Quarkus - Using Apache Kafka with Reactive Messaging

This guide demonstrates how your Quarkus application can utilize MicroProfile Reactive Messaging to interact with Apache Kafka.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- A running Kafka cluster, or Docker Compose to start a development cluster
- GraalVM installed if you want to run in native mode.

Architecture

In this guide, we are going to generate (random) prices in one component. These prices are written in a Kafka topic (`prices`). A second component reads from the `prices` Kafka topic and apply some magic conversion to the price. The result is sent to an in-memory stream consumed by a JAX-RS resource. The data is sent to a browser using server-sent events.
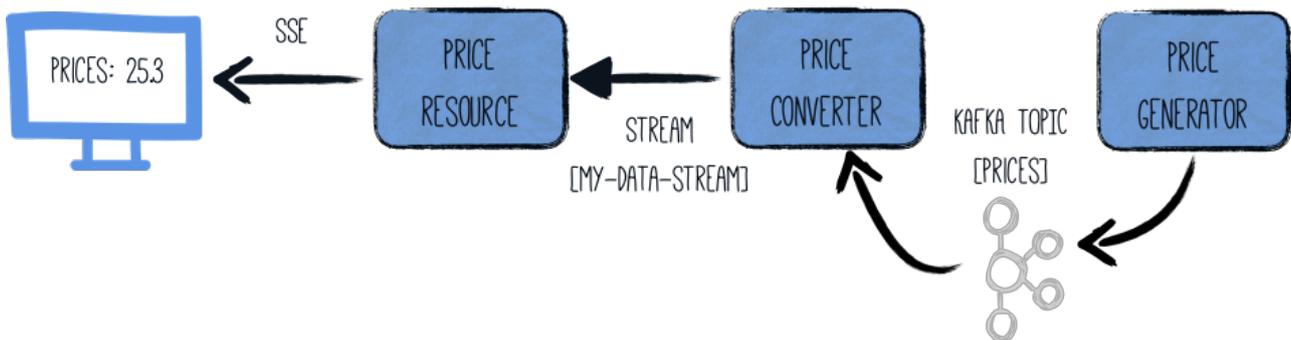
Solution

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-quickstart` directory.

Creating the Maven Project

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

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.Alpha1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=kafka-quickstart \
    -Dextensions="kafka"
cd kafka-quickstart
```

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

Starting Kafka

Then, we need a Kafka cluster. You can follow the instructions from the Apache Kafka web site or create a `docker-compose.yaml` file with the following content:

```yaml
version: '2'

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

  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}"
    ]
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      LOG_DIR: "/tmp/logs"
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

Once created, run `docker-compose up`.

ⓘ   This is a development cluster, do not use in production.

The price generator

Create the `src/main/java/org/acme/quarkus/sample/PriceGenerator.java` file, with the following content:

```java
package org.acme.quarkus.sample;

import io.reactivex.Flowable;
import org.eclipse.microprofile.reactive.messaging.Outgoing;

import javax.enterprise.context.ApplicationScoped;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
 * A bean producing random prices every 5 seconds.
 * The prices are written to a Kafka topic (prices). The Kafka
configuration is specified in the application configuration.
 */
@ApplicationScoped
public class PriceGenerator {

    private Random random = new Random();

    @Outgoing("generated-price")                          ①
    public Flowable<Integer> generate() {                 ②
        return Flowable.interval(5, TimeUnit.SECONDS)
                .map(tick -> random.nextInt(100));
    }

}
```

① Instruct Reactive Messaging to dispatch the items from returned stream to `generated-price`.

② The method returns a RX Java 2 *stream* (`Flowable`) emitting a random *price* every 5 seconds.

The method returns a *Reactive Stream*. The generated items are sent to the stream named `generated-price`. This stream is mapped to Kafka using the `application.properties` file that we will create soon.

The price converter

The price converter reads the prices from Kafka, and transforms them. Create the `src/main/java/org/acme/quarkus/sample/PriceConverter.java` file with the following content:

```java
package org.acme.quarkus.sample;

import io.smallrye.reactive.messaging.annotations.Broadcast;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;

import javax.enterprise.context.ApplicationScoped;

/**
 * A bean consuming data from the "prices" Kafka topic and applying
some conversion.
 * The result is pushed to the "my-data-stream" stream which is an
in-memory stream.
 */
@ApplicationScoped
public class PriceConverter {

    private static final double CONVERSION_RATE = 0.88;

    @Incoming("prices")                                           ①
    @Outgoing("my-data-stream")                                   ②
    @Broadcast                                                    ③
    public double process(int priceInUsd) {
        return priceInUsd * CONVERSION_RATE;
    }

}
```

① Indicates that the method consumes the items from the `prices` topic

② Indicates that the objects returned by the method are sent to the `my-data-stream` stream

③ Indicates that the item are dispatched to all *subscribers*

The `process` method is called for every Kafka *record* from the `prices` topic (configured in the application configuration). Every result is sent to the `my-data-stream` in-memory stream.

The price resource

Finally, let's bind our stream to a JAX-RS resource. Creates the `src/main/java/org/acme/quarkus/sample/PriceResource.java` file with the following content:

```java
package org.acme.quarkus.sample;

import io.smallrye.reactive.messaging.annotations.Channel;
import org.reactivestreams.Publisher;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.jboss.resteasy.annotations.SseElementType;

/**
 * A simple resource retrieving the in-memory "my-data-stream" and
sending the items as server-sent events.
 */
@Path("/prices")
public class PriceResource {

    @Inject
    @Channel("my-data-stream") Publisher<Double> prices;  ①

    @GET
    @Path("/stream")
    @Produces(MediaType.SERVER_SENT_EVENTS)  ②
    @SseElementType("text/plain")  ③
    public Publisher<Double> stream() {  ④
        return prices;
    }
}
```

① Injects the `my-data-stream` channel using the `@Channel` qualifier

② Indicates that the content is sent using `Server Sent Events`

③ Indicates that the data contained within the server sent events is of type `text/plain`

④ Returns the stream (*Reactive Stream*)

Configuring the Kafka connector

We need to configure the Kafka connector. This is done in the `application.properties` file. The keys are structured as follows:

`mp.messaging.[outgoing|incoming].{channel-name}.property=value`

The `channel-name` segment must match the value set in the `@Incoming` and `@Outgoing` annotation:

- generated-price → sink in which we write the prices

- prices → source in which we read the prices

```
# Configure the Kafka sink (we write to it)
mp.messaging.outgoing.generated-price.connector=smallrye-kafka
mp.messaging.outgoing.generated-price.topic=prices
mp.messaging.outgoing.generated-
price.value.serializer=org.apache.kafka.common.serialization.Intege
rSerializer

# Configure the Kafka source (we read from it)
mp.messaging.incoming.prices.connector=smallrye-kafka
mp.messaging.incoming.prices.value.deserializer=org.apache.kafka.co
mmon.serialization.IntegerDeserializer
```

More details about this configuration is available on the Producer configuration and Consumer configuration section from the Kafka documentation.

> ℹ️ What about my-data-stream? This is an in-memory stream, not connected to a message broker.

The HTML page

Final touch, the HTML page reading the converted prices using SSE.

Create the src/main/resources/META-INF/resources/prices.html file, with the following content:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Prices</title>

    <link rel="stylesheet" type="text/css"
          href=
"https://cdnjs.cloudflare.com/ajax/libs/patternfly/3.24.0/css/patte
rnfly.min.css">
    <link rel="stylesheet" type="text/css"
          href=
"https://cdnjs.cloudflare.com/ajax/libs/patternfly/3.24.0/css/patte
rnfly-additions.min.css">
</head>
<body>
<div class="container">

    <h2>Last price</h2>
    <div class="row">
    <p class="col-md-12">The last price is <strong><span id=
"content">N/A</span> &euro;</strong>.</p>
    </div>
</div>
</body>
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script>
    var source = new EventSource("/prices/stream");
    source.onmessage = function (event) {
        document.getElementById("content").innerHTML = event.data;
    };
</script>
</html>
```

Nothing spectacular here. On each received price, it updates the page.

Get it running

If you followed the instructions, you should have Kafka running. Then, you just need to run the application using:

```
./mvnw compile quarkus:dev
```

Open http://localhost:8080/prices.html in your browser.

> ℹ️ If you started the Kafka broker with docker compose, stop it using `CTRL+C` followed by `docker-compose down`.

Running Native

You can build the native executable with:

```
./mvnw package -Pnative
```

Imperative usage

Sometimes, you need to have an imperative way of sending messages.

For example, if you need to send a message to a stream, from inside a REST endpoint, when receiving a POST request. In this case, you cannot use `@Output` because your method has parameters.

For this, you can use an `Emitter`.

```java
import io.smallrye.reactive.messaging.annotations.Channel;
import io.smallrye.reactive.messaging.annotations.Emitter;

import javax.inject.Inject;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Consumes;
import javax.ws.rs.core.MediaType;

@Path("/prices")
public class PriceResource {

    @Inject @Channel("price-create") Emitter<Double> priceEmitter;

    @POST
    @Consumes(MediaType.TEXT_PLAIN)
    public void addPrice(Double price) {
        priceEmitter.send(price);
    }
}
```

> ℹ️ The `Emitter` configuration is done the same way as the other stream configuration used by `@Incoming` and `@Outgoing`. In addition, you can use `@OnOverflow` to configure back-pressure strategy.

Kafka Health Check

If you are using the `quarkus-smallrye-health` extension, `quarkus-kafka` can add a readiness health check to validate the connection to the broker. This is disabled by default.

If enabled, when you access the `/health/ready` endpoint of your application you will have information about the connection validation status.

This behavior can be enabled by setting the `quarkus.kafka.health.enabled` property to `true` in your `application.properties`. You also need to point `quarkus.kafka.bootstrap-servers` to your Kafka cluster.

JSON serialization

Quarkus has built-in capabilities to deal with JSON Kafka messages.

Imagine we have a `Fruit` pojo as follows:

```java
public class Fruit {

    public String name;
    public int price;

    public Fruit() {
    }

    public Fruit(String name, int price) {
        this.name = name;
        this.price = price;
    }
}
```

And we want to use it to receive messages from Kafka, make some price transformation, and send messages back to Kafka.

```java
import io.smallrye.reactive.messaging.annotations.Broadcast;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;

import javax.enterprise.context.ApplicationScoped;

/**
 * A bean consuming data from the "fruit-in" Kafka topic and
applying some price conversion.
 * The result is pushed to the "fruit-out" stream.
 */
@ApplicationScoped
public class FruitProcessor {

    private static final double CONVERSION_RATE = 0.88;

    @Incoming("fruit-in")
    @Outgoing("fruit-out")
    @Broadcast
    public double process(Fruit fruit) {
        fruit.price = fruit.price * CONVERSION_RATE;
        return fruit;
    }

}
```

To do this, we will need to setup JSON serialization with JSON-B or Jackson.

> With JSON serialization correctly configured, you can also use `Publisher<Fruit>` and `Emitter<Fruit>`.

Serializing via JSON-B

First, you need to include the `quarkus-resteasy-jsonb` extension (if you already use the `quarkus-resteasy-jsonb` extension, this is not needed).

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jsonb</artifactId>
</dependency>
```

There is an existing `JsonbSerializer` that can be used to serialize all pojos via JSON-B, but the corresponding deserializer is generic, so it needs to be subclassed.

So, let's create a `FruitDeserializer` that extends the generic `JsonbDeserializer`.

```java
package com.acme.fruit.jsonb;

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

public class FruitDeserializer extends JsonbDeserializer<Fruit> {
    public FruitDeserializer(){
        // pass the class to the parent.
        super(Fruit.class);
    }
}
```

> If you don't want to create a deserializer for each of your pojo, you can use the generic `io.vertx.kafka.client.serialization.JsonObjectDeserializer` that will deserialize to a `javax.json.JsonObject`. The corresponding serializer can also be used: `io.vertx.kafka.client.serialization.JsonObjectSerializer`.

Finally, configure your streams to use the JSON-B serializer and deserializer.

```
# Configure the Kafka source (we read from it)
mp.messaging.incoming.fruit-in.connector=smallrye-kafka
mp.messaging.incoming.fruit-in.topic=fruit-in
mp.messaging.incoming.fruit-in.value.deserializer=com.acme.fruit.jsonb.FruitDeserializer

# Configure the Kafka sink (we write to it)
mp.messaging.outgoing.fruit-out.connector=smallrye-kafka
mp.messaging.outgoing.fruit-out.topic=fruit-out
mp.messaging.outgoing.fruit-out.value.serializer=io.quarkus.kafka.client.serialization.JsonbSerializer
```

Now, your Kafka messages will contain a JSON-B serialized representation of your Fruit pojo.

Serializing via Jackson

First, you need to include the `quarkus-resteasy-jackson` extension (if you already use the `quarkus-jackson-jsonb` extension, this is not needed).

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jackson</artifactId>
</dependency>
```

There is an existing `ObjectMapperSerializer` that can be used to serialize all pojos via Jackson, but the corresponding deserializer is generic, so it needs to be subclassed.

So, let's create a `FruitDeserializer` that extends the `ObjectMapperDeserializer`.

```java
package com.acme.fruit.jackson;

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

public class FruitDeserializer extends ObjectMapperDeserializer<Fruit> {
    public FruitDeserializer(){
        // pass the class to the parent.
        super(Fruit.class);
    }
}
```

Finally, configure your streams to use the Jackson serializer and deserializer.

```
# Configure the Kafka source (we read from it)
mp.messaging.incoming.fruit-in.connector=smallrye-kafka
mp.messaging.incoming.fruit-in.topic=fruit-in
mp.messaging.incoming.fruit-in.value.deserializer=com.acme.fruit.jackson.FruitDeserializer

# Configure the Kafka sink (we write to it)
mp.messaging.outgoing.fruit-out.connector=smallrye-kafka
mp.messaging.outgoing.fruit-out.topic=fruit-out
mp.messaging.outgoing.fruit-out.value.serializer=io.quarkus.kafka.client.serialization.ObjectMapperSerializer
```

Now, your Kafka messages will contain a Jackson serialized representation of your Fruit pojo.

Sending JSON Server-Sent Events (SSE)

If you want RESTEasy to send JSON Server-Sent Events, you need to use the `@SseElementType` annotation to define the content type of the events, as the method will be annotated with `@Produces(MediaType.SERVER_SENT_EVENTS)`.

The following example shows how to use SSE from a Kafka topic source.

```java
import io.smallrye.reactive.messaging.annotations.Channel;
import org.reactivestreams.Publisher;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.jboss.resteasy.annotations.SseElementType;

@Path("/fruits")
public class PriceResource {

    @Inject
    @Channel("fruit-out") Publisher<Fruit> fruits;

    @GET
    @Path("/stream")
    @Produces(MediaType.SERVER_SENT_EVENTS)
    @SseElementType(MediaType.APPLICATION_JSON)
    public Publisher<Fruit> stream() {
        return fruits;
    }
}
```

Going further

This guide has shown how you can interact with Kafka using Quarkus. It utilizes MicroProfile Reactive Messaging to build data streaming applications.

If you want to go further check the documentation of SmallRye Reactive Messaging, the implementation used in Quarkus.