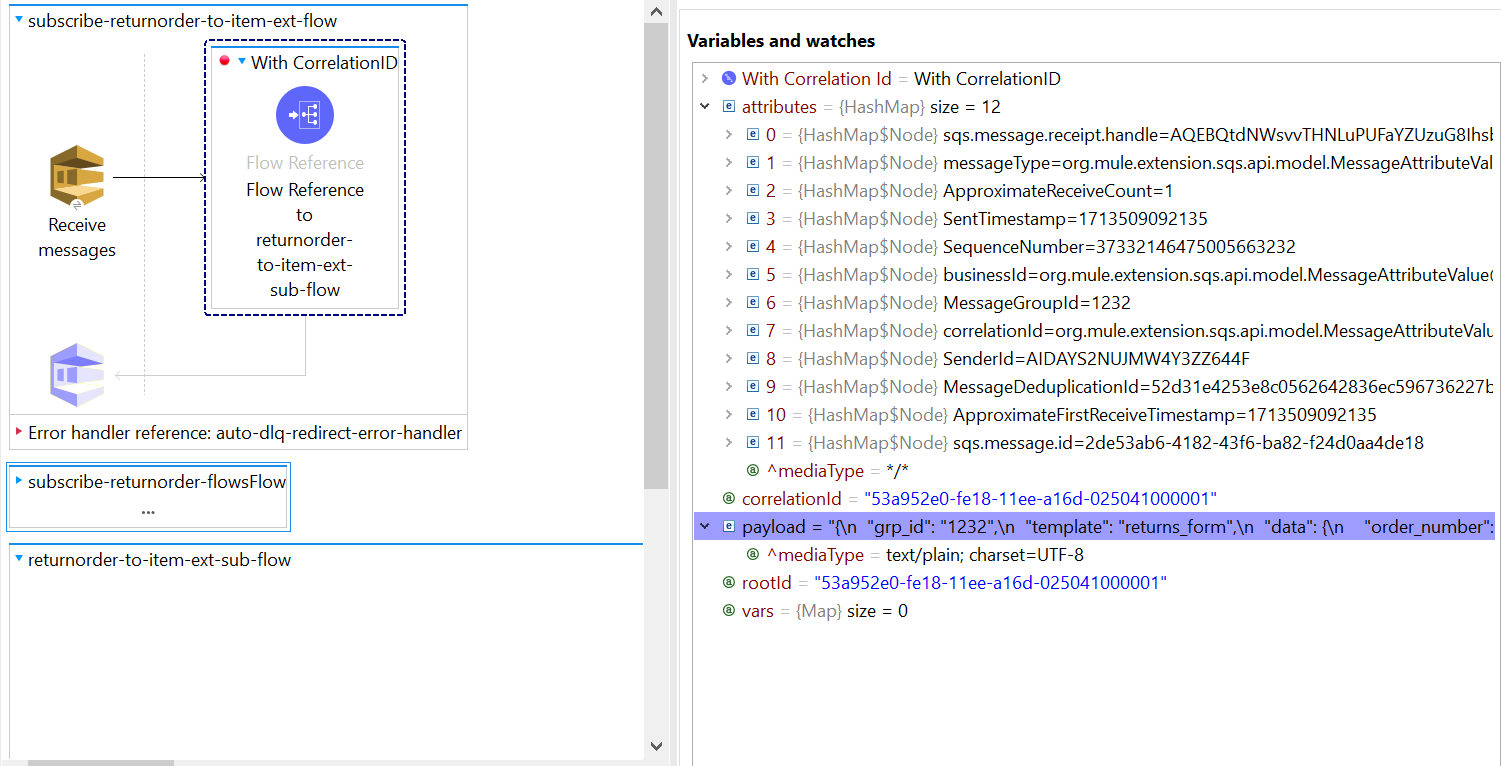


* 1. **A message subscription from sqs standard queue binded to SNS topic.**



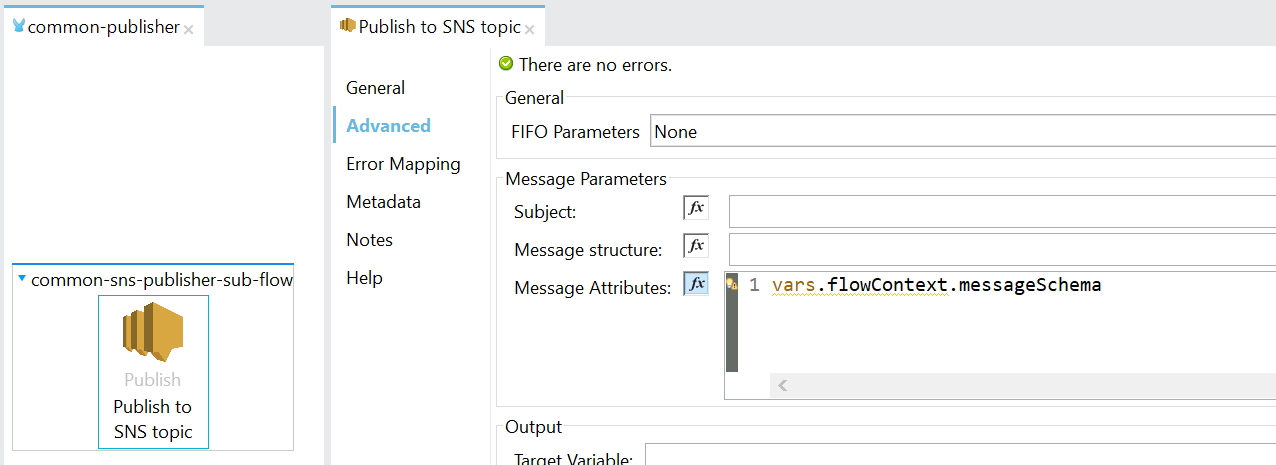


* 1. **A message subscription from sqs fifo queue.**

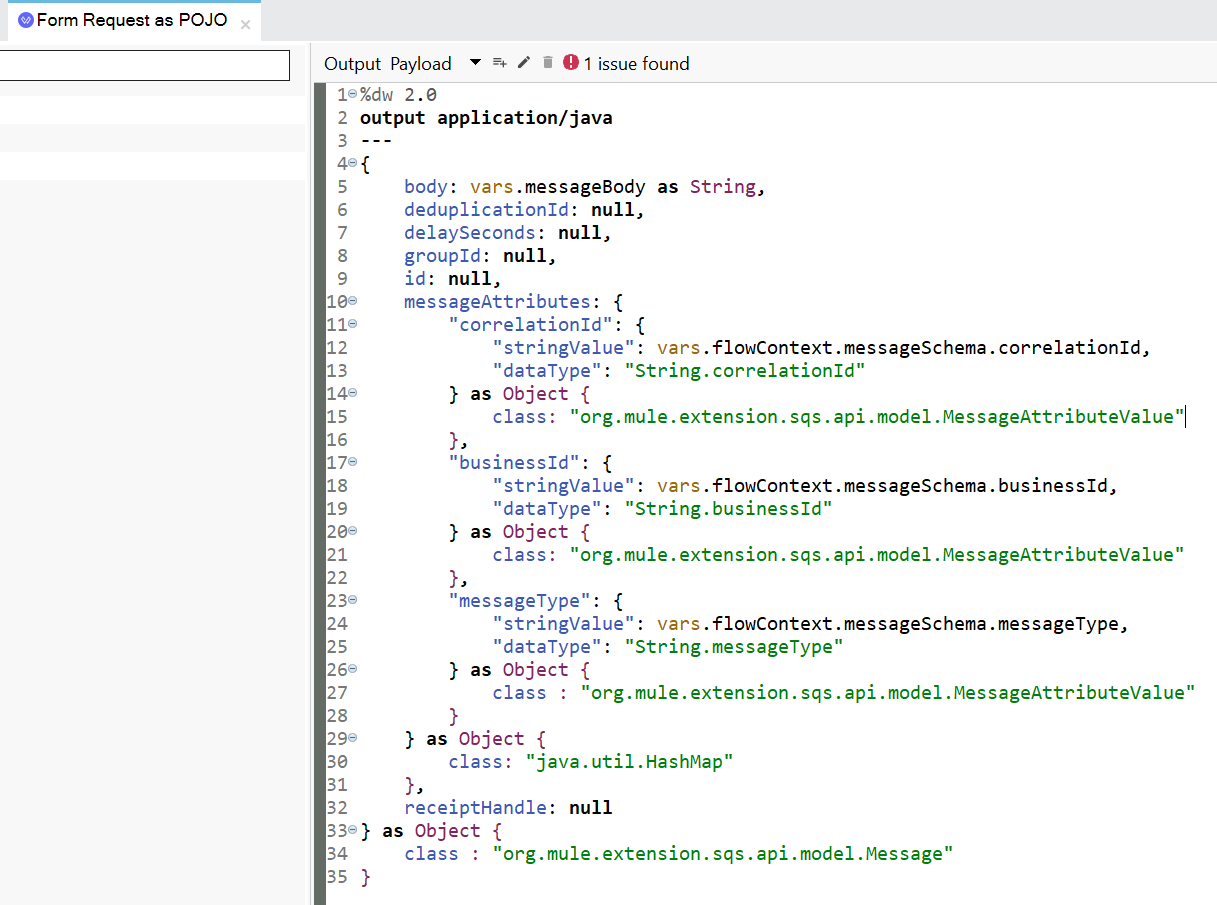


* 1. **Transaction tracking and message type categorization.**

Sending message attirbutes like correlationId, businessId and messageType from flowContext variable to **SNS topic**



Sending message attirbutes like correlationId, businessId and messageType from flowContext variable to **SQS Queue**



**Error Handling**

1. **Automatic DLQ Redirect Error Handler**

If message delivery fails,

If failure is one of retry able error types,

then extend message visibility timeout by 40 seconds as back off time and end the flow.

Message will be picked up after 40 seconds and retried for maxRecieveCount = 3,

If message delivery failed on third attempt, then it will be automatically sent to dlq as it is with **no change in payload and message attributes**.

If failure is one of non retry able error types,

Then simply log the message and delete it from queue

Raise Service now incident

1. **Manual DLQ Redirect Error Handler**

If message delivery fails,

If failure is one of retry able error types,

then extend message visibility timeout by 40 seconds as back off time and end the flow.

Message will be picked up after 40 seconds and retried for maxRecieveCount = 3,

If message delivery failed on third attempt, then set the message body and attributes with error attributes, description, and publish the message to dlq.

If failure is because duplicate message delivery:

Log and delete the message from queue.

If failure is one of non retry able error types,

then set the message body and attributes with error attributes, description, and publish the message to dlq.

Raise Service now incident

1. **Idempotency Logic With Mule**

attributes.sqs.message.Id is unique identifier for every message in sqs queue.

If this messageId alone is kept as deduplication criteria, then we can avoid getting duplicate messages, but only setback would be that it will also stop messages that are getting retried because of target failure because they will have same messageId.

That is why We can leverage Object Store to solve this problem

Keep **IdExpression** to filter duplicate message as (**messageId + approximateReceiveCount)**

* **Success scenario**

Message picked up with idExpression (1234-arc-1) stored in OS.

If message is successfully delivered to target then remove key (1234-arc-1) and update it to (1234-arc-2) , this will let the idempotent filter to stop message subscription with same message id and approximateReceiveCount of 2.

* **Failure Scenario**

Anyway message is getting retried so , no need to update OS with next approximateRecevieCount because it wont let retry if update happens.

**When to Use Manual DLQ Redirect Error Handler at consumer end**

* if you want messages in dlq to have additional attributes like error types, description of the cause of error.
* If you want to use common dlq for multiple queues, then additional attribute would be needed for a message to determine what is its main queue, you can add message type attribute to a message before publishing it to dlq.
* If you are using Manual dlq redirect error handler, then using click ops to redrive messages from dlq to main queue will show error **Failed: CouldNotDetermineMessageSource** on AWS management console because message was manually published to dlq not implicitly.

**When to Use Auto DLQ Redirect Error Handler at consumer end**

* if you want messages in dlq to be in same state as they arrived in main queue and don’t want to lose the capability to perform click ops (Start Dlq Redrive to redrive messages from dlq to main queue).
* If you don’t want to send non retry able error messages to dlq.

**Rest Service Encapsulation**

Activities like creating queue, topic, setting queue and topic attributes and policies, deleting message , purge queues, binding topics-queue, setting redrive policy, read/send messages in batch, Start DLQ redrive (Click Ops) can be managed by **AWS management console** or **AWS CDK**.

if the IAM user have **read** **only access** to **AWS services** and **write access** only via third party programs with SDK to perform these operations, then in that case we could perform all these tasks via **API gateway**.

For this reason I have created REST service to perform these operations along with postman collection and example response.

Detailed API specification is created on Anypoint exchange.

All the services are accessible via APIKit router.

Remark: Every thing is tested on local development environment as well as cloudhub.

These flows could be used as reusable component in other flows provided inbound payload to this flows should be formed in compliance to the request format.

Rest service encapsulation heavily relies on object serial/deserialization , AWS SQS/SNS connectors.

**Reprocessing Strategies:**

Lets define routine x as below:

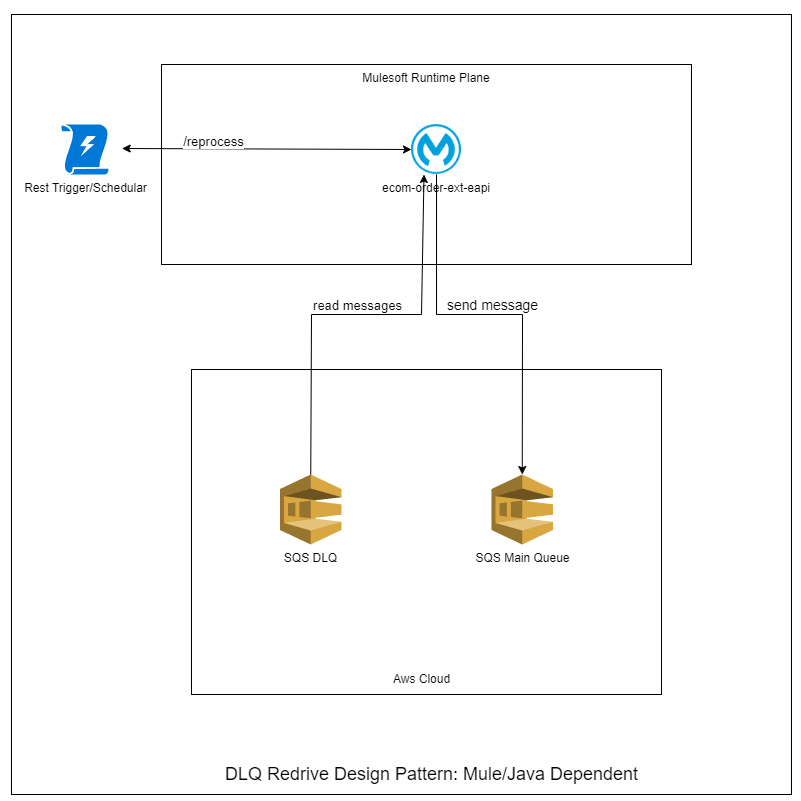
**routine(x): x is max no of messages to redrive from dlq**

* + Read top X messages from dead letter queue
  + Filter retry able errors and send them in batches to source queue.
  + For non retryable errors raise SNOW incident here or at the time of publishing them to dlq.
  + Then delete receiptHandle in batch for all the messages which were successfully reprocessed to source queue.
  + Let the user know of success and failed messageId so that these failed messageIds could be retried.**(TBD)**

1. **Mule based dlq reprocessor**

Trigger could be a Rest endpoint or Schedular

Implement routine(x) with Mule SQS connectors



1. **Mule-Java based dlq reprocessor**

Trigger could be a Rest endpoint or Schedular

Implement routine(x) with java JDK 8 and call with Java static invoke connector

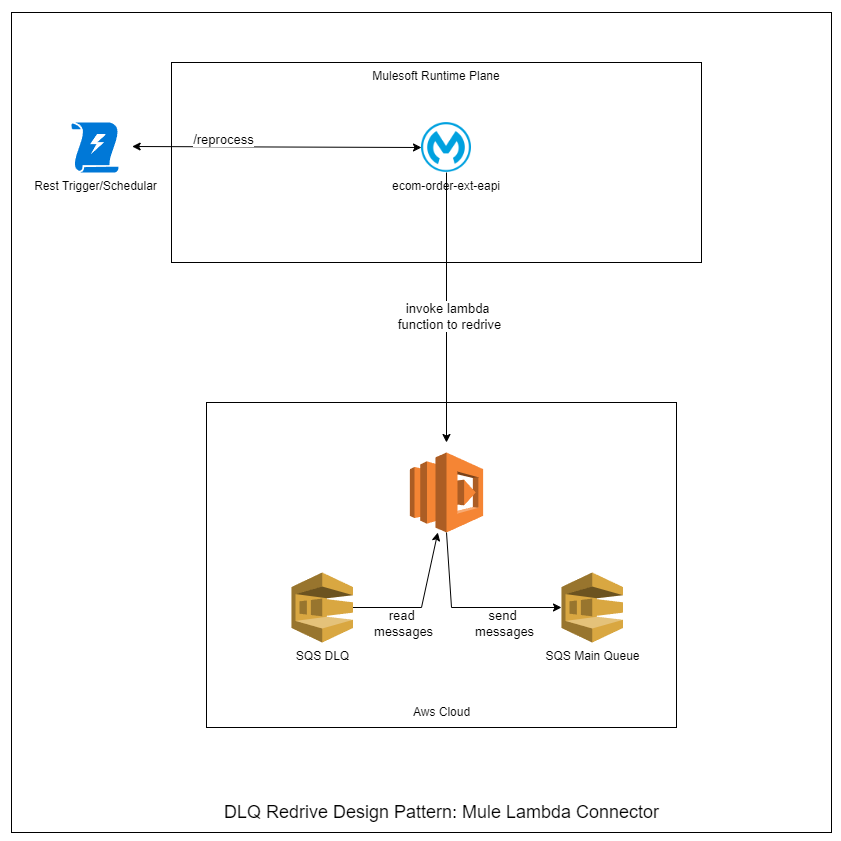
1. **Mule-Lambda based dlq reprocessor**

Trigger could be a Rest endpoint or Schedular

Create Maven Quickstart Project and write lambda function in java to perform routine(x)

Deploy that project to AWS cloud lambda

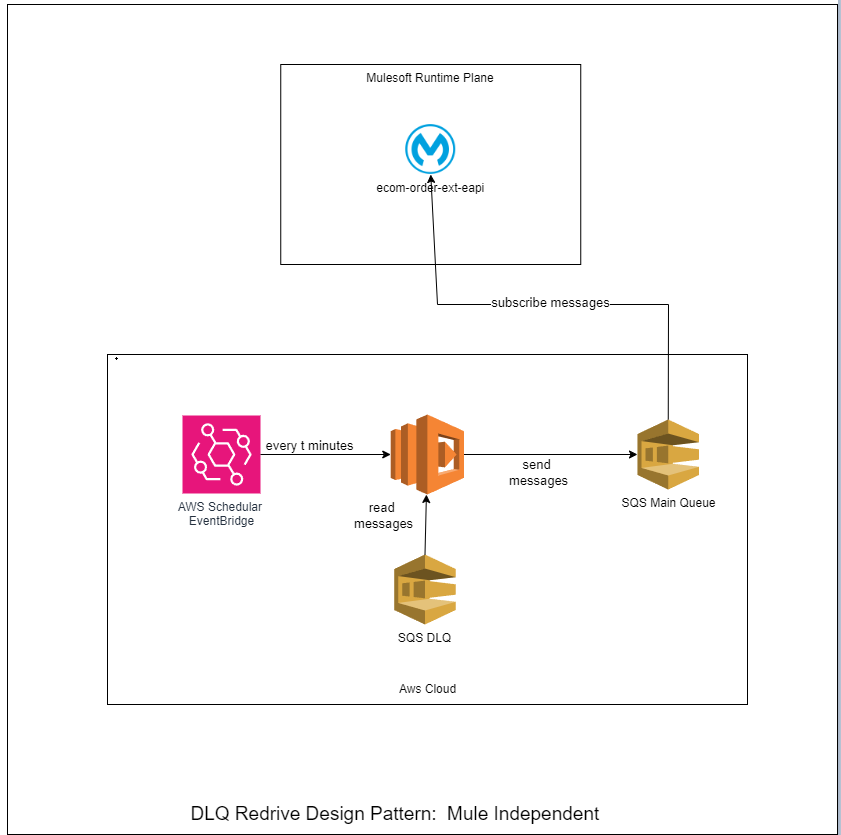
Call lambda function on AWS cloud to perform redrive using **Mule Lambda Connector.**



1. **AWS Eventbridge-Lambda reprocessor (Mule Independent)**

AWS EventBridge schedular every t minutes

Invoke Lambda function to perform dlq redrive.



**Notes:**

1. While dealing with SQS connectors , we have to deal with POJO like Message Objects, SendMessageBatchResult object, and in order to deserialize them we may need to depend on **Java Interceptor connectors.**
2. When used common dlq for multiple queues, you have to incorporate messageType grouping before sending message to source queue.

**Troubleshooting tips (Important Points)**

1. **Setting Message Visibility Timeout at queue level**

When a consumer claims a message from queue, a visibility timeout of 30 seconds (default) starts, means message wouldn’t be visible to other consumers or other threads of same consumer until 30 seconds has lapsed.

Once the consumer has processed the message, it has to delete it from queue with the receiptHandle which comes along with it, and it has to be done before visibility timeout expires.

That is why value of visibility timeout should be always greater than time required to process and delete the message.

Ex if max time required to process the message is 20 seconds then its safe to set visibility timeout at 30 seconds while creating the queue, that’s why we should load test our target and record max time it takes to respond.

1. **Visibility Timeout Expires**

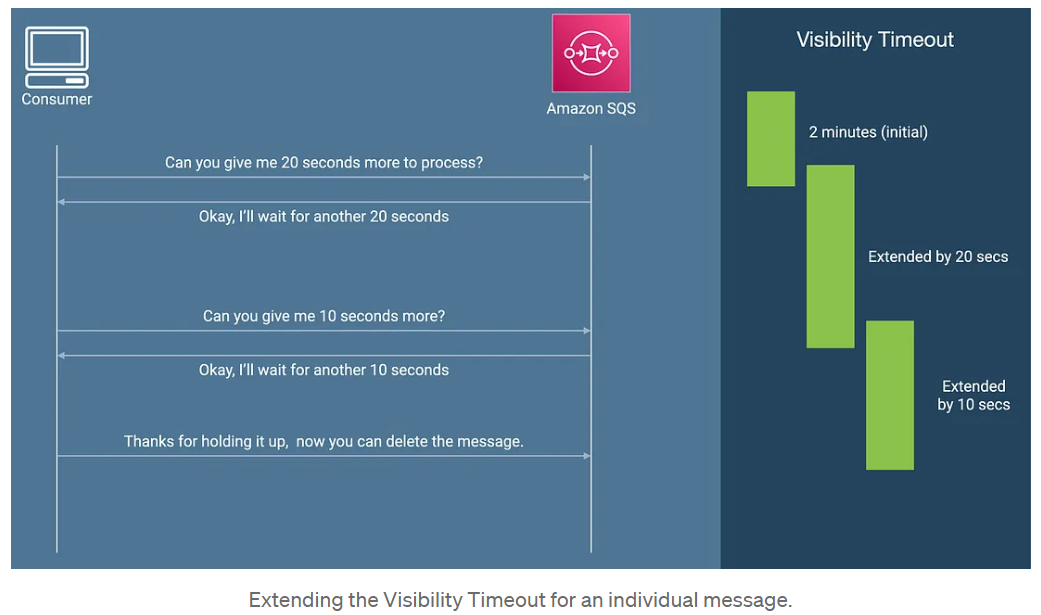
if message visibility timeout expires while you are still processing the message, then message will be returned to the queue, and you will end up picking up same message twice. If your flow is idempotent then no need to worry.

1. **Extending Message Visibility timeout at message level**

If target took longer to respond and you are seconds away from exceeding visibility timeout then its safe to extend visibility timeout by some seconds so that message is not returned to queue again.

1. **Extending Message Visibility timeout when target response time is not known**

set the timeout to a baseline value and keep on extending it until your consumer is done with the message.



1. **When a consumer doesn’t want to process and delete a message,**

it can tell the SQS to terminate the visibility timeout for that message immediately. This can be done by calling ChangeMessageVisibility API operation with VisibilityTimeout set to 0 seconds.

1. **Setting Message Visibility Timeout to be small value (not right)**

"Value AQEB… for parameter ReceiptHandle is invalid. Reason: Message does not exist or is not available for visibility timeout change. (Service: Sqs, Status Code: 400, Request ID: 0975a118-a788-5915-b85b-ff8d3892537c)"

The ReceiptHandle is what the watcher container uses to either remove a message from SQS (if processing was a success), or return it (if it needs a retry).

The handle is provided to the watcher when it receives a message from SQS. If the message gets received again before that first one gets returned, then the handle from the first receive becomes invalid.

messageTimeout values are not quite tuned right, and a message got received a 2nd time before the first one finished.

1. **With short polling**,

the ReceiveMessage request queries only a subset of the servers (based on a weighted random distribution) to find messages that are available to include in the response. Amazon SQS sends the response right away, even if the query found no messages.

1. **With long polling,**

the ReceiveMessage request queries all of the servers for messages. Amazon SQS sends a response after it collects at least one available message, up to the maximum number of messages specified in the request. Amazon SQS sends an empty response only if the polling wait time expires.

1. **Change message visibility**, **Delete message operation on Standard SQS doesn’t cause failure if tried to deleted outside of visibility timeout window,**

because receiptHandle is different for different instance of message and normally for standard SQS there’s **at least** once delivery, that’s why you might get delivered message twice with different receipt handles each time, that’s why it won’t cause Invalid receipt handle error.

That’s why for standard sqs, even if you fail to delete message out of visibility timeout, you wont get error, you will only get same message twice because of SQS distributed architecture and you can take care of that with **idempotency at mule**,

1. **Change message visibility, Delete message causes failure when tried to delete outside visibility timeout in FIFO queues**

because a single instance of message is only retried , that’s why the moment vis timeout expires receipt handle resets and trying to delete it causes error. **Invalid receiptHandle**

That’s why at no cost you should exceed vis timeout to delete or extend message visibility timeout in FIFO queues.

1. **MessageId Obeservations:**

**Manual Redirect:**

If message redirected using manual dlq redirector, then message id resets no matter if you mirror it from original message Id.

For message redirected to dlq, approximateReceiveCount will be reset to 1.

**Auto Redirect:**

If message redirected using auto dlq redirector, then message id is not changed.

For message redirected to dlq, approximateReceiveCount will be incremented to 3 because of retries done in the source queue.

1. **Bug in Runtime version 4.6-** use 4.6-java8 or 4.6-java17 as workaround instead of 4.6.2 in <app.runtime>
2. If using mule maven plugin, first deploy mule application to exchange and then deploy to cloud hub. Skipping former will result into deployment failure.
3. Add <source> tag in pom under maven compiler plugin to prevent java compilation error.
4. **HTTP Request Configuration**

When configuring outbound call after message subscription, make sure to set connection timeout under client socket properties, setting generic connection timeout won’t be enough because for some reason despite setting connection timeout to 20 seconds, its taking 30 seconds, and message visibility timeout is set at 30 secs, so receiptHandle will expire by the time connection closes and will give SQS error **“SQS:INVALID Reason: Message does not exist or is not available for visibility timeout change”.**

1. **SQS Read message Connector Anomaly**

Read connector may or may not give top x messages in case of **standard queues/dlq** that’s why sometimes you will get empty response from read connector, even if the message is present whereas in case of fifo queue/dlq you will 100% get top x messages at once.

**Performance Metrics/Regulations**

* Free tier offers 1 million free API calls.
* Two different threads can’t compete for one single message
* Serverless processing with backpressure control achieved using queue in conjunction with lambda
* Lambda can scale to high throughputs , you can also rate limit lambda.
* Many thread/process can poll a queue at once. Only a single thread/process can process a message at once.
* Long polling is supported and encouraged, allows to save compute resources and cycles asking for messages.
* Support for cross account publishing.
* 256KB size per message in SQS
* Long polling is better than short polling.

1. **Demo Setup:**
   * Briefly explain the technical environment and tools utilized for the demo.
   * Demonstrate any necessary prerequisites or configurations required for seamless integration.

**Prerequisites:**

* Aws trial account: IAM user (Services: SQS,SNS,Lambda,EC2,EventBridge,API Gateway)
* JDK 17 for lambda code development.
* Putty, SSH Client for aws EC2 (Optional)
* Maven plugin and dependency Configuration, AWS Module Dependency Addition,
* IntelliJ/ Eclipse for creating **aws-stub-service** with **maven quickstart archetype**.
* Mule POM file configuration to compile Java 8 code.
* Postman.

1. **Live Demo:**
   * Conduct a step-by-step demonstration of the integration solution in action.

* b. Showcase the seamless flow of data between different systems or applications.
* c. Highlight any automated processes, data transformations, or validations occurring during the integration.

**Walkthrough:**

1. Explain a bit about dummy backend service created to simulate error scenarios
2. walkthrough REST Service
3. walkthrough Queues and Topics
4. walkthrough Publisher and Subscriber flows
5. explain error handling strategies.
6. explain idempotency.
7. walkthrough reprocessing flows.

**Test Plan:**

**Objective:** To showcase lifecycle of successful and failure message and features of REST api along with reprocessing strategies.

**Happy Path:**

* **Salesorder**
  + Publish 2 Salesorder,
  + Open backend to show Logs
  + Highlight Logs
* **Returnorder**
  1. **Deduplication with deduplication scope set to mesageGroup, and Content based deduplication set as true**
     + Publish 2 returnorder with same messageGrpId and same payload ,
     + Open backend to show that only one unique is received because of fifo queues features.
     + Highlight Logs
  2. **Show that message with different payload and same messageGropId will always be subscribed in order in single thread, queue will remain blocked until fierst message is processed and deleted**
     + Publish 2 returnorder with same messageGrpId different payload i.e for single customer ,
     + Show Logs that single thread is only spawned to process messages within single groupId
     + Open backend to show messages came in order
     + Highlight Logs
  3. **Show that message with different payload and different messageGropId will always be subscribed in different threads, this is high throughput mode because parallelism is achieved here.**
     + Publish 2 returnorder with different messageGrpId i.e returnorder for different customers,
     + Show Logs that two different threads spawned to process messages from different grpIds because returnorder from different customers are independent of each other and don’t need to be processed in order
     + Open backend to show returnorder came concurrently since they had different msgGrpIds
     + Highlight Logs
* **Purchaseorder**
  + Publish 1 Purchaseorder,
  + Show logs that product and shipping queue have picked message from purchaseorder topic.
  + Open backend to show Logs
  + Highlight Logs

**Error Path:**

* **Salesorder using Standard Queues and *MANUAL DLQ REDIRECT ERROR HANDLER***

1. **Show that non retyrable error messages are redirected to dlq without retry with error attributes.**
   * + Publish 1 salesorder with invalid data
     + Expect **HTTP:BAD\_REQUEST** from backend
     + Send message to dlq with setting error type and description
     + Delete message from current queue.
     + Raise Service now incident
2. **Show that retryable errors are retried for 3 times then redirected to dlq with error attributes.**
   * + Publish 1 salesorder with valid data.
     + Turn backend down or keep it waiting to simulate **HTTP:CONNECTIVITY(500) or HTTP:TIMEOUT(504).**
     + Expect connectivity or timeout error from backend.
     + Retry message for 3 times after some backoff time.
     + Then Send message to dlq with setting error type and description
     + Delete message from current queue.
     + Raise Service now incident
3. **Show duplicate message errors are suppressed as success response**

**Observation: Three cases in which duplicate message can come:**

**Case1 :** if **message is processed successfully** and **deleted out side of visibility timeout** then you will get same message twice with ApproximateReceiveCount = 2 (because AWS thinks you were not able to process message as you deleted it outside vis timout)

**Case 2:** if **message is processed successfully** and **deleted within visibility timout**, then you may or maynot get same message twice with same messageId and approximateReceiveCount=2 (because of AWS distributed architechure, i.e in case standard SQS multiple instances of messages are stored in different servers, and even when you delete the message within vis timout there are chances that it might not have got deleted in one server, so you might get delivered that message, (at least once delivery)

**Case 3:** if **message is retried** because of retry policy in case target is down.

**Outcomes of observation:**

We only want to allow **Case 3** types of **duplicate message** and restrict **Case 1** and **2** types of messages,

Its very hard to simulate Case 2 because it happens very occasionally and in case of large number of messages in flight.

We can reproduce Case 1 by putting breakpoint just before deleting the receipt handle and the going over it when visibility timeout exceeds. This way we get message retried because of Case 1 and we will reject this duplicate message with idempotent filter.

* **Returnorder using Standard Queues *and AUTO DLQ REDIRECT ERROR HANDLER***

1. **Show that non retyrable error messages are not redirected to dlq** 
   * + Publish 1 returnorder with invalid data
     + Expect **HTTP:BAD\_REQUEST** from backend
     + Simply log the non retryable error message and description and raise SNOW ticket
     + Delete message from current queue.
     + Raise Service now incident
2. **Show that retryable errors are retried for 3 times then automatically redirected to dlq without error attributes.**
   * + Publish 1 returnorder with valid data.
     + Turn backend down or keep it waiting to simulate **HTTP:CONNECTIVITY(500) or HTTP:TIMEOUT(504).**
     + Expect connectivity or timeout error from backend.
     + Retry message for 3 times after some backoff time.
     + Check if message is automatically Sent to dlq without error attributes and description
     + Message automatically Deleted from current queue.
     + Raise Service now incident
3. **Show implicit idempotency of fifo queues if possible**

Here we have not implemented idempotent strategy because Case 1 and Case 2 duplicate messages are not possible because of nature of FIFO queues ***(Exactly Once Delivery).***

And as for case 3, it will not need idempotency. It’s a retry message

WARNING: All the target configuration in mule is done keeping in mind that response timout is already known beforehand. Cases where response time is not know from target, will require dynamic message visibility extension capabilities.

**Its important that you load test target and record maximum and average response time, message visibility timeout will have to be set accordingly**

* **Purchaseorder using Standard Queues *and MANUAL DLQ REDIRECT ERROR HANDLER***

1. **Already shown in salesorder flow**

**Reprocessing Path:**

**ReprocessAlgoForManualMessages(x):**

* read max X messages from dlq,
* filter retryable messages,
* remove error attributes and mirror flowContext variables and payloads as it is
* then send them back to their queue in batch, once send is successful, delete all the message in batch from dlq.
* Respond the details of successful and failure messageIds while reprocessing

**ReprocessAlgoForManualMessagesCommonDlq(x):**

* read max X messages from dlq,
* filter retryable messages,
* group messages by messageTypes
* remove error attributes and mirror flowContext variables and payloads as it is
* then send them back to their source queues in batch, once send is successful, delete all the message in batch from dlq.
* Respond the details of successful and failure messageIds while reprocessing

**ReprocessAlgoForAutoMessages(x):**

* read max X messages from dlq,
* mirror flowContext variables and payloads as it is
* then send them back to their queue in batch, once send is successful, delete all the message in batch from dlq.
* Respond the details of successful and failure messageIds while reprocessing
* **Salesorder**
  + **Using Mule-SQS based Reprocessor**
    - Trigger MULE REST endpoint to reprocess salesorder dlq with param X= maxNumberOfMessageToRedrive
    - Call **ReprocessAlgoForManualMessages(X)**
    - Check if backend service is getting all the reprocessed messages or not.
  + **Using Mule-Java based Reprocessor**
    - Trigger MULE REST endpoint to reprocess salesorder dlq with param X= maxNumberOfMessageToRedrive.
    - Call **ReprocessAlgoForManualMessages(X)**
    - Check if backend service is getting all the reprocessed messages or not.
  + **Using Mule-Lambda based Reprocessor**
    - Trigger MULE REST endpoint to reprocess salesorder dlq with param X= maxNumberOfMessageToRedrive.
    - Call **ReprocessAlgoForManualMessages(X)** on remote server on AWS via Mule Lambda connector.
    - Check if backend service is getting all the reprocessed messages or not.
  + **Using AWS Eventbridge-Lambda-SQS based Reprocessor**
    - Enable AWS Eventbridge service
    - Event bridge will Call **ReprocessAlgoForManualMessages(X)** in lambda service in fixed interval.
    - Check if backend service is getting all the reprocessed messages or not.
* **Returnorder**
  + **Using Mule-SQS based Reprocessor**
    - Trigger MULE REST endpoint to reprocess returnorder dlq with param X= maxNumberOfMessageToRedrive
    - Call **ReprocessAlgoForAutoMessages(X).**
    - Check if backend service is getting all the reprocessed messages or not.
* **Purchaseorder**
  + **Using Mule-SQS based Reprocessor**
    - Trigger MULE REST endpoint to reprocess returnorder dlq with param X= maxNumberOfMessageToRedrive
    - Call **ReprocessAlgoForManualMessagesCommonDlq(X)**
    - Check if backend service is getting all the reprocessed messages or not.

**Rest Service:**

Create dummy queue/topic and set properties for them if asked by panelists otherwise leave it. Focus on important stuff

1. **Benefits and Value Proposition:**
   * Discuss the tangible benefits offered by the integration solution.
   * Emphasize how it effectively addresses the challenges identified earlier.
   * Present evidence of efficiency gains, cost savings, or enhanced user experiences resulting from the integration.
2. **Use Case Examples:**
   * Provide additional use cases or scenarios where the integration solution proves valuable.
   * Illustrate its versatility and applicability across various industries or domains.

**Airline ticket system:** Fifo queues could come in handy in ATS where you want to maintain order of events like you don’t want user to give ability to select seats before payment.

**GPS location tracking**: To avoid showing out of order pings we can leverage fifo queues.

Using standard queues as utility for simple data orchestration across organization.

1. **Q&A Session:**
   * Invite questions and encourage interactive discussions from the audience.
   * Address any queries or concerns raised, ensuring clarity and understanding.

Open to any questions/suggestions.

1. **Closing Remarks:**

* Summarize the key takeaways from the demo.
* Reiterate the benefits of adopting the integration solution.

**Future Scopes:**

* Retry logic for handling failed messages while reprocessing from dlq.
* Incorporate **dynamic visibility timeout** extension when processing time is not known beforehand (In experiment).

**Limitation of this approach**

* + MuleSoft restricts the nested call depth to default value of 25 i.e. in case nested call depth reaches beyond 25 MuleSoft throws below exception:

Message: Too many child contexts nested.

Error type: MULE:CRITICAL

To overcome this issue, set: org.mule.runtime.core.privileged.event.BaseEventContext.maxDepth

Higher than 25 and as per your requirement.

As we are accumulating all the data, so it can also cause Memory issues. Keep this point in mind while using this approach.

Scenarios where this can be implemented:

* + This approach can be used when we don’t know how many records we need to fetch and some looping is not achievable using For Each Loop.

**Resources:**

* **Commands**
  + **Maven/Mule Commands:**
    - **Deploy Application on Exchange:**

mvn clean deploy -DclientID=23c7e68d140a4a9588511e7c3bc416bf -DclientSecret=26EAFe4D5D764c9C96941cfb58e1255a -s settings.xml

* + - **Deploy Application on Cloudhub 1.0 using mule maven plugin**

mvn clean deploy -DmuleDeploy -Dregion=us-east-2 -Dworkers=1 -DworkerType=MICRO -Denvironment=Sandbox -DappName=ecom-order-ext-eapi -DclientID=23c7e68d140a4a9588511e7c3bc416bf -DclientSecret=26EAFe4D5D764c9C96941cfb58e1255a -Denv=dev -Dkey=1234567812345678 -Danypoint.platform.client\_id=f3c1d78dbe504863b4e4c73b96f3eac7 -Danypoint.platform.client\_secret=7633383955B941f29b3eD2f16Bae7305 -s settings.xml

* + - **Deploy Application on Cloudhub 2.0 using mule maven plugin**

mvn clean deploy -DmuleDeploy -DtargetName=Cloudhub-US-East-2 -Denvironment=Sandbox -Dreplicas=1 -DvCores=0.1 -DappName=ecom-order-ext-eapi -DclientID=23c7e68d140a4a9588511e7c3bc416bf -DclientSecret=26EAFe4D5D764c9C96941cfb58e1255a -Denv=dev -Dkey=1234567812345678 1234567812345678 -Danypoint.platform.client\_id=\*\*\*\* -Danypoint.platform.client\_secret=\*\*\*\*\* -s settings.xml

* + - **Encrypt yaml files**

java -cp secure-properties-tool.jar com.mulesoft.tools.SecurePropertiesTool file encrypt AES CBC 1234567812345678 config-secure-dev.yaml dev-out.yaml

* + **Putty Commands:**
    - **Upload jar file to ec2 instance**

pscp -scp -i ec2-key.ppk oms-backend-0.0.1-SNAPSHOT.jar [ec2-user@ec2-13-211-167-26.ap-southeast-2.compute.amazonaws.com:~](mailto:ec2-user@ec2-13-211-167-26.ap-southeast-2.compute.amazonaws.com:~)

* + - **Perform write permission on pem key**

chmod 400 OmsPemKey.pem

* + - **Connect to ec2 instance**

ssh -i OmsPemKey.pem [ec2-user@ec2-13-211-167-26.ap-southeast-2.compute.amazonaws.com](mailto:ec2-user@ec2-13-211-167-26.ap-southeast-2.compute.amazonaws.com)

* + - **Install JDK 17 if instance created for the first time**

sudo yum install java-17-amazon-corretto

* + - **Deploy the jar file with output logs enabled**

java -jar oms-backend-0.0.1-SNAPSHOT.jar > output.log &

* **Linkes/Indexs/Codes/Collections:**
  + **AWS Links**

**IAM account ID:** 891376909160

**Root user:**awasthi.kedareshwar@gmail.com

**IAM user:** testuser

Console sign-in URL: <https://891376909160.signin.aws.amazon.com/console>

AWS Policy generator: <https://awspolicygen.s3.amazonaws.com/policygen.html>

* + **SNOW credentials**

Instance name: **dev191862**

Instance URL: <https://dev191862.service-now.com/>

Username: **admin**

* **Postman collections**
  + **Mule: ecom-order-ext-eapi collection url:**

**Cloudhub:** <https://github.com/kedarawasthi/aws-poc-files/blob/main/Postman%20Collections/ecom-order-ext-eapi%20cloudhub.postman_collection.json>

**Local:** <https://github.com/kedarawasthi/aws-poc-files/blob/main/Postman%20Collections/ecom-order-ext-eapi.postman_collection.json>

* + **Others: oms-backend collection url ( (lambda,EC2,Local) inclusive):**

<https://github.com/kedarawasthi/aws-poc-files/blob/main/Postman%20Collections/oms-backend-api.postman_collection.json>

* **API specifications:**
  + **Raml Code Repository:** <https://github.com/kedarawasthi/aws-poc-files/tree/main/Api%20Specification/ecom-order-ext-eapi>
  + **Exchange url:** <https://anypoint.mulesoft.com/exchange/portals/deloitte-india-llp/a41e7b82-f610-496e-8898-19374947af3a/ecom-order-ext-eapi/>
* **Code Repositories:**
  + **Mule: ecom-order-ext-eapi:** <https://github.com/kedarawasthi/ecom-order-ext-eapi>
  + **Mule-Aws Lambda DLQ service:** <https://github.com/kedarawasthi/aws-stub-service>
  + **Springboot oms-backend:** <https://github.com/kedarawasthi/oms-backend>
  + **Lambda oms-backend url:** <https://github.com/kedarawasthi/lambda-oms-backend>
* **Documentation:**

**Document Url:** https://github.com/kedarawasthi/aws-poc-files/blob/main/aws-poc-document.docx