

Quarkus - Building applications with Maven

Creating a new project

With Maven, you can scaffold a new project with:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
  -DprojectId=my-groupId \
  -DprojectArtifactId=my-artifactId \
  -DprojectVersion=my-version \
  -DclassName="org.my.group.MyResource"
```



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.

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

Attribute	Default Value	Description
<code>projectId</code>	<code>org.acme.sample</code>	The group id of the created project
<code>projectArtifactId</code>	<i>mandatory</i>	The artifact id of the created project. Not passing it triggers the interactive mode.
<code>projectVersion</code>	<code>1.0-SNAPSHOT</code>	The version of the created project
<code>platformGroupId</code>	<code>io.quarkus</code>	The group id of the target platform. Given that all the existing platforms are coming from <code>io.quarkus</code> this one won't practically be used explicitly. But it's still an option.
<code>platformArtifactId</code>	<code>quarkus-universe-bom</code>	The artifact id of the target platform BOM. It should be <code>quarkus-bom</code> in order to use the locally built Quarkus.

Attribute	Default Value	Description
<code>platformVersion</code>	If it's not specified, the latest one will be resolved.	The version of the platform you want the project to use. It can also accept a version range, in which case the latest from the specified range will be used.
<code>className</code>	<i>Not created if omitted</i>	The fully qualified name of the generated resource
<code>path</code>	<code>/hello</code>	The resource path, only relevant if <code>className</code> is set.
<code>extensions</code>	<code>[]</code>	The list of extensions to add to the project (comma-separated)

By default, the command will target the latest version of `quarkus-universe-bom` (unless specific coordinates have been specified). If you run offline however, it will look for the latest locally available and if `quarkus-universe-bom` (satisfying the default version range which is currently up to 2.0) is not available locally, it will fallback to the bundled platform based on `quarkus-bom` (the version will match the version of the plugin).

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 generated in a directory named after the passed `artifactId`. If the directory already exists, the generation fails.

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.

Dealing with extensions

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

```
./mvnw quarkus:list-extensions
```

You can enable an extension using:

```
./mvnw quarkus:add-extension -Dextensions="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.

```
$ ./mvnw quarkus:add-extensions -Dextensions=jdbc,agroal,non-existent
[...]
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-existent', maybe a typo?
[...]
```

You can install all extensions which match a globbing pattern :

```
./mvnw quarkus:add-extension -Dextensions="hibernate-*
```

Development mode

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

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

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.

quarkus:dev 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.

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 application in dev mode.

To do this you must build a mutable application, using the `mutable-jar` format. Set the following properties in `application.xml`:

```
quarkus.package.type=mutable-jar ①
quarkus.live-reload.password=changeit ②
quarkus.live-reload.url=http://my.cluster.host.com:8080 ③
```

- ① This tells Quarkus to use the mutable-jar format. Mutable applications also include the deployment time parts of Quarkus, so they take up a bit more disk space. If run normally they start just as fast and use the same memory as an immutable application, however they can also be started in dev mode.
- ② The password that is used to secure communication between the remote side and the local side.
- ③ The URL that your app is going to be running in dev mode at. This is only needed on the local side, so you may want to leave it out of the properties file and specify it as a system property on the command line.

Before you start Quarkus on the remote host set the environment variable `QUARKUS_LAUNCH_DEVMODE=true`. If you are on bare metal you can just set this via the `export QUARKUS_LAUNCH_DEVMODE=true` command, if you are running using docker start the image with `-e QUARKUS_LAUNCH_DEVMODE=true`. When the application starts you should now see the following line in the logs: `Profile dev activated. Live Coding activated.`



The remote side does not need to include Maven or any other development tools. The normal `fast-jar` Dockerfile that is generated with a new Quarkus application is all you need. If you are using bare metal launch the Quarkus runner jar, do not attempt to run normal devmode.

Now you need to connect your local agent to the remote host, using the `remote-dev` command:

```
./mvnw quarkus:remote-dev -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. This is done via a HTTP based long polling transport, that will synchronize your local workspace and the remote application via HTTP calls.

If you do not want to use the HTTP feature then you can simply run the `remote-dev` command without specifying the URL. In this mode the command will continuously rebuild the local application, so you can use an external tool such as `odo` or `rsync` to sync to the remote application.



It is recommended you use SSL when using remote dev mode, however even if you are using an unencrypted connection your password is never sent directly over the wire. For the initial connection request the password is hashed with the initial state data, and subsequent requests hash it with a random session id generated by the server and any body contents for POST requests, and the path for DELETE requests, as well as an incrementing counter to prevent replay attacks.

Configuring Development Mode

By default, the Maven plugin picks up compiler flags to pass to `javac` from `maven-compiler-plugin`.

If you need to customize the compiler flags used in development mode, add a `configuration` section to the `plugin` block and set the `compilerArgs` property just as you would when configuring `maven-compiler-plugin`. You can also set `source`, `target`, and `jvmArgs`. For example, to pass `--enable-preview` to both the JVM and `javac`:

```
<plugin>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-maven-plugin</artifactId>
  <version>${quarkus-plugin.version}</version>

  <configuration>
    <source>${maven.compiler.source}</source>
    <target>${maven.compiler.target}</target>
    <compilerArgs>
      <arg>--enable-preview</arg>
    </compilerArgs>
    <jvmArgs>--enable-preview</jvmArgs>
  </configuration>

  ...
</plugin>
```

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 `./mvnw compile quarkus:dev -Ddebug` which is a shorthand for `./mvnw compile quarkus:dev -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 Maven project.

Eclipse

In Eclipse, click on: `File` → `Import`. In the wizard, select: `Maven` → `Existing Maven 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 `./mvnw compile quarkus:dev`, 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 `Maven`
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 `./mvnw compile quarkus:dev`. 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 `./mvnw compile quarkus:dev`. 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 Maven project.

Logging Quarkus application build classpath tree

Usually, dependencies of an application (which is a Maven project) could be displayed using `mvn dependency:tree` command. In case of a Quarkus application, however, this command will list only the runtime dependencies of the application. Given that the Quarkus build process adds deployment dependencies of the extensions used in the application to the original application classpath, it could be useful to know which dependencies and which versions end up on the build classpath. Luckily, the `quarkus-bootstrap` Maven plugin includes the `build-tree` goal which displays the build dependency tree for the application.

To be able to use it, the following plugin configuration has to be added to the `pom.xml`:

```
<plugin>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-bootstrap-maven-
plugin</artifactId>
  <version>${quarkus-plugin.version}</version>
</plugin>
```

Now you should be able to execute `./mvnw quarkus-bootstrap:build-tree` on your project and see something like:

```
[INFO] --- quarkus-bootstrap-maven-plugin:1.8.0.CR1:build-tree
(default-cli) @ getting-started ---
[INFO] org.acme:getting-started:jar:1.0-SNAPSHOT
[INFO] └─ io.quarkus:quarkus-resteasy-deployment:jar:1.8.0.CR1
(compile)
[INFO]   └─ io.quarkus:quarkus-resteasy-server-common-
deployment:jar:1.8.0.CR1 (compile)
[INFO]     └─ io.quarkus:quarkus-core-deployment:jar:1.8.0.CR1
(compile)
[INFO]       └─ commons-beanutils:commons-beanutils:jar:1.9.3
(compile)
[INFO]         └─ commons-logging:commons-logging:jar:1.2
(compile)
[INFO]           └─ commons-collections:commons-
collections:jar:3.2.2 (compile)
...
```

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). Verify that your `pom.xml` has the proper `native` profile (see [Maven configuration](#)).

Create a native executable using: `./mvnw package -Pnative`. A native executable will be present in `target/`.

To run Integration Tests on the native executable, make sure to have the proper Maven plugin configured (see [Maven configuration](#)) and launch the `verify` goal.


```

./mvnw verify -Pnative
...
[quarkus-quickstart-runner:50955]    universe:      391.96 ms
[quarkus-quickstart-runner:50955]    (parse):      904.37 ms
[quarkus-quickstart-runner:50955]    (inline):    1,143.32 ms
[quarkus-quickstart-runner:50955]    (compile):   6,228.44 ms
[quarkus-quickstart-runner:50955]    compile:    9,130.58 ms
[quarkus-quickstart-runner:50955]    image:      2,101.42 ms
[quarkus-quickstart-runner:50955]    write:      803.18 ms
[quarkus-quickstart-runner:50955]    [total]:   33,520.15 ms
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.0:integration-test (default)
@ quarkus-quickstart-native ---
[INFO]
[INFO] -----
[INFO]  T E S T S
[INFO] -----
[INFO] Running org.acme.quickstart.GreetingResourceIT
Executing [/Users/starksm/Dev/JBoss/Quarkus/starksm64-quarkus-
quickstarts/getting-started-native/target/quarkus-quickstart-
runner, -Dquarkus.http.port=8081, -Dtest.url=http://localhost:8081,
-Dquarkus.log.file.path=target/quarkus.log]
2019-02-28 16:52:42,020 INFO  [io.quarkus] (main) Quarkus started
in 0.007s. Listening on: http://localhost:8080
2019-02-28 16:52:42,021 INFO  [io.quarkus] (main) Installed
features: [cdi, resteasy]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 1.081 s - in org.acme.quickstart.GreetingResourceIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
...

```

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:

```
./mvnw package -Dnative -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.

You can follow the [Build a native executable guide](#) as well as [Deploying Application to Kubernetes and OpenShift](#) for more information.

Maven configuration

If you have not used [project scaffolding](#), add the following elements in your `pom.xml`

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

<build>
  <plugins>
    <plugin> ②
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-maven-plugin</artifactId>
      <version>${quarkus-plugin.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>build</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin> ③
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
```

```

        <version>${surefire-plugin.version}</version>
        <configuration>
            <systemPropertyVariables>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.ut
il.logging.manager>

            <maven.home>${maven.home}</maven.home>
            </systemPropertyVariables>
        </configuration>
    </plugin>
</plugins>
</build>

<profiles>
    <profile> ④
        <id>native</id>
        <properties> ⑤
            <quarkus.package.type>native</quarkus.package.type>
        </properties>
        <build>
            <plugins>
                <plugin> ⑥
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-failsafe-plugin</artifactId>
                    <version>${surefire-plugin.version}</version>
                    <executions>
                        <execution>
                            <goals>
                                <goal>integration-test</goal>
                                <goal>verify</goal>
                            </goals>
                            <configuration>
                                <systemPropertyVariables>

<native.image.path>${project.build.directory}/${project.build.final
Name}-runner</native.image.path>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.ut
il.logging.manager>

<maven.home>${maven.home}</maven.home>
                                </systemPropertyVariables>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>

```

```
</profiles>
```

- ① Optionally use a BOM file to omit the version of the different Quarkus dependencies.
- ② Use the Quarkus Maven plugin that will hook into the build process.
- ③ Add system properties to `maven-surefire-plugin`.
`maven.home` is only required if you have custom configuration in `${maven.home}/conf/settings.xml`.
- ④ Use a specific `native` profile for native executable building.
- ⑤ Enable the `native` package type. The build will therefore produce a native executable.
- ⑥ If you want to test your native executable with Integration Tests, add the following plugin configuration. Test names `*IT` and annotated `@NativeImageTest` will be run against the native executable. See the [Native executable guide](#) for more info.

Uber-Jar Creation

Quarkus Maven plugin supports the generation of Uber-Jars by specifying a `quarkus.package.type=uber-jar` configuration option in your `application.properties`.

The original jar will still be present in the `target` directory but it will be renamed to contain the `.original` suffix.

When building an Uber-Jar you can specify entries that you want to exclude from the generated jar by using the `quarkus.package.ignored-entries` configuration option, this takes a comma separated list of entries to ignore.

Uber-Jar creation by default excludes [signature files](#) that might be present in the dependencies of the application.

Uber-Jar's final name is configurable via a Maven's build settings `finalName` option.

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 the `jandex-maven-plugin`, unless it is the main application module already configured with the `quarkus-maven-plugin`, in which case it will indexed automatically.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.jandex</groupId>
      <artifactId>jandex-maven-plugin</artifactId>
      <version>1.0.7</version>
      <executions>
        <execution>
          <id>make-index</id>
          <goals>
            <goal>jandex</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>


```



More information on this topic can be found on the [Bean Discovery](#) section of the CDI guide.








Configuring the Project Output



There are a several configuration options that will define what the output of your project build will be. These are provided in `application.properties` the same as any other config property.

The properties are shown below:

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

Configuration property	Type	Default
 <code>quarkus.package.type</code> The requested output type. The default built in types are 'jar', 'fast-jar' (a prototype more performant version of the default 'jar' type), 'uber-jar' and 'native'.	string	<code>jar</code>
 <code>quarkus.package.manifest.add-implementation-entries</code> If the Implementation information should be included in the runner jar's MANIFEST.MF.	boolean	<code>true</code>

 <code>quarkus.package.main-class</code> The entry point of the application. This can either be a fully qualified name of a standard Java class with a main method, or <code>io.quarkus.runtime.QuarkusApplication</code> . If your application has main classes annotated with <code>io.quarkus.runtime.annotations.QuarkusMain</code> then this can also reference the name given in the annotation, to avoid the need to specify fully qualified names in the config.	string	
 <code>quarkus.package.user-configured-ignored-entries</code> Files that should not be copied to the output artifact	list of string	
 <code>quarkus.package.runner-suffix</code> The suffix that is applied to the runner jar and native images	string	<code>-runner</code>
 <code>quarkus.package.output-directory</code> The output folder in which to place the output, this is resolved relative to the build systems target directory.	string	
 <code>quarkus.package.output-name</code> The name of the final artifact	string	
 <code>quarkus.package.create-appcds</code> Whether to automate the creation of AppCDS. This has not effect when a native binary is needed and will be ignored in that case. Furthermore, this option only works for Java 11+ and is considered experimental for the time being. Finally, care must be taken to use the same exact JVM version when building and running the application.	boolean	<code>false</code>
 <code>quarkus.package.user-providers-directory</code> This is an advanced option that only takes effect for the mutable-jar format. If this is specified a directory of this name will be created in the jar distribution. Users can place jar files in this directory, and when re-augmentation is performed these will be processed and added to the class-path. Note that before reaugmentation has been performed these jars will be ignored, and if they are updated the app should be reaugmented again.	string	

<p> <code>quarkus.package.manifest.manifest-sections</code></p> <p>Custom manifest sections to be added to the MANIFEST.MF file. An example of the user defined property: <code>quarkus.package.manifest.manifest-sections.{Section-Name}.Entry-Key1={Value1}</code> <code>quarkus.package.manifest.manifest-sections.{Section-Name}.Entry-Key2={Value2}</code></p>	<p><code>Map<String, Map<String, String>></code></p>	<p>required </p>
---	--	---

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 Maven Surefire and Maven Failsafe configurations 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.

```

<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>${surefire-plugin.version}</version>
        <configuration>
          <systemPropertyVariables>
            <quarkus.test.profile>foo</quarkus.test.profile> ①

<buildDirectory>${project.build.directory}</buildDirectory>
      [...]
    </systemPropertyVariables>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>${failsafe-plugin.version}</version>
  <configuration>
    <systemPropertyVariables>
      <quarkus.test.profile>foo</quarkus.test.profile> ①

<buildDirectory>${project.build.directory}</buildDirectory>
      [...]
    </systemPropertyVariables>
  </configuration>
</plugin>
</plugins>
</build>
  [...]
</project>

```

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.