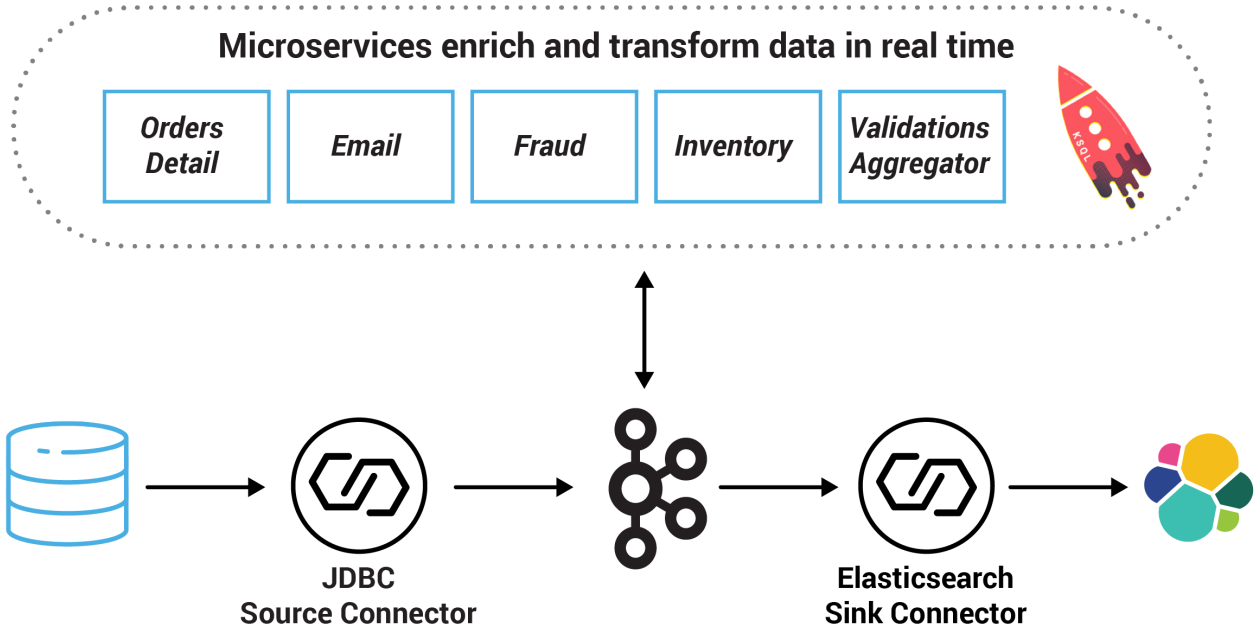
Tutorial: Introduction to Streaming Application Development

This self-paced tutorial provides exercises for developers to apply the basic principles of streaming applications.

Overview

The tutorial is based on a small microservices ecosystem, showcasing an order management workflow, such as one might find in retail and online shopping. It is built using Kafka Streams, whereby business events that describe the order management workflow propagate through this ecosystem. The blog post Building a Microservices Ecosystem with Kafka Streams and KSQLoutlines the approach used.



Note: this is demo code, not a production system and certain elements are left for further work.

Microservices

In this example, the system centers on an Orders Service which exposes a REST interface to POST and GET Orders. Posting an Order creates an event in Kafka that is recorded in the topic *orders*. This is picked up by different validation engines (Fraud Service, Inventory Service and Order Details Service), which validate the order in parallel, emitting a PASS or FAIL based on whether each validation succeeds.

The result of each validation is pushed through a separate topic, Order Validations, so that we retain the *single writer* status of the Orders Service —> Orders Topic (Ben Stopford's book discusses several options for managing consistency in event collaboration). The results of the various validation checks are aggregated in the Validation Aggregator Service, which then moves the order to a Validated or Failed state, based on the combined result.

To allow users to GET any order, the Orders Service creates a queryable materialized view (embedded inside the Orders Service), using a state store in each instance of the service, so that any Order can be requested historically. Note also that the Orders Service can be scaled out over a number of nodes, in which case GET requests must be routed to the correct node to get a certain key. This is handled automatically using the interactive queries functionality in Kafka Streams.

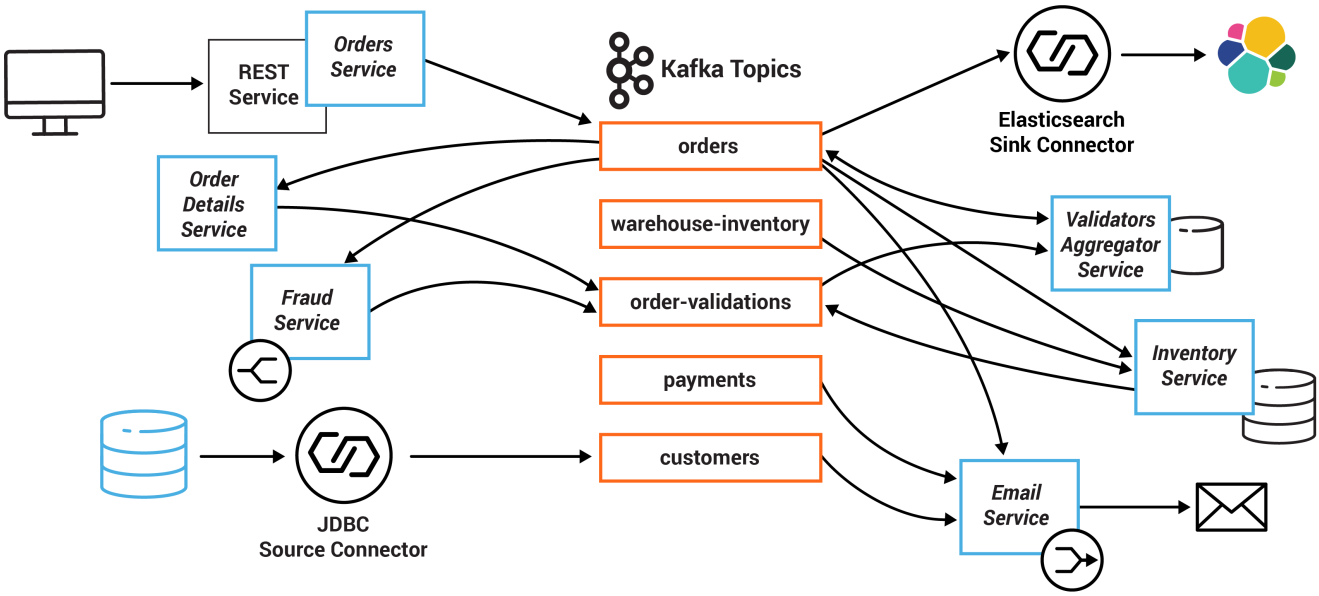
The Orders Service also includes a blocking HTTP GET so that clients can read their own writes. In this way, we bridge the synchrono

paradigm of a RESTful interface with the asynchronous, non-blocking processing performed server-side.

There is a simple service that sends emails, and another that collates orders and makes them available in a search index using Elasticsearch.

Finally, KSQL is running with persistent queries to enrich streams and to also check for fraudulent behavior.

Here is a diagram of the microservices and the related Kafka topics.



All the services are client applications written in Java, and they use the Kafka Streams API. The java source code for these microservices are in the kafka-streams-examples repo.

Summary of services and the topics they consume from and produce to:

Service	Consumes From	Produces To
InventoryService	*orders, warehouse-inventory*	*order-validations*
FraudService	*orders*	*order-validations*
OrderDetailsService	*orders*	*order-validations*
ValidationsAggregatorService	*order-validations, orders*	*orders*
EmailService	*orders, payments, customers*	*platinum, gold, silver, bronze*
OrdersService	•	*orders*

End-to-end Streaming ETL

This demo showcases an entire end-to-end streaming ETL deployment, built around the microservices described above. It is build on the Confluent Platform, including:

- JDBC source connector: reads from a sqlite database that has a table of customers information and writes the data to a Kafka topic, using Connect transforms to add a key to each message
- Elasticsearch sink connector: pushes data from a Kafka topic to Elasticsearch
- KSQL: another variant of a fraud detection microservice

Other Clients	Consumes From	Produces To
JDBC source connector	DB	*customers*
Elasticsearch sink connector	*orders*	ES
KSQL	*orders*, *customers*	KSQL streams and tables

For the end-to-end demo, the code that creates the order events via REST calls to the Orders Service and generates the initial inventory is provided by the following applications:

Application (Datagen)	Consumes From	Produces To
PostOrdersAndPayments	•	*payments*
AddInventory	•	*warehouse-inventory*

Pre-requisites

Reading

You will get a lot more out of this tutorial if you have first learned the concepts which are foundational for this tutorial. To learn how service-based architectures and stream processing tools such as Apache Kafka® can help you build business-critical systems, we recommend:

- If you have lots of time:Designing Event-Driven Systems, a book by Ben Stopford.
- If you do not have lots of time:Building a Microservices Ecosystem with Kafka Streams and KSQLor Build Services on a Backbone of Events.

For more learning on Kafka Streams API that you can use as a reference while working through this tutorial, we recommend:

- Kafka Streams documentation

Environment Setup

1. Make sure you have the following pre-requisites, depending on whether you are running Confluent Platform locally or in Docker

Local:

- Confluent Platform: download Confluent Platform with commercial features to use topic management, KSQL and Confluent Schema Registry integration, and streams monitoring capabilities
- Java 1.8 to run the demo application
- Maven to compile the demo application
- (optional) Elasticsearch 5.6.5 to export data from Kafka
  - If you do not want to use Elasticsearch, comment out `check_running_elasticsearch` in the `start.sh` script
- (optional) Kibana 5.5.2 to visualize data
  - If you do not want to use Kibana, comment out `check_running_kibana` in the `start.sh` script

Docker:

- Docker version 17.06.1-ce

- Docker Compose version 1.14.0 with Docker Compose file format 2.1
- In Docker's advanced settings, increase the memory dedicated to Docker to at least 8GB (default is 2GB)

2. Clone the examples GitHub repository:

```
git clone https://github.com/confluentinc/examples
```

3. Change directory to this project.

```
cd examples/microservices-orders
```

Tutorial

How to use the tutorial

As a pre-requisite, follow the "Environment Setup" instructions.

Then run the full end-to-end working solution, which requires no code development, to see a customer-representative deployment of a streaming application.. This provides context for each of the exercises in which you will develop pieces of the microservices.

- Exercise 0: Run end-to-end demo

After you have successfully run the full solution, go through the execises in the tutorial to better understand the basic principles of streaming applications:

- Exercise 1: Persist events
- Exercise 2: Event-driven applications
- Exercise 3: Enriching streams with joins
- Exercise 4: Filtering and branching
- Exercise 5: Stateful operations
- Exercise 6: State stores
- Exercise 7: Enrichment with KSQL

For each exercise:

1. Read the description to understand the focus area for the exercise
2. Edit the file specified in each exercise and fill in the missing code
3. Copy the file to the project, then compile the project and run the test for the service to ensure it works

Exercise 0: Run end-to-end demo

Running the fully working demo end-to-end provides context for each of the later exercises.

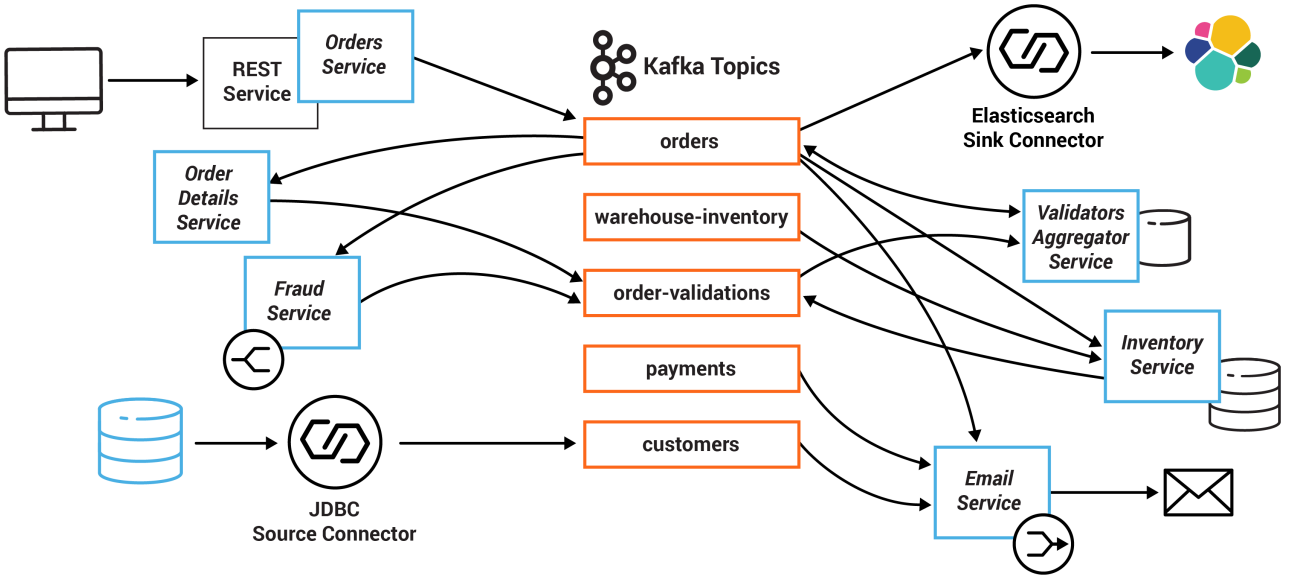
1. Start the demo
    - If you are have Confluent Platform downloaded locally, then run the full solution (this also starts a local Confluent Platform cluster using Confluent CLI):

      ```
      ./start.sh
      ```

- If you are running Docker, then run the full solution (this also starts a local Confluent Platform cluster in Docker containers).

```
docker-compose up -d --build
```

2. After starting the demo with one of the above two commands, the microservices applications will be running and Kafka topics will have data in them.



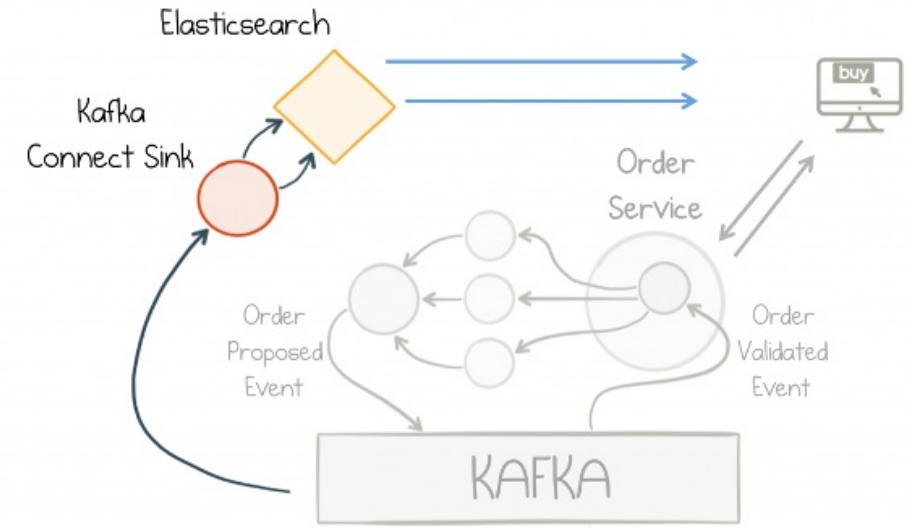
- If you are running locally, you can sample topic data by running:

```
./read-topics.sh
```

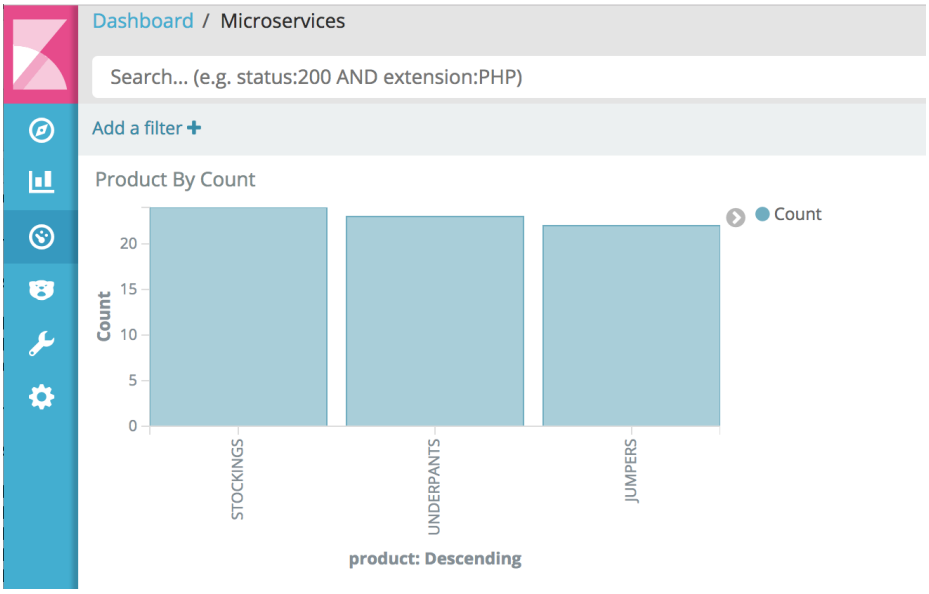
- If you are running Docker, you can sample topic data by running:

```
./read-topics-docker.sh
```

3. The Kibana dashboard is populated by Elasticsearch.

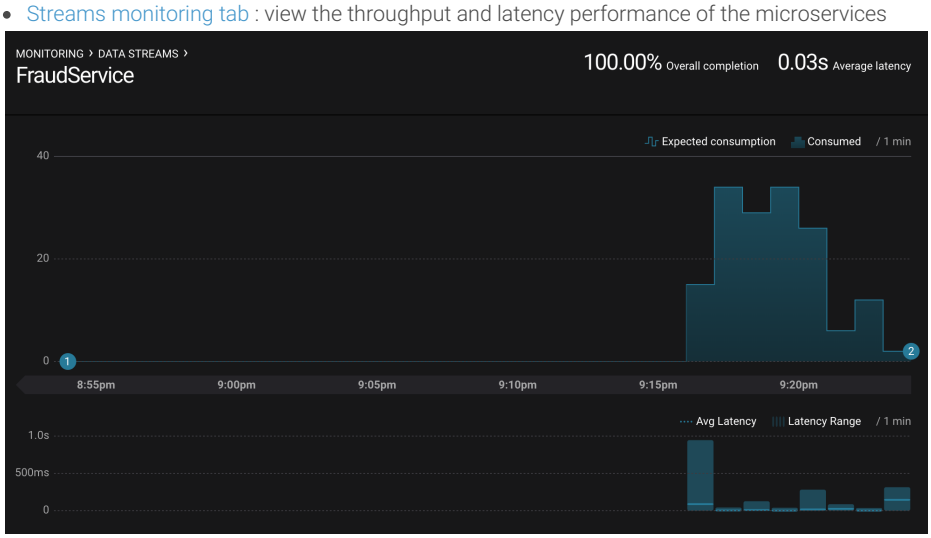


Full-text search is added via an Elasticsearch database connected through Kafka's Connect API (source). View the Kibana dashboard at http://localhost:5601/app/kibana#/dashboard/Microservices



4. Use Confluent Control Center to view Kafka data, write KSQL queries, manage Kafka connectors, and monitoring your applications:

   - KSQL tab : view KSQL streams and tables, and to create KSQL queries. Otherwise, run the KSQL CLI *ksql http://localhost:8088*. To get started, run the query `SELECT * FROM ORDERS;`
   - Kafka Connect tab : view the JDCB source connector and Elasticsearch sink connector.
   - Streams monitoring tab : view the throughput and latency performance of the microservices



5. When you are done, make sure to stop the demo before proceeding to the exercises.

   - If you are running Confluent Platform locally:

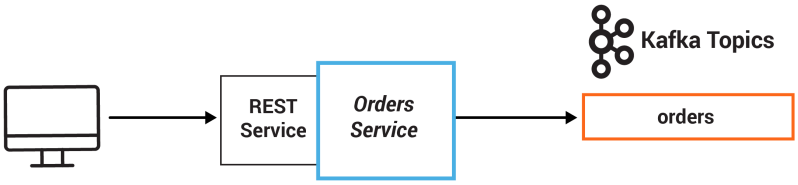
     ```
     ./stop.sh
     ```

   - If you are running Docker:

     ```
     docker-compose down
     ```

Exercise 1: Persist events

An _event_ is simply a thing that happened or occurred. An event in a business is some fact that occurred, such as a sale, an invoice, a trade, a customer experience, etc., and it is the source of truth. In event-oriented architectures, events are first-class citizens that constantly push data into applications. Client applications can then react to these streams of events in real time and decide what to do next.



In this exercise, you will persist events into Kafka by producing records that represent customer orders. This event happens in the Orders Service, which provides a REST interface to POST and GET Orders. Posting an Order is essentially a REST call, and it creates the event in Kafka.

Implement the *TODO* lines of the file exercises/OrdersService.java

1. TODO 1.1: create a new *ProducerRecord* with a key specified by *bean.getId()* and value of the bean, to the orders topic whose name is specified by *ORDERS.name()*
2. TODO 1.2: produce the newly created record using the existing *producer* and pass use the *OrdersService#callback* function to send the *response* and the record key

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/clients/javadocs/org/apache/kafka/clients/producer/ProducerRecord.html#ProducerRecord-java.lang.String-K-V-
- https://docs.confluent.io/current/clients/javadocs/org/apache/kafka/clients/producer/Callback.html
- kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/domain/Schemas.java
- kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/domain/beans/OrderBean.java

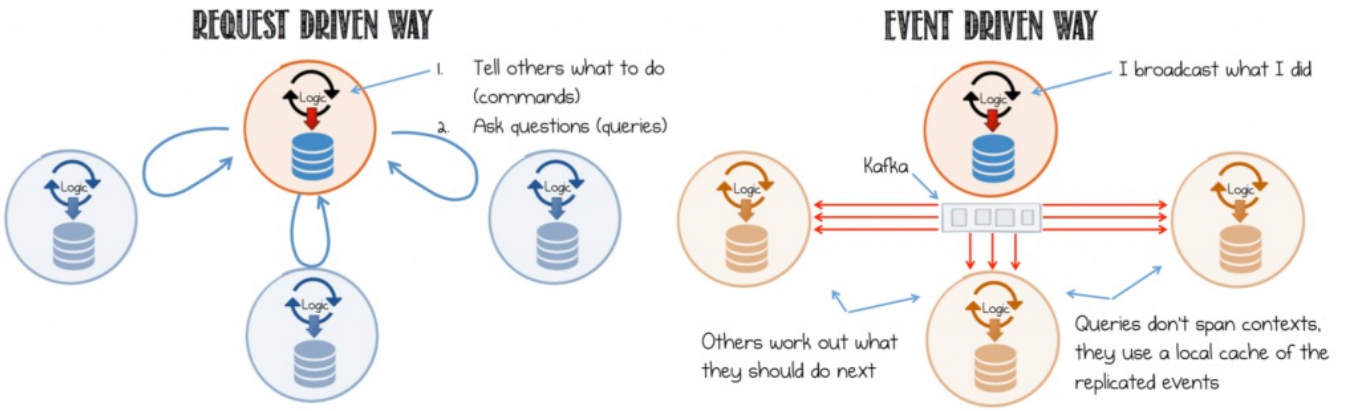
If you get stuck, here is the complete solution.

To test your code, save off the project's working solution, copy your version of the file to the main project, compile, and run the unit test.

```
# Clone and compile kafka-streams-examples
./get-kafka-streams-examples.sh

# Save off the working microservices client application to /tmp/
cp kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/OrdersService.java /tmp/.

# Copy your exercise client application to the project
cp exercises/OrdersService.java kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/.

# Compile the project and resolve any compilation errors
mvn clean compile -DskipTests -f kafka-streams-examples/pom.xml

# Run the test and validate that it passes
mvn compile -Dtest=io.confluent.examples.streams.microservices.OrdersServiceTest test -f kafka-streams-examples/pom.xml
```

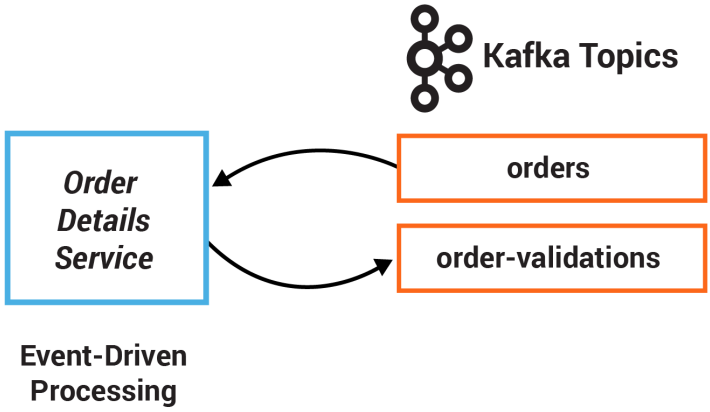
Exercise 2: Event-driven applications

Service-based architectures are often designed to be request-driven, in which services send commands to other services to tell them what to do, await a response, or send queries to get the resulting state. Building services on a protocol of requests and responses forces a complicated web of synchronous dependencies that bound services together.



A visual summary of commands, events and queries (source)

In contrast, in an event-driven design, the event stream is the inter-service communication that enables services to cross deployment boundaries and avoids synchronous execution. When and how downstream services respond to those events is within their control, which reduces the coupling between services and enables an architecture with more pluggability. Read more on Build Services on a Backbone of Events.



In this exercise, you will write a service that validates customer orders. Instead of using a series of synchronous calls to submit and validate orders, the order event itself triggers the *OrderDetailsService*. When a new order is created, it is written to the topic *orders*, from which *OrderDetailsService* has a consumer polling for new records.

Implement the *TODO* lines of the file exercises/OrderDetailsService.java

1. TODO 2.1: subscribe the existing *consumer* to a *Collections#singletonList* with the orders topic whose name is specified by *Topics.ORDERS.name()*
2. TODO 2.2: validate the order using *OrderDetailsService#isValid* and save the validation result to type *OrderValidationResult*
3. TODO 2.3: create a new record using *OrderDetailsService#result()* that takes the order and validation result
4. TODO 2.4: produce the newly created record using the existing *producer*

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/clients/javadocs/org/apache/kafka/clients/consumer/KafkaConsumer.html#subscribe-java.util.Collection-
- https://docs.confluent.io/current/clients/javadocs/org/apache/kafka/clients/producer/KafkaProducer.html#send-org.apache.kafka.clients.producer.ProducerRecord-
- https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-
- kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/domain/Schemas.java

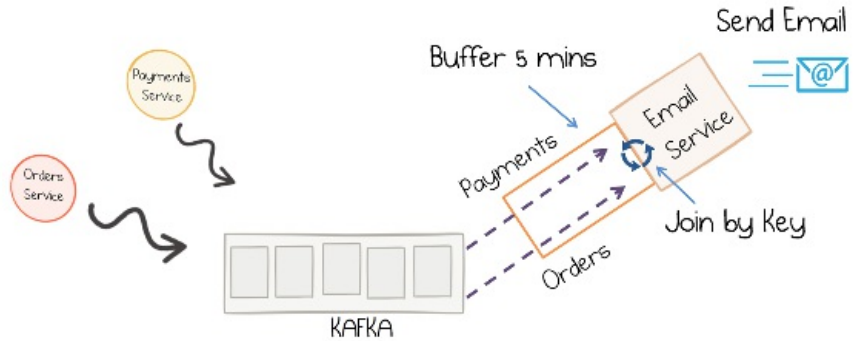
If you get stuck, here is the complete solution.

To test your code, save off the project's working solution, copy your version of the file to the main project, compile, and run the unit test.

```
# Clone and compile kafka-streams-examples
./get-kafka-streams-examples.sh

# Save off the working microservices client application to /tmp/
cp kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/OrderDetailsService.java /tmp/.

# Copy your exercise client application to the project
cp exercises/OrderDetailsService.java kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/
.

# Compile the project and resolve any compilation errors
mvn clean compile -DskipTests -f kafka-streams-examples/pom.xml

# Run the test and validate that it passes
mvn compile -Dtest=io.confluent.examples.streams.microservices.OrderDetailsServiceTest test -f kafka-streams-examples/
pom.xml
```

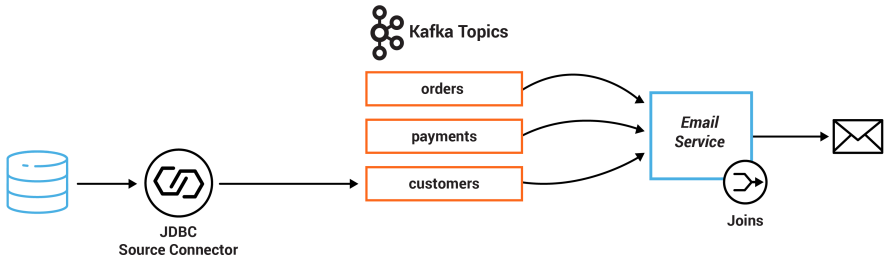
Exercise 3: Enriching streams with joins

Streams can be enriched with data from other streams or tables through joins. A join enriches data by performing lookups in a streaming context where data is updated continuously and concurrently. For example, applications backing an online retail store might enrich new data records with information from multiple databases. In this scenario, it may be that a stream of customer transactions is enriched with sales price, inventory, customer information, etc. These lookups can be performed at very large scale and with a low processing latency.



A stateful streaming service that joins two streams at runtime (source)

A popular pattern is to make the information in the databases available in Kafka through so-called change data capture (CDC), together with Kafka's Connect API to pull in the data from the database. Once the data is in Kafka, client applications can perform very fast and efficient joins of such tables and streams, rather than requiring the application to make a query to a remote database over the network for each record. Read more on an overview of distributed, real-time joins and implementing joins in Kafka Streams.



In this exercise, you will write a service that joins streaming order information with streaming payment information and data from a customer database. First, the payment stream needs to be rekeyed to match the same key info as the order stream before joined together. The resulting stream is then joined with the customer information that was read into Kafka by a JDBC source from a customer database. Additionally, this service performs dynamic routing: an enriched order record is written to a topic that is determined from the value of level field of the corresponding customer.

Implement the *TODO* lines of the fileexercises/EmailService.java

1. TODO 3.1: create a new *KStream* called *payments* from *payments_original*, using *KStream#selectKey* to rekey on order id specified by *payment.getOrderId()* instead of payment id
2. TODO 3.2: do a stream-table join with the customers table, which requires three arguments:
    1. the GlobalKTable for the stream-table join

2. customer Id, specified by *order.getCustomerId()*, using a KeyValueMapper that gets the customer id from the tuple in the record's value
3. method that computes a value for the result record, in this case *EmailTuple::setCustomer*

3. TODO 3.3: route an enriched order record to a topic that is dynamically determined from the value of the customerLevel field of the corresponding customer

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/Consumed.html#with-org.apache.kafka.common.serialization.Serde-org.apache.kafka.common.serialization.Serde-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/StreamsBuilder.html#stream-java.lang.String-org.apache.kafka.streams.kstream.Consumed-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KStream.html#selectKey-org.apache.kafka.streams.kstream.KeyValueMapper-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KStream.html#join-org.apache.kafka.streams.kstream.KTable-org.apache.kafka.streams.kstream.ValueJoiner-org.apache.kafka.streams.kstream.Joined-
- kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/domain/Schemas.java
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KStream.html#to-org.apache.kafka.streams.processor.TopicNameExtractor-org.apache.kafka.streams.kstream.Produced-

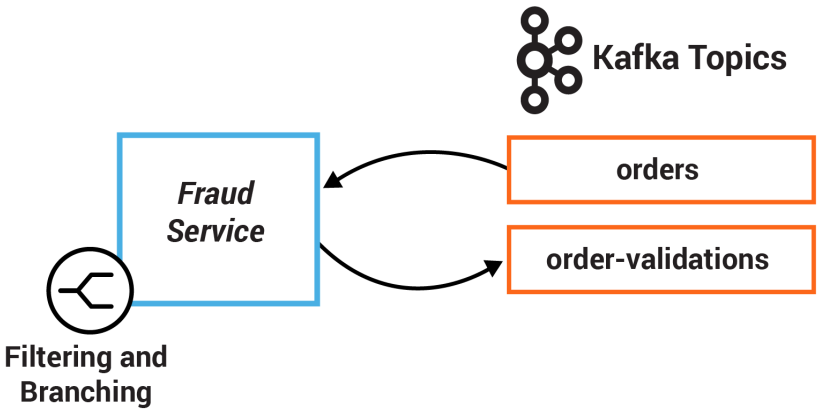
If you get stuck, here is thecomplete solution.

To test your code, save off the project's working solution, copy your version of the file to the main project, compile, and run the unit test.

```
# Clone and compile kafka-streams-examples
./get-kafka-streams-examples.sh

# Save off the working microservices client application to /tmp/
cp kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/EmailService.java /tmp/.

# Copy your exercise client application to the project
cp exercises/EmailService.java kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/.

# Compile the project and resolve any compilation errors
mvn clean compile -DskipTests -f kafka-streams-examples/pom.xml

# Run the test and validate that it passes
mvn compile -Dtest=io.confluent.examples.streams.microservices.EmailServiceTest test -f kafka-streams-examples/pom.xml
```

Exercise 4: Filtering and branching

A stream of events can be captured in a Kafka topic. Client applications can then manipulate this stream based on some user-defined criteria, even creating new streams of data that they can act on or downstream services can act on. These help create new streams with more logically consistent data. In some cases, the application may need to filter events from an input stream that match certain critera, which results in a new stream with just a subset of records from the original stream. In other cases, the application may need to branch events, whereby each event is tested against a predicate and then routed to a stream that matches, which results in multiple new streams split from the original stream.



In this exercise, you will define one set of criteria to filter records in a stream based on some criteria. Then you will define define another set of criteria to branch records into two different streams.

Implement the *TODO* lines of the file exercises/FraudService.java

1. TODO 4.1: filter this stream to include only orders in "CREATED" state, i.e., it should satisfy the predicate *OrderState.CREATED.equals(order.getState())*
2. TODO 4.2: create a *KStream<String, OrderValue>* array from the *ordersWithTotals* stream by branching the records based on *OrderValue#getValue*
   1. First branched stream: FRAUD_CHECK will fail for predicate where order value >= FRAUD_LIMIT
   2. Second branched stream: FRAUD_CHECK will pass for predicate where order value < FRAUD_LIMIT

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KStream.html#filter-org.apache.kafka.streams.kstream.Predicate-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KStream.html#branch-org.apache.kafka.streams.kstream.Predicate...-
- kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/domain/beans/OrderBean.java

If you get stuck, here is the complete solution.

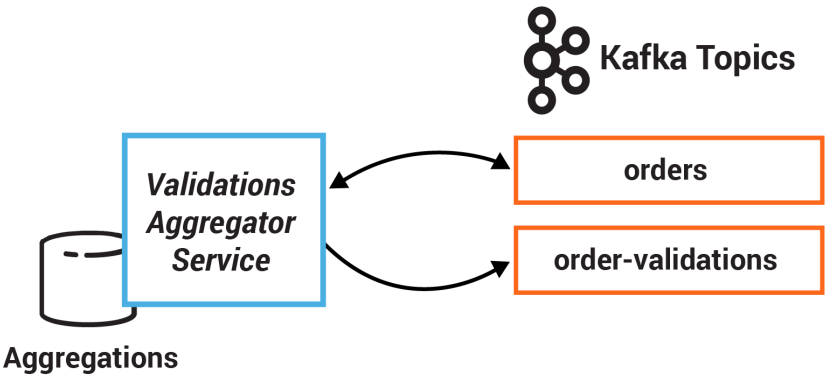
To test your code, save off the project's working solution, copy your version of the file to the main project, compile, and run the unit test.

```
# Clone and compile kafka-streams-examples
./get-kafka-streams-examples.sh

# Save off the working microservices client application to /tmp/
cp kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/FraudService.java /tmp/.

# Copy your exercise client application to the project
cp exercises/FraudService.java kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/.

# Compile the project and resolve any compilation errors
mvn clean compile -DskipTests -f kafka-streams-examples/pom.xml

# Run the test and validate that it passes
mvn compile -Dtest=io.confluent.examples.streams.microservices.FraudServiceTest test -f kafka-streams-examples/pom.xml
```

Exercise 5: Stateful operations

An aggregation operation takes one input stream or table, and yields a new table by combining multiple input records into a single output record. Examples of aggregations are computing `count` or `sum`, because they combine current record values with previous record values. These are stateful operations because they maintain data during processing. Aggregations are always key-based operations, and Kafka's Streams API ensures that records for the same key are always routed to the same stream processing task. Oftentimes, these are combined with windowing

capabilities in order to run computations in real time over a window of time.



In this exercise, you will create a session window to define five-minute windows for processing. Additionally, you will use a stateful operation *reduce* to collapse duplicate records in a stream. Before running *reduce*, you will group the records to repartition the data, which is generally required before using an aggregation operator.

Implement the *TODO* lines of the file exercises/ValidationsAggregatorService.java

1. TODO 5.1: window the data using *KGroupedStream#windowedBy*, specifically using *SessionWindows.with* to define 5-minute windows
2. TODO 5.2: group the records by key using *KStream#groupByKey*, providing the existing Serialized instance for ORDERS
3. TODO 5.3: use an aggregation operator *KTable#reduce* to collapse the records in this stream to a single order for a given key

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/SessionWindows.html#with-java.time.Duration-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KGroupedStream.html#windowedBy-org.apache.kafka.streams.kstream.SessionWindows-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/KStream.html#groupByKey-org.apache.kafka.streams.kstream.Serialized-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/kstream/SessionWindowedKStream.html#reduce-org.apache.kafka.streams.kstream.Reducer-

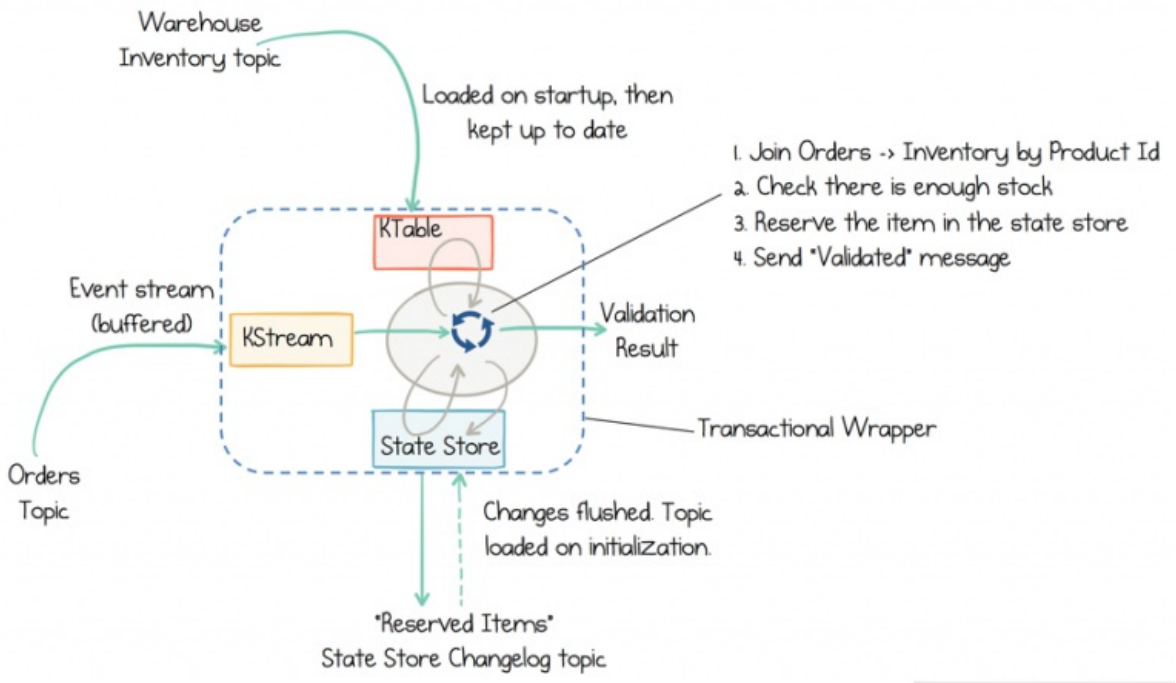
If you get stuck, here is the complete solution.

To test your code, save off the project's working solution, copy your version of the file to the main project, compile, and run the unit test.

```
# Clone and compile kafka-streams-examples
./get-kafka-streams-examples.sh

# Save off the working microservices client application to /tmp/
cp kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/ValidationsAggregatorService.java /t
p/.

# Copy your exercise client application to the project
cp exercises/ValidationsAggregatorService.java kafka-streams-examples/src/main/java/io/confluent/examples/streams/micros
rvices/.

# Compile the project and resolve any compilation errors
mvn clean compile -DskipTests -f kafka-streams-examples/pom.xml

# Run the test and validate that it passes
mvn compile -Dtest=io.confluent.examples.streams.microservices.ValidationsAggregatorServiceTest test -f kafka-streams-
examples/pom.xml
```

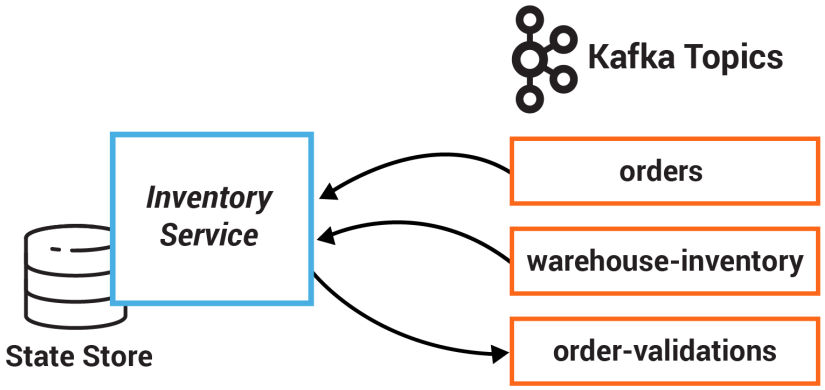
Exercise 6: State stores

Kafka Streams provides so-called state stores, which are disk-resident hash tables, held inside the API for the client application. The state store can be used within stream processing applications to store and query data, an important capability when implementing stateful operations. It can be used to remember recently received input records, to track rolling aggregates, to de-duplicate input records, etc.



State stores in Kafka Streams can be used to create use-case-specific views right inside the service (source)

It is also backed by a Kafka topic and comes with all the Kafka guarantees. Consequently, other applications can also interactively query another application's state store. Querying state stores is always read-only to guarantee that the underlying state stores will never be mutated out-of-band (i.e., you cannot add new entries).



In this exercise, you will create a state store for the Inventory Service. This state store is initialized with data from a Kafka topic before the service starts processing, and then it is updated as new orders are created.

Implement the *TODO* lines of the file exercises/InventoryService.java

1. TODO 6.1: create a state store called *RESERVED_STOCK_STORE_NAME*, using *Stores#keyValueStoreBuilder* and *Stores#persistentKeyValueStore*
    1. the key Serde is derived from the topic specified by *WAREHOUSE_INVENTORY*
    2. the value Serde is derived from *Serdes.Long()* because it represents a count

2. TODO 6.2: update the reserved stock in the KeyValueStore called *reservedStocksStore*
    1. the key is the product in the order, using *OrderBean#getProduct*
    2. the value is the sum of the current reserved stock and the quantity in the order, using *OrderBean#getQuantity*

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/state/Stores.html#persistentKeyValueStore-java.lang.String-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/state/Stores.html#keyValueStoreBuilder-org.apache.kafka.streams.state.KeyValueBytesStoreSupplier-org.apache.kafka.common.serialization.Serde-org.apache.kafka.common.serialization.Serde-
- https://docs.confluent.io/current/streams/javadocs/org/apache/kafka/streams/state/KeyValueStore.html#put-K-V-
- kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/domain/Schemas.java

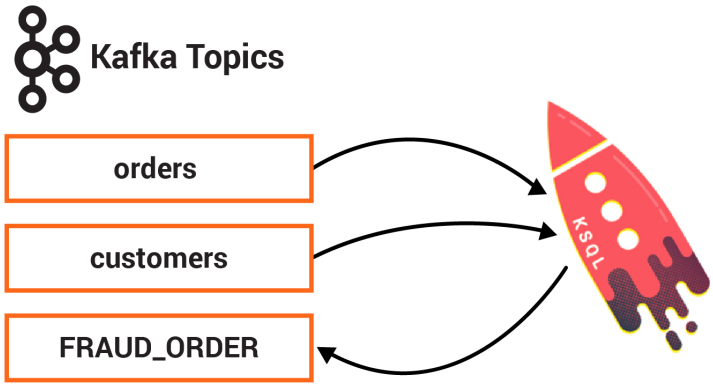
If you get stuck, here is the complete solution.

To test your code, save off the project's working solution, copy your version of the file to the main project, compile, and run the unit test.

```
# Clone and compile kafka-streams-examples
./get-kafka-streams-examples.sh

# Save off the working microservices client application to /tmp/
cp kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/InventoryService.java /tmp/.

# Copy your exercise client application to the project
cp exercises/InventoryService.java kafka-streams-examples/src/main/java/io/confluent/examples/streams/microservices/.

# Compile the project and resolve any compilation errors
mvn clean compile -DskipTests -f kafka-streams-examples/pom.xml

# Run the test and validate that it passes
mvn compile -Dtest=io.confluent.examples.streams.microservices.InventoryServiceTest test -f kafka-streams-examples/pom
.xml
```

Exercise 7: Enrichment with KSQL

Confluent KSQL is the streaming SQL engine that enables real-time data processing against Apache Kafka. It provides an easy-to-use, yet powerful interactive SQL interface for stream processing on Kafka, without requiring you to write code in a programming language such as Java or Python. KSQL is scalable, elastic, fault tolerant, and it supports a wide range of streaming operations, including data filtering, transformations, aggregations, joins, windowing, and sessionization.



You can use KSQL to merge streams of data in real time by using a SQL-like *join* syntax. A KSQL join and a relational database join are similar in that they both combine data from two sources based on common values. The result of a KSQL join is a new stream or table that's populated with the column values that you specify in a *SELECT* statement. KSQL also supports several aggregate functions, like *COUNT* and *SUM*. You can use these to build stateful aggregates on streaming data.

In this exercise, you will create one persistent query that enriches the *orders* stream with customer information using a stream-table join. You will create another persistent query that detects fraudulent behavior by counting the number of orders in a given window.

If you are running on local install, then type *ksql* to get to the KSQL CLI prompt. If you are running on Docker, then type *docker-compose exec*

ksql-cli ksql http://ksql-server:8088 to get to the KSQL CLI prompt.

Assume you already have a KSQL stream of orders called *orders* and a KSQL table of customers called *customers_table*. From the KSQL CLI prompt, type *DESCRIBE orders;* and *DESCRIBE customers_table;* to see the respective schemas. Then create the following persistent queries:

1. TODO 7.1: create a new KSQL stream that does a stream-table join between *orders* and *customers_table* based on customer id.
2. TODO 7.2: create a new KSQL table that counts if a customer submits more than 2 orders in a 30 second time window.

Tip

The following APIs will be helpful:

- https://docs.confluent.io/current/ksql/docs/developer-guide/create-a-stream.html#create-a-persistent-streaming-query-from-a-stream
- https://docs.confluent.io/current/ksql/docs/tutorials/examples.html#joining
- https://docs.confluent.io/current/ksql/docs/tutorials/examples.html#aggregating-windowing-and-sessionization

If you get stuck, here is the complete solution.

The CLI parser will give immediate feedback whether your KSQL queries worked or not. Use `SELECT * FROM <stream or table name>;` to see the rows in each query.

Additional Resources

- Kafka Streams videos
- Kafka Streams documentation
- Designing Event-Driven Systems
- Building a Microservices Ecosystem with Kafka Streams and KSQL
- Build Services on a Backbone of Events
- No More Silos: How to Integrate Your Databases with Apache Kafka and CDC

Please report any inaccuracies on this page or suggest an edit.

Rate this page

Last updated on Sep 10, 2019.