

Quarkus - Using AMQP with Reactive Messaging

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



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

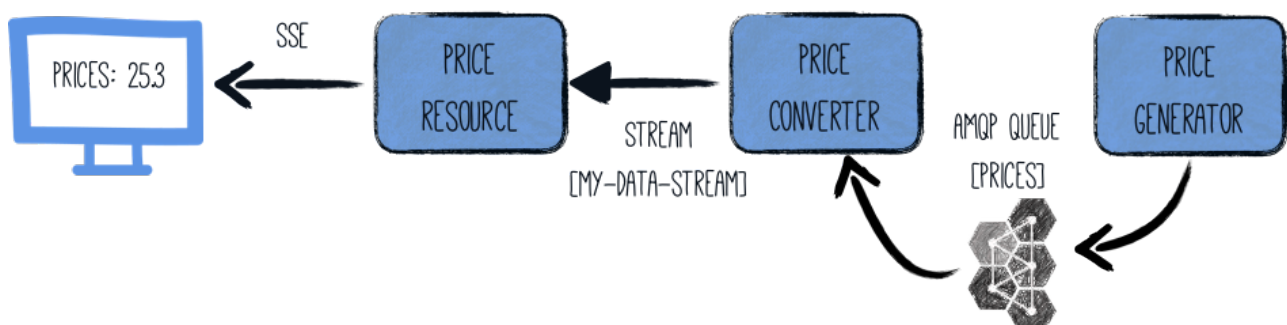
Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- A running AMQP 1.0 broker, 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 an AMQP queue (`prices`). A second component reads from the `prices` queue 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 `amqp-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:1.7.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=amqp-quickstart \
  -Dextensions="amqp"
cd amqp-quickstart
```

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

Starting an AMQP broker

Then, we need an AMQP broker. You can follow the instructions from the [Apache Artemis web site](#) or create a `docker-compose.yml` file with the following content:

```
# A docker compose file to start an Artemis AMQP broker
# more details on https://github.com/vromero/activemq-artemis-docker.
version: '2'

services:

  artemis:
    image: vromero/activemq-artemis:2.8.0-alpine
    ports:
      - "8161:8161"
      - "61616:61616"
      - "5672:5672"
    environment:
      ARTEMIS_USERNAME: quarkus
      ARTEMIS_PASSWORD: quarkus
```

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/amqp/PriceGenerator.java` file, with the following content:

```
package org.acme.amqp;

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 an AMQP queue (prices). The AMQP
 * configuration is specified in the
 * application configuration.
 */
@ApplicationScoped
public class PriceGenerator {

    private Random random = new Random();

    @Outgoing("generated-price")
    public Flowable<Integer> generate() {
        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 an AMQP queue using the `application.properties` file that we will create soon.

The price converter

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

```
package org.acme.amqp;

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" AMQP queue 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` channel
- ② Indicates that the objects returned by the method are sent to the `my-data-stream` channel
- ③ Indicates that the item are dispatched to all *subscribers*

The `process` method is called for every AMQP messages from the `prices` queue (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/amqp/PriceResource.java` file with the following content:

```

package org.acme.amqp;

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;

/**
 * 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
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }

    @GET
    @Path("/stream")
    @Produces(MediaType.SERVER_SENT_EVENTS) ②
    public Publisher<Double> stream() { ③
        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 Returns the stream (*Reactive Stream*)

Configuring the AMQP connector

We need to configure the AMQP 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

```
# Configures the AMQP broker credentials.
amqp-username=quarkus
amqp-password=quarkus

# Configure the AMQP connector to write to the `prices` address
mp.messaging.outgoing.generated-price.connector=smallrye-amqp
mp.messaging.outgoing.generated-price.address=prices

# Configure the AMQP connector to read from the `prices` queue
mp.messaging.incoming.prices.connector=smallrye-amqp
mp.messaging.incoming.prices.durable=true
```

More details about this configuration is available in the [SmallRye Reactive Messaging AMQP connector](#) documentation.



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:

```

<!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/
patternfly.min.css">
  <link rel="stylesheet" type="text/css"

href="https://cdnjs.cloudflare.com/ajax/libs/patternfly/3.24.0/css/
patternfly-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>&nbsp;&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 the AMQP broker running. Then, you just need to run the application using:

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

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



If you started the AMQP 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 **@Outgoing** because your method has parameters.

For this, you can use an **Emitter**.

```
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.eclipse.microprofile.reactive.messaging.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);
    }
}
```



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 a back-pressure strategy.

Deprecation

The `io.smallrye.reactive.messaging.annotations.Emitter`, `io.smallrye.reactive.messaging.annotations.Channel` and `io.smallrye.reactive.messaging.annotations.OnOverflow` classes are now deprecated and replaced by:



- `org.eclipse.microprofile.reactive.messaging.Emitter`
- `org.eclipse.microprofile.reactive.messaging.Channel`
- `org.eclipse.microprofile.reactive.messaging.OnOverflow`

The new `Emitter.send` method returns a `CompletionStage` completed when the produced message is acknowledged.

Going further

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

If you did the Kafka quickstart, you have realized that it's the same code. The only difference is the connector configuration.

If you want to go further check the documentation of [SmallRye Reactive Messaging](#), the implementation used in Quarkus.