

Quarkus - Using JMS

This guide demonstrates how your Quarkus application can use JMS messaging via the Apache Qpid JMS AMQP client, or alternatively the Apache ActiveMQ Artemis JMS client.



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 Artemis server, or Docker to start one
- GraalVM, or Docker, 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 to a queue (`prices`) using a JMS client. Another component reads from the `prices` queue and stores the latest price. The data can be fetched from a browser using a fetch button from a JAX-RS resource.

The guide can be used either via the Apache Qpid JMS AMQP client as detailed immediately below, or alternatively with the Apache ActiveMQ Artemis JMS client given some different configuration as [detailed later](#).

Qpid JMS - AMQP

In the detailed steps below we will use the [Apache Qpid JMS](#) client via the [Quarkus Qpid JMS extension](#). Qpid JMS uses the AMQP 1.0 ISO standard as its wire protocol, allowing it to be used with a variety of AMQP 1.0 servers and services such as ActiveMQ Artemis, ActiveMQ 5, Qpid Broker-J, Qpid Dispatch router, Azure Service Bus, and more.

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/amqphub/quarkus-qpid-jms-quickstart.git`, or download an [archive](#).

Creating the Maven Project

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

```
mvn io.quarkus:quarkus-maven-plugin:1.3.2.Final:create \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=jms-quickstart \
  -Dextensions="qpid-jms"
cd jms-quickstart
```

This command generates a Maven project, with its `pom.xml` importing the `quarkus-qpid-jms` extension:

```
<dependency>
  <groupId>org.amqphub.quarkus</groupId>
  <artifactId>quarkus-qpid-jms</artifactId>
</dependency>
```

Starting the broker

Then, we need an AMQP broker. In this case we will use an ActiveMQ Artemis server. You can follow the instructions from the [Apache Artemis web site](#) or start a broker via docker:

```
docker run -it --rm -p 8161:8161 -p 61616:61616 -p 5672:5672 -e
ARTEMIS_USERNAME=quarkus -e ARTEMIS_PASSWORD=quarkus
vromero/activemq-artemis:2.11.0-alpine
```

The price producer

Create the `src/main/java/org/acme/jms/PriceProducer.java` file, with the following content:

```

package org.acme.jms;

import java.util.Random;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.inject.Inject;
import javax.jms.ConnectionFactory;
import javax.jms.JMSContext;
import javax.jms.Session;

import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;

/**
 * A bean producing random prices every 5 seconds and sending them
 * to the prices JMS queue.
 */
@ApplicationScoped
public class PriceProducer implements Runnable {

    @Inject
    ConnectionFactory connectionFactory;

    private final Random random = new Random();
    private final ScheduledExecutorService scheduler = Executors
        .newSingleThreadScheduledExecutor();

    void onStart(@Observes StartupEvent ev) {
        scheduler.scheduleWithFixedDelay(this, 0L, 5L, TimeUnit
            .SECONDS);
    }

    void onStop(@Observes ShutdownEvent ev) {
        scheduler.shutdown();
    }

    @Override
    public void run() {
        try (JMSContext context = connectionFactory.createContext(
            Session.AUTO_ACKNOWLEDGE)) {
            context.createProducer().send(context.createQueue(
                "prices"), Integer.toString(random.nextInt(100)));
        }
    }
}

```

The price consumer

The price consumer reads the prices from JMS, and stores the last one. Create the `src/main/java/org/acme/jms/PriceConsumer.java` file with the following content:

```
package org.acme.jms;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.inject.Inject;
import javax.jms.ConnectionFactory;
import javax.jms.JMSConsumer;
import javax.jms.JMSContext;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;

/**
 * A bean consuming prices from the JMS queue.
 */
@ApplicationScoped
public class PriceConsumer implements Runnable {

    @Inject
    ConnectionFactory connectionFactory;

    private final ExecutorService scheduler = Executors
        .newSingleThreadExecutor();

    private volatile String lastPrice;

    public String getLastPrice() {
        return lastPrice;
    }

    void onStart(@Observes StartupEvent ev) {
        scheduler.submit(this);
    }

    void onStop(@Observes ShutdownEvent ev) {
        scheduler.shutdown();
    }
}
```

```

@Override
public void run() {
    try (JMSContext context = connectionFactory.createContext(
        Session.AUTO_ACKNOWLEDGE)) {
        JMSConsumer consumer = context.createConsumer(context
            .createQueue("prices"));
        while (true) {
            Message message = consumer.receive();
            if (message == null) return;
            lastPrice = message.getBody(String.class);
        }
    } catch (JMSException e) {
        throw new RuntimeException(e);
    }
}
}

```

The price resource

Finally, let's create a simple JAX-RS resource to show the last price. Create the `src/main/java/org/acme/jms/PriceResource.java` file with the following content:

```

package org.acme.jms;

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 showing the last price.
 */
@Path("/prices")
public class PriceResource {

    @Inject
    PriceConsumer consumer;

    @GET
    @Path("last")
    @Produces(MediaType.TEXT_PLAIN)
    public String last() {
        return consumer.getLastPrice();
    }
}

```

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/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"><button id="fetch">Fetch</button>The last
price is <strong><span id="content">N/A</span>&nbsp;&euro;
</strong>.</p>
  </div>
</div>
</body>
<script>
  document.getElementById("fetch").addEventListener("click",
function() {
    fetch("/prices/last").then(function (response) {
      response.text().then(function (text) {
        document.getElementById("content").textContent =
text;
      })
    })
  })
</script>
</html>
```

Nothing spectacular here. On each fetch, it updates the page.

Configure the Qpid JMS properties

We need to configure the Qpid JMS properties used by the extension when injecting the `ConnectionFactory`.

This is done in the `src/main/resources/application.properties` file.

```
# Configures the Qpid JMS properties.
quarkus.qpid-jms.url=amqp://localhost:5672
quarkus.qpid-jms.username=quarkus
quarkus.qpid-jms.password=quarkus
```

More detail about the configuration are available in the [Quarkus Qpid JMS](#) documentation.

Get it running

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

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

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

Running Native

You can build the native executable with:

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

Or, if you don't have GraalVM installed, you can instead use Docker to build the native executable using:

```
./mvnw package -Pnative -Dquarkus.native.container-build=true
```

and then run with:

```
./target/jms-quickstart-1.0-SNAPSHOT-runner
```

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

Artemis JMS

The above steps detailed using the Qpid JMS AMQP client, however the guide can also be used with the Artemis JMS client. Many of the individual steps are exactly as previously [detailed above for Qpid JMS](#). The individual component code is the same. The only differences are in the dependency for the initial project creation, and the configuration properties used. These changes are detailed below and should be substituted for the equivalent step during the sequence above.

Solution

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 Artemis JMS solution is located in the `jms-quickstart` directory.

Creating the Maven Project

Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.2.Final:create \
    -DprojectId=org.acme \
    -DprojectArtifactId=jms-quickstart \
    -Dextensions="artemis-jms"
cd jms-quickstart
```

This creates a Maven project, with its `pom.xml` importing the `quarkus-artemis-jms` extension:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-artemis-jms</artifactId>
</dependency>
```

With the project created, you can resume from [Starting the broker](#) in the detailed steps above and proceed until configuring the `application.properties` file, when you should use the Artemis configuration below instead.


Configure the Artemis properties

We need to configure the Artemis connection properties. This is done in the `src/main/resources/application.properties` file.


```
# Configures the Artemis properties.
quarkus.artemis.url=tcp://localhost:61616
quarkus.artemis.username=quarkus
quarkus.artemis.password=quarkus
```

With the Artemis properties configured, you can resume the steps above from [Get it running](#).

Configuration Reference

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

Configuration property	Type	Default
 <code>quarkus.artemis.health.enabled</code> Whether or not an health check is published in case the smallrye-health extension is present	boolean	<code>true</code>
<code>quarkus.artemis.url</code> Artemis connection url	string	required 
<code>quarkus.artemis.username</code> Username for authentication, only used with JMS	string	
<code>quarkus.artemis.password</code> Password for authentication, only used with JMS	string	