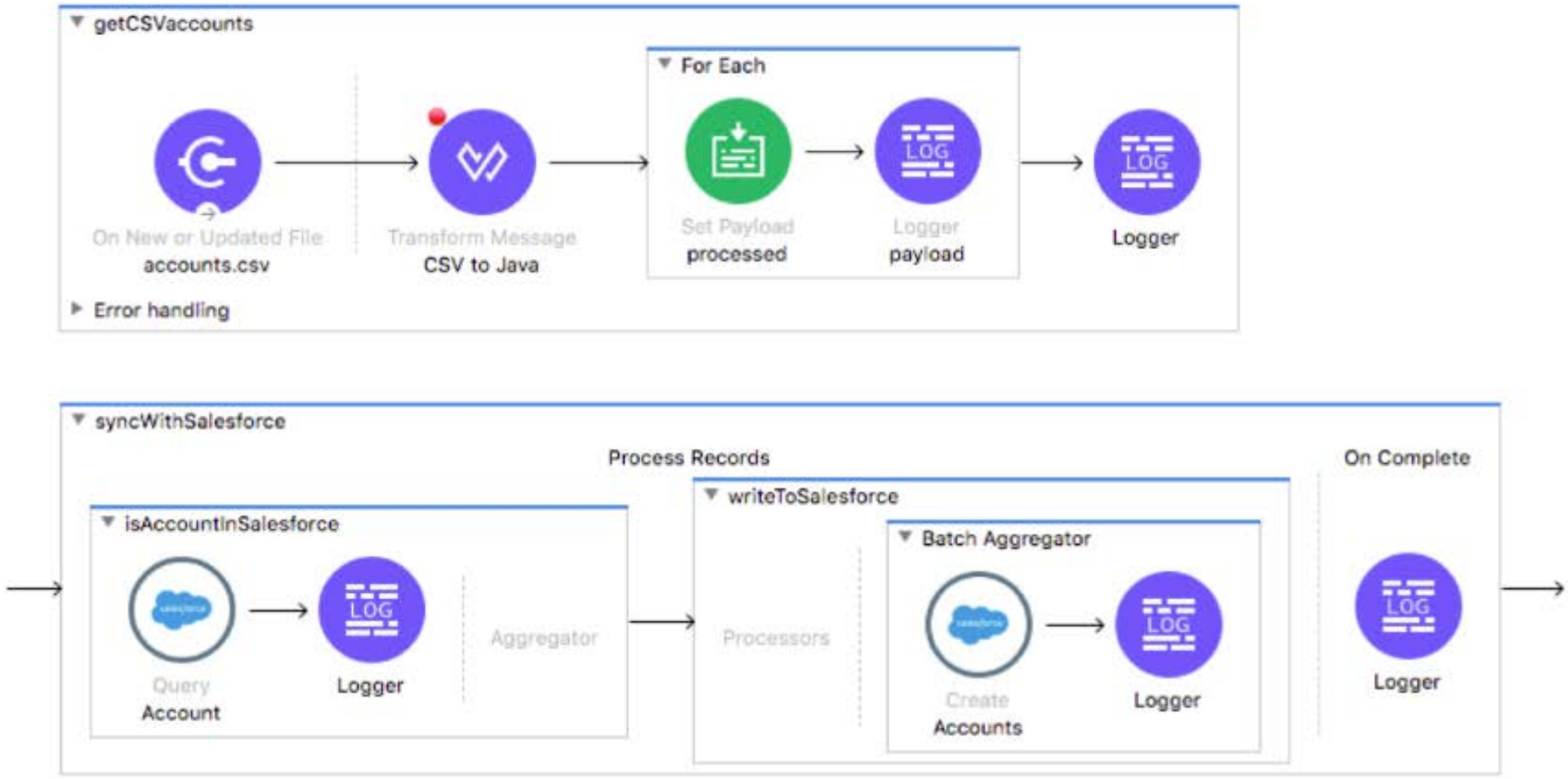

MuleSoft
Module 13: Processing Records

Goal



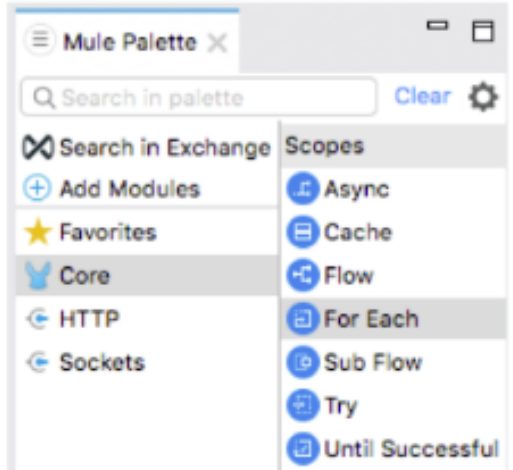
At the end of this module, you should be able to

- Process items in a collection using the For Each scope
- Process records using the Batch Job scope
- Use filtering and aggregation in a batch step

Processing items in a collection with the For Each scope

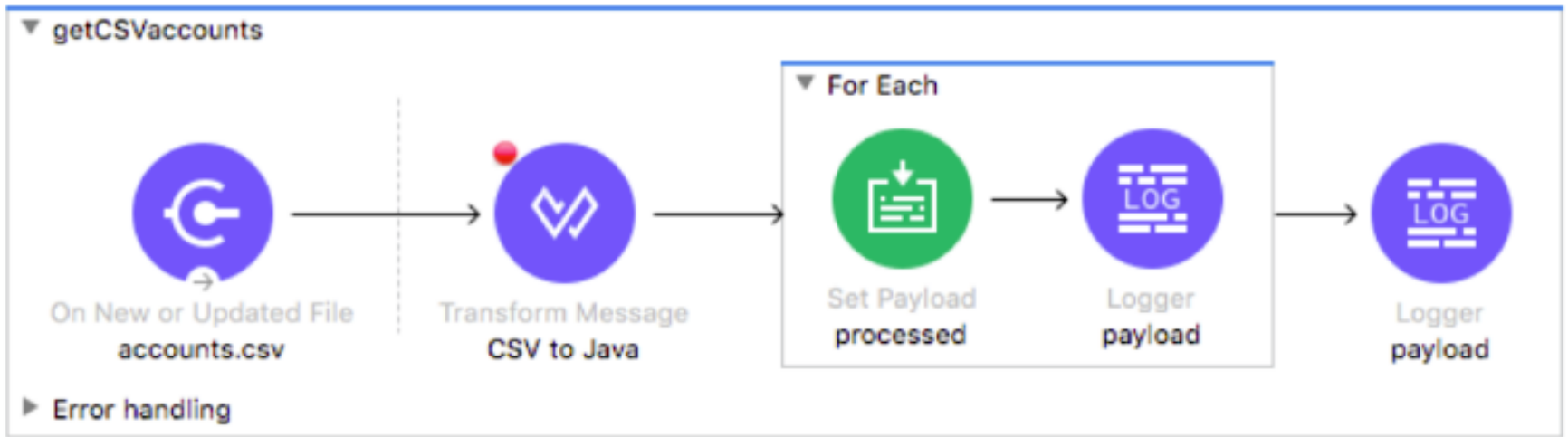
The For Each scope

- Splits a payload collection and processes the individual elements
  - Collection can be any supported content type, including application/json, application/java, or application/xml
- Returns the original payload
  - Regardless of any modifications made inside the scope
- Stops processing and invokes an error handler if one element throws an exception



Walkthrough 13-1: Process items in a collection individually

- Use the For Each scope element to process each item in a collection individually
- Change the value of an item inside the scope
- Examine the payload before, during, and after the scope
- Look at the thread used to process each item



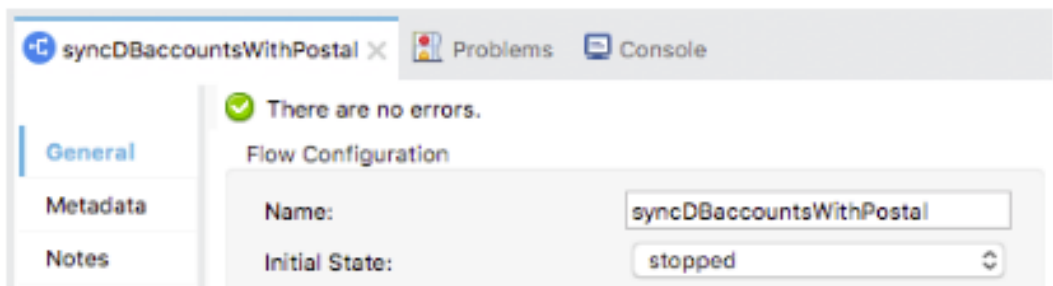
Walkthrough 13-1: Process items in a collection using the For Each scope

In this walkthrough, you split a collection and process each item in it. You will:

- Use the For Each scope element to process each item in a collection individually.
- Change the value of an item inside the scope.
- Examine the payload before, during, and after the scope.
- Look at the thread used to process each item.

Stop the syncDBaccountsWithPostal flow so it does not run

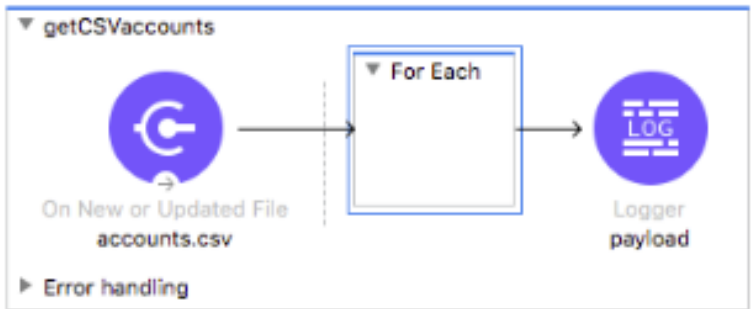
1. Return to accounts.xml.
2. In the properties view for the syncDBaccountsWithPostal flow, set the initial state to stopped.



3. Right-click in the canvas and select Collapse All.

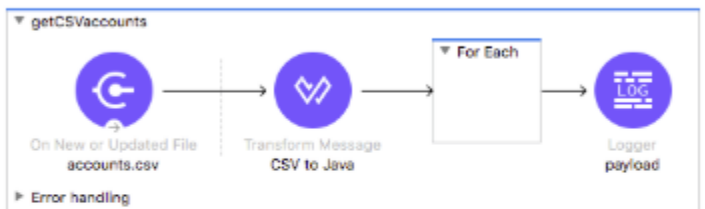
Add a For Each scope

4. Expand getCSVaccounts.
5. In the Mule Palette, select Core.
6. Locate the For Each scope and drag and drop it before the Logger.

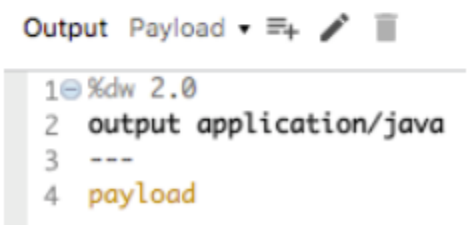


Transform the input to a collection

7. Add a Transform Message component before the For Each scope.
8. Set its display name to CSV to Java.

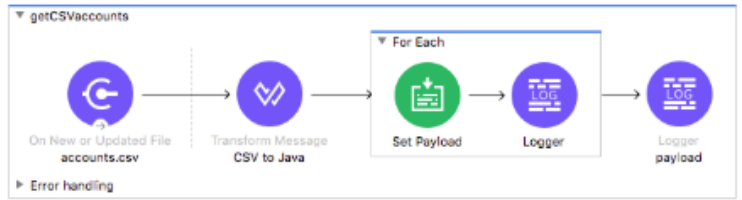


9. In the Transform Message properties view, leave the output type set to java and set the expression to payload.

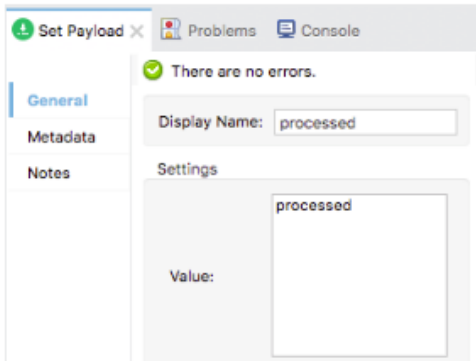


Process each element in the For Each scope

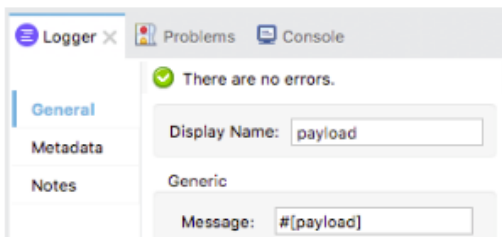
10. Add a Set Payload transformer and a Logger to the For Each scope.



11. In the Set Payload properties view, set the display name and value to the string: processed.



12. Set the Logger display name to payload and have it display the payload.

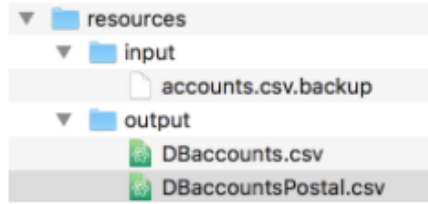


Change the File listener so it does not rename files

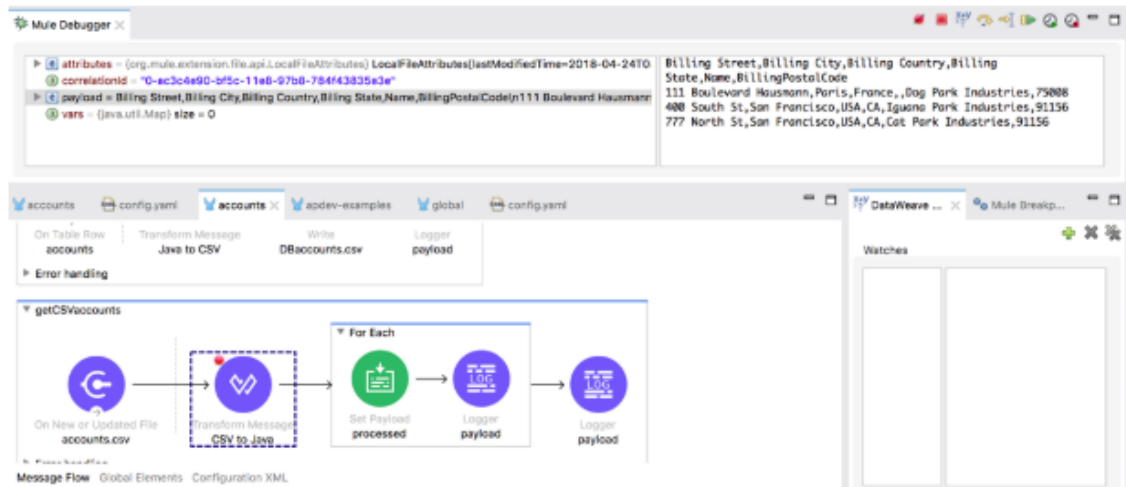
13. In the On New or Updated File properties view, delete the rename to value.

Debug the application

14. Add a breakpoint to the Transform Message component.
15. Debug the project and do not clear application data.
16. Return to the course student files in your computer's file browser.

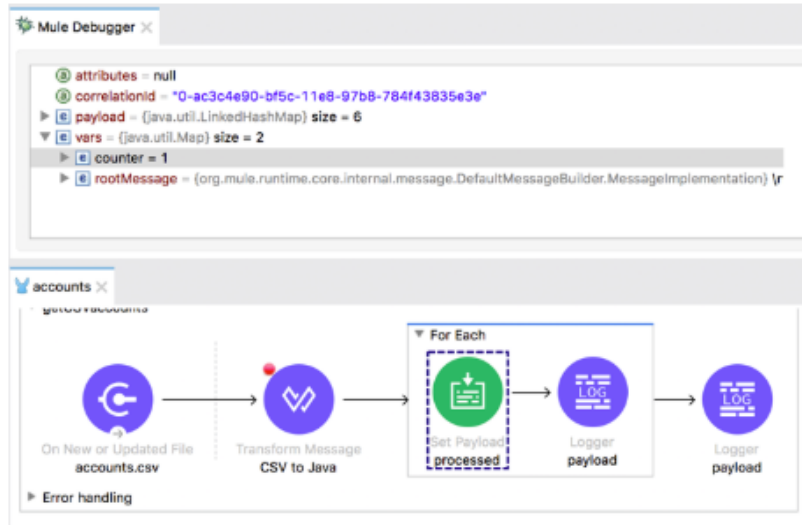


17. Rename accounts.csv.backup to accounts.csv.
18. Return to Anypoint Studio; application execution should have stopped at the Transform Message component.
19. In the Mule Debugger view, look at the payload type and value.

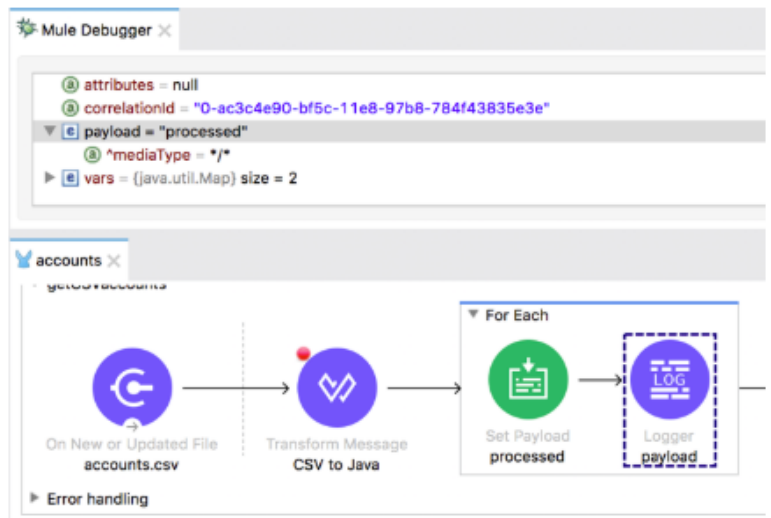


20. Step to the For Each scope; the payload should now be an ArrayList of LinkedHashMaps.

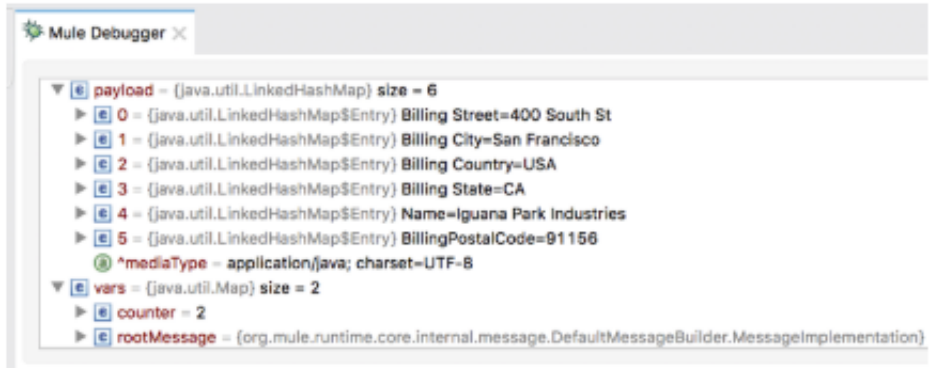
21. Step into the Set Payload in For Each scope; the payload should be a LinkedHashMap.
22. Expand Variables; you should see a counter variable.



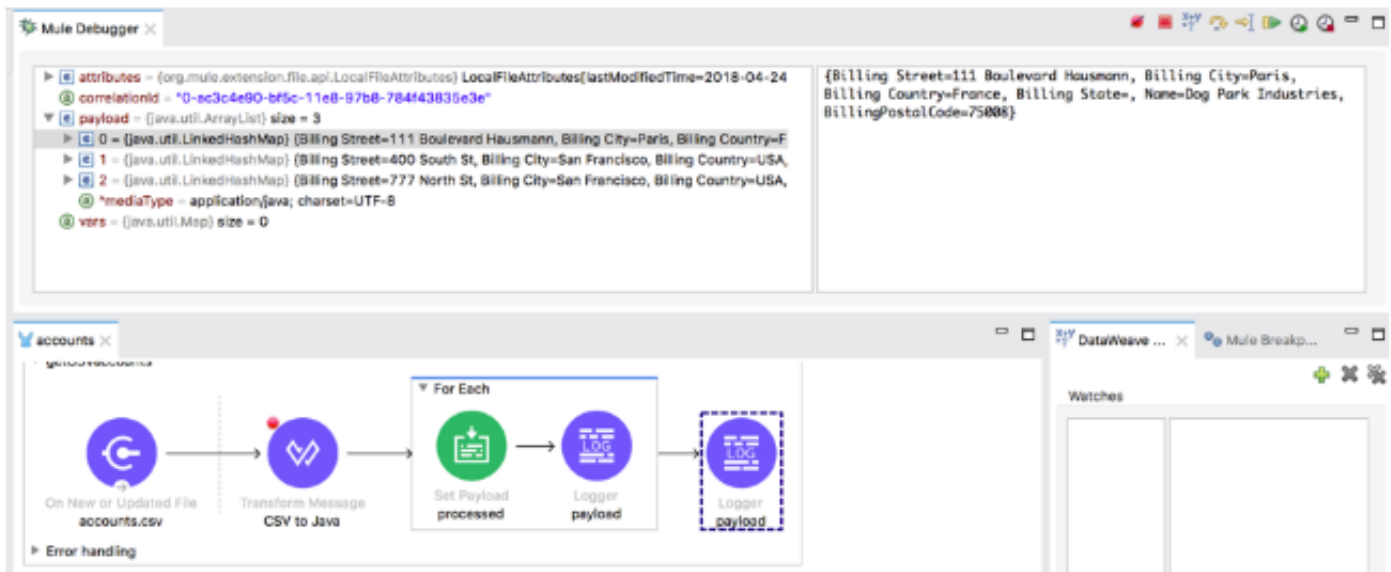
23. Step again; you should see the payload for this record inside the scope has been set to the string, processed.



24. Step again and look at the payload and counter for the second record.



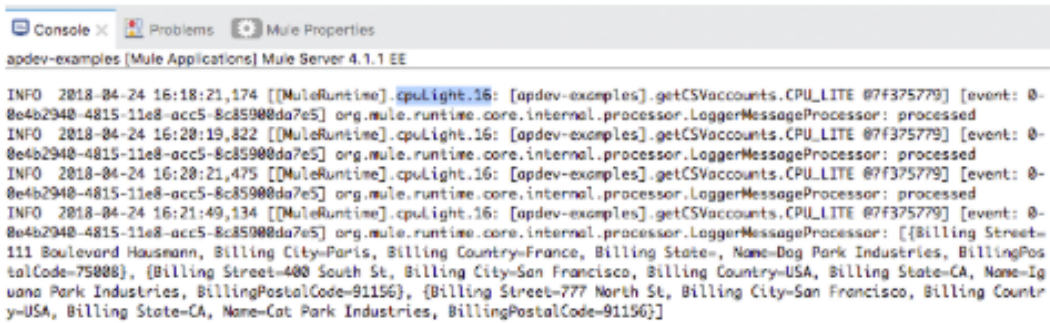
25. Step through the application to the Logger after the For Each scope; the payload should be equal to the original ArrayList of HashMaps and not a list of processed strings.



26. Step to the end of the application.
27. Stop the project and switch perspectives.

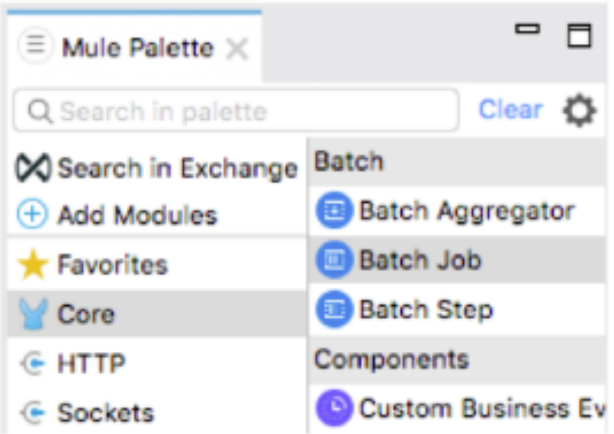
Look at the processing threads

28. In the console, locate the thread number used to process each item in the collection; the same thread should be used for each: cpuLight.16 in the following screenshot.



Processing records with the Batch Job scope

The Batch Job scope



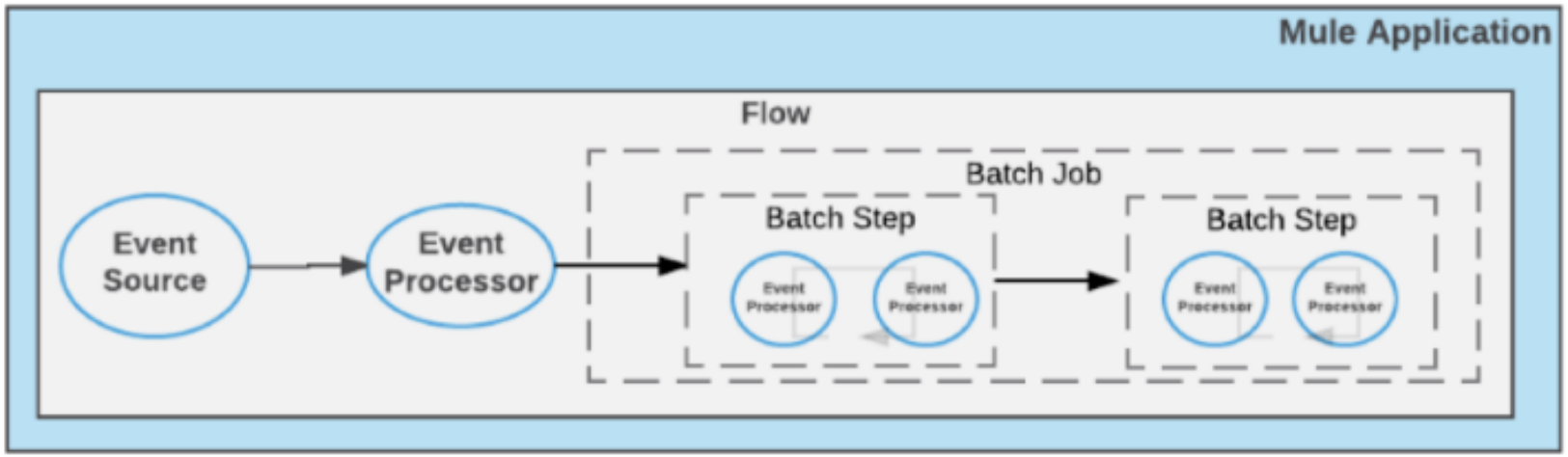
- Provides ability to split large messages into records that are processed asynchronously in a batch job
- Created especially for processing data sets
  - Splits large or streamed messages into individual records
  - Performs actions upon each record
  - Handles record level failures that occur so batch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

Example use cases

- Integrating data sets to parallel process records
  - Small or large data sets, streaming or not
- Engineering "near real-time" data integration
  - Synchronizing data sets between business applications
  - Like syncing contacts between NetSuite and Salesforce
- Extracting, transforming, and loading (ETL) info into a target system
  - Like uploading data from a flat file (CSV) to Hadoop
- Handling large quantities of incoming data from an API into a legacy system

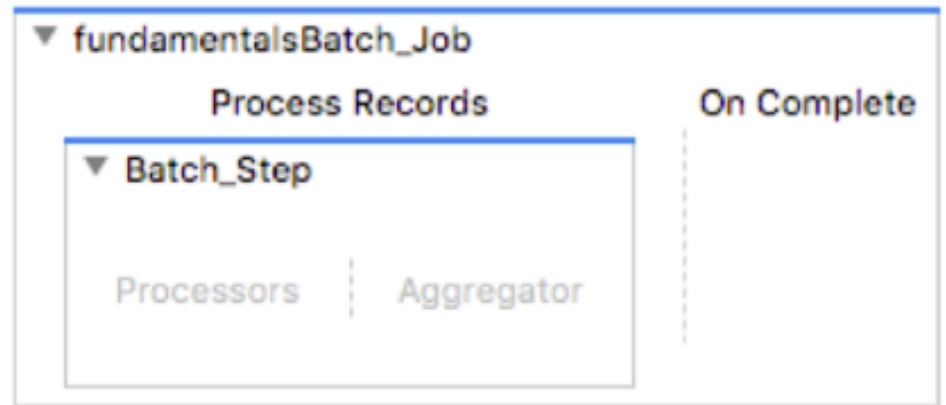
How a batch job works

- A batch job contains one or more batch steps that act upon records as they move through the batch job



A batch job contains three different phases

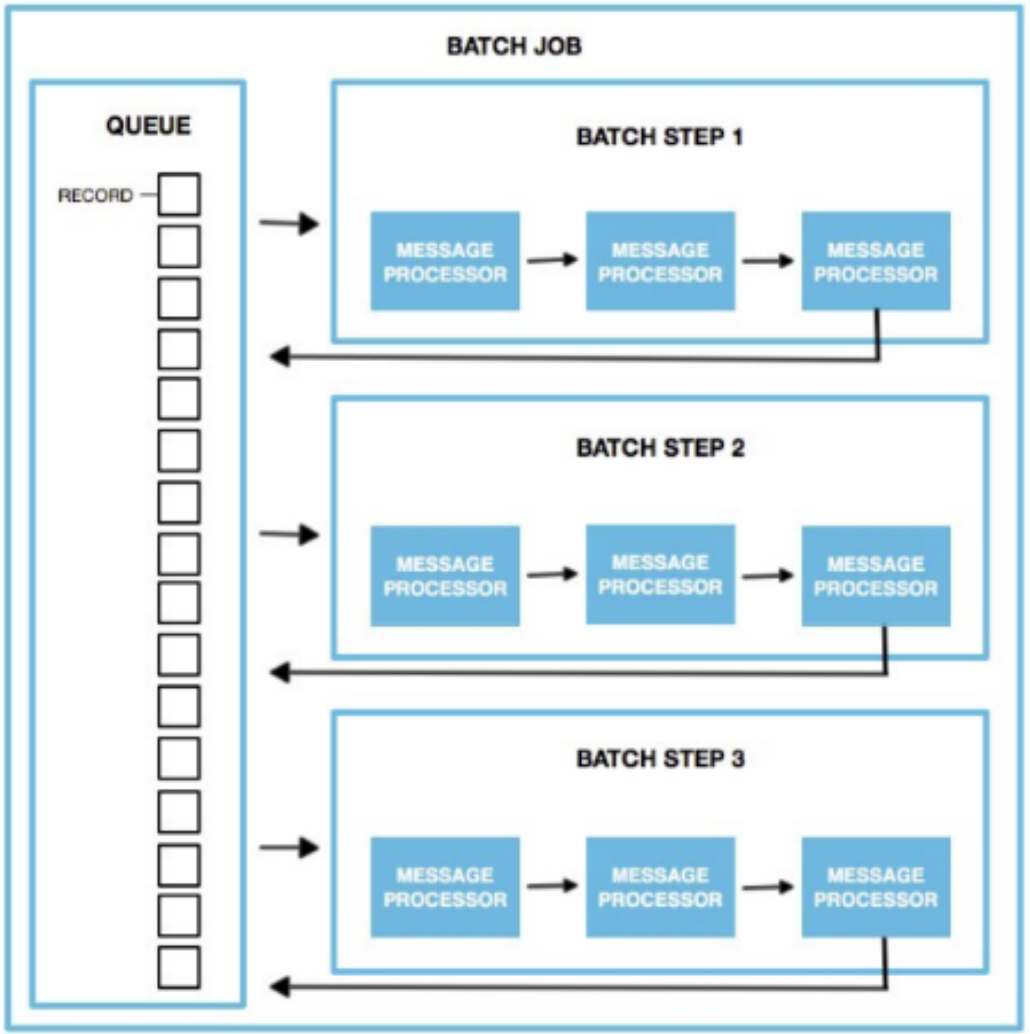
- **Load and dispatch** (implicit)
  - Performs "behind-the-scene" work
    - Splits payload into a collection of records
    - Creates a persistent queue and stores each record in it
- **Process** (required)
  - Asynchronously processes the records
  - Contains one or more batch steps



- **On complete** (optional)
  - Reports summary of records processed
  - Provides insight into which records failed so you can address issues

How record processing works

- One queue exists
- Each record
  - Keeps track of what steps it has been processed through
  - Moves through the processors in the first batch step
  - Is sent back to the queue
  - Waits to be processed by the second step
- This repeats until each record has passed through every batch step
- **Note**: All records do not have to finish processing in one step before any of them are sent to the next step



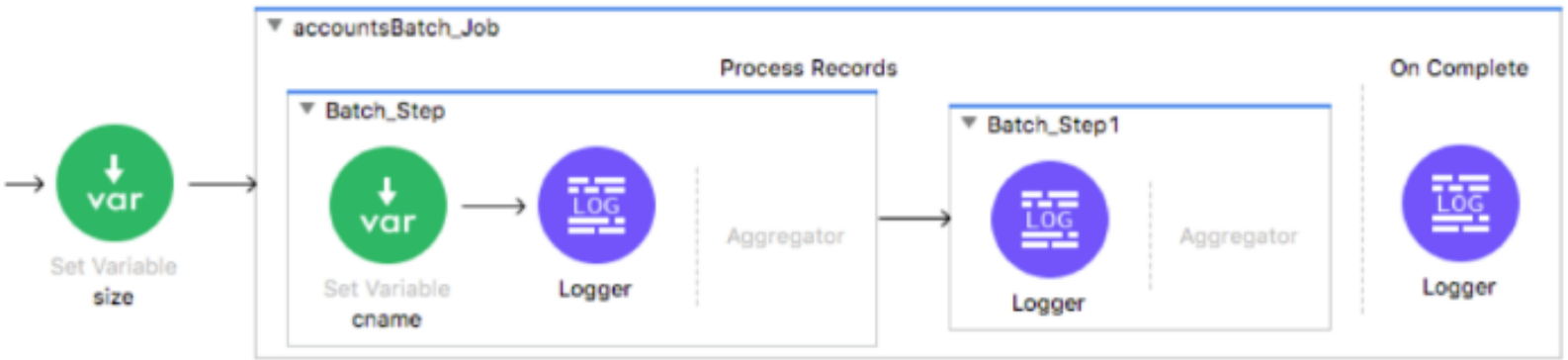
Batch job performance

- Batch records are queued and scheduled in blocks of 100
  - This lessens the amount of I/O requests and improves an operation's load
- A threading profile of 16 threads per job is used
  - Each of the 16 threads processes a block of 100 records
  - Each thread iterates through that block processing each record, and then each block is queued back and the process continues
- This configuration works for most use cases, but can be customized to improve batch's performance in certain use cases

Variables in a batch job

- Variables created before a batch job are available in all batch steps

- Variables created inside a batch step are record-specific
  - Persist across all batch steps in the processing phase
  - Commonly used to capture whether or not a record already exists in a database

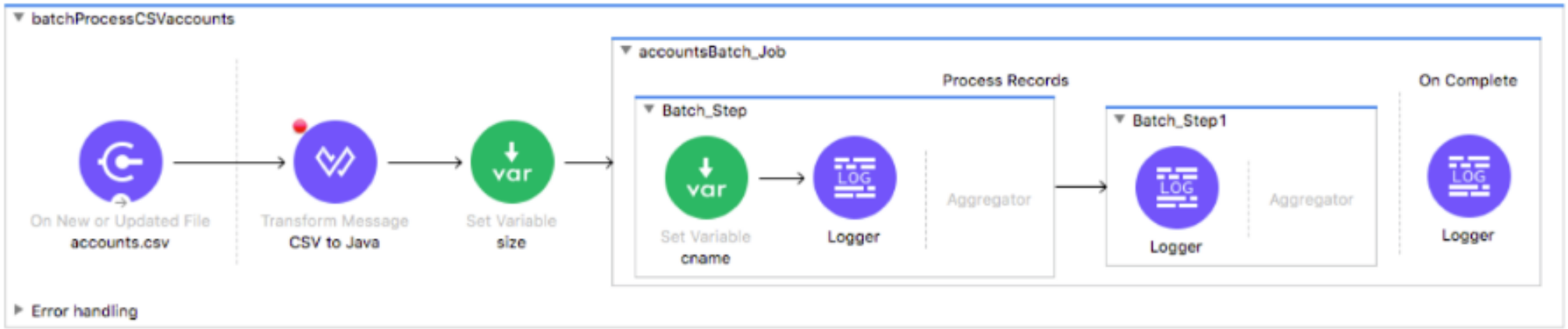


Handling record-level errors during processing

- If a record fails to be processed by a processor in a batch step, there are three options
  - **Stop processing** the entire batch (default)
    - In-flight steps are finished, but all other steps are skipped and the on complete phase is invoked
  - **Continue processing** the batch
    - You need to specify how subsequent batch steps should handle failed records
      - To do this, use batch step filters that are covered in the next section
  - **Continue processing** the batch **until a max number** of failed records is reached
    - At that point, the on complete phase is invoked

Walkthrough 13-2: Create a batch job for records in a file

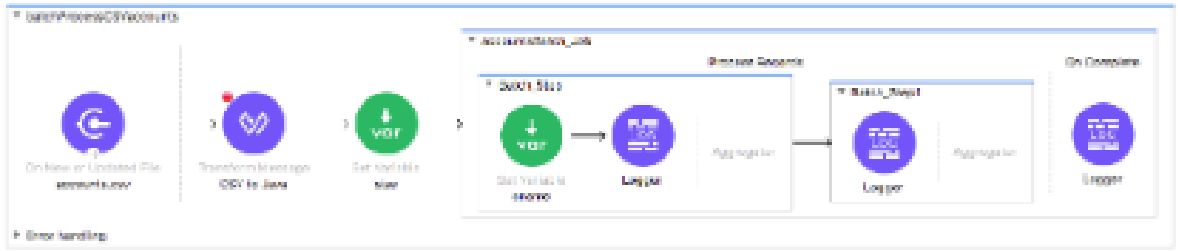
- Use the Batch Job scope to process items in a collection
- Examine the payload as it moves through the batch job
- Explore variable persistence across batch steps and phases
- Examine the payload that contains information about the job in the on complete phase
- Look at the threads used to process the records in each step



Walkthrough 13-2: Process records using the Batch Job scope

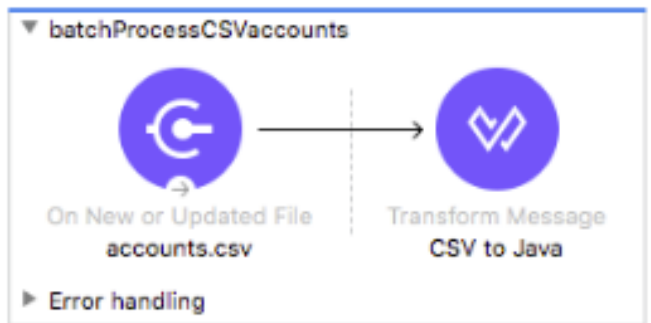
In this walkthrough, you create a batch job to process the records in a CSV file. You will:

- Use the Batch Job scope to process items in a collection.
- Examine the payload as it moves through the batch job.
- Explore variable persistence across batch steps and phases.
- Examine the payload that contains information about the job in the on complete phase.
- Look at the threads used to process the records in each step.

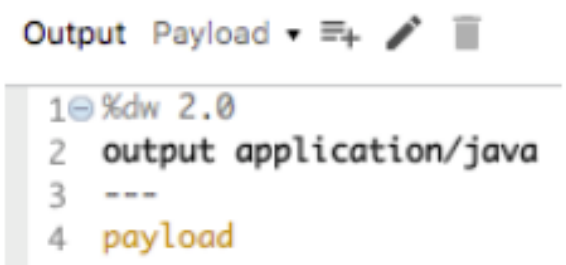


Create a new flow to read CSV files

1. Return to accounts.xml.
2. Drag a Flow scope from the Mule Palette and drop it above getCSVaccounts.
3. Change the flow name to batchProcessCSVaccounts.
4. Select the On New or Updated File operation in getCSVaccounts and select Edit > Copy.
5. Click the Source section of batchProcessCSVaccounts and select Edit > Paste.
6. Modify the display name.
7. Add a Transform Message component to the flow.
8. Set the display name to CSV to Java.

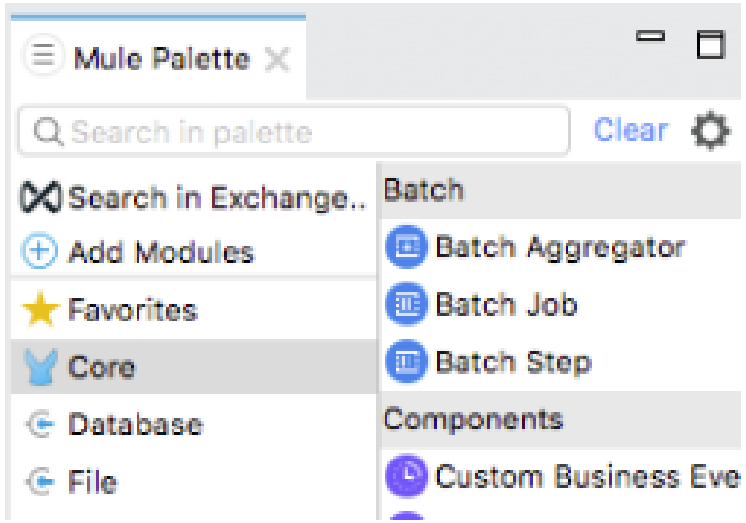


9. In the Transform Message properties view, change the body expression to payload.

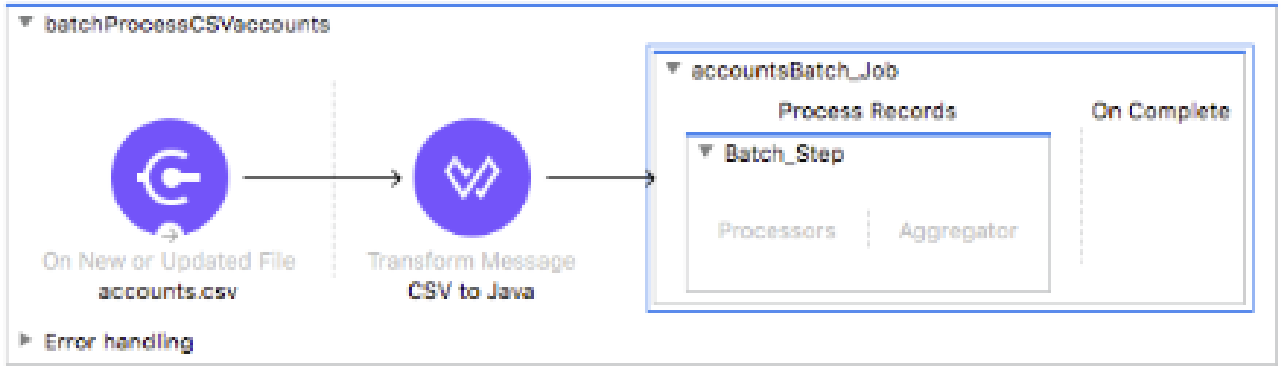


Add a Batch Job scope

10. In the Mule Palette, select Core and locate the Batch elements.

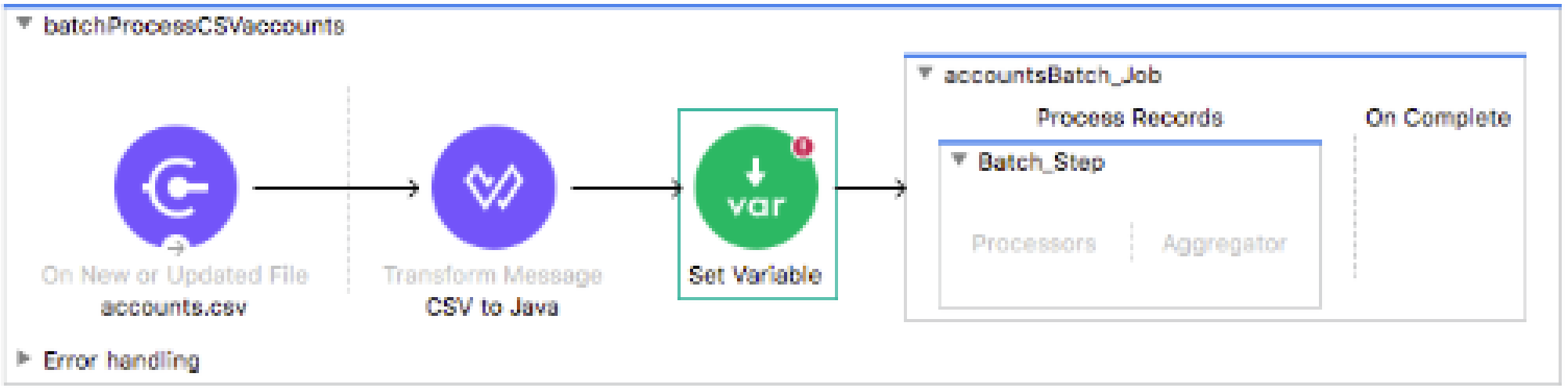


11. Drag a Batch Job scope and drop it after the Transform Message component.



Set a variable before the Batch Job scope

12. Add a Set Variable transformer before the Batch Job scope.

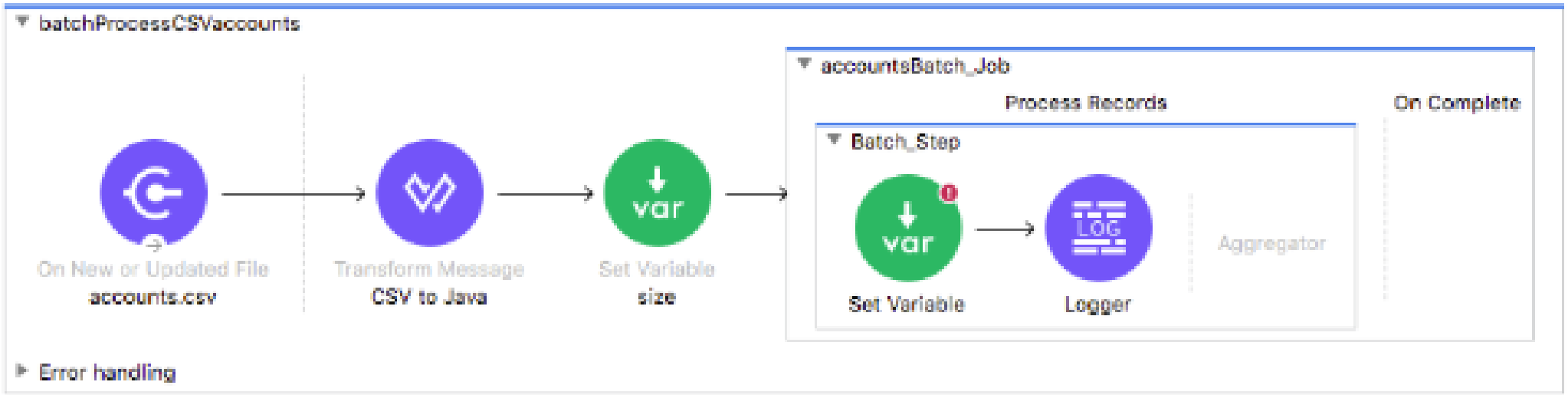


13. In the Set Variable properties view, set the following:

- Display Name: size
- Name: size
- Value: #[sizeOf(payload)]

Set a variable inside the Batch Job scope

14. Add a Set Variable transformer to the batch step in the processors phase of the batch step.
15. Add a Logger component to the batch step after the transformer.

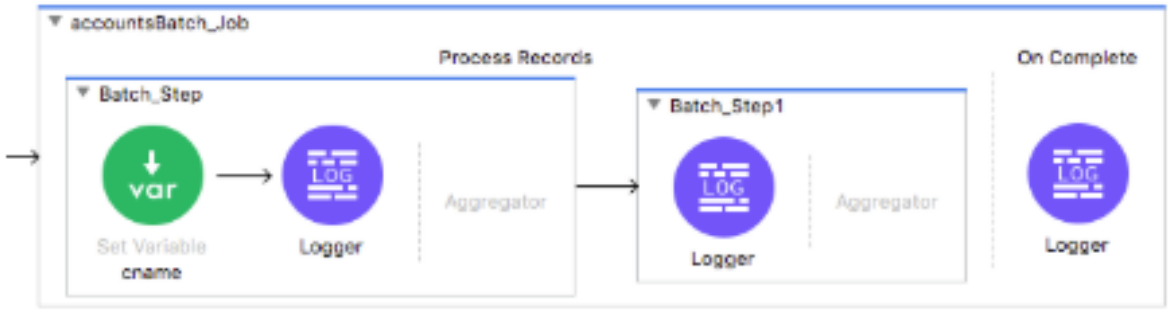


16. In the Set Variable properties view, set the following:

- Display Name: cname
- Name: cname
- Value: #[payload.Name]

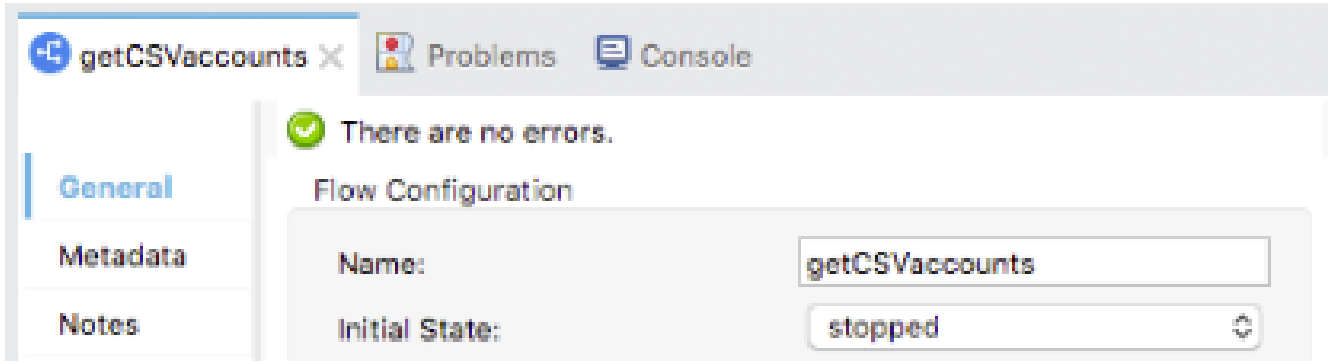
Create a second batch step

17. Drag a Batch Step scope from the Mule Palette and drop it in the process records phase of the Batch Job scope after the first batch step.
18. Add a Logger to the processors section of the second batch step.
19. Add a Logger to the On Complete phase.



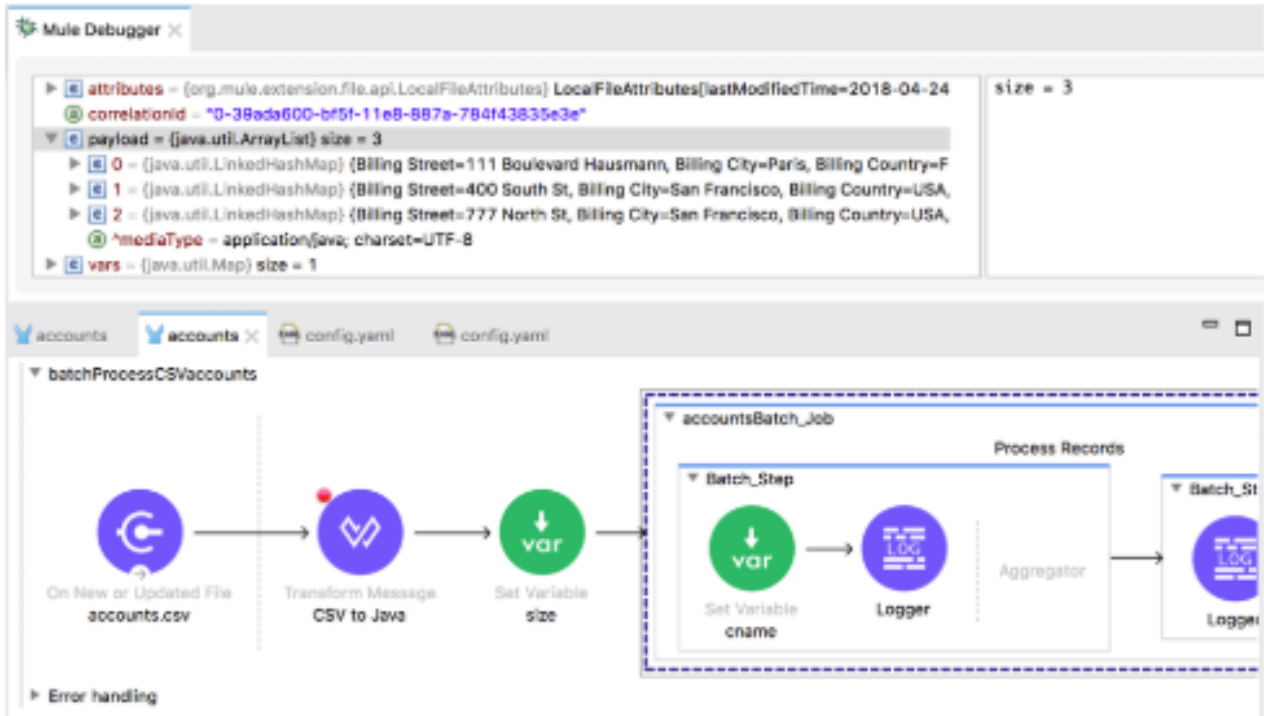
Stop the other flow watching the same file directory from being executed

20. In the properties view for the getCSVaccounts flow, set the initial state to stopped.



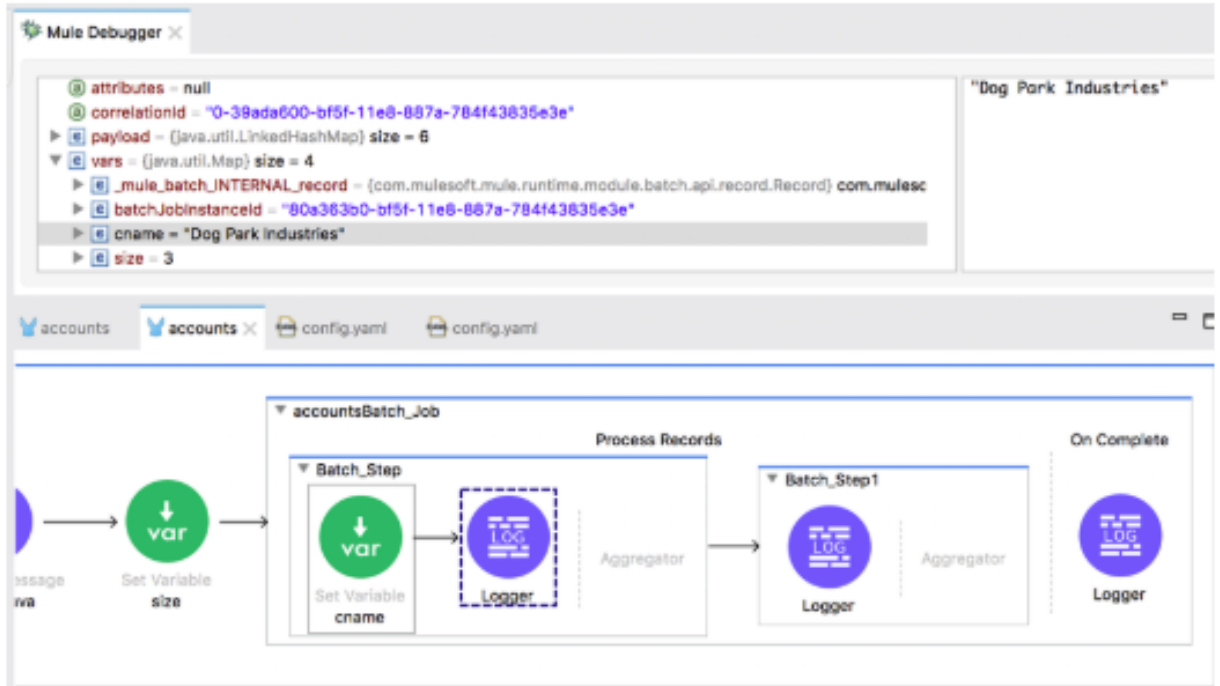
Debug the application

21. In batchProcessCSVaccounts, add a breakpoint to the Transform Message component.
22. Save the file to redeploy the application in debug mode.
23. Return to the course student files in your computer's file browser and move accounts.csv from the output folder to the input folder.
24. In the Mule Debugger view, watch the payload as you step to the Batch Job scope; it should go from CSV to an ArrayList.
25. In the Mule Debugger view, expand Variables; you should see the size variable.



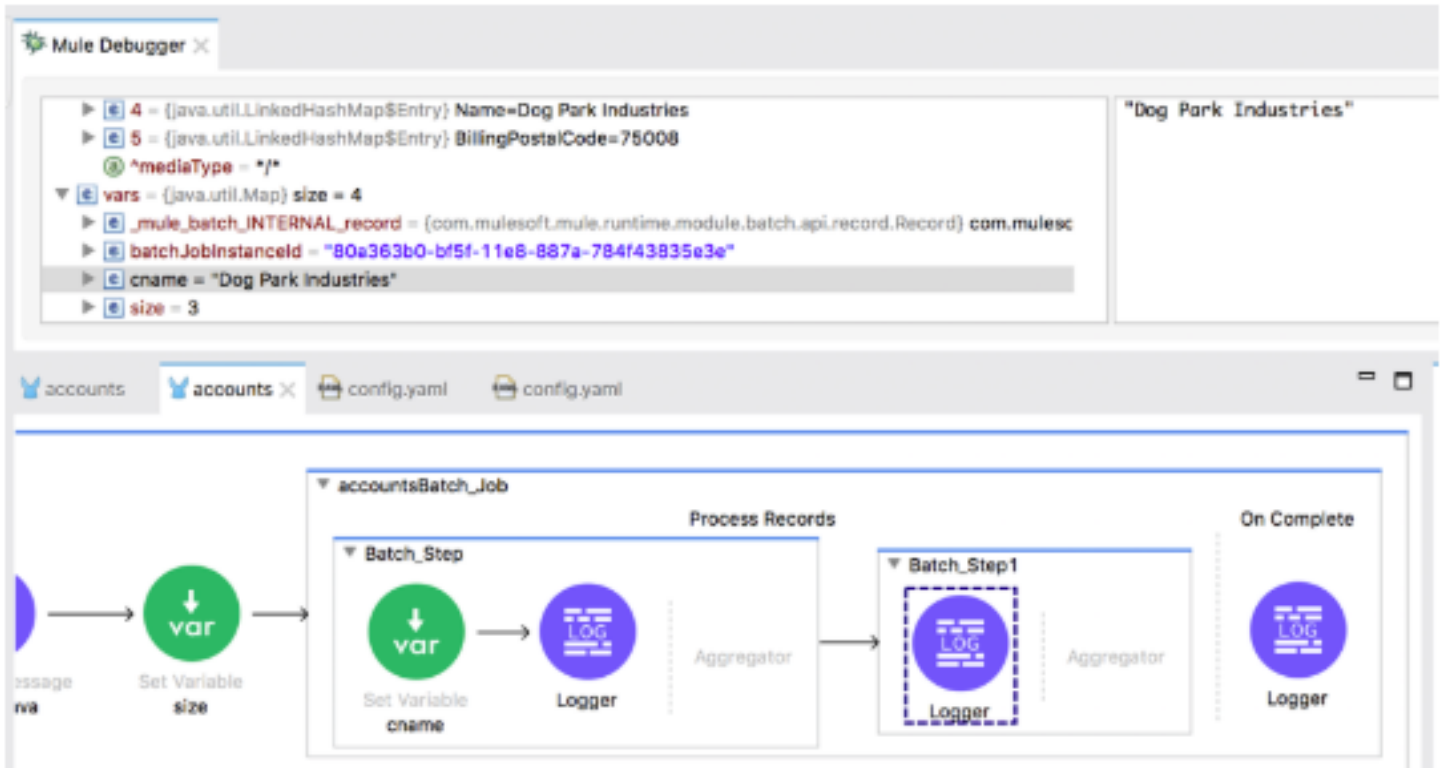
26. Step to the Logger in the first batch step.
27. In the Mule Debugger view, look at the value of the payload; it should be a HashMap.

28. Expand Variables; you should see the size variable and the cname variable specific for that record.



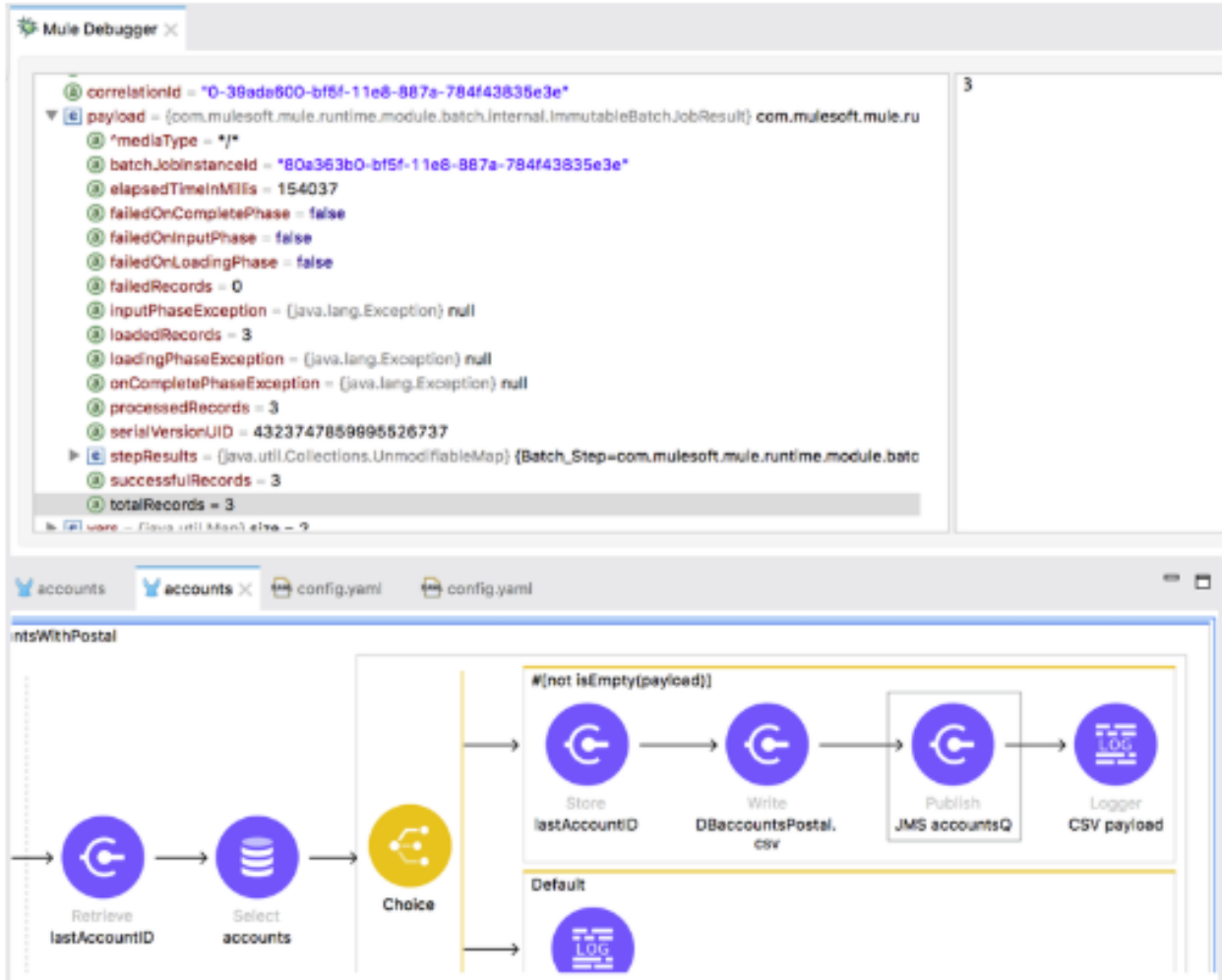
29. Step through the rest of the records in the first batch step and watch the payload and the cname variable change.

30. Step into the second batch step and look at the payload and the cname variable; you should see the cname variable is defined and has a value.



31. Step through the rest of the records in the second batch step and watch the value of cname.
32. Step into the on complete phase; you should see the payload is an ImmutableBatchJobResult.

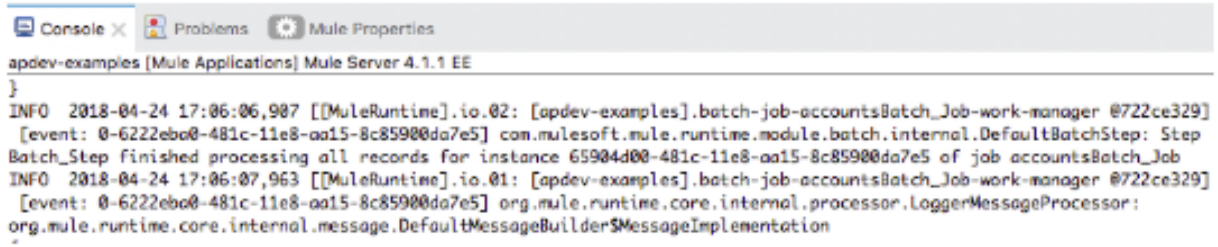
33. Expand the payload and locate the values for totalRecords, successfulRecords, and failedRecords.



34. Step through the rest of the application and switch perspectives.
35. Stop the project.

Look at the processing threads

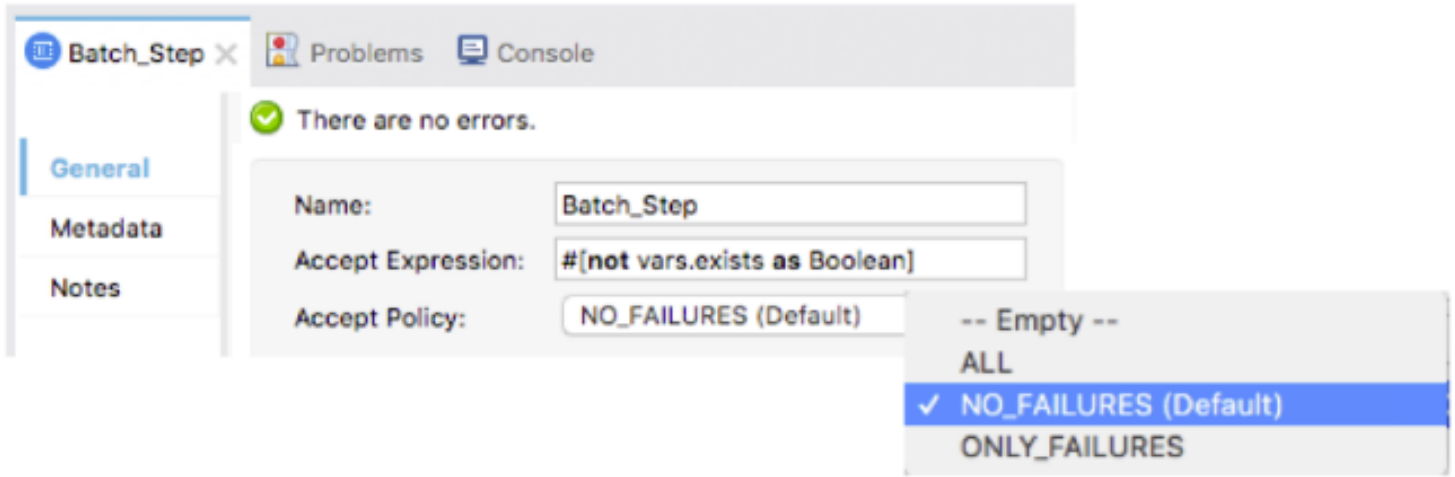
36. In the console, locate the thread number used to process each record in the collection in each step of the batch process; you should see more than one thread used.



Using filtering and aggregation in a batch step

Using filters to specify when a batch step is executed

- A batch job has two attributes to filter the records it processes
  - An accept expression
  - An accept policy

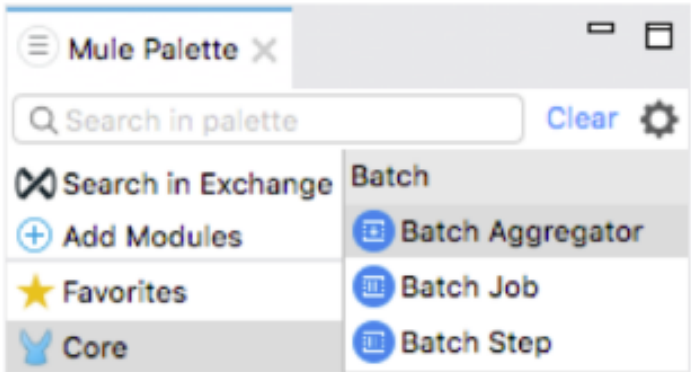


- Examples
  - Prevent a step from processing any records which failed processing in the preceding step
  - In one step, check and see if the record exists in some data store and then in the next only upload it to that data store if it does not exist

Aggregating records in a batch step for for bulk insert

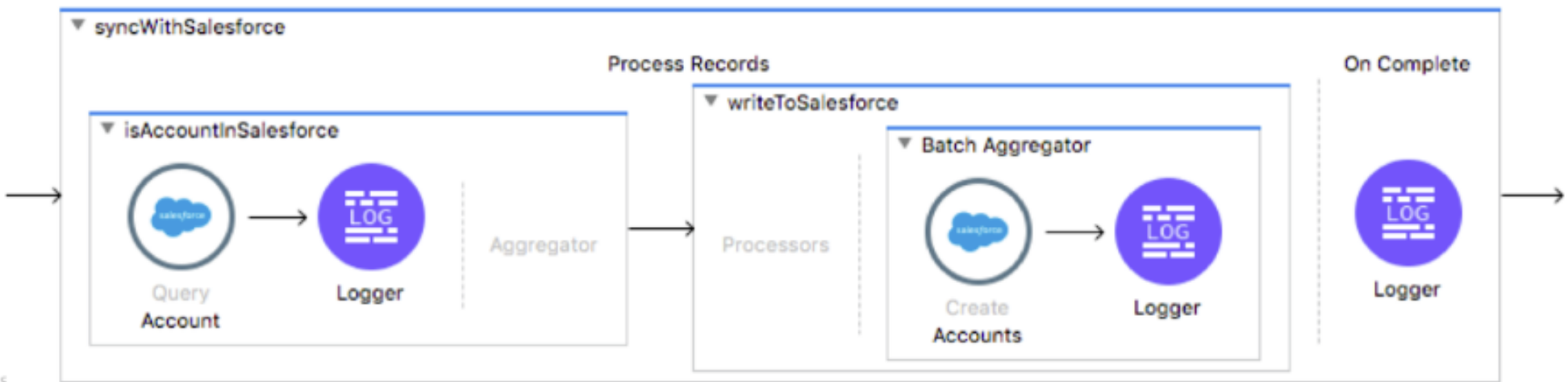
- To accumulate records, use a **Batch Aggregator** scope inside the Aggregator section of a batch step

- For example, instead of using a separate API call to upsert each record to a service, upload them in a batch of 100



Walkthrough 13-3: Use filtering and aggregation in a batch step

- Use a batch job to synchronize database records to Salesforce
- In a first batch step, check to see if the record exists in Salesforce
- In a second batch step, add the record to Salesforce
- Use a batch step filter so the second step is only executed for specific records
- Use a Batch Aggregator scope to commit records in batches



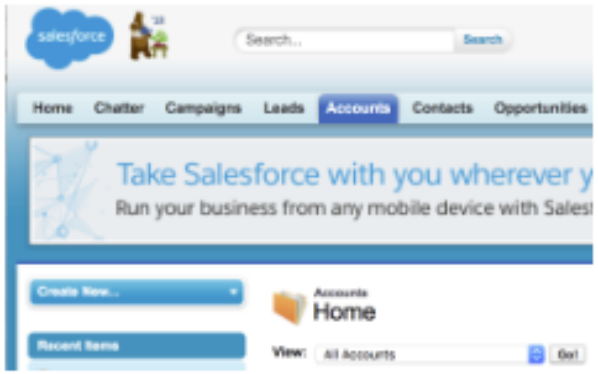
Walkthrough 13-3: Use filtering and aggregation in a batch step

In this walkthrough, you use a batch job to synchronize account database records to Salesforce. You will:

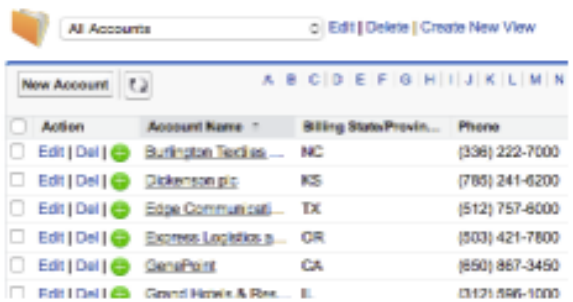
- Use a batch job to synchronize database records (with your postal code) to Salesforce.
- In a first batch step, check to see if the record already exists in Salesforce.
- In a second batch step, add the record to Salesforce.
- Use a batch step filter so the second batch step is only executed for specific records.
- Use a Batch Aggregator scope to commit records in batches.

Look at existing Salesforce account data

1. In a web browser, navigate to http://login.salesforce.com/ and log in with your Salesforce Developer account.
2. Click the Accounts link in the main menu bar.
3. In the view drop-down menu, select All Accounts and click the Go button.

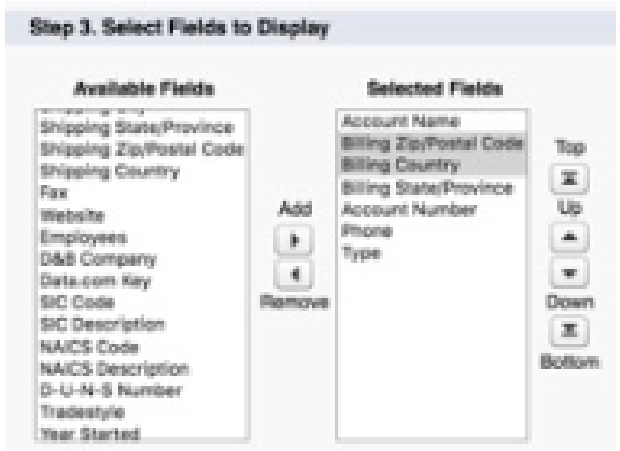


4. Look at the existing account data; a Salesforce Developer account is populated with some sample data.

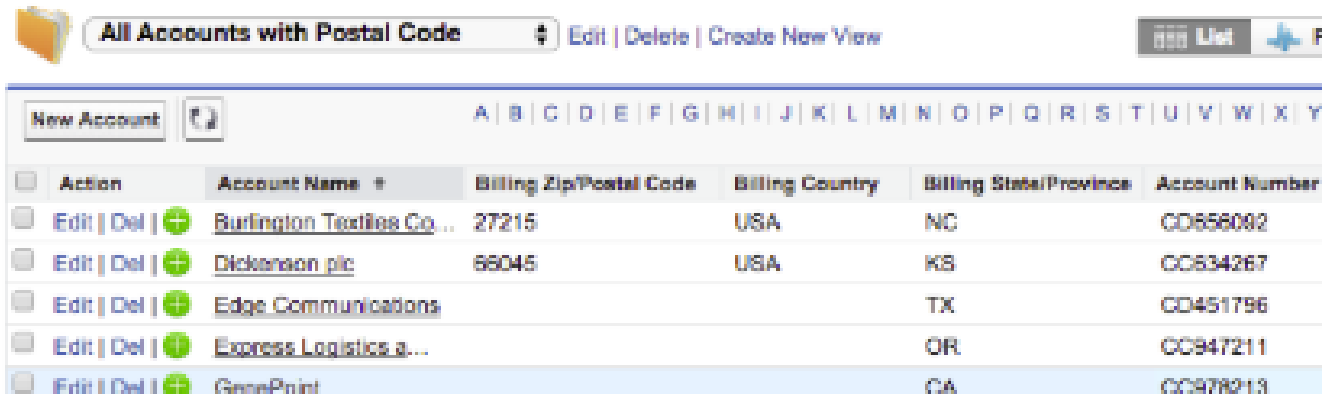


5. Notice that countries and postal codes are not displayed by default.
6. Click the Create New View link next to the drop-down menu displaying All Accounts.
7. Set the view name to All Accounts with Postal Code.
8. Locate the Select Fields to Display section.

9. Select Billing Zip/Postal Code as the available field and click the Add button.
10. Add the Billing Country field.
11. Use the Up and Down buttons to order the fields as you prefer.

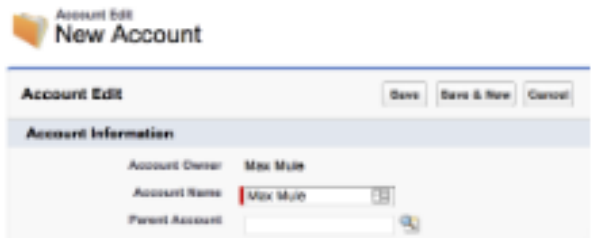


12. Click the Save button; you should now see all the accounts with postal codes and countries.



Add an account to Salesforce with a name matching one of the database records

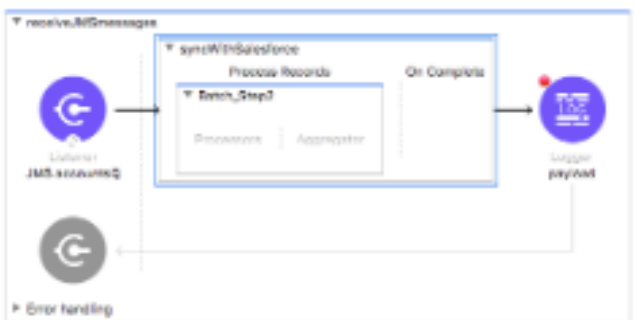
13. Click the New Account button.
14. Enter an account name (one that matches one of the accounts you added with your postal code to the database) and click Save.



15. Leave this window open.

Add a Batch Job scope to the receiveJMSmessages flow

16. Return to accounts.xml in Anypoint Studio.
17. Drag a Batch Job scope from the Mule Palette and drop it before the Logger in the receiveJMSmessages flow.
18. Change the display name of the batch job to syncWithSalesforce.



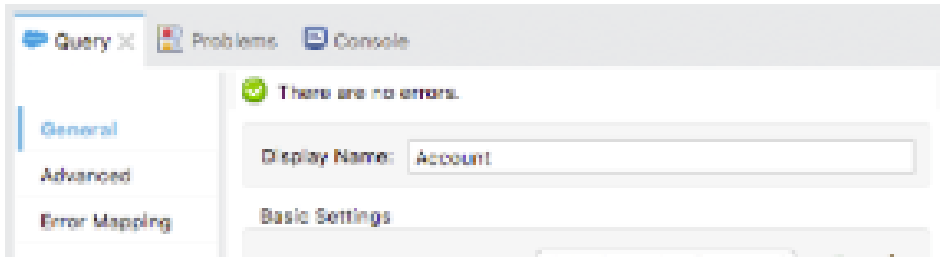
Query Salesforce to see if an account already exists

19. Change the name of the batch step inside the batch job to isAccountInSalesforce.
20. Drag a Salesforce Query operation from the Mule Palette and drop it in the processors phase in the batch step.

21. In the Query properties view, set the following values:

    - Display Name: Account
    - Connector configuration: Salesforce_Config
    - Salesforce query: SELECT Name FROM Account

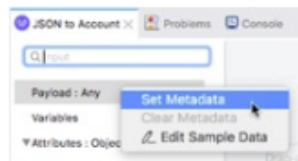
    Note: You will add to this query shortly.



Transform the input JSON data to Salesforce Account objects

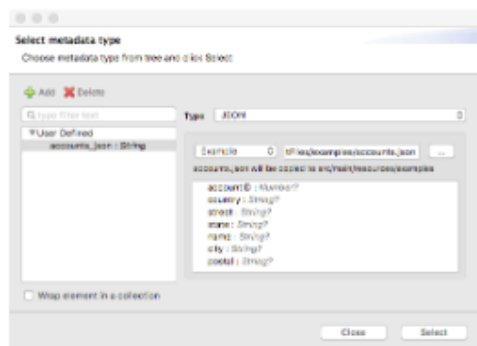
22. Add a Transform Message component before the Batch Job.
23. Change the display name to JSON to Accounts.

24. In the Transform Message properties view, look at the metadata in the input section.
25. Right-click Payload and select Set Metadata.



26. Create a new metadata type with the following information:

- Type id: accounts_json
- Type: JSON
- Example: *Use the accounts.json file in the examples folder of the course student files*



27. In the Transform Message properties view, map the following fields:

- country: BillingCountry
- street:BillingStreet
- state: BillingState
- name: BillingPostalCode
- city: BillingCity
- postal:BillingPostalCode

Finish the batch step to check if an account already exists in Salesforce

28. In the Query properties view, add an input parameter named cname.
29. Set the parameter value to payload.Name, ensure it is a String, and give it a default value.

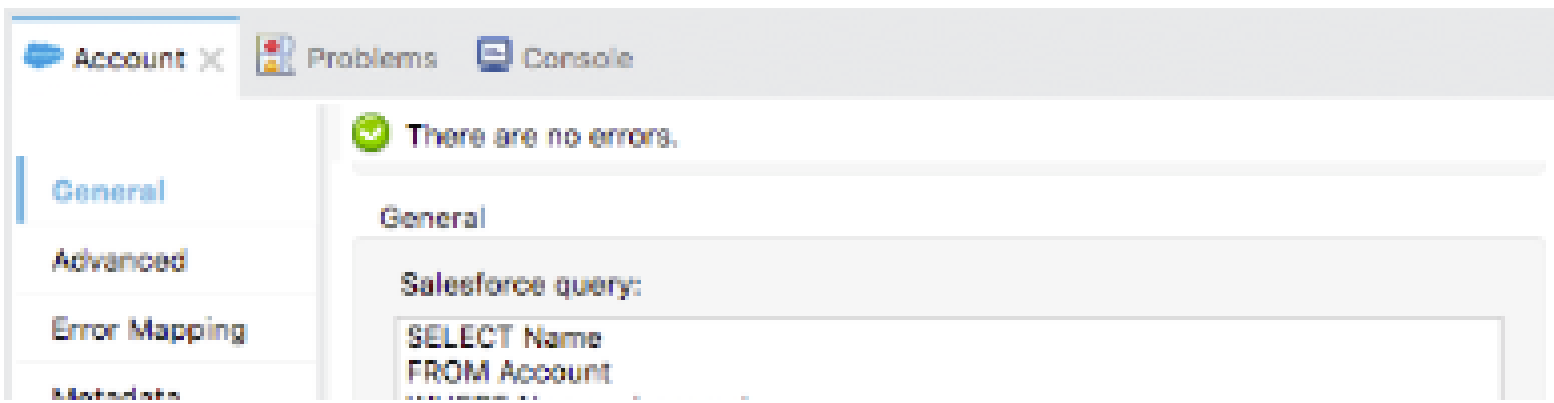
    ```
    payload.Name default "" as String
    ```

30. Modify the Salesforce query to look for accounts with this name.

    ```
    SELECT Name

    FROM Account

    WHERE Name= ':cname'
    ```



Store the result in a variable instead of overriding the payload

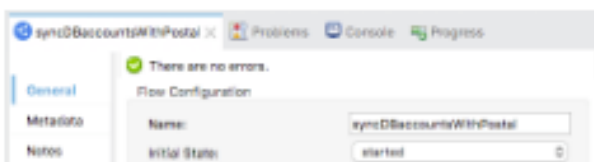
31. Select the Advanced tab.
32. Set the target variable to exists.
33. Set the target value to

```
#[(sizeOf(payload as Array) > 0)]
```

34. Add a Logger after the Query operation.
35. Add a Logger to the on complete phase.

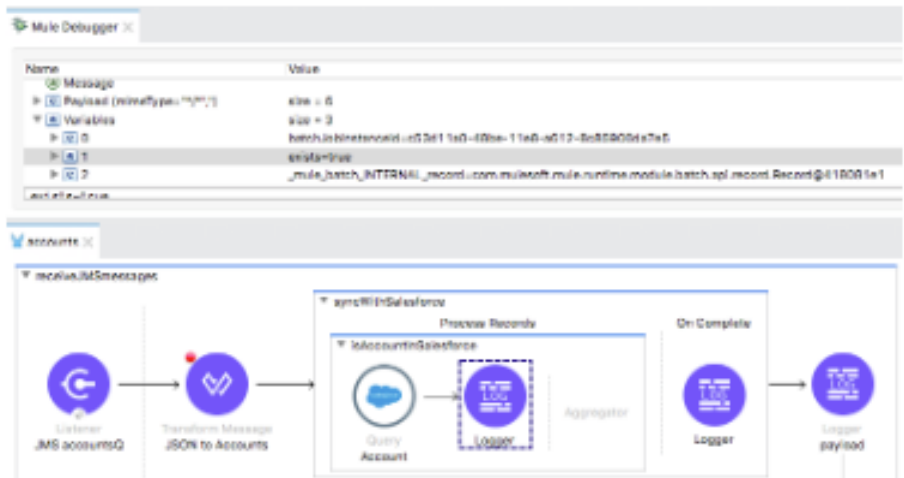
Change the initial state of the flow

36. In the properties view for the syncDBaccountsWithPostal flow, change the initial state to started.



Debug the application

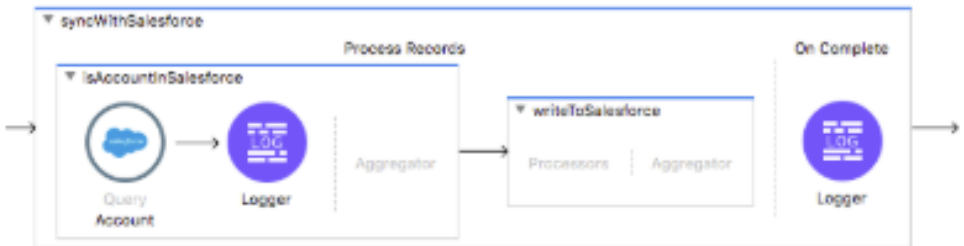
37. Add a breakpoint to the Transform Message component.
38. Debug the project.
39. Clear the application data.
40. In the Mule Debugger, wait until application execution stops in the receiveJMSmessages flow.
41. Step to the Logger in the first batch step and expand Variables; you should see the exists variable set to true or false.



42. Step through the application; you should see the exists variable set to false for records with names that don't exist in Salesforce and true for those that do.
43. Stop the project and switch perspectives.

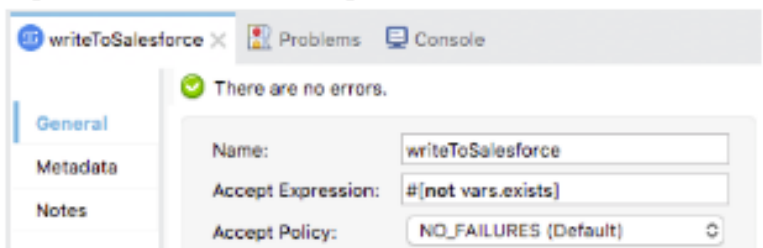
Set a filter for the insertion step

44. Add a second batch step to the batch job.
45. Change its display name to writeToSalesforce.



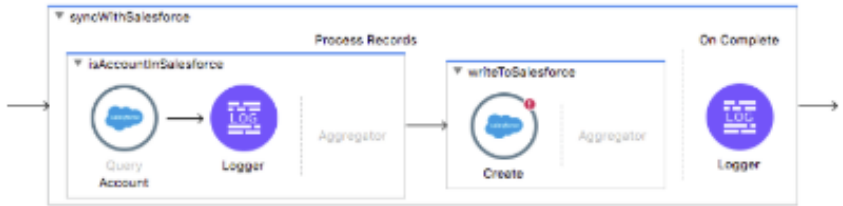
46. In the properties view for the second batch step, set the accept expression so that only records that have the variable exists set to false are processed.

```
#[not vars.exists]
```



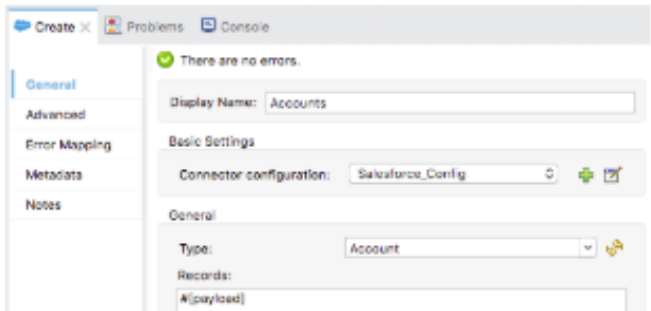
Use the Salesforce Create operation to add new account records to Salesforce

47. Add Salesforce Create operation to the second batch step in the processing phase.

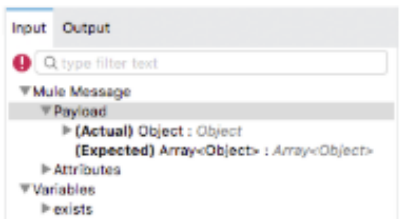


48. In the Create properties view, set the following:

- Display Name: Accounts
- Connector configuration: Salesforce_Config
- Type: Account
- Records: #[payload]

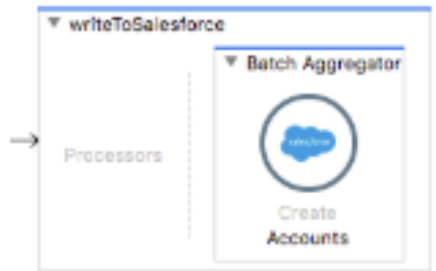


49. Select the Input tab in the DataSense Explorer; you should see this operation expects an Array of Account objects – not just one.

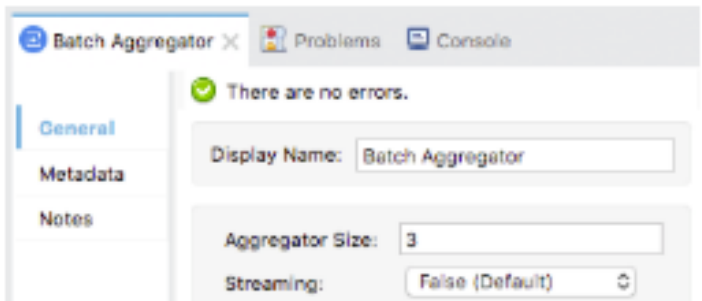


Create the Array of objects that the operation is expecting

50. Drag a Batch Aggregator scope from the Mule Palette and drop it in the aggregator section of the second batch step.
51. Move the Salesforce Create operation into the Batch Aggregator.



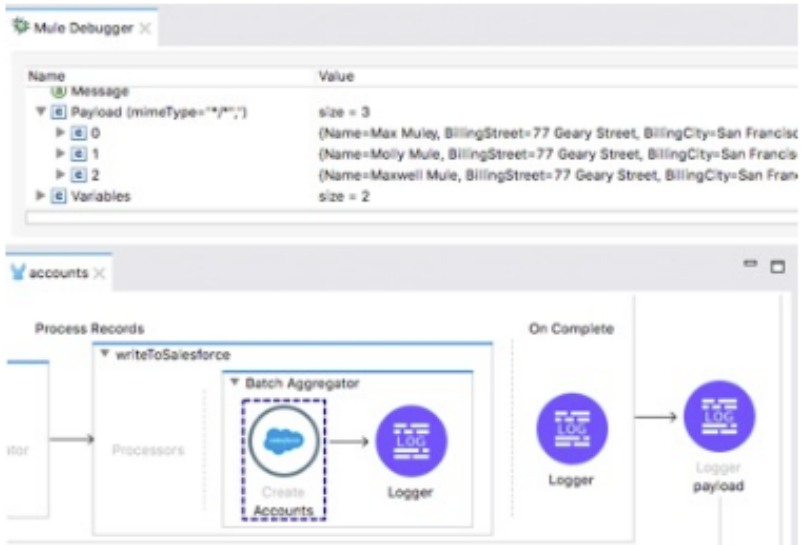
52. In the Batch Aggregator properties view, leave the aggregator size set to 3.



53. Add a Logger after the Create operation.

Test the application

54. Debug the project and clear the application data.
55. Step through the application until you step into the Batch Aggregator.
56. Expand Payload.



57. Step through the rest of the flow.
58. Return to Salesforce and look at the accounts; you should see the records from the legacy MySQL database are now in Salesforce.

    Note: You could also check for the records by making a request to http://localhost:8081/sfdc.

59. Run the application again but do not clear the application data; no records should be processed.
60. Return to salesforce.com and locate your new record(s); they should have been inserted only once.
61. Return to Anypoint Studio and stop the project.

Summary

Summary

- Use the **For Each** scope to process individual collection elements sequentially and return the original payload
- Use the **Batch Job** scope (EE only) for complex batch jobs
  - Created especially for processing data sets
  - Splits messages into individual records and performs actions upon each record
  - Can have multiple batch steps and these can have filters
  - Record-level data is persisted across steps using variables
  - Can handle record level failures so the job is not aborted
  - The batch job returns an object with the results of the job for insight into which records were processed or failed

DIY Exercise 13-1: Create a batch job to process records

Time estimate: 2 hours

Objectives

In this exercise, you create a batch processing system. You will:

- Control batch step processing using batch filters.
- Exchange data between batch steps using variables.
- Trigger batch processing on a schedule.
- Use watermarks to avoid duplicate processing.
- Improve performance by using batch aggregators.
- Improve performance and share data with other Mule applications using VM queues.

Scenario

The Finance department needs to audit certain transactions and needs a Mule application to consistently retrieve data from a database and write these transactions as CSV files to a server.

To meet compliance standards, a CSV file can have no more than 50 records and the Mule application must be deployed to a private server where the Mule application will share the same Mule domain with other financial compliance Mule applications. You do not, however, need to create a new Mule domain project yourself; another developer will be responsible for deploying your project into an existing Mule domain.

Create a project that retrieves new transactions from the database using batch

Create a new Mule application that retrieves data from the flights_transactions table in the database using the following information:

- Host: mudb.learn.mulesoft.com
- Port: 3306
- User: mule
- Password: mule
- Database: training
- Table: flights_transactions

Schedule the main flow to automatically run every 5 seconds. Retrieve new database records based on the value of the primary key field transactionID. Use an ObjectStore to save the maximum transactionID processed for any batch session.

Hint: For test development, limit the query to only retrieve 10 records at a time.

Add a flow to mock the financial compliance application logic

Add a new flow to the Mule application with a VM Listener on the VM queue named validate. Add a Transform Message component to this flow and add DataWeave code to simulate the transactionID validation logic. It expects one record and returns the value true or false, where true indicates that the record needs to be audited. In this simple mock flow, return a Boolean value true if the transactionID is divisible by 4, and false otherwise:

```
%dw 2.0

output application/java

---

if ( mod(payload.transactionID as Number, 4) == 0 )

    true

else

    false
```

Send each transaction record to a VM queue for conditional testing

Publish each transaction record to the validate VM queue and wait to consume the Boolean response. Save the result in a variable to filter the current record in the next batch step.

Add batch filters to only process transactions that need auditing

Configure target variables and the accept expression in each batch step to keep track of your records throughout each batch step.

Hint: For test development, you can create a flow that listens on the validate queue path and arbitrarily return true or false for each record processed.

Write out transactions as CSV files

In a second batch step, configure an accept expression to only process this second batch step if the previous VM queue response was true. Inside this batch step, transform the database results to a CSV file and save this CSV file to this Mule application's file system. Use a property placeholder for the file location so the file location can be modified by Ops staff at deployment time. Add a batch aggregator so no more than 50 records at a time are written to each CSV file.

Log the batch processing summary

In the On Complete phase of the Batch Job, log the batch processing summary.

Test your solution

Debug your solution. Step through several polling steps and verify some queries return true from the VM queue and are processed by the second batch step, but other database queries return false and skip the second batch step. Also verify the output CSV files contain at most 50 records each.

Verify your solution

Import the solution /files/module13/audit-mod13-file-write-solution.jar deployable archive file (in the MUFundamentals_DIYexercises.zip that you can download from the Course Resources) and compare your solution.

Going further: Handle errors

Add logic to the first batch step to throw errors.

- Call a flow at the beginning of the first batch step and add event processors to this flow that would sometimes throw an error, but not for every record.
- Experiment with what happens when you handle the error in the flow versus if you don't handle the error in the flow.
- Add a third batch step with an ONLY_FAILURES accept policy to report failed messages to a dead letter VM queue for failed batch steps.
- In the Batch Job scope's general configuration, change the max failed records to 1 and observe the behavior of subsequent batch records after a record throws an error. Change this value to 2 and observe any changes in behavior.
- In the Batch Job scope's general configurations, change the scheduling strategies options to ROUND_ROBIN and observe the behavior, and compare it with the default ORDERED_SEQUENTIAL option's behavior.
- Look at the logs for the On Complete phase to see how many times the same error is reported for each record of the batch job.
- In the first batch step's referenced flow, add a Choice router and a sleep timer that sleeps for a minute. Add logic to the Choice router to only call the sleep timer if the transactionID ends with 6. Observe if later records in the batch job can skip ahead while some records are paused by the sleep timer.

Note: For info about handling batch errors, see: https://blogs.mulesoft.com/dev/mule-dev/handle-errors-batch-job/

Going further: Refactor the validation logic to another Mule application in a new shared Mule domain

Create a Mule domain project named finance, then change the Mule application's Mule domain from default to finance. Move the VM connector global element to the finance Mule domain project. Move the validation flow to a new Mule application and configure this Mule application to also use the finance Mule domain.

Deploy the Mule domain and both Mule applications to a customer-hosted Mule runtime. Verify batch jobs are still processed correctly.

Going further: Deploy both Mule applications to CloudHub

Instead of using customer-hosted Mule runtimes, configure both Mule applications to use an external online JMS server that is accessible over the public internet. Deploy both Mule applications to CloudHub and verify batch jobs are still processed correctly.