Example Java Applications

We provide several example Java applications to demonstrate how to interface with the Confluent Kafka Platform here at M&T, with or without the help of Spring Boot or Spring Cloud.

Below is a description of each example application, how it works, and some instructions for configuring the example application and of course building on top of the code.

> Note: All topics under naming convention `fs-example-` have a retention period of 15 minutes. Meaning for any example application you use, the messages to any topic will be automatically deleted after 15 minutes.

All apps can be found here: https://gitlab.mtb.com/foundational-services/cp-example-java-apps

- Basic Java
- Spring with Kafka Client
- Spring Kafka
- Spring Cloud Stream Kafka
    - Basic Producer and Consumer
    - Streams Processing
        - Producer as Controller
        - Consumer Expanded
        - Filter and Transform Example
        - Lookup Table Example
        - Stream Branch Example
    - Other Useful Documentation for Spring Cloud Stream
- Kafka State Store
    - Lookup Table Example
    - Select Query Example
    - KTable Inventory Example
    - Delete Query Example

Basic Java

GitLab - Basic Java Example Apps

We provide a basic implementation of a producer and a consumer using Java. These examples are utilizing the `kafka-clients` library as the primary interface for our Kafka platform. The `JSON` and `AVRO` dependencies, including `avro-maven-plugin` are for data structuring and message serialization/deserialization.

This producer example is conducted from the `src/test` directory. You'll see there are three different test cases designed to demonstrate some common serialization formats. In each test, simply run the `main` method and you will see a large burst of random `Payment` objects produced to the Kafka broker under the topic `fs-example-avro-payments`.

This consumer example is also conducted from the `src/test` directory. You'll see there are three examples in which a consumer is subscribing to the topic `fs-example-avro-payments` and doing something with the retrieved data (printing to console, persisting to database, etc.). Run each `main` method and check the output. Don't forget you may need to produce messages to the topic before seeing any results with the consumer example!

To use these example apps, follow the instructions in the README file.

GitLab - Basic Java Producer

GitLab - Basic Java Consumer

Topics used:

- `fs-example-avro-payments`

Spring with Kafka Client

GitLab - Spring with Kafka Client Example Apps

These examples demonstrate how to build basic `kafka-clients` producers and consumers (just like in the above section) only this time using Spring Boot.

This producer example is conducted from the `src/main` directory. This time, there is only one method to run a producer and we are only using the AVRO message body format. Our single method produces ten random `Payment` objects to topic `fs-example-avro-payments`. You will see the producer is triggered by the REST endpoint `/test`.

This consumer example is also conducted from the `src/main` directory. On starting the application, this consumer is automatically subscribed to topic `fs-example-avro-payments` and will print out to the console any message received. We recommend booting up the consumer and running the producer to observe test.

For this example you are driving the application using REST endpoints. Take a look at the instructions in the README file.

> GitLab - Spring with Kafka Client Producer
>
> GitLab - Spring with Kafka Client Consumer

Topics used:

- `fs-example-avro-payments`

Spring Kafka

GitLab - Spring Kafka Example Apps

We've also provided a producer and consumer example using Spring's library designed for using Apache Kafka: Spring-Kafka. Using Spring-Kafka, many components of the application are abstracted away by Spring. For example, all your configurations go into one `.properties` or `.yml` file and Spring auto-connects those configs to any Kafka components you call into your script. The Kafka components are also provided by Spring, and implement their own versions of Kafka-Clients objects. You'll also see annotations making light work of some of the steps to using the Spring-Kafka components themselves.

This producer example is again using the `Payment` object and producing messages to topic `fs-example-avro-payments`. There are two endpoints you can use to observe this producer implementation in action using a `POST Request`. Endpoint `/publish-many` will allow you to put an integer into the body and send N number of random `Payment` objects, and `/publish` will allow you to craft your own `Payment` object by writing the `JSON` body.

This consumer is automatically subscribed to topic `fs-example-avro-payments` on startup, using the `@KafkaListener` annotation provided by Spring. That is all you need to make a consumer using Spring-Kafka, however you will need to use the REST endpoints to view the consumer output. The methods implemented in the REST Controller are just examples and used simply for viewing the data on user trigger. The data is already aggregated in Lists by consumption and accessed when the user calls each endpoint.

Like the last example with Spring, you are driving the application using REST endpoints. Take a look at the instructions in the README file.

> GitLab - Spring Kafka Producer
>
> GitLab - Spring Kafka Consumer

Topics used:

- `fs-example-avro-payments`

Spring Cloud Stream Kafka

GitLab - Spring Cloud Stream Kafka

We've provided for your reference an implementation of Spring Cloud Stream using Kafka. Spring Cloud Stream provides binders for Kafka which can be mapped to streams of data moving to or from topics. Spring Cloud Stream is a popular data and event streaming framework, so the Kafka implementation can definitely be useful. However, since you can write configurations for `spring.kafka`, `spring.cloud.stream`, `spring.cloud.stream.kafka`, or `spring.cloud.stream.kafka.streams` all in the same `.yml` or `.properties` file, configuration can get extremely tricky.

Basic Producer and Consumer

This producer example on startup is sending a collection of pre-written data to three different topics:

- `fs-example-customer-info` details about customers (in this case rock stars)
- `fs-example-group-info` details about the groups these customers belong to
- `fs-example-account-info` some random balance details associated with customer's accounts

And of course the consumer is just subscribed to these three topics.

Because of all of the nuances surrounding Spring Cloud Stream configuration with Kafka, it's imperative you review the instructions in the README file and also the **Other Useful Documentation for Spring Cloud Stream** section below.

GitLab - Spring Cloud Stream Kafka Basic Producer

GitLab - Spring Cloud Stream Kafka Basic Consumer

Topics used:

- `fs-example-account-info`
- `fs-example-customer-info`
- `fs-example-group-info`

Streams Processing

GitLab - Spring Cloud Stream Kafka - Streams Processing Examples

Producer as Controller

A simple re-design of the above Spring Cloud Stream Producer Example, this time utilizing REST endpoints for user to control producer tests. This is recommended to run when testing stream processing examples.

GitLab - Spring Cloud Stream Kafka Producer as Controller

Topics used:

- `fs-example-account-info`
- `fs-example-customer-info`
- `fs-example-group-info`

Consumer Expanded

A simple re-design of the above Spring Cloud Stream Consumer Example, this time expanded to all topics involved in all Spring Cloud Stream Kafka examples. This is recommended to run when testing stream processing examples.

GitLab - Spring Cloud Stream Kafka Consumer Expanded

Topics used:

- `fs-example-account-details`
- `fs-example-account-info`
- `fs-example-balance-output`
- `fs-example-customer-details`
- `fs-example-customer-details-2`
- `fs-example-customer-info`
- `fs-example-group-info`
- `fs-example-group-output`
- `fs-example-group-transformed`

Filter and Transform Example

This example demonstrates how to set up streams processing using spring cloud stream binders. This is the method for configuring processors that is documented by Spring in their official Spring Cloud Stream Kafka docs.

We are making use of `java.util` objects Supplier, Consumer, and Function to craft binders for streams. Spring's configuration just requires the `@Bean` annotation and some configurations behind the scenes in `application.yml`.

The processor Beans labeled "filter" are simply reading the records in a topic and filtering for pre-defined values. Once that value is received, they output a certain record to the same topic that it had been listening to.

The processor Beans labeled "transform" are simply reading the records in one topic and turning their data into new objects and new topics as output.

The naming conventions are super important to the functionality of this example. You can find more details in the application's README file.

GitLab - Spring Cloud Stream Kafka Filter and Transform Example

Topics used:

- `fs-example-account-info`
- `fs-example-balance-output`
- `fs-example-customer-info`
- `fs-example-group-info`
- `fs-example-group-output`
- `fs-example-group-transformed`

Lookup Table Example

Here we are making explicit use of `java.util` object `BiFunction` object to craft binders with multiple inputs for streams.

In the first example, the Bean labeled `cloudLookupTable1`, we are implementing a join on messages from topic `fs-example-account-info` and `fs-example-customer-info`, keying on ID, where `AccountInfo` messages are checked against the state store of `CustomerInfo` messages.

In the second example, the Bean labeled `cloudLookupTable2`, we are implementing a join on messages from topic `fs-example-customer-info` and `fs-example-group-info`, keying on ID, where `CustomerInfo` messages are checked against the state store of `GroupInfo` messages.

GitLab - Spring Cloud Stream Kafka Lookup Table Example

Topics used:

- `fs-example-account-details`
- `fs-example-account-info`
- `fs-example-customer-details`
- `fs-example-customer-info`
- `fs-example-group-info`
- ** `fs-example-cloudLookupTable1-applicationId-KSTREAM-TOTABLE-STATE-STORE-0000000003-changelog`
- ** `fs-example-cloudLookupTable2-applicationId-KSTREAM-TOTABLE-STATE-STORE-0000000003-changelog`

** These topics are state store topics used by the stream processing topology.

Stream Branch Example

Here we are making explicit use of the `KStream<>[]` and `Predicate<>` classes from Kafka-Streams.

In the example, the Bean labeled `streamBranch`, we are subscribed to the topic `fs-example-customer-info` and creating an output stream of `CustomerDetails` objects, but it is only directed to a topic based on a conditional. You'll see you can set up multiple conditionals that dictate the output's routing to multiple different topics.

GitLab - Spring Cloud Stream Kafka Stream Branch Example

Topics used:

- `fs-example-customer-details`
- `fs-example-customer-details-2`
- `fs-example-customer-info`

Other Useful Documentation for Spring Cloud Stream

https://docs.spring.io/spring-cloud-stream-binder-kafka/docs/3.0.10.RELEASE/reference/html/spring-cloud-stream-binder-kafka.html

https://spring.io/projects/spring-cloud

https://docs.spring.io/spring-cloud-stream/docs/3.1.0/reference/html/spring-cloud-stream.html#spring-cloud-stream-overview-producing-consuming-messages

Kafka State Store

GitLab - Kafka State Store

This collection of examples explores Kafka's state stores and their ability to serve as a queryable data source, also called a "single source of truth".

Inspired by https://github.com/confluentinc/kafka-streams-examples

Lookup Table Example

This time we built a lookup table using Kafka-Client, Kafka-Streams and Spring, but without Spring Cloud Stream.

This application's consumer, stream processor, and one pre-loading producer all boot up when you start it. The initial producer is designed to populate the topic with `Warehouse` data. Once the app is running, you can navigate to any of the endpoints below, but we recommend going in this order:

1. Request the `/test1` endpoint to produce a handful of `Shipment` records:
2. Request the `/test2` endpoint to produce a new, different handful of `Shipment` records:
3. Request the `/test3` endpoint to produce another new, different handful of `Shipment` records:

The purpose of running in this order is to make evident that the contents of the KTable (`Warehouse` data)
remains in state, like a database, and all incoming `Shipment` records are checked against it. You can observe this behavior in the output console as you request each endpoint. The streams processing topology we put together will join incoming Shipments with existing Warehouse records based on ID.

GitLab - Kafka State Store Lookup Table Example

Topics used:

- `fs-example-shipment`
- `fs-example-warehouse`
- `fs-example-shipping-details`
- ** `fs-example-shipping-details-example-app-test-warehouse-STATE-STORE-0000000000-changelog`

** These topics are state store topics used by the stream processing topology.

Select Query Example

Here we have a demonstration of a `SELECT` or `SELECT WHERE` query using Kafka state store as a source of data.

The application's stream processor and producer boot up when you start it. The producer is designed to routinely push a random sentence to topic `fs-example-text-lines` every 5 seconds. Once the app is running, you can navigate to any of the endpoints below to observe how to treat the Kafka state store like a data source:

- The state store name is automatically created as `word-count` in the `Materialized.as()` command in the streams `Topology`.

1. Request the `/{storeName}/all` endpoint to see every word that has gone to topic `fs-example-text-lines` and the count of its occurrences.
2. Request the `/{storeName}/get/{key}` endpoint to see the word count of a specific word. The word is the key.
3. Request the `/{storeName}/range/{from}/{to}` endpoint to see the word count of a specific range of words. The word is the key and also the `from` and `to` fields.

GitLab - Kafka State Store Select Query Example

Topics used:

- `fs-example-text-lines`
- ** `fs-example-select-query-example-word-count-changelog`
- ** `fs-example-select-query-example-word-count-repartition`

** These topics are state store topics used by the stream processing topology.

KTable Inventory Example

In this example, we demonstrate how Kafka state store can be used to build a microservice, in this case an inventory service for a musical instrument warehouse.

On startup, this application spins up a producer, consumer, and stream processor. Inside the stream processor's `Topology` is where two state stores are created and maintained. These will be for total inventory and inventory reserved for approved orders. The producer is designed to push a pre-defined inventory to the `fs-example-instrument-inventory` topic. This will become the KTable in the lookup table operation.

The consumer is configured to the output topic `fs-example-order-validation` which returns whether the order is a `PASS` or `FAIL` depending on quantity ordered vs. inventory amounts.

Say we are placing an order for guitars. The logic of calculating a `PASS` or `FAIL` is as follows:

- The inventory state store tracks our `fs-example-instrument-inventory` topic and holds our total warehouse inventory of guitars.
- Another state store holds the number of guitars that have been ordered since the application started. This is the reserve state store (reserved for orders)
- The lookup table takes the total inventory minus guitars reserved for any accrued orders and determines if there is enough quantity to fill the new order.

Once the application is running, there are endpoints to watch the procedure in action:

1. Request the `/new-order/guitars/{quantity}` endpoint to place an order for guitars. You'll see there are identical endpoints for amplifiers and drum sets.
2. Request the `/inventory/all` endpoint to see the total inventory.
3. Request the `/inventory/reserve` endpoint to see the reserved inventory (reserved for successfully processed orders).
4. Request the `/inventory/available` endpoint to see the available inventory.
5. Request the `/restart` endpoint to restart the demonstration and return the reserve state store to all zeroes.

About this Example's Streams Processor Topology

```
static Topology getTopology() {

   final StreamsBuilder builder = new StreamsBuilder();
   final OrderJoiner joiner = new OrderJoiner();

   // The orders come in as a stream
   final KStream<String, Order> orders = builder
           .stream(ORDERS_TOPIC);

   // The inventory topic is converted to KTable to use as reference
table
   final KTable<String, Integer> warehouseInventory = builder
           .table(WAREHOUSE_INVENTORY_TOPIC,
                   Consumed.with(Serdes.String(), Serdes.Integer()));
```

```
    // Creating a store to reserve inventory while the order is processed
    // This will be pre-populated from producer
    final StoreBuilder<KeyValueStore<String, Long>> reservedStock =
Stores
            .keyValueStoreBuilder(Stores.persistentKeyValueStore
(RESERVED_STOCK_STORE_NAME),
                    Serdes.String(), Serdes.Long())
            .withLoggingEnabled(new HashMap<>());
    builder.addStateStore(reservedStock);

    // Joining and storing stock inventory
    orders.selectKey((id, order) -> order.getProduct().toString())
            .filter((id, order) -> order.getState().toString().equals
("CREATED"))
            .join(warehouseInventory, joiner)
            .transform(InventoryValidator::new,
RESERVED_STOCK_STORE_NAME)
            .to(ORDER_VALIDATION_TOPIC);

    return builder.build();
}
```

This streams topology may look intimidating, but it's actually quite similar to the Lookup Table example elsewhere in this repository.

Basically what's happening here, is we are setting up a lookup table like we did in the other example. This lookup table is reading incoming orders via input stream and looking up the inventory of the ordered product via ID (exactly like the shipment example). We know this will instantiate a state store on the inventory topic, which is the total inventory.

However, we have one additional step. We also add a state store for reserved stock, which is made from successfully validated orders. So, when an order is confirmed to have enough stock to fill, the order is pushed to the output topic `fs-example-order-validation` with either a `PASS` or a `FAIL`. When it's a `PASS`, that reserve state store is updated to reflect the quantity that must go out to fill that order we just approved.

As you can see, this is a fully functioning inventory manager, all using Kafka!

About this Example's Restart Demonstration Feature

This application features a "restart demonstration" capability, where the messages produced to reserve state store are nulled. This is accomplished by using the Kafka-Client library method `deleteRecords()`, which abstracts away the procedure of pushing tombstone messages to Kafka to identify those records as null.

Deleting messages is explored in more detail in the next example: Delete Query Example. For this particular reset case, we are telling Kafka-Client to delete all messages before offset -1 (which means all messages) and we input each partition number for the topic name. This sends tombstone messages for all records in that topic, and then we simply use `streams.cleanUp()` to reset the state store before booting up the processor again.

GitLab - Kafka State Store KTable Inventory Example

Topics used:

- `fs-example-instrument-order`
- `fs-example-instrument-order-validation`
- `fs-example-instrument-inventory`
- ** `fs-example-instrument-inventory-example-KSTREAM-FILTER-0000000005-repartition`
- ** `fs-example-instrument-inventory-example-fs-example-instrument-inventory-STATE-STORE-0000000001-changelog`
- ** `fs-example-instrument-inventory-example-store-of-reserved-stock-changelog`

** These topics are state store topics used by the stream processing topology.

Delete Query Example

In this example, we are continuing to explore Kafka's state store behavior as a data source. After all, what good is a data source that doesn't allow records to be deleted? Luckily, Kafka provides an implementation for this as well.

Here we are simply producing some sample messages to topic `fs-example-airline-ticket` which are basic representations of airline tickets being purchased one at a time. Like the previous examples, we are setting up a state store with the count of each type of airline ticket purchased.

You'll notice we have including the Kafka-Client restart demonstration method from the KTable Inventory Example. However, we also dive into some delete behavior that resembles a simple SQL `DELETE` or `DELETE WHERE` query.

Check out the README for details on the specific delete methods used. For additional information on topic compaction or deletion of topic records, check out https://www.confluent.io/blog/handling-gdpr-log-forget/

GitLab - Kafka State Store Delete Query Example

Topics used:

- `fs-example-airline-ticket`
- ** `fs-example-delete-query-example-ticket-count-changelog`
- ** `fs-example-delete-query-example-ticket-count-repartition`

** These topics are state store topics used by the stream processing topology.