Building Quarkus apps with Gradle

🔥 Quarkus Gradle support is considered preview. You can use Gradle to create Quarkus projects as outlined in our guides. If you go beyond there will be cases where the Gradle tasks does not behave as expected. This is just a caution, and we recommend if you like Gradle you try it out and give us feedback.

Creating a new project

The easiest way to scaffold a Gradle project, is currently to use the Quarkus Maven plugin like so:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
    -DprojectGroupId=my-groupId \
    -DprojectArtifactId=my-artifactId \
    -DprojectVersion=my-version \
    -DclassName="org.my.group.MyResource" \
    -Dextensions="resteasy-jsonb" \
    -DbuildTool=gradle
```

ℹ️ If you just launch `mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create` the Maven plugin asks for user inputs. You can disable (and use default values) this interactive mode by passing `-B` to the Maven command.

ℹ️ Quarkus project scaffolding automatically installs the Gradle wrapper (`./gradlew`) in your project.

If you prefer to use a standalone Gradle installation, please use Gradle 6.6.1.

The following table lists the attributes you can pass to the `create` command:

Attribute	Default Value	Description
`projectGroupId`	`org.acme.sample`	The group id of the created project
`projectArtifactId`	*mandatory*	The artifact id of the created project. Not passing it triggers the interactive mode.
`projectVersion`	`1.0-SNAPSHOT`	The version of the created project
`className`	*Not created if omitted*	The fully qualified name of the generated resource

Attribute	Default Value	Description
`path`	`/hello`	The resource path, only relevant if `className` is set.
`extensions`	*[]*	The list of extensions to add to the project (comma-separated)

If you decide to generate a REST resource (using the `className` attribute), the endpoint is exposed at: `http://localhost:8080/$path`. If you use the default `path`, the URL is: http://localhost:8080/hello.

The project is either generated in the current directory or in a directory named after the passed artifactId. If the current directory is empty, the project is generated in-place.

A pair of Dockerfiles for native and jvm mode are also generated in `src/main/docker`. Instructions to build the image and run the container are written in those Dockerfiles.

Custom test configuration profile in JVM mode

By default, Quarkus tests in JVM mode are run using the `test` configuration profile. If you are not familiar with Quarkus configuration profiles, everything you need to know is explained in the Configuration Profiles Documentation.

It is however possible to use a custom configuration profile for your tests with the Gradle build configuration shown below. This can be useful if you need for example to run some tests using a specific database which is not your default testing database.

```
test {
    systemProperty "quarkus.test.profile", "foo" ①
}
```

or, if you use the Gradle Kotlin DSL:

```
tasks.test {
    systemProperty("quarkus.test.profile", "foo") ①
}
```

① The `foo` configuration profile will be used to run the tests.

> ⚠ It is not possible to use a custom test configuration profile in native mode for now. Native tests are always run using the `prod` profile.

Dealing with extensions

From inside a Quarkus project, you can obtain a list of the available extensions with:

```
./gradlew listExtensions
```

You can enable an extension using:

```
./gradlew addExtension --extensions="hibernate-validator"
```

Extensions are passed using a comma-separated list.

The extension name is the GAV name of the extension: e.g. `io.quarkus:quarkus-agroal`. But you can pass a partial name and Quarkus will do its best to find the right extension. For example, `agroal`, `Agroal` or `agro` will expand to `io.quarkus:quarkus-agroal`. If no extension is found or if more than one extensions match, you will see a red check mark in the command result.

```
./gradlew addExtension --extensions="jdbc,agroal,non-exist-ent"
[...]
  Multiple extensions matching 'jdbc'
      * io.quarkus:quarkus-jdbc-h2
      * io.quarkus:quarkus-jdbc-mariadb
      * io.quarkus:quarkus-jdbc-postgresql
      Be more specific e.g using the exact name or the full gav.
  Adding extension io.quarkus:quarkus-agroal
  Cannot find a dependency matching 'non-exist-ent', maybe a typo?
[...]
```

You can install all extensions which match a globbing pattern:

```
./gradlew addExtension --extensions="hibernate*"
```

Development mode

Quarkus comes with a built-in development mode. Run your application with:

```
./gradlew quarkusDev
```

You can then update the application sources, resources and configurations. The changes are automatically reflected in your running application. This is great to do development spanning UI and database as you see changes reflected immediately.

`quarkusDev` enables hot deployment with background compilation, which means that when you modify your Java files 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. The act of refreshing the browser triggers a scan of the workspace, and if any changes are detected the Java files are compiled, and the application is redeployed, then your request is serviced by the redeployed

application. If there are any issues with compilation or deployment an error page will let you know.

Hit `CTRL+C` to stop the application.

You can change the working directory the development environment runs on:

```
quarkusDev {
    workingDir = rootProject.projectDir
}
```

> 💡 By default, the `quarkusDev` task uses `compileJava` compiler options. These can be overridden by setting the `compilerArgs` property in the task.

Remote Development Mode

It is possible to use development mode remotely, so that you can run Quarkus in a container environment (such as OpenShift) and have changes made to your local files become immediately visible.

This allows you to develop in the same environment you will actually run your app in, and with access to the same services.

> ⚠️ Do not use this in production. This should only be used in a development environment. You should not run production applications in dev mode.

To do this you must have the `quarkus-undertow-websockets` extension installed:

```
./gradlew addExtension --extensions="undertow-websockets"
```

You must also have the following config properties set:

- `quarkus.live-reload.password`
- `quarkus.live-reload.url`

These can be set via `application.properties`, or any other way (e.g. system properties, environment vars etc). The password must be set on both the local and remote processes, while the url only needs to be set on the local host.

Start Quarkus in dev mode on the remote host. Now you need to connect your local agent to the remote host:

```
./gradlew quarkusRemoteDev -Dquarkus.live-reload.url=http://my
-remote-host:8080
```

Now every time you refresh the browser you should see any changes you have made locally

immediately visible in the remote app.

Debugging

In development mode, Quarkus starts by default with debug mode enabled, listening to port `5005` without suspending the JVM.

This behavior can be changed by giving the `debug` system property one of the following values:

- `false` - the JVM will start with debug mode disabled
- `true` - The JVM is started in debug mode and will be listening on port `5005`
- `client` - the JVM will start in client mode and attempt to connect to `localhost:5005`
- `{port}` - The JVM is started in debug mode and will be listening on `{port}`

An additional system property `suspend` can be used to suspend the JVM, when launched in debug mode. `suspend` supports the following values:

- `y` or `true` - The debug mode JVM launch is suspended
- `n` or `false` - The debug mode JVM is started without suspending

> 💡 You can also run a Quarkus application in debug mode with a suspended JVM using `./gradlew quarkusDev -Dsuspend -Ddebug` which is a shorthand for `./gradlew quarkusDev -Dsuspend=true -Ddebug=true`.
>
> Then, attach your debugger to `localhost:5005`.

Import in your IDE

Once you have a [project generated](), you can import it in your favorite IDE. The only requirement is the ability to import a Gradle project.

Eclipse

In Eclipse, click on: `File → Import`. In the wizard, select: `Gradle → Existing Gradle Project`. On the next screen, select the root location of the project. The next screen list the found modules; select the generated project and click on `Finish`. Done!

In a separated terminal, run `./gradlew quarkusDev`, and enjoy a highly productive environment.

IntelliJ

In IntelliJ:

1. From inside IntelliJ select `File → New → Project From Existing Sources…` or, if you are on the welcome dialog, select `Import project`.
2. Select the project root

3. Select `Import project from external model` and `Gradle`

4. Next a few times (review the different options if needed)

5. On the last screen click on Finish

In a separated terminal or in the embedded terminal, run `./gradlew quarkusDev`. Enjoy!

Apache NetBeans

In NetBeans:

1. Select `File → Open Project`

2. Select the project root

3. Click on `Open Project`

In a separated terminal or the embedded terminal, go to the project root and run `./gradlew quarkusDev`. Enjoy!

Visual Studio Code

Open the project directory in VS Code. If you have installed the Java Extension Pack (grouping a set of Java extensions), the project is loaded as a Gradle project.

Building a native executable

Native executables make Quarkus applications ideal for containers and serverless workloads.

Make sure to have `GRAALVM_HOME` configured and pointing to GraalVM version 20.2.0 (Make sure to use a Java 11 version of GraalVM).

Create a native executable using: `./gradlew build -Dquarkus.package.type=native`. A native executable will be present in `build/`.

> ⚠️ The `buildNative` task has been deprecated in favor of `build -Dquarkus.package.type=native`.

Native related properties can either be added in `application.properties` file, as command line arguments or in the `quarkusBuild` task. Configuring the `quarkusBuild` task can be done as following:

```
quarkusBuild {
    nativeArgs {
        containerBuild = true ①
        buildImage = "quay.io/quarkus/ubi-quarkus-native-
image:20.2.0-java11" ②
    }
}
```

① Set `quarkus.native.containerBuild` property to `true`

② Set `quarkus.native.buildImage` property to `quay.io/quarkus/ubi-quarkus-native-image:20.2.0-java11`

Build a container friendly executable

The native executable will be specific to your operating system. To create an executable that will run in a container, use the following:

```
./gradlew build -Dquarkus.package.type=native
-Dquarkus.native.container-build=true
```

The produced executable will be a 64 bit Linux executable, so depending on your operating system it may no longer be runnable. However, it's not an issue as we are going to copy it to a Docker container. Note that in this case the build itself runs in a Docker container too, so you don't need to have GraalVM installed locally.

By default, the native executable will be generated using the `quay.io/quarkus/ubi-quarkus-native-image:20.2.0-java11` Docker image.

If you want to build a native executable with a different Docker image (for instance to use a different GraalVM version), use the `-Dquarkus.native.builder-image=<image name>` build argument.

The list of the available Docker images can be found on quay.io. Be aware that a given Quarkus version might not be compatible with all the images available.

Running native tests

Run the native tests using:

```
./gradlew testNative
```

This task depends on `quarkusBuild`, so it will generate the native image before running the tests.

Building Uber-Jars

Quarkus Gradle plugin supports the generation of Uber-Jars by specifying an `--uber-jar` argument as follows:

```
./gradlew quarkusBuild --uber-jar
```

When building an Uber-Jar you can specify entries that you want to exclude from the generated jar by using the `--ignored-entry` argument:

```
./gradlew quarkusBuild --uber-jar --ignored-entry=META
-INF/file1.txt
```

The entries are relative to the root of the generated Uber-Jar. You can specify multiple entries by adding extra `--ignored-entry` arguments.

Working with multi-module projects

By default, Quarkus will not discover CDI beans inside another module.

The best way to enable CDI bean discovery for a module in a multi-module project would be to include a `META-INF/beans.xml` file, unless it is the main application module already configured with the quarkus-maven-plugin, in which case it will indexed automatically.

Alternatively, there is some unofficial Gradle Jandex plugins) that can be used instead of the `META-INF/beans.xml` file.

More information on this topic can be found on the Bean Discovery section of the CDI guide.

Building with `./gradlew build`

Starting from 1.1.0.Final, `./gradlew build` will no longer build the native image. Add `-Dquarkus.package.type=native` build argument explicitly as explained above if needed.