**Solution Design - Affirm Use Case**

**Overview**

The primary task for this solution to replicate and solve is to create a system by which data flows in batches that contain specific object/table names and acceptable values. The system should run on a scheduler and implement an objectstore to only capture the data since the last run.

*Requirements*

* Scheduled singular daily call to source system, gathering all data since last run
* Data to be transported from source system to destination system, then replicated into tables based on a country value in the payload
* There are at least three valid country values: USA, CAN, AUS
* Must be able to handle high traffic, high data transactions using batching

**Proposed Solution**

Starting with a simple raml specification in Design Center, build an API that runs a scheduled call at 10PM EST to scrape data from a source tables to one of four tables present in another database. Three tables are the requested country tables, the fourth is a cases\_FAILED table to hold any records that do not have a valid country code and any future data that does not fit into the required tables.

*Workflow*

The API starts with a scheduler with a cron expression, running every day at 10PM EST. When the workflow is triggered, the API checks for a last run date in the Object Store, then attempts to select all records from the source system with a date greater than or equal to the date from the store.

Once the records have been pulled, the batch processor transforms each into a json payload. Each payload has its country data value checked against the three valid options and then is either sorted into the correct one or written to a separate table to hold invalid/failed records.

The batch processing is then output in a logger and the last run date is update to today’s date/time in the Object Store.

*Dataset*

Each destination table and the source table have the following columns

* caseID - Primary Key, varchar(10)
* fname - first name, varchar(15)
* lname - last name, varchar(15)
* language - user language, varchar(10)
* country - record country, varchar(3)
* lastWriteDate - date last updated, datetime (MM-DD-YYYY HH:MM:SS)

*Table Structure*

* Source System
  + cases\_SOURCE - initial data source
* Destination System
  + cases\_USA - table for all records with payload.country == USA
  + cases\_CAN - table for all records with payload.country == CAN
  + cases\_AUS - table for all records with payload.country == AUS
  + cases\_FAILED - table for all records without one of the three valid payload.country values

*Assumptions*

* Data format is consistent between source and destination
* Transitional data format is JSON, source and destination data sources are SQL
* Scheduled job is run during low-traffic hours for the region it services to avoid impacting user uptime (assumed downtime window is 10PM EST to 5AM EST)
  + If there is no currently stored Last Write Date, the current timestamp minus 24 hours is used to calculate a date for the Select
* The acceptable amount of failed writes is 1000, or 1% of the expected average dataset size
  + Note, failed writes in this case would mean a failed write from source system to destination system, not an invalid country value. Those are considered a successful write but are placed in their own cases\_FAILED table.

*Dependencies*

* ObjectStore (Mulesoft provided)
* Mule Secure Properties (Mulesoft provided)

**Outcomes**

*Limitations*

The core use cases have been successfully implemented, but there are some limitations. Primarily, the limitations are related to how the system handles imperfect/flawed data: without one of the 3 valid country code assigned to the payload, the only way to rectify imperfect data would be to manually update the records with a correct value after the system has processed all of them. Future refinement could be used to add more checks and validation modules to not only confirm the key data is correct but to also filter out data errors that might be rectifiable in an algorithmic way from data that must be corrected by a user.

The tables are currently the only part of the call that isn’t configured as Muelsoft system variable, due to constraints on the provided Mulesoft SQL modules. There are workarounds but they are moderately “hacky” and might lead to a small loss of efficiency that isn’t offset by the gain in ease of configuration. The tables have been hardcoded.

Another large limitation comes in the form of a lack of sample data to test with. Though batching has been applied and handles small datasets easily, before moving forward with validation/deployment this API would need strenuous testing with a dataset more appropriate to the use case presented. There are elements present in the system to start handling large data, an example being the max batch block size (concurrent processing records) increasing from the default of 10 to 1000, but true testing of those capabilities is best done with a much larger dataset than present in the spun-up system for this technical assessment.

Finally, error notification outside of standard Mule logging has been mostly ignored. Similar to the testing limitation above, this would be something to work out with IT support groups (if any) to either fully leverage Anypoint Monitoring or a third party solution like Splunk to handle notification of IT resources about errors in processing.

*Future Work*

Though the base features requested are complete and functional, there are a number of outstanding parts of the code that could be expanded upon:

* Reprocessing of failed records, which are currently written to cases\_FAILED in destination table and not reprocessed.
  + Proposed solution: additional API support structure to scrape the cases\_FAILED table
* Error notification is still handled within the Mule API itself, without external communications or Anypoint Platform being included.
  + Proposed Solution: integration with a logging solution (such as splunk) or configuration in Anypoint Platform for logging notifications (Anypoint Monitoring)
* API currently removes all records from the initial data source that it processes, this behavior could be detrimental to business data requirements/reprocessing needs.
  + Proposed Solution: determine the preferred handling of failed records and create a secondary flow/subflow that notifies business/IT representatives of the data failure + possible rectifications
  + An example: an error subflow that would attempt to reprocess the cases\_FAILED table by stripping special characters, unwanted newlines/whitespaces, and other elements that typically cause writes to fail and attempting to run a batch job on the fixed records. Any records left after that processing would likely require a notification subflow to have a user rectify the data manually.
* The API currently only triggers at the specified time, it has proven useful in the past to have some kind of manual trigger to initiate the API in case of emergency data needs/reprocessing.
  + Proposed Solution: Add an additional Mule flow that replaces the scheduler with an HTTP Listener that allows for the passing of lastWriteDate and any other input parameters for the initial select
  + Configuration properties have been left in for an HTTP listener to be added in later refinement and the original HTTP listener used in development is only commented out in demo-case-scheduledjob-sapi