Quarkus - Creating Your First Application

Learn how to create a Hello World Quarkus app. This guide covers:

- Bootstrapping an application

- Creating a JAX-RS endpoint

- Injecting beans

- Functional tests

- Packaging of the application

1. Prerequisites

To complete this guide, you need:

- less than 15 minutes

- an IDE

- JDK 8 or 11+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.6.3

💡 *Verify Maven is using the Java you expect*

If you have multiple JDK's installed it is not certain Maven will pick up the expected java and you could end up with unexpected results. You can verify which JDK Maven uses by running `mvn --version`.

2. Architecture

In this guide, we create a straightforward application serving a `hello` endpoint. To demonstrate dependency injection, this endpoint uses a `greeting` bean.

This guide also covers the testing of the endpoint.

3. Solution

We recommend that you follow the instructions from Boostrapping project and onwards to create the application step by step.

However, you can go right to the completed example.

Download an archive or clone the git repository:

```
git clone https://github.com/quarkusio/quarkus-quickstarts.git
```

The solution is located in the `getting-started` directory.

4. Bootstrapping the project

The easiest way to create a new Quarkus project is to open a terminal and run the following command:

For Linux & MacOS users

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=getting-started \
    -DclassName="org.acme.getting.started.GreetingResource" \
    -Dpath="/hello"
cd getting-started
```

For Windows users

- If using cmd , (don't use forward slash `\`)

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.Final:create
-DprojectGroupId=org.acme -DprojectArtifactId=getting-started
-DclassName="org.acme.getting.started.GreetingResource"
-Dpath="/hello"
```

- If using Powershell , wrap `-D` parameters in double quotes

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.Final:create "-
DprojectGroupId=org.acme" "-DprojectArtifactId=getting-started" "-
DclassName=org.acme.getting.started.GreetingResource" "-
Dpath=/hello"
```

It generates the following in `./getting-started`:

- the Maven structure

- an `org.acme.getting.started.GreetingResource` resource exposed on `/hello`

- an associated unit test

- a landing page that is accessible on `http://localhost:8080` after starting the application

- example `Dockerfile` files for both `native` and `jvm` modes in `src/main/docker`

- the application configuration file

Once generated, look at the `pom.xml`. You will find the import of the Quarkus BOM, allowing you to omit the version on the different Quarkus dependencies. In addition, you can see the `quarkus-maven-plugin` responsible of the packaging of the application and also providing the development mode.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-bom</artifactId>
            <version>${quarkus.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-maven-plugin</artifactId>
            <version>${quarkus.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>build</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

If we focus on the dependencies section, you can see the extension allowing the development of REST applications:

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

4.1. The JAX-RS resources

During the project creation, the src/main/java/org/acme/getting/started/GreetingResource.java file has been created with the following content:

```
package org.acme.getting.started;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

It's a very simple REST endpoint, returning "hello" to requests on "/hello".

Differences with vanilla JAX-RS

With Quarkus, there is no need to create an `Application` class. It's supported, but not required. In addition, only one instance of the resource is created and not one per request. You can configure this using the different `*Scoped` annotations (`ApplicationScoped`, `RequestScoped`, etc).

5. Running the application

Now we are ready to run our application. Use: `./mvnw compile quarkus:dev`:

```
$ ./mvnw compile quarkus:dev
[INFO] --------------------< org.acme:getting-started
>--------------------
[INFO] Building getting-started 1.0-SNAPSHOT
[INFO] ----------------------------[ jar
]----------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources)
@ getting-started ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory
/Users/starksm/Dev/JBoss/Quarkus/starksm64-quarkus-
quickstarts/getting-started/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @
getting-started ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to
/Users/starksm/Dev/JBoss/Quarkus/starksm64-quarkus-
quickstarts/getting-started/target/classes
[INFO]
[INFO] --- quarkus-maven-plugin:<version>:dev (default-cli) @
getting-started ---
Listening for transport dt_socket at address: 5005
2019-02-28 17:05:22,347 INFO  [io.qua.dep.QuarkusAugmentor] (main)
Beginning quarkus augmentation
2019-02-28 17:05:22,635 INFO  [io.qua.dep.QuarkusAugmentor] (main)
Quarkus augmentation completed in 288ms
2019-02-28 17:05:22,770 INFO  [io.quarkus] (main) Quarkus started
in 0.668s. Listening on: http://localhost:8080
2019-02-28 17:05:22,771 INFO  [io.quarkus] (main) Installed
features: [cdi, resteasy]
```

Once started, you can request the provided endpoint:

```
$ curl -w "\n" http://localhost:8080/hello
hello
```

Hit CTRL+C to stop the application, or keep it running and enjoy the blazing fast hot-reload.

Automatically add newline with `curl -w "\n"`

We are using `curl -w "\n"` in this example to avoid your terminal printing a '%' or put both result and next command prompt on the same line.

6. Using injection

Dependency injection in Quarkus is based on ArC which is a CDI-based dependency injection solution tailored for Quarkus' architecture. You can learn more about it in the Contexts and Dependency Injection guide.

ArC comes as a dependency of `quarkus-resteasy` so you already have it handy.

Let's modify the application and add a companion bean. Create the `src/main/java/org/acme/getting/started/GreetingService.java` file with the following content:

```
package org.acme.getting.started;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class GreetingService {

    public String greeting(String name) {
        return "hello " + name;
    }

}
```

Edit the `GreetingResource` class to inject the `GreetingService` and create a new endpoint using it:

```
package org.acme.getting.started;

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.jaxrs.PathParam;

@Path("/hello")
public class GreetingResource {

    @Inject
    GreetingService service;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    public String greeting(@PathParam String name) {
        return service.greeting(name);
    }

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

If you stopped the application, restart the application with `./mvnw compile quarkus:dev`. Then check that the endpoint returns `hello quarkus` as expected:

```
$ curl -w "\n" http://localhost:8080/hello/greeting/quarkus
hello quarkus
```

7. Development Mode

`quarkus:dev` runs Quarkus in development mode. This enables hot deployment with background compilation, which means that when you modify your Java files and/or your resource files and refresh your browser, these changes will automatically take effect. This works too for resource files like the configuration property file. Refreshing the browser triggers a scan of the workspace, and if any changes are detected, the Java files are recompiled and the application is redeployed; your request is then serviced by the redeployed application. If there are any issues with compilation or deployment an error page will let you know.

This will also listen for a debugger on port 5005. If you want to wait for the debugger to attach before running you can pass `-Dsuspend` on the command line. If you don't want the debugger at all you can use `-Ddebug=false`.

8. Testing

All right, so far so good, but wouldn't it be better with a few tests, just in case.

In the generated `pom.xml` file, you can see 2 test dependencies:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

Quarkus supports Junit 5 tests. Because of this, the version of the Surefire Maven Plugin must be set, as the default version does not support Junit 5:

```xml
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${surefire-plugin.version}</version>
    <configuration>
       <systemPropertyVariables>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
            <maven.home>${maven.home}</maven.home>
       </systemPropertyVariables>
    </configuration>
</plugin>
```

We also set the `java.util.logging` system property to make sure tests will use the correct logmanager and `maven.home` to ensure that custom configuration from `${maven.home}/conf/settings.xml` is applied (if any).

The generated project contains a simple test. Edit the `src/test/java/org/acme/getting/started/GreetingResourceTest.java` to match the following content:

```
package org.acme.getting.started;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import java.util.UUID;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class GreetingResourceTest {

    @Test        ①
    public void testHelloEndpoint() {
        given()
          .when().get("/hello")
          .then()
             .statusCode(200)     ②
             .body(is("hello"));
    }

    @Test
    public void testGreetingEndpoint() {
        String uuid = UUID.randomUUID().toString();
        given()
          .pathParam("name", uuid)
          .when().get("/hello/greeting/{name}")
          .then()
            .statusCode(200)
            .body(is("hello " + uuid));
    }

}
```

① By using the `QuarkusTest` runner, you instruct JUnit to start the application before the tests.

② Check the HTTP response status code and content

These tests use RestAssured, but feel free to use your favorite library.

You can run these using Maven:

```
./mvnw test
```

You can also run the test from your IDE directly (be sure you stopped the application first).

By default, tests will run on port 8081 so as not to conflict with the running application. We

automatically configure RestAssured to use this port. If you want to use a different client you should use the `@TestHTTPResource` annotation to directly inject the URL of the tested application into a field on the test class. This field can be of the type `String`, `URL` or `URI`. This annotation can also be given a value for the test path. For example, if I want to test a Servlet mapped to `/myservlet` I would just add the following to my test:

```
@TestHTTPResource("/myservlet")
URL testUrl;
```

The test port can be controlled via the `quarkus.http.test-port` config property. Quarkus also creates a system property called `test.url` that is set to the base test URL for situations where you cannot use injection.

9. Working with multi-module project or external modules

Quarkus heavily utilizes Jandex at build time, to discover various classes or annotations. One immediately recongizable application of this, is CDI bean discovery. As a result, most of the Quarkus extensions will not work properly if this build time discovery isn't properly setup.

This index is created by default on the project on which Quarkus is configured for, thanks to our Maven and Gradle plugins.

However, when working with a multi-module project, be sure to read the `Working with multi-module projects` section of the Maven or Gradle guides.

If you plan to use external modules (for example, an external library for all your domain objects), you will need to make these modules known to the indexing process either by adding the Jandex plugin (if you can modify them) or via the `quarkus.index-dependency` property inside your `application.properties` (useful in cases where you can't modify the module).

Be sure to read the Bean Discovery section of the CDI guide for more information.

10. Packaging and run the application

The application is packaged using `./mvnw package`. It produces 2 jar files in `/target`:

- `getting-started-1.0-SNAPSHOT.jar` - containing just the classes and resources of the projects, it's the regular artifact produced by the Maven build;

- `getting-started-1.0-SNAPSHOT-runner.jar` - being an executable *jar*. Be aware that it's not an *über-jar* as the dependencies are copied into the `target/lib` directory.

You can run the application using: `java -jar target/getting-started-1.0-SNAPSHOT-runner.jar`

> **i** The `Class-Path` entry of the `MANIFEST.MF` from the *runner jar* explicitly lists the jars from the `lib` directory. So if you want to deploy your application somewhere, you need to copy the *runner* jar as well as the *lib* directory.

> **i** Before running the application, don't forget to stop the hot reload mode (hit `CTRL+C`), or you will have a port conflict.

11. Configuring the banner

By default when a Quarkus application starts (in regular or dev mode), it will display an ASCII art banner. The banner can be disabled by setting `quarkus.banner.enabled=false` in `application.properties`, by setting the `-Dquarkus.banner.enabled=false` Java System Property, or by setting the `QUARKUS_BANNER_ENABLED` environment variable to `false`. Furthermore, users can supply a custom banner by placing the banner file in `src/main/resources` and configuring `quarkus.banner.path=name-of-file` in `application.properties`.

12. What's next?

This guide covered the creation of an application using Quarkus. However, there is much more. We recommend continuing the journey with the building a native executable guide, where you learn about creating a native executable and packaging it in a container. If you are interested in reactive, we recommend the Getting started with reactive guide, where you can see how to implement reactive applications with Quarkus.

In addition, the tooling guide document explains how to:

- scaffold a project in a single command line

- enable the *development mode* (hot reload)

- import the project in your favorite IDE

- and more