Quarkus - Using Apache Kafka Streams

This guide demonstrates how your Quarkus application can utilize the Apache Kafka Streams API to implement stream processing applications based on Apache Kafka.

Prerequisites

To complete this guide, you need:

- less than 30 minutes

- an IDE

- JDK 1.8+ installed with JAVA_HOME configured appropriately

- Apache Maven 3.6.3

- Docker Compose to start an Apache Kafka development cluster

- GraalVM installed if you want to run in native mode.

It is recommended, that you have read the Kafka quickstart before.

The Quarkus extension for Kafka Streams allows for very fast turnaround times during development by supporting the Quarkus Dev Mode (e.g. via `./mvnw compile quarkus:dev`). After changing the code of your Kafka Streams topology, the application will automatically be reloaded when the next input message arrives.

A recommended development set-up is to have some producer which creates test messages on the processed topic(s) in fixed intervals, e.g. every second and observe the streaming application's output topic(s) using a tool such as `kafkacat`. Using the dev mode, you'll instantly see messages on the output topic(s) as produced by the latest version of your streaming application when saving.

For the best development experience, we recommend applying the following configuration settings to your Kafka broker:

```
group.min.session.timeout.ms=250
```

Also specify the following settings in your Quarkus `application.properties`:

```
kafka-streams.consumer.session.timeout.ms=250
kafka-streams.consumer.heartbeat.interval.ms=200
```

Together, these settings will ensure that the application can very quickly reconnect to the broker after being restarted in dev mode.

Architecture

In this guide, we are going to generate (random) temperature values in one component (named `generator`). These values are associated to given weather stations and are written in a Kafka topic (`temperature-values`). Another topic (`weather-stations`) contains just the master data about the weather stations themselves (id and name).

A second component (`aggregator`) reads from the two Kafka topics and processes them in a streaming pipeline:

- the two topics are joined on weather station id

- per weather station the min, max and average temperature is determined

- this aggregated data is written out to a third topic (`temperatures-aggregated`)

The data can be examined by inspecting the output topic. By exposing a Kafka Streams interactive query, the latest result for each weather station can alternatively be obtained via a simple REST query.

The overall architecture looks like so:
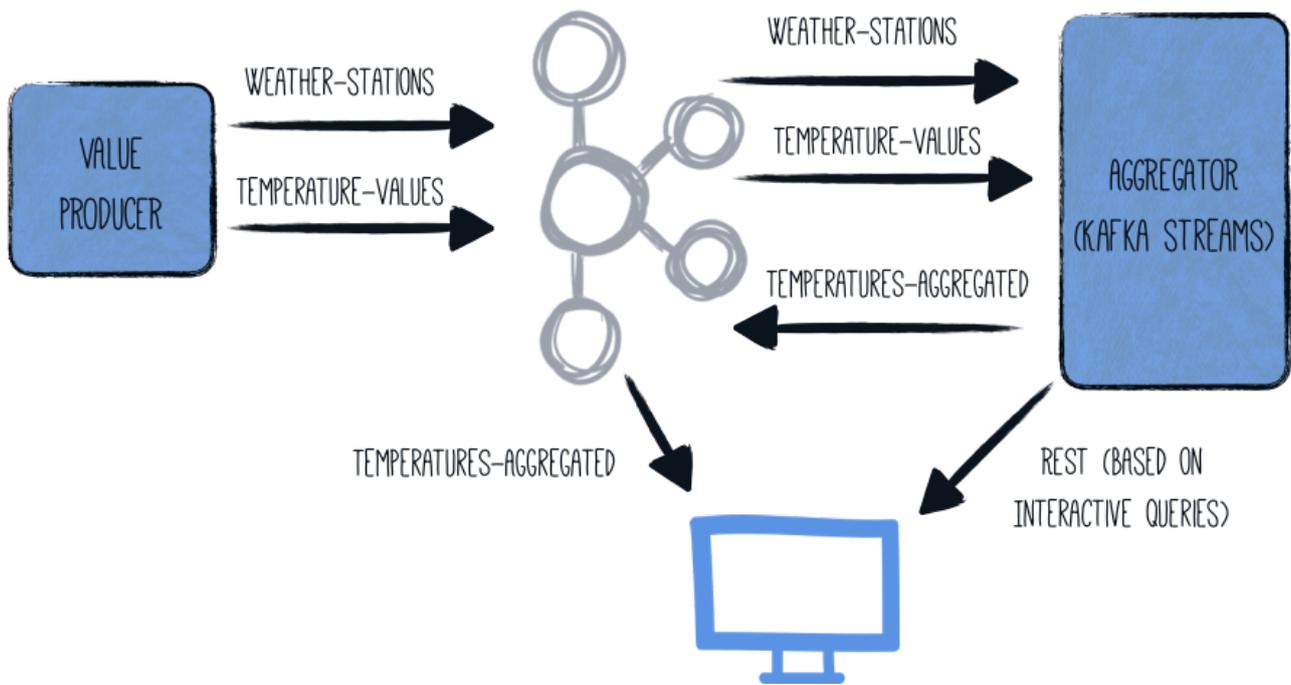
Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `kafka-streams-quickstart` directory.

Creating the Producer Maven Project

First, we need a new project with the temperature value producer. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=kafka-streams-quickstart-producer \
    -Dextensions="kafka" \
    && mv kafka-streams-quickstart-producer producer
```

This command generates a Maven project, importing the Reactive Messaging and Kafka connector extensions.

If you already have your Quarkus project configured, you can add the `smallrye-reactive-messaging-kafka` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="kafka"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-reactive-messaging-
kafka</artifactId>
</dependency>
```

The Temperature Value Producer

Create   the
`producer/src/main/java/org/acme/kafka/streams/producer/generator/ValuesGenerator.java` file, with the following content:

```
package org.acme.kafka.streams.producer.generator;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.Instant;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;

import javax.enterprise.context.ApplicationScoped;

import org.eclipse.microprofile.reactive.messaging.Outgoing;
import org.jboss.logging.Logger;

import io.reactivex.Flowable;
import io.smallrye.reactive.messaging.kafka.KafkaRecord;

/**
 * A bean producing random temperature data every second.
 * The values are written to a Kafka topic (temperature-values).
 * Another topic contains the name of weather stations (weather-
stations).
 * The Kafka configuration is specified in the application
configuration.
 */
@ApplicationScoped
public class ValuesGenerator {
```

```
    private static final Logger LOG =
Logger.getLogger(ValuesGenerator.class);

    private Random random = new Random();

    private List<WeatherStation> stations =
Collections.unmodifiableList(
            Arrays.asList(
                    new WeatherStation(1, "Hamburg", 13),
                    new WeatherStation(2, "Snowdonia", 5),
                    new WeatherStation(3, "Boston", 11),
                    new WeatherStation(4, "Tokio", 16),
                    new WeatherStation(5, "Cusco", 12),
                    new WeatherStation(6, "Svalbard", -7),
                    new WeatherStation(7, "Porthsmouth", 11),
                    new WeatherStation(8, "Oslo", 7),
                    new WeatherStation(9, "Marrakesh", 20)
            ));

    @Outgoing("temperature-values")                         ①
    public Flowable<KafkaRecord<Integer, String>> generate() {

        return Flowable.interval(500, TimeUnit.MILLISECONDS)   ②
                .onBackpressureDrop()
                .map(tick -> {
                    WeatherStation station =
stations.get(random.nextInt(stations.size()));
                    double temperature =
BigDecimal.valueOf(random.nextGaussian() * 15 +
station.averageTemperature)
                            .setScale(1, RoundingMode.HALF_UP)
                            .doubleValue();

                    LOG.infov("station: {0}, temperature: {1}",
station.name, temperature);
                    return KafkaRecord.of(station.id, Instant.now()
+ ";" + temperature);
                });
    }

    @Outgoing("weather-stations")                           ③
    public Flowable<KafkaRecord<Integer, String>> weatherStations()
{
        List<KafkaRecord<Integer, String>> stationsAsJson =
stations.stream()
            .map(s -> KafkaRecord.of(
                    s.id,
```

```
                        "{ \"id\" : " + s.id +
                        ", \"name\" : \"" + s.name + "\" }"))
              .collect(Collectors.toList());

        return Flowable.fromIterable(stationsAsJson);
    };

    private static class WeatherStation {

        int id;
        String name;
        int averageTemperature;

        public WeatherStation(int id, String name, int
 averageTemperature) {
            this.id = id;
            this.name = name;
            this.averageTemperature = averageTemperature;
        }
    }
 }
```

① Instruct Reactive Messaging to dispatch the items from the returned `Flowable` to `temperature-values`.

② The method returns a RX Java 2 *stream* (`Flowable`) emitting a random temperature value every 0.5 seconds.

③ Instruct Reactive Messaging to dispatch the items from the returned `Flowable` (static list of weather stations) to `weather-stations`.

The two methods each return a *reactive stream* whose items are sent to the streams named `temperature-values` and `weather-stations`, respectively.

Topic Configuration

The two channels are mapped to Kafka topics using the Quarkus configuration file `application.properties`. For that, add the following to the file `producer/src/main/resources/application.properties`:

```
# Configure the Kafka broker location
kafka.bootstrap.servers=localhost:9092

mp.messaging.outgoing.temperature-values.connector=smallrye-kafka
mp.messaging.outgoing.temperature-
values.key.serializer=org.apache.kafka.common.serialization.Integer
Serializer
mp.messaging.outgoing.temperature-
values.value.serializer=org.apache.kafka.common.serialization.Strin
gSerializer

mp.messaging.outgoing.weather-stations.connector=smallrye-kafka
mp.messaging.outgoing.weather-
stations.key.serializer=org.apache.kafka.common.serialization.Integ
erSerializer
mp.messaging.outgoing.weather-
stations.value.serializer=org.apache.kafka.common.serialization.Str
ingSerializer
```

This configures the Kafka bootstrap server, the two topics and the corresponding (de-)serializers. More details about the different configuration options are available on the Producer configuration and Consumer configuration section from the Kafka documentation.

Creating the Aggregator Maven Project

With the producer application in place, it's time to implement the actual aggregator application, which will run the Kafka Streams pipeline. Create another project like so:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=kafka-streams-quickstart-aggregator \
    -Dextensions="kafka-streams,resteasy-jsonb" \
    && mv kafka-streams-quickstart-aggregator aggregator
```

This creates the `aggregator` project with the Quarkus extension for Kafka Streams and with RESTEasy support for JSON-B.

If you already have your Quarkus project configured, you can add the `kafka-streams` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="kafka-streams"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-kafka-streams</artifactId>
</dependency>
```

The Pipeline Implementation

Let's begin the implementation of the stream processing application by creating a few value objects for representing temperature measurements, weather stations and for keeping track of aggregated values.

First, create the file aggregator/src/main/java/org/acme/kafka/streams/aggregator/model/WeatherStation.java, representing a weather station, with the following content:

```
package org.acme.kafka.streams.aggregator.model;

import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection ①
public class WeatherStation {

    public int id;
    public String name;
}
```

① By adding the @RegisterForReflection annotation, it is ensured that this type can be instantiated reflectively when running the application in native mode.

Then the file aggregator/src/main/java/org/acme/kafka/streams/aggregator/model/TemperatureMeasurement.java, representing temperature measurements for a given station:

```
package org.acme.kafka.streams.aggregator.model;

import java.time.Instant;

public class TemperatureMeasurement {

    public int stationId;
    public String stationName;
    public Instant timestamp;
    public double value;

    public TemperatureMeasurement(int stationId, String
stationName, Instant timestamp,
            double value) {
        this.stationId = stationId;
        this.stationName = stationName;
        this.timestamp = timestamp;
        this.value = value;
    }
}
```

And   finally
`aggregator/src/main/java/org/acme/kafka/streams/aggregator/model/Aggregati`
`on.java`, which will be used to keep track of the aggregated values while the events are processed in
the streaming pipeline:

```
package org.acme.kafka.streams.aggregator.model;

import java.math.BigDecimal;
import java.math.RoundingMode;

import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection
public class Aggregation {

    public int stationId;
    public String stationName;
    public double min = Double.MAX_VALUE;
    public double max = Double.MIN_VALUE;
    public int count;
    public double sum;
    public double avg;

    public Aggregation updateFrom(TemperatureMeasurement
measurement) {
        stationId = measurement.stationId;
        stationName = measurement.stationName;

        count++;
        sum += measurement.value;
        avg = BigDecimal.valueOf(sum / count)
                .setScale(1, RoundingMode.HALF_UP).doubleValue();

        min = Math.min(min, measurement.value);
        max = Math.max(max, measurement.value);

        return this;
    }
}
```

Next, let's create the actual streaming query implementation itself in the
aggregator/src/main/java/org/acme/kafka/streams/aggregator/streams/Topolog
yProducer.java file. All we need to do for that is to declare a CDI producer method which returns
the Kafka Streams Topology; the Quarkus extension will take care of configuring, starting and
stopping the actual Kafka Streams engine.

```
package org.acme.kafka.streams.aggregator.streams;

import java.time.Instant;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;
```

```java
import org.acme.kafka.streams.aggregator.model.Aggregation;
import
org.acme.kafka.streams.aggregator.model.TemperatureMeasurement;
import org.acme.kafka.streams.aggregator.model.WeatherStation;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.GlobalKTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;
import org.apache.kafka.streams.state.KeyValueBytesStoreSupplier;
import org.apache.kafka.streams.state.Stores;

import io.quarkus.kafka.client.serialization.JsonbSerde;

@ApplicationScoped
public class TopologyProducer {

    static final String WEATHER_STATIONS_STORE = "weather-stations-
store";

    private static final String WEATHER_STATIONS_TOPIC = "weather-
stations";
    private static final String TEMPERATURE_VALUES_TOPIC =
"temperature-values";
    private static final String TEMPERATURES_AGGREGATED_TOPIC =
"temperatures-aggregated";

    @Produces
    public Topology buildTopology() {
        StreamsBuilder builder = new StreamsBuilder();

        JsonbSerde<WeatherStation> weatherStationSerde = new
JsonbSerde<>(
                WeatherStation.class);
        JsonbSerde<Aggregation> aggregationSerde = new
JsonbSerde<>(Aggregation.class);

        KeyValueBytesStoreSupplier storeSupplier =
Stores.persistentKeyValueStore(
                WEATHER_STATIONS_STORE);

        GlobalKTable<Integer, WeatherStation> stations =
builder.globalTable( ①
                WEATHER_STATIONS_TOPIC,
                Consumed.with(Serdes.Integer(),
weatherStationSerde));

        builder.stream(
```

```
②                            TEMPERATURE_VALUES_TOPIC,
                             Consumed.with(Serdes.Integer(),
Serdes.String())
                )
                .join(
③
                        stations,
                        (stationId, timestampAndValue) ->
stationId,

                        (timestampAndValue, station) -> {
                            String[] parts =
timestampAndValue.split(";");
                            return new
TemperatureMeasurement(station.id, station.name,
                                  Instant.parse(parts[0]),
Double.valueOf(parts[1]));
                        }
                )
                .groupByKey()
④
                .aggregate(
⑤
                        Aggregation::new,
                        (stationId, value, aggregation) ->
aggregation.updateFrom(value),
                        Materialized.<Integer, Aggregation>
as(storeSupplier)
                            .withKeySerde(Serdes.Integer())
                            .withValueSerde(aggregationSerde)
                )
                .toStream()
                .to(
⑥
                        TEMPERATURES_AGGREGATED_TOPIC,
                        Produced.with(Serdes.Integer(),
aggregationSerde)
                );

        return builder.build();
    }
}
```

① The `weather-stations` table is read into a `GlobalKTable`, representing the current state of each weather station

② The `temperature-values` topic is read into a `KStream`; whenever a new message arrives to this topic, the pipeline will be processed for this measurement

③ The message from the `temperature-values` topic is joined with the corresponding weather station, using the topic's key (weather station id); the join result contains the data from the measurement and associated weather station message

④ The values are grouped by message key (the weather station id)

⑤ Within each group, all the measurements of that station are aggregated, by keeping track of minimum and maximum values and calculating the average value of all measurements of that station (see the `Aggregation` type)

⑥ The results of the pipeline are written out to the `temperatures-aggregated` topic

The Kafka Streams extension is configured via the Quarkus configuration file `application.properties`. Create the file `aggregator/src/main/resources/application.properties` with the following contents:

```
quarkus.kafka-streams.bootstrap-servers=localhost:9092
quarkus.kafka-streams.application-server=${hostname}:8080
quarkus.kafka-streams.topics=weather-stations,temperature-values

# pass-through options
kafka-streams.cache.max.bytes.buffering=10240
kafka-streams.commit.interval.ms=1000
kafka-streams.metadata.max.age.ms=500
kafka-streams.auto.offset.reset=earliest
kafka-streams.metrics.recording.level=DEBUG
```

The options with the `quarkus.kafka-streams` prefix can be changed dynamically at application startup, e.g. via environment variables or system properties. `bootstrap-servers` and `application-server` are mapped to the Kafka Streams properties `bootstrap.servers` and `application.server`, respectively. `topics` is specific to Quarkus: the application will wait for all the given topics to exist before launching the Kafka Streams engine. This is to done to gracefully await the creation of topics that don't yet exist at application startup time.

All the properties within the `kafka-streams` namespace are passed through as-is to the Kafka Streams engine. Changing their values requires a rebuild of the application.

Building and Running the Applications

We now can build the `producer` and `aggregator` applications:

```
./mvnw clean package -f producer/pom.xml
./mvnw clean package -f aggregator/pom.xml
```

Instead of running them directly on the host machine using the Quarkus dev mode, we're going to package them into container images and launch them via Docker Compose. This is done in order to demonstrate scaling the `aggregator` aggregation to multiple nodes later on.

The `Dockerfile` created by Quarkus by default needs one adjustment for the `aggregator` application in order to run the Kafka Streams pipeline. To do so, edit the file `aggregator/src/main/docker/Dockerfile.jvm` and replace the line `FROM fabric8/java-alpine-openjdk8-jre` with `FROM fabric8/java-centos-openjdk8-jdk`.

Next create a Docker Compose file (`docker-compose.yaml`) for spinning up the two applications as well as Apache Kafka and ZooKeeper like so:

```yaml
version: '3.5'

services:
  zookeeper:
    image: strimzi/kafka:0.11.3-kafka-2.1.0
    command: [
      "sh", "-c",
      "bin/zookeeper-server-start.sh config/zookeeper.properties"
    ]
    ports:
      - "2181:2181"
    environment:
      LOG_DIR: /tmp/logs
    networks:
      - kafkastreams-network
  kafka:
    image: strimzi/kafka:0.11.3-kafka-2.1.0
    command: [
      "sh", "-c",
      "bin/kafka-server-start.sh config/server.properties --override listeners=$${KAFKA_LISTENERS} --override advertised.listeners=$${KAFKA_ADVERTISED_LISTENERS} --override zookeeper.connect=$${KAFKA_ZOOKEEPER_CONNECT} --override num.partitions=$${KAFKA_NUM_PARTITIONS}"
    ]
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      LOG_DIR: "/tmp/logs"
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_NUM_PARTITIONS: 3
    networks:
      - kafkastreams-network

  producer:
    image: quarkus-quickstarts/kafka-streams-producer:1.0
    build:
```

```
        context: producer
        dockerfile: src/main/docker/Dockerfile.${QUARKUS_MODE:-jvm}
      environment:
        KAFKA_BOOTSTRAP_SERVERS: kafka:9092
      networks:
        - kafkastreams-network

  aggregator:
    image: quarkus-quickstarts/kafka-streams-aggregator:1.0
    build:
      context: aggregator
      dockerfile: src/main/docker/Dockerfile.${QUARKUS_MODE:-jvm}
      environment:
        QUARKUS_KAFKA_STREAMS_BOOTSTRAP_SERVERS: kafka:9092
      networks:
        - kafkastreams-network

networks:
  kafkastreams-network:
    name: ks
```

To launch all the containers, building the `producer` and `aggregator` container images, run `docker-compose up --build`.

You should see log statements from the `producer` application about messages being sent to the "temperature-values" topic.

Now run an instance of the *debezium/tooling* image, attaching to the same network all the other containers run in. This image provides several useful tools such as *kafkacat* and *httpie*:

```
docker run --tty --rm -i --network ks debezium/tooling:1.0
```

Within the tooling container, run *kafkacat* to examine the results of the streaming pipeline:

```
kafkacat -b kafka:9092 -C -o beginning -q -t temperatures-
aggregated

{"avg":34.7,"count":4,"max":49.4,"min":16.8,"stationId":9,"stationN
ame":"Marrakesh","sum":138.8}
{"avg":15.7,"count":1,"max":15.7,"min":15.7,"stationId":2,"stationN
ame":"Snowdonia","sum":15.7}
{"avg":12.8,"count":7,"max":25.5,"min":-
13.8,"stationId":7,"stationName":"Porthsmouth","sum":89.7}
...
```

You should see new values arrive as the producer continues to emit temperature measurements, each value on the outbound topic showing the minimum, maximum and average temperature values of the

represented weather station.

Interactive Queries

Subscribing to the `temperatures-aggregated` topic is a great way to react to any new temperature values. It's a bit wasteful though if you're just interested in the latest aggregated value for a given weather station. This is where Kafka Streams interactive queries shine: they let you directly query the underlying state store of the pipeline for the value associated to a given key. By exposing a simple REST endpoint which queries the state store, the latest aggregation result can be retrieved without having to subscribe to any Kafka topic.

Let's begin by creating a new class `InteractiveQueries` in the file `aggregator/src/main/java/org/acme/kafka/streams/aggregator/streams/InteractiveQueries.java`:

one more method to the `KafkaStreamsPipeline` class which obtains the current state for a given key:

```
package org.acme.kafka.streams.aggregator.streams;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.acme.kafka.streams.aggregator.model.Aggregation;
import org.acme.kafka.streams.aggregator.model.WeatherStationData;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.errors.InvalidStateStoreException;
import org.apache.kafka.streams.state.QueryableStoreTypes;
import org.apache.kafka.streams.state.ReadOnlyKeyValueStore;

@ApplicationScoped
public class InteractiveQueries {

    @Inject
    KafkaStreams streams;

    public GetWeatherStationDataResult getWeatherStationData(int
id) {
        Aggregation result = getWeatherStationStore().get(id);

        if (result != null) {
            return
GetWeatherStationDataResult.found(WeatherStationData.from(result));
①
        }
        else {
            return GetWeatherStationDataResult.notFound();
②
        }
    }

    private ReadOnlyKeyValueStore<Integer, Aggregation>
getWeatherStationStore() {
        while (true) {
            try {
                return
streams.store(TopologyProducer.WEATHER_STATIONS_STORE,
QueryableStoreTypes.keyValueStore());
            } catch (InvalidStateStoreException e) {
                // ignore, store not ready yet
            }
        }
    }
}
```

① A value for the given station id was found, so that value will be returned

② No value was found, either because a non-existing station was queried or no measurement exists yet for the given station

Also create the method's return type in the file `aggregator/src/main/java/org/acme/kafka/streams/aggregator/streams/GetWeatherStationDataResult.java`:

```java
package org.acme.kafka.streams.aggregator.streams;

import java.util.Optional;
import java.util.OptionalInt;

import org.acme.kafka.streams.aggregator.model.WeatherStationData;

public class GetWeatherStationDataResult {

    private static GetWeatherStationDataResult NOT_FOUND =
            new GetWeatherStationDataResult(null);

    private final WeatherStationData result;

    private GetWeatherStationDataResult(WeatherStationData result) {
        this.result = result;
    }

    public static GetWeatherStationDataResult found(WeatherStationData data) {
        return new GetWeatherStationDataResult(data);
    }

    public static GetWeatherStationDataResult notFound() {
        return NOT_FOUND;
    }

    public Optional<WeatherStationData> getResult() {
        return Optional.ofNullable(result);
    }
}
```

Also create `aggregator/src/main/java/org/acme/kafka/streams/aggregator/model/WeatherStationData.java`, which represents the actual aggregation result for a weather station:

```
package org.acme.kafka.streams.aggregator.model;

import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection
public class WeatherStationData {

    public int stationId;
    public String stationName;
    public double min = Double.MAX_VALUE;
    public double max = Double.MIN_VALUE;
    public int count;
    public double avg;

    private WeatherStationData(int stationId, String stationName,
double min, double max,
            int count, double avg) {
        this.stationId = stationId;
        this.stationName = stationName;
        this.min = min;
        this.max = max;
        this.count = count;
        this.avg = avg;
    }

    public static WeatherStationData from(Aggregation aggregation)
{
        return new WeatherStationData(
                aggregation.stationId,
                aggregation.stationName,
                aggregation.min,
                aggregation.max,
                aggregation.count,
                aggregation.avg);
    }
}
```

We now can add a simple REST endpoint (`aggregator/src/main/java/org/acme/kafka/streams/aggregator/rest/WeatherStationEndpoint.java`), which invokes `getWeatherStationData()` and returns the data to the client:

```
package org.acme.kafka.streams.aggregator.rest;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.List;
```

```
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

import
org.acme.kafka.streams.aggregator.streams.GetWeatherStationDataResu
lt;
import
org.acme.kafka.streams.aggregator.streams.KafkaStreamsPipeline;

@ApplicationScoped
@Path("/weather-stations")
public class WeatherStationEndpoint {

    @Inject
    InteractiveQueries interactiveQueries;

    @GET
    @Path("/data/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response getWeatherStationData(@PathParam("id") int id)
{
        GetWeatherStationDataResult result =
interactiveQueries.getWeatherStationData(id);

        if (result.getResult().isPresent()) {   ①
            return Response.ok(result.getResult().get()).build();
        }
        else {
            return
Response.status(Status.NOT_FOUND.getStatusCode(),
                    "No data found for weather station " +
id).build();
        }
    }
}
```

① Depending on whether a value was obtained, either return that value or a 404 response

With this code in place, it's time to rebuild the application and the `aggregator` service in Docker

Compose:

```
./mvnw clean package -f aggregator/pom.xml
docker-compose stop aggregator
docker-compose up --build -d
```

This will rebuild the `aggregator` container and restart its service. Once that's done, you can invoke the service's REST API to obtain the temperature data for one of the existing stations. To do so, you can use `httpie` in the tooling container launched before:

```
http aggregator:8080/weather-stations/data/1

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 85
Content-Type: application/json
Date: Tue, 18 Jun 2019 19:29:16 GMT

{
    "avg": 12.9,
    "count": 146,
    "max": 41.0,
    "min": -25.6,
    "stationId": 1,
    "stationName": "Hamburg"
}
```

Scaling Out

A very interesting trait of Kafka Streams applications is that they can be scaled out, i.e. the load and state can be distributed amongst multiple application instances running the same pipeline. Each node will then contain a subset of the aggregation results, but Kafka Streams provides you with an API to obtain the information which node is hosting a given key. The application can then either fetch the data directly from the other instance, or simply point the client to the location of that other node.

Launching multiple instances of the `aggregator` application will make look the overall architecture like so:
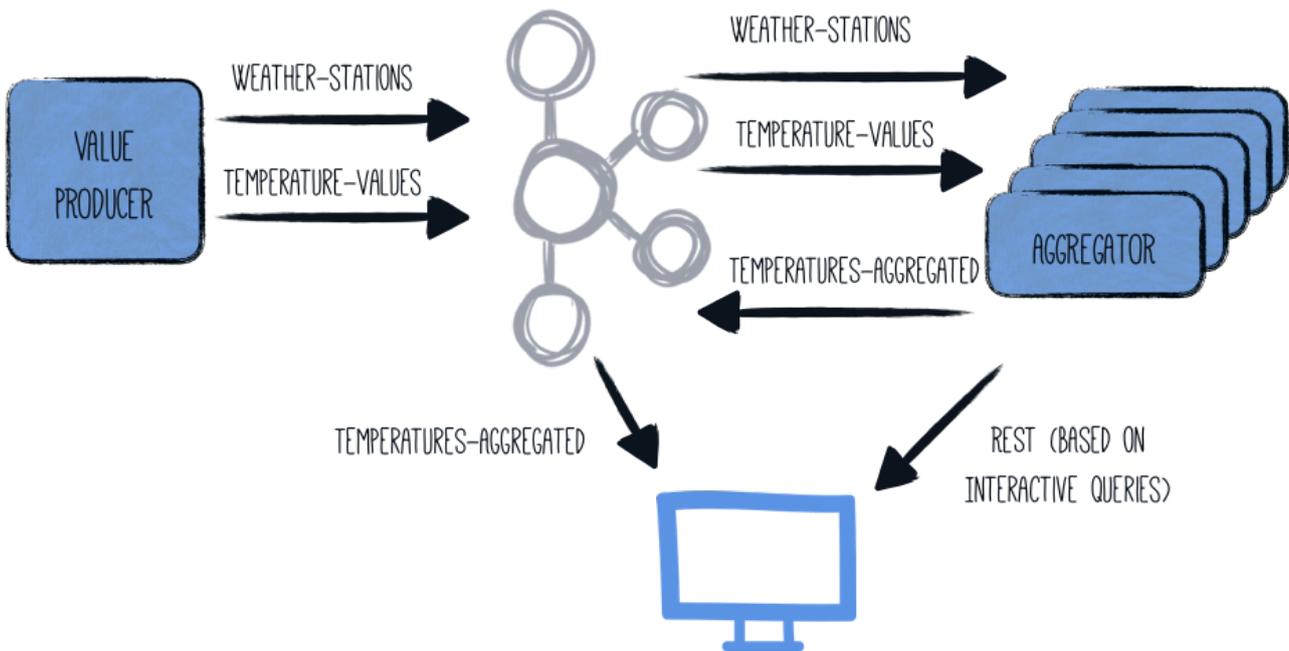
The `InteractiveQueries` class must be adjusted slightly for this distributed architecture:

```java
public GetWeatherStationDataResult getWeatherStationData(int id) {
    StreamsMetadata metadata = streams.metadataForKey(
①
            TopologyProducer.WEATHER_STATIONS_STORE,
            id,
            Serdes.Integer().serializer()
    );

    if (metadata == null || metadata ==
StreamsMetadata.NOT_AVAILABLE) {
        LOG.warn("Found no metadata for key {}", id);
        return GetWeatherStationDataResult.notFound();
    }
    else if (metadata.host().equals(host)) {
②
        LOG.info("Found data for key {} locally", id);
        Aggregation result = getWeatherStationStore().get(id);

        if (result != null) {
            return
GetWeatherStationDataResult.found(WeatherStationData.from(result));
        }
        else {
            return GetWeatherStationDataResult.notFound();
        }
    }
    else {
③
        LOG.info(
```

```
                "Found data for key {} on remote host {}:{}",
                id,
                metadata.host(),
                metadata.port()
        );
        return
GetWeatherStationDataResult.foundRemotely(metadata.host(),
metadata.port());
    }
}

public List<PipelineMetadata> getMetaData() {
④
    return
streams.allMetadataForStore(TopologyProducer.WEATHER_STATIONS_STORE
)
            .stream()
            .map(m -> new PipelineMetadata(
                    m.hostInfo().host() + ":" +
m.hostInfo().port(),
                    m.topicPartitions()
                        .stream()
                        .map(TopicPartition::toString)
                        .collect(Collectors.toSet()))
            )
            .collect(Collectors.toList());
}
```

① The streams metadata for the given weather station id is obtained

② The given key (weather station id) is maintained by the local application node, i.e. it can answer the query itself

③ The given key is maintained by another application node; in this case the information about that node (host and port) will be returned

④ The `getMetaData()` method is added to provide callers with a list of all the nodes in the application cluster.

The `GetWeatherStationDataResult` type must be adjusted accordingly:

```
package org.acme.kafka.streams.aggregator.streams;

import java.util.Optional;
import java.util.OptionalInt;

import org.acme.kafka.streams.aggregator.model.WeatherStationData;

public class GetWeatherStationDataResult {
```

```java
    private static GetWeatherStationDataResult NOT_FOUND =
            new GetWeatherStationDataResult(null, null, null);

    private final WeatherStationData result;
    private final String host;
    private final Integer port;

    private GetWeatherStationDataResult(WeatherStationData result, String host,
            Integer port) {
        this.result = result;
        this.host = host;
        this.port = port;
    }

    public static GetWeatherStationDataResult
found(WeatherStationData data) {
        return new GetWeatherStationDataResult(data, null, null);
    }

    public static GetWeatherStationDataResult foundRemotely(String
host, int port) {
        return new GetWeatherStationDataResult(null, host, port);
    }

    public static GetWeatherStationDataResult notFound() {
        return NOT_FOUND;
    }

    public Optional<WeatherStationData> getResult() {
        return Optional.ofNullable(result);
    }

    public Optional<String> getHost() {
        return Optional.ofNullable(host);
    }

    public OptionalInt getPort() {
        return port != null ? OptionalInt.of(port) :
OptionalInt.empty();
    }
}
```

Also the return type for `getMetaData()` must be defined (`aggregator/src/main/java/org/acme/kafka/streams/aggregator/streams/PipelineMetadata.java`):

```java
package org.acme.kafka.streams.aggregator.streams;

import java.util.Set;

public class PipelineMetadata {

    public String host;
    public Set<String> partitions;

    public PipelineMetadata(String host, Set<String> partitions) {
        this.host = host;
        this.partitions = partitions;
    }
}
```

Lastly, the REST endpoint class must be updated:

```java
package org.acme.kafka.streams.aggregator.rest;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.List;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

import
org.acme.kafka.streams.aggregator.streams.GetWeatherStationDataResu
lt;
import
org.acme.kafka.streams.aggregator.streams.KafkaStreamsPipeline;
import org.acme.kafka.streams.aggregator.streams.PipelineMetadata;

@ApplicationScoped
@Path("/weather-stations")
public class WeatherStationEndpoint {

    @Inject
    InteractiveQueries interactiveQueries;
```

```java
    @GET
    @Path("/data/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response getWeatherStationData(@PathParam("id") int id)
{
        GetWeatherStationDataResult result =
interactiveQueries.getWeatherStationData(id);

        if (result.getResult().isPresent()) {
①
            return Response.ok(result.getResult().get()).build();
        }
        else if (result.getHost().isPresent()) {
②
            URI otherUri = getOtherUri(result.getHost().get(),
result.getPort().getAsInt(),
                    id);
            return Response.seeOther(otherUri).build();
        }
        else {
③
            return
Response.status(Status.NOT_FOUND.getStatusCode(),
                    "No data found for weather station " +
id).build();
        }
    }

    @GET
    @Path("/meta-data")
    @Produces(MediaType.APPLICATION_JSON)
    public List<PipelineMetadata> getMetaData() {
④
        return interactiveQueries.getMetaData();
    }

    private URI getOtherUri(String host, int port, int id) {
        try {
            return new URI("http://" + host + ":" + port +
"/weather-stations/data/" + id);
        }
        catch (URISyntaxException e) {
            throw new RuntimeException(e);
        }
    }
}
```

① The data was found locally, so return it

② The data is maintained by another node, so reply with a redirect (HTTP status code 303) if the data for the given key is stored on one of the other nodes.

③ No data was found for the given weather station id

④ Exposes information about all the hosts forming the application cluster

Now stop the `aggregator` service again and rebuild it. Then let's spin up three instances of it:

```
./mvnw clean package -f aggregator/pom.xml
docker-compose stop aggregator
docker-compose up --build -d --scale aggregator=3
```

When invoking the REST API on any of the three instances, it might either be that the aggregation for the requested weather station id is stored locally on the node receiving the query, or it could be stored on one of the other two nodes.

As the load balancer of Docker Compose will distribute requests to the `aggregator` service in a round-robin fashion, we'll invoke the actual nodes directly. The application exposes information about all the host names via REST:

```
http aggregator:8080/weather-stations/meta-data

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 202
Content-Type: application/json
Date: Tue, 18 Jun 2019 20:00:23 GMT

[
    {
        "host": "2af13fe516a9:8080",
        "partitions": [
            "temperature-values-2"
        ]
    },
    {
        "host": "32cc8309611b:8080",
        "partitions": [
            "temperature-values-1"
        ]
    },
    {
        "host": "1eb39af8d587:8080",
        "partitions": [
            "temperature-values-0"
        ]
    }
]
```

Retrieve the data from one of the three hosts shown in the response (your actual host names will differ):

```
http 2af13fe516a9:8080/weather-stations/data/1
```

If that node holds the data for key "1", you'll get a response like this:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 74
Content-Type: application/json
Date: Tue, 11 Jun 2019 19:16:31 GMT

{
  "avg": 11.9,
  "count": 259,
  "max": 50.0,
  "min": -30.1,
  "stationId": 1,
  "stationName": "Hamburg"
}
```

Otherwise, the service will send a redirect:

```
HTTP/1.1 303 See Other
Connection: keep-alive
Content-Length: 0
Date: Tue, 18 Jun 2019 20:01:03 GMT
Location: http://1eb39af8d587:8080/weather-stations/data/1
```

You can also have *httpie* automatically follow the redirect by passing the `--follow option`:

```
http --follow 2af13fe516a9:8080/weather-stations/data/1
```

Running Natively

The Quarkus extension for Kafka Streams enables the execution of stream processing applications natively via GraalVM without further configuration.

To run both the `producer` and `aggregator` applications in native mode, the Maven builds can be executed using the `native` profile:

```
./mvnw clean package -f producer/pom.xml -Pnative -Dnative
-image.container-runtime=docker
./mvnw clean package -f aggregator/pom.xml -Pnative -Dnative
-image.container-runtime=docker
```

Now create an environment variable named `QUARKUS_MODE` and with value set to "native":

```
export QUARKUS_MODE=native
```

This is used by the Docker Compose file to use the correct `Dockerfile` when building the `producer` and `aggregator` images. The Kafka Streams application can work with less than 50 MB RSS in native mode. To do so, add the `Xmx` option to the program invocation in `aggregator/src/main/docker/Dockerfile.native`:

```
CMD ["./application", "-Dquarkus.http.host=0.0.0.0", "-Xmx32m"]
```

Now start Docker Compose as described above (don't forget to rebuild the container images).

Kafka Streams Health Checks

If you are using the `quarkus-smallrye-health` extension, `quarkus-kafka-streams` will automatically add:

- a readiness health check to validate that all topics declared in the `quarkus.kafka-streams.topics` property are created,
- a liveness health check based on the Kafka Streams state.

So when you access the `/health` endpoint of your application you will have information about the state of the Kafka Streams and the available and/or missing topics.

This is an example of when the status is `DOWN`:

```
curl -i http://aggregator:8080/health

HTTP/1.1 503 Service Unavailable
content-type: application/json; charset=UTF-8
content-length: 454

{
    "status": "DOWN",
    "checks": [
        {
            "name": "Kafka Streams state health check",   ①
            "status": "DOWN",
            "data": {
                "state": "CREATED"
            }
        },
        {
            "name": "Kafka Streams topics health check",   ②
            "status": "DOWN",
            "data": {
                "available_topics": "weather-stations,temperature-
values",
                "missing_topics": "hygrometry-values"
            }
        }
    ]
}
```

① Liveness health check. Also available at `/health/live` endpoint.

② Readiness health check. Also available at `/health/ready` endpoint.

So as you can see, the status is `DOWN` as soon as one of the `quarkus.kafka-streams.topics` is missing or the Kafka Streams `state` is not `RUNNING`.

If no topics are available, the `available_topics` key will not be present in the `data` field of the `Kafka Streams topics health check`. As well as if no topics are missing, the `missing_topics` key will not be present in the `data` field of the `Kafka Streams topics health check`.

You can of course disable the health check of the `quarkus-kafka-streams` extension by setting the `quarkus.kafka-streams.health.enabled` property to `false` in your `application.properties`.

Obviously you can create your liveness and readiness probes based on the respective endpoints `/health/live` and `/health/ready`.

Liveness health check

Here is an example of the liveness check:

```
curl -i http://aggregator:8080/health/live

HTTP/1.1 503 Service Unavailable
content-type: application/json; charset=UTF-8
content-length: 225

{
    "status": "DOWN",
    "checks": [
        {
            "name": "Kafka Streams state health check",
            "status": "DOWN",
            "data": {
                "state": "CREATED"
            }
        }
    ]
}
```

The `state` is coming from the `KafkaStreams.State` enum.

Readiness health check

Here is an example of the readiness check:

```
curl -i http://aggregator:8080/health/ready

HTTP/1.1 503 Service Unavailable
content-type: application/json; charset=UTF-8
content-length: 265

{
    "status": "DOWN",
    "checks": [
        {
            "name": "Kafka Streams topics health check",
            "status": "DOWN",
            "data": {
                "missing_topics": "weather-stations,temperature-
values"
            }
        }
    ]
}
```

Going Further

This guide has shown how you can build stream processing applications using Quarkus and the Kafka Streams APIs, both in JVM and native modes. For running your KStreams application in production, you could also add health checks and metrics for the data pipeline. Refer to the Quarkus guides on metrics and health checks to learn more.

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.kafka-streams.health.enabled` Whether or not a health check is published in case the smallrye-health extension is present (defaults to true).	boolean	`true`
`quarkus.kafka-streams.application-id` A unique identifier for this Kafka Streams application. If not set, defaults to quarkus.application.name.	string	`${quarkus.application.name}`

`quarkus.kafka-streams.bootstrap-servers` A comma-separated list of host:port pairs identifying the Kafka bootstrap server(s)	list of host:port	`localhost:9012`
`quarkus.kafka-streams.application-server` A unique identifier of this application instance, typically in the form host:port.	string	
`quarkus.kafka-streams.topics` A comma-separated list of topic names. The pipeline will only be started once all these topics are present in the Kafka cluster.	list of string	required ❗
`quarkus.kafka-streams.schema-registry-key` The schema registry key. e.g. to diff between different registry impls / instances as they have this registry url under different property key. Red Hat / Apicurio - apicurio.registry.url Confluent - schema.registry.url	string	`schema.registry.url`
`quarkus.kafka-streams.schema-registry-url` The schema registry url.	string	
`quarkus.kafka-streams.security.protocol` The security protocol to use See https://docs.confluent.io/current/streams/developer-guide/security.html#security-example	string	
`quarkus.kafka-streams.sasl.jaas-config` JAAS login context parameters for SASL connections in the format used by JAAS configuration files	string	
`quarkus.kafka-streams.sasl.client-callback-handler-class` The fully qualified name of a SASL client callback handler class	string	
`quarkus.kafka-streams.sasl.login-callback-handler-class` The fully qualified name of a SASL login callback handler class	string	
`quarkus.kafka-streams.sasl.login-class` The fully qualified name of a class that implements the Login interface	string	

`quarkus.kafka-streams.sasl.kerberos-service-name` The Kerberos principal name that Kafka runs as	string	
`quarkus.kafka-streams.sasl.kerberos-kinit-cmd` Kerberos kinit command path	string	
`quarkus.kafka-streams.sasl.kerberos-ticket-renew-window-factor` Login thread will sleep until the specified window factor of time from last refresh	double	
`quarkus.kafka-streams.sasl.kerberos-ticket-renew-jitter` Percentage of random jitter added to the renewal time	double	
`quarkus.kafka-streams.sasl.kerberos-min-time-before-relogin` Percentage of random jitter added to the renewal time	long	
`quarkus.kafka-streams.sasl.login-refresh-window-factor` Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached-	double	
`quarkus.kafka-streams.sasl.login-refresh-window-jitter` The maximum amount of random jitter relative to the credential's lifetime	double	
`quarkus.kafka-streams.sasl.login-refresh-min-period` The desired minimum duration for the login refresh thread to wait before refreshing a credential	Duration ❓	
`quarkus.kafka-streams.sasl.login-refresh-buffer` The amount of buffer duration before credential expiration to maintain when refreshing a credential	Duration ❓	
`quarkus.kafka-streams.ssl.protocol` The SSL protocol used to generate the SSLContext	string	
`quarkus.kafka-streams.ssl.provider` The name of the security provider used for SSL connections	string	

`quarkus.kafka-streams.ssl.cipher-suites` A list of cipher suites	string	
`quarkus.kafka-streams.ssl.enabled-protocols` The list of protocols enabled for SSL connections	string	
`quarkus.kafka-streams.ssl.truststore.type` Store type	string	
`quarkus.kafka-streams.ssl.truststore.location` Store location	string	
`quarkus.kafka-streams.ssl.truststore.password` Store password	string	
`quarkus.kafka-streams.ssl.keystore.type` Store type	string	
`quarkus.kafka-streams.ssl.keystore.location` Store location	string	
`quarkus.kafka-streams.ssl.keystore.password` Store password	string	
`quarkus.kafka-streams.ssl.key.type` Store type	string	
`quarkus.kafka-streams.ssl.key.location` Store location	string	
`quarkus.kafka-streams.ssl.key.password` Store password	string	
`quarkus.kafka-streams.ssl.keymanager-algorithm` The algorithm used by key manager factory for SSL connections	string	

`quarkus.kafka-streams.ssl.trustmanager-algorithm` The algorithm used by trust manager factory for SSL connections	string	
`quarkus.kafka-streams.ssl.endpoint-identification-algorithm` The endpoint identification algorithm to validate server hostname using server certificate	string	`https`
`quarkus.kafka-streams.ssl.secure-random-implementation` The SecureRandom PRNG implementation to use for SSL cryptography operations	string	

About the Duration format

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.