Quarkus - Building my first extension

Quarkus extensions enhance your application just as projects dependencies do. The role of the extensions is to leverage Quarkus paradigms to integrate seamlessly a library into Quarkus architecture - e.g. do more things at build time. This is how you can use your battle-tested ecosystem and take advantage of Quarkus performance and native compilation. Go to code.quarkus.io to get the list of the supported extensions.

In this guide we are going to develop the **Sample Greeting Extension**. The extension will expose a customizable HTTP endpoint which simply greets the visitor.

> *Disclaimer*
>
> To be sure it's extra clear you don't need and extension to add a Servlet to your application. This guide is a simplified example to explain the concepts of extensions development. Keep in mind it's not representative of the power of moving things to build time or simplifying the build of native images.

Prerequisites

To complete this guide, you need:

- less than 30 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.5.3+

Solution

We recommend that you follow the instructions in the next sections and create the extension 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 `getting-started-extension` directory.

Basic Concepts

First things first, we will need to start with some basic concepts.

- JVM mode vs Native mode

- Quarkus is first and foremost a Java framework, that means you can develop, package and run classic JAR applications, that's what we call **JVM mode.**

- Thanks to GraalVM you can compile your Java application into machine specific code (like you do in Go or C++) and that's what we call **Native mode.**

The operation of compiling Java bytecode into a native system-specific machine code is named **Ahead of Time Compilation** (aka AoT).

- build time vs runtime in classic Java frameworks

  - The build time corresponds to all the actions you apply to your Java source files to convert them into something runnable (class files, jar/war, native images). Usually this stage is composed by the compilation, annotation processing, bytecode generation, etc. At this point, everything is under developer's scope and control.

  - The runtime is all the actions that happen when you execute your application. It's obviously focused on starting your business-oriented actions but it relies on a lot of technical actions like loading libraries and configuration files, scanning the application's classpath, configuring the dependency injection, setting up your Object-Relational Mapping, instantiating your REST controllers, etc.

Usually, Java frameworks do their bootstrapping during the runtime before actually starting the application "Business oriented layer". During bootstrap, frameworks dynamically collect metadata by scanning the classpath to find configurations, entity definitions, dependency injection binding, etc. in order to instantiate proper objects through reflection. The main consequences are:

- Delaying the readiness of your application: you need to wait a couple of seconds before actually serving a business request.

- Having a peak of resource consumption at bootstrap: In a constrained environment, you will need to size the needed resources based on your technical boostrap need rather than your actual business needs.

Quarkus' philosophy is to prevent as much as possible slow and memory intensive dynamic code execution by shifting left these actions and eventually do them during the build time. A Quarkus extension is a Java piece of code acting as an adapter layer for your favorite libary or technology.

Description of a Quarkus extension

A Quarkus extension consists of two parts:

- Runtime module : Represent the capabilities the extension developer exposes to the application's developer (an authentication filter, an enhanced data layer API, etc). Runtime dependencies are the ones the users will add into their POMs.

- Deployment module: Used during the augmentation phase of the build, it describes how to "deploy" a library following the Quarkus philosophy. In other words, it applies all the Quarkus optimizations to your application during the build. The deployment module is also where we prepare things for GraalVM's native compilation.

At this point, you should have understood that most of the magic will happen at the Augmentation

build time thanks to the deployment module.

Quarkus Application Bootstrap

There are three distinct bootstrap phases of a Quarkus application.

- Augmentation: During the build time, the Quarkus extensions will load and scan your application's bytecode (including the dependencies) and configuration. At this stage, the extension can read configuration files, scan classes for specific annotations, etc. Once all the metadata collected, the extensions can pre-process the libraries bootstrap actions like your ORM, DI or REST controllers configurations. The result of the bootstrap is directly recorded into bytecode and will be part of your final application package.

- Static Init: During the run time, Quarkus will execute first a static init method which contains some extensions actions/configurations. When you will do your native packaging, this static method will be pre-processed during the build time and the eventual generated objects will be serialized in the final native executable, so this code will not be executed in a native mode (Imagine you execute a Fibonacci function during this phase, the result of the computation will be directly recorded in the native executable). When running the application in JVM mode, this static init phase is executed at the start of the application.

- Runtime Init: Well nothing fancy here, we do classic run time code execution. So, the more code you run during the 2 phases above, the faster your application will start.

Now that everything is explained, we can start coding!!

Maven setup

Quarkus provides a `create-extension` Maven Mojo to initialize your extension. The team is working hard to improve the dev experience but at the moment, you must manually create a root `pom.xml` before invoking the Mojo.

```
$ mkdir getting-started-extension && cd getting-started-extension
$ cat << EOF > pom.xml
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.acme</groupId>
    <artifactId>getting-started-extension</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <properties>
        <quarkus.version>{quarkus-version}</quarkus.version>
        <project.build.sourceEncoding>UTF-
```

```
8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-platform-bom-
deployment</artifactId> ①
                <version>\${quarkus.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId> ②
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>${maven.compiler.version}</version>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

</project>
EOF
```

① The `quarkus-universe-bom-deployment` BOM gives access to all the existing Quarkus extensions (runtime and deployment modules).

② Quarkus requires a recent version of the Maven compiler plugin supporting the `annotationProcessorPaths` configuration.

Now you can invoke `quarkus-maven-plugin:create-extension` MOJO to initialize your extension.

```
$ mvn io.quarkus:quarkus-maven-plugin:{quarkus-version}:create-
extension -N \
    -Dquarkus.artifactIdBase=greeting \
    -Dquarkus.artifactIdPrefix=quarkus- \
    -Dquarkus.nameBase="Greeting Extension"
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------< org.acme:getting-started-extension
>----------------
[INFO] Building getting-started-extension 1.0-SNAPSHOT
[INFO] ------------------------------[ pom
]------------------------------
[INFO]
[INFO] --- quarkus-maven-plugin:{quarkus-version}:create-extension
(default-cli) @ getting-started-extension ---
[INFO] Adding module [greeting] to [/tmp/quarkus-
quickstarts/getting-started-extension/pom.xml]
[INFO]
------------------------------------------------------------------
-----
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------
-----
[INFO] Total time:  1.324 s
[INFO] Finished at: 2020-01-26T16:57:32+01:00
[INFO]
------------------------------------------------------------------
-----
```

Maven has generated a `greeting` directory and added the freshly create module into the current `getting-started-extension/pom.xml`.

```
<modules>
    <module>greeting</module>
</modules>
```

Maven has generated couple of files inside `getting-started-extension/greeting`, let's see together what their role is.

The parent module

Your extension is a multi-module project. So let's start by the parent `./greeting/pom.xml`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.acme</groupId>
        <artifactId>getting-started-extension</artifactId>
        <version>1.0-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>

    <artifactId>quarkus-greeting-parent</artifactId>
    <name>Greeting Extension - Parent</name>

    <packaging>pom</packaging>
    <modules>  ①
        <module>deployment</module>
        <module>runtime</module>
    </modules>
</project>
```

① Your extension declares 2 sub-modules `deployment` and `runtime`.

The Deployment module

Let's have a look at the deployment's `./greeting/deployment/pom.xml`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.acme</groupId>
        <artifactId>quarkus-greeting-parent</artifactId>
        <version>1.0-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>

    <artifactId>quarkus-greeting-deployment</artifactId>  ①
    <name>Greeting Extension - Deployment</name>

    <dependencies>
        <dependency>
```

```xml
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-core-deployment</artifactId>  ②
                <version>${quarkus.version}</version>
            </dependency>
            <dependency>
                <groupId>org.acme</groupId>
                <artifactId>quarkus-greeting</artifactId>
                <version>${project.version}</version>
            </dependency>
        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <annotationProcessorPaths>
                            <path>
                                <groupId>io.quarkus</groupId>
                                <artifactId>quarkus-extension-
processor</artifactId>  ③
                                <version>${quarkus.version}</version>
                            </path>
                        </annotationProcessorPaths>
                    </configuration>
                </plugin>
            </plugins>
        </build>

</project>
```

The key points are:

① By convention, the deployment module has the `-deployment` suffix (`greeting-deployment`).

② The deployment module depends on the `quarkus-core-deployment` artifact. We will see later which dependencies are convenient to add.

③ We add the `quarkus-extension-processor` to the compiler annotation processors.

In addition of the `pom.xml` Maven also created the `org.acme.quarkus.greeting.deployment.GreetingProcessor` class.

```java
package org.acme.quarkus.greeting.deployment;

import io.quarkus.deployment.annotations.BuildStep;
import io.quarkus.deployment.builditem.FeatureBuildItem;

class GreetingProcessor {

    private static final String FEATURE = "greeting";

    @BuildStep
    FeatureBuildItem feature() {
        return new FeatureBuildItem(FEATURE);
    }

}
```

Be patient, we will explain the `Build Step Processor` concept and all the extension deployment API later on. At this point, you just need to understand that this class explains to Quarkus how to deploy a feature named `greeting` which is your extension. In other words, you are augmenting your application to use the `greeting` extension with all the Quarkus benefits (build time optimization, native support, etc.).

The Runtime module

Finally `./greeting/runtime/pom.xml`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.acme</groupId>
        <artifactId>quarkus-greeting-parent</artifactId>
        <version>1.0-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>

    <artifactId>quarkus-greeting</artifactId>   ①
    <name>Greeting Extension - Runtime</name>

    <build>
        <plugins>
            <plugin>
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-bootstrap-maven-
```

```
plugin</artifactId>   ②
                <version>${quarkus.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>extension-descriptor</goal>
                        </goals>
                        <phase>compile</phase>
                        <configuration>

<deployment>${project.groupId}:${project.artifactId}-
deployment:${project.version}
                        </deployment>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>io.quarkus</groupId>
                        <artifactId>quarkus-extension-
processor</artifactId>  ③

                        <version>${quarkus.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

The key points are:

① By convention, the runtime module has no suffix (`greeting`) as it is the artifact exposed to the end user.

② We add the `quarkus-bootstrap-maven-plugin` to generate the Quarkus extension descriptor included into the runtime artifact which links it with the corresponding deployment artifact.

③ We add the `quarkus-extension-processor` to the compiler annotation processors.

Basic version of the Sample Greeting extension

Implementing the Greeting feature

The (killer) feature proposed by our extension is to greet the user. To do so, our extension will deploy, in the user application, a Servlet exposing the HTTP endpoint `/greeting` which responds to the GET verb with a plain text `Hello`.

The `runtime` module is where you develop the feature you want to propose to your users, so it's time to create our Web Servlet.

To use Servlets you need to have a Servlet Container such as Undertow. There is a Quarkus Undertow extension to use the web server with all the Quarkus benefits. We need to add the extension as a dependency to `./greeting/runtime/pom.xml`.

```xml
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-undertow</artifactId>
    </dependency>
</dependencies>
```

Now we can create our Servlet `org.acme.quarkus.greeting.GreetingServlet` in the `runtime` module.

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

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet
public class GreetingServlet extends HttpServlet { ①

    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws IOException { ②
        resp.getWriter().write("Hello");
    }
}
```

① As usual, defining a servlet requires to extend `javax.servlet.http.HttpServlet`.

② Since we want to respond to the HTTP GET verb, we override the `doGet` method and write `Hello` in the Servlet response's output stream.

Deploying the Greeting feature

Quarkus magic relies on bytecode generation at build time rather than waiting runtime code evaluation, that's the role of your extension's `deployment` module. Calm down, we know, bytecode is hard and you don't want to do it manually, Quarkus proposes a high level API to make your life easier. Thanks to basic concepts, you will describe the items to produce/consume and the corresponding steps in order to generate the bytecode to produce during the deployment time.

The `io.quarkus.builder.item.BuildItem` concept represents object instances you will produce or consume (and at some point convert into bytecode) thanks to methods annotated with `@io.quarkus.deployment.annotations.BuildStep` which describe your extension's deployment tasks.

Go back to the generated `org.acme.quarkus.greeting.deployment.GreetingProcessor` class.

```
package org.acme.quarkus.greeting.deployment;

import io.quarkus.deployment.annotations.BuildStep;
import io.quarkus.deployment.builditem.FeatureBuildItem;

class GreetingProcessor {

    private static final String FEATURE = "greeting";

    @BuildStep ①
    FeatureBuildItem feature() {
        return new FeatureBuildItem(FEATURE); ②
    }

}
```

① `feature()` method is annotated with `@BuildStep` which means it is identified as a deployment task Quarkus will have to execute during the deployment. `BuildStep` methods are run concurrently at augmentation time to augment the application. They use a producer/consumer model, where a step is guaranteed not to be run until all the items that it is consuming have been produced.

② `io.quarkus.deployment.builditem.FeatureBuildItem` is an implementation of `BuildItem` which represents the description of an extension. This `BuildItem` will be used by Quarkus to display information to the users when the application is starting.

There are many `BuildItem` implementations, each one represents an aspect of the deployment process. Here are some examples:

- `ServletBuildItem`: describes a Servlet (name, path, etc.) we want to generate during the deployment.

- `BeanContainerBuildItem`: describes a container used to store and retrieve object instances during the deployment.

If you don't find a `BuildItem` for what you want to achieve, you can create your own implementation. Keep in mind that a `BuildItem` should be as fine-grained as possible, representing a specific part of the deployment. To create your `BuildItem` you can extend:

- `io.quarkus.builder.item.SimpleBuildItem` if you need only a single instance of the item during the deployment (e.g. `BeanContainerBuildItem`, you only want one container).

- `io.quarkus.builder.item.MultiBuildItem` if you want to have multiple instances (e.g. `ServletBuildItem`, you can produce many Servlets during the deployment).

It's now time to declare our HTTP endpoint. To do so, we need to produce a `ServletBuildItem`. At this point, we are sure you understood that if the `quarkus-undertow` dependency proposes Servlet support for our `runtime` module, we will need the `quarkus-undertow-deployment` dependency into our `deployment` module to have access to the `io.quarkus.undertow.deployment.ServletBuildItem`.

Update the `./greeting/deployment/pom.xml` as follows:

```xml
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-core-deployment</artifactId>
        <version>${quarkus.version}</version>
    </dependency>
    <dependency>
        <groupId>org.acme</groupId>
        <artifactId>quarkus-greeting</artifactId>
        <version>${project.version}</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-undertow-deployment</artifactId>
    </dependency>
</dependencies>
```

We can now update `org.acme.quarkus.greeting.deployment.GreetingProcessor`:

```java
package org.acme.quarkus.greeting.deployment;

import io.quarkus.deployment.annotations.BuildStep;
import io.quarkus.deployment.builditem.FeatureBuildItem;
import org.acme.quarkus.greeting.GreetingServlet;
import io.quarkus.undertow.deployment.ServletBuildItem;

class GreetingProcessor {

    private static final String FEATURE = "greeting";

    @BuildStep
    FeatureBuildItem feature() {
        return new FeatureBuildItem(FEATURE);
    }

    @BuildStep
    ServletBuildItem createServlet() { ①
      ServletBuildItem servletBuildItem = ServletBuildItem.builder
("greeting", GreetingServlet.class.getName())
        .addMapping("/greeting")
        .build(); ②
      return servletBuildItem;
    }

}
```

① We add a `createServlet` method which returns a `ServletBuildItem` and annotate it with `@BuildStep`. Now, Quarkus will process this new task which will result in the bytecode generation of the Servlet registration at build time.

② `ServletBuildItem` proposes a fluent API to instantiate a Servlet named `greeting` of type `GreetingServlet` (it's our class provided by our extension `runtime` module), and map it the `/greeting` path.

Testing the Greeting feature

When developing a Quarkus extension, you mainly want to test your feature is properly deployed in an application and works as expected. That's why the tests will be hosted in the `deployment` module.

Let's add the testing dependencies into the `./greeting/deployment/pom.xml` and `maven-surefire` configuration

```xml
    <?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```xml
<modelVersion>4.0.0</modelVersion>
<parent>
    <groupId>org.acme</groupId>
    <artifactId>quarkus-greeting-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
</parent>

<artifactId>quarkus-greeting-deployment</artifactId> ①
<name>Greeting Extension - Deployment</name>

<properties>
    <maven.surefire.version>3.0.0-M4</maven.surefire.version>
</properties>

<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-arc-deployment</artifactId> ②
        <version>${quarkus.version}</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-core-deployment</artifactId> ②
        <version>${quarkus.version}</version>
    </dependency>
    <dependency>
        <groupId>org.acme</groupId>
        <artifactId>quarkus-greeting</artifactId>
        <version>${project.version}</version>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-undertow-deployment</artifactId>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-junit5-internal</artifactId> ①
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.rest-assured</groupId>
        <artifactId>rest-assured</artifactId> ②
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>io.quarkus</groupId>
                            <artifactId>quarkus-extension-
processor</artifactId>
                            <version>${quarkus.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId> ③
                <version>${maven.surefire.version}</version>
                <configuration>
                    <systemProperties>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.ut
il.logging.manager>
                    </systemProperties>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

① Quarkus proposes facilities to test extensions via the `quarkus-junit5-internal` artifact, in particular the `io.quarkus.test.QuarkusUnitTest` runner which starts an application with your extension.

② We will use RestAssured (massively used in Quarkus) to test our HTTP endpoint.

③ In order to not fallback to JUnit 4 legacy mode you need to define a recent version of `maven-surefire` plugin.

> **ℹ** As for now, the `create-extension` Maven Mojo does not create the test structure. You need to create it:

```
mkdir -p ./greeting/deployment/src/test/java/
```

To start testing your extension, create the following `org.acme.quarkus.greeting.deployment.GreetingTest` test class:

```java
package org.acme.quarkus.greeting.deployment;

import io.quarkus.test.QuarkusUnitTest;
import io.restassured.RestAssured;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

import static org.hamcrest.Matchers.containsString;

public class GreetingTest {

  @RegisterExtension
  static final QuarkusUnitTest config = new QuarkusUnitTest()
    .setArchiveProducer(() -> ShrinkWrap.create(JavaArchive.class)
); ①

  @Test
  public void testGreeting() {
    RestAssured.when().get("/greeting").then().statusCode(200).
body(containsString("Hello")); ②
  }

}
```

① We register a Junit Extension which will start a Quarkus application with the Greeting extension.

② We verify the application has a `greeting` endpoint responding to a HTTP GET request with a OK status (200) and a plain text body containing `Hello`

Time to test!!

```
$ mvn clean test
[INFO] Scanning for projects...
[INFO]
------------------------------------------------------------------------
-----
[INFO] Reactor Build Order:
[INFO]
[INFO] Greeting Extension - Parent
[pom]
[INFO] Greeting Extension - Runtime
[jar]
[INFO] Greeting Extension - Deployment
[jar]
[INFO]
...
```

```
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.acme.quarkus.greeting.deployment.GreetingTest
2020-01-28 22:40:54,913 INFO  [io.quarkus] (main) Quarkus
1.2.0.Final started in 0.289s. Listening on: http://0.0.0.0:8081
2020-01-28 22:40:54,921 INFO  [io.quarkus] (main) Profile test
activated.
2020-01-28 22:40:54,922 INFO  [io.quarkus] (main) Installed
features: [cdi, greeting, servlet]
2020-01-28 22:40:55,691 INFO  [io.quarkus] (main) Quarkus stopped
in 0.021s
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 1.714 s - in
org.acme.quarkus.greeting.deployment.GreetingTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
-----------------------------------------------------------------
-----
[INFO] Reactor Summary for getting-started-extension 1.0-SNAPSHOT:
[INFO]
[INFO] getting-started-extension .......................... SUCCESS
[  0.076 s]
[INFO] Greeting Extension - Parent ....................... SUCCESS
[  0.002 s]
[INFO] Greeting Extension - Runtime ...................... SUCCESS
[  1.467 s]
[INFO] Greeting Extension - Deployment ................... SUCCESS
[  4.099 s]
[INFO]
-----------------------------------------------------------------
-----
[INFO] BUILD SUCCESS
[INFO]
-----------------------------------------------------------------
-----
[INFO] Total time:  5.745 s
[INFO] Finished at: 2020-01-28T22:40:56+01:00
[INFO]
-----------------------------------------------------------------
-----
```

Looks good!! Congratulations you just finished your first extension.

Debugging your extension

If debugging is the process of removing bugs, then programming must be the process of putting them in.
Edsger W. Dijkstra

Debugging your application build

Since your extension deployment is made during the application build, this process is triggered by your build tool. That means if your want to debug this phase you need to launch your build tool with the remote debug mode switched one.

Maven

You can activate Maven remote debugging by using `mvnDebug`. You can launch your application with the following command line:

```
mvnDebug clean compile quarkus:dev
```

By default, Maven will wait for a connection on `localhost:8000`. Now, you can run your IDE `Remote` configuration to attach it to `localhost:8000`.

Gradle

You can activate Gradle remote debugging by using the flags `org.gradle.debug=true` or `org.gradle.daemon.debug=true` in daemon mode. You can launch your application with the following command line:

```
./gradlew quarkusDev -Dorg.gradle.daemon.debug=true
```

By default, Maven will wait for a connection on `localhost:5005`. Now, you can run your IDE `Remote` configuration to attach it to `localhost:5005`.

Debugging your extension tests

We have seen together how to test your extension and sometimes things don't go so well and you want to debug your tests. Same principle here, the trick is to enable the Maven Surefire remote debugging in order to attach an IDE `Remote` configuration.

```
$ cd ./greeting
$ mvn clean test -Dmaven.surefire.debug
```

By default, Maven will wait for a connection on `localhost:5005`.

Extension publication

Now that you just finish to build your first extension you should be eager to use it in a Quarkus application!

Classic Maven publication

Because your extension produces a classics JARs, the easiest way to share your extension is to publish it to a Maven repository. Once published you can simply declare it with your project dependencies

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-platform-bom</artifactId>
            <version>{quarkus-version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-resteasy</artifactId>
    </dependency>
    <dependency>
        <groupId>org.acme</groupId>
        <artifactId>quarkus-greeting</artifactId>
    </dependency>
</dependencies>
```

Run the application and notice the `Install Features` list contains the `greeting` extension.

```
$ mvn clean compile quarkus:dev
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< org.acme:code-with-quarkus
>--------------------
[INFO] Building code-with-quarkus 1.0.0-SNAPSHOT
[INFO] --------------------------------[ jar
]--------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ code-
with-quarkus ---
[INFO] Deleting /tmp/code-with-quarkus/target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources)
@ code-with-quarkus ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @
code-with-quarkus ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /tmp/code-with-
quarkus/target/classes
[INFO]
[INFO] --- quarkus-maven-plugin:1.2.0.Final:dev (default-cli) @
code-with-quarkus ---
Listening for transport dt_socket at address: 5005
2020-01-28 22:49:40,585 INFO  [io.quarkus] (main) code-with-quarkus
1.0.0-SNAPSHOT (running on Quarkus 1.2.0.Final) started in 1.025s.
Listening on: http://0.0.0.0:8080
2020-01-28 22:49:40,600 INFO  [io.quarkus] (main) Profile dev
activated. Live Coding activated.
2020-01-28 22:49:40,600 INFO  [io.quarkus] (main) Installed
features: [cdi, greeting, resteasy, servlet]
```

From an extension developer standpoint the Maven publication strategy is very handy and fast but Quarkus wants to go one step further by also ensuring a reliability of the ecosystem for the people who will use the extensions. Think about it, we all had a poor Developer Experience with an unmaintained library, an incompatibility between dependencies (and we don't even talk about legal issues). That's why there is the Quarkus Platform.

Quarkus Platform

Quarkus proposes a `quarkus-platform-bom` which is a certified list of extensions placed under the Quarkus Platform label. From an application developper, the objectives of the platform are:

- To guarantee a supportability of the extension (bugfix, security issues, dependency upgrades)

- To ease the extension discovery through the Quarkus CLI or https://code.quarkus.io/

- To ensure a global consistency of the extension ecosystem

Should I publish my extension to the platform?

If you feel your extensions is for you or a limited group, simply publishing to Maven is fine. If the extension solves a general problem, it is very handy for Quarkus users to see it on https://code.quarkus.io. But this comes with some responsibility for you, keeping it up to date with Quarkus minor releases (every month or so at the moment). When in doubt, have a conversation with the community in the Quarkus Google Group. We can make a collective decision.

As for now, the process to propose a new extension is not defined yet. Your best chance is to present your extension on the Quarkus Google Group and wait for an official invitation to join the Quarkus Platform.

Conclusion

Creating new extensions may appear to be an intricate task at first but once you understood the Quarkus game-changer paradigm (build time vs runtime) the structure of an extension makes perfectly sense.

As usual, along the path Quarkus simplifies things under the hood (Maven Mojo, bytecode generation or testing) to make it pleasant to develop new features.