Contributing guide

Want to contribute? Great! We try to make it easy, and all contributions, even the smaller ones, are more than welcome. This includes bug reports, fixes, documentation, examples... But first, read this page (including the small print at the end).

Table of contents generated with markdown-toc

Legal

All original contributions to Quarkus are licensed under the ASL - Apache License, version 2.0 or later, or, if another license is specified as governing the file or directory being modified, such other license.

All contributions are subject to the Developer Certificate of Origin (DCO). The DCO text is also included verbatim in the dco.txt file in the root directory of the repository.

Reporting an issue

This project uses GitHub issues to manage the issues. Open an issue directly in GitHub.

If you believe you found a bug, and it's likely possible, please indicate a way to reproduce it, what you are seeing and what you would expect to see. Don't forget to indicate your Quarkus, Java, Maven/Gradle and GraalVM version.

Checking an issue is fixed in main

Sometimes a bug has been fixed in the `main` branch of Quarkus and you want to confirm it is fixed for your own application. There are two simple options for testing the `main` branch:

- either use the snapshots we publish daily on https://s01.oss.sonatype.org/content/repositories/snapshots
- or build Quarkus locally

The following is a quick summary aimed at allowing you to quickly test `main`. If you are interested in learning more details, refer to the Build section and the Usage section.

Using snapshots

Snapshots are published daily with version `999-SNAPSHOT`, so you will have to wait for a snapshot containing the commits you are interested in.

Then just add https://s01.oss.sonatype.org/content/repositories/snapshots as a Maven repository **and** a plugin repository in your settings xml:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
        <profile>
            <id>quarkus-snapshots</id>
            <repositories>
                <repository>
                    <id>quarkus-snapshots-repository</id>
                    <url>https://s01.oss.sonatype.org/content/repositories/snapshots/</url>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </repository>
            </repositories>
            <pluginRepositories>
                <pluginRepository>
                    <id>quarkus-snapshots-plugin-repository</id>
                    <url>https://s01.oss.sonatype.org/content/repositories/snapshots/</url>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </pluginRepository>
            </pluginRepositories>
        </profile>
    </profiles>
    <activeProfiles>
        <activeProfile>quarkus-snapshots</activeProfile>
    </activeProfiles>
</settings>
```

You can check the last publication date here: https://s01.oss.sonatype.org/content/repositories/snapshots/io/quarkus/.

Building main

Just do the following:

```
git clone git@github.com:quarkusio/quarkus.git
cd quarkus
MAVEN_OPTS="-Xmx4g" ./mvnw -Dquickly
```

Building can take a few minutes depending on the hardware being used.

Note For Apple Silicon computer, Rosetta must be installed. It can be done using `softwareupdate --install-rosetta`

Updating the version

When using the `main` branch, you need to use the group id `io.quarkus` instead of `io.quarkus.platform` for both the Quarkus BOM and the Quarkus Maven Plugin.

In a standard Quarkus pom.xml set the `quarkus.platform.group-id` property to `io.quarkus` and the `quarkus.platform.version` property to `999-SNAPSHOT` to build your application against the locally installed main branch.

You can now test your application.

Before you contribute

To contribute, use GitHub Pull Requests, from your **own** fork.

Also, make sure you have set up your Git authorship correctly:

```
git config --global user.name "Your Full Name"
git config --global user.email your.email@example.com
```

If you use different computers to contribute, please make sure the name is the same on all your computers.

We use this information to acknowledge your contributions in release announcements.

Code reviews

All submissions, including submissions by project members, need to be reviewed by at least one Quarkus committer before being merged.

GitHub Pull Request Review Process is followed for every pull request.

Coding Guidelines

- We decided to disallow `@author` tags in the Javadoc: they are hard to maintain, especially in a very active project, and we use the Git history to track authorship. GitHub also has this nice page with your contributions. For each major Quarkus release, we also publish the list of contributors in the announcement post.
- Commits should be atomic and semantic. Please properly squash your pull requests before submitting them. Fixup commits can be used temporarily during the review process but things should be squashed at the end to have meaningful commits. We use merge commits so the GitHub Merge button cannot do that for us. If you don't know how to do that, just ask in your pull request, we will be happy to help!
- Please limit the use of lambdas and streams as much as possible in code that executes at runtime, in order to minimize runtime footprint.

Continuous Integration

Because we are all humans, and to ensure Quarkus is stable for everyone, all changes must go through Quarkus continuous integration. Quarkus CI is based on GitHub Actions, which means that everyone has the ability to automatically execute CI in their forks as part of the process of making changes. We ask that all non-trivial changes go through this process, so that the contributor gets immediate feedback, while at the same time keeping our CI fast and healthy for everyone.

The process requires only one additional step to enable Actions on your fork (clicking the green button in the actions tab). See the full video walkthrough for more details on how to do this.

To keep the caching of non-Quarkus artifacts efficient (speeding up CI), you should occasionally sync the `main` branch of your fork with `main` of this repo (e.g. monthly).

Tests and documentation are not optional

Don't forget to include tests in your pull requests. Also don't forget the documentation (reference documentation, javadoc...).

Be sure to test your pull request in:

1. Java mode
2. Native mode

Also, make sure that any native tests you add will actually get executed on CI. In the interest of speeding up CI, the native build job `native-tests` have been split into multiple categories which are run in parallel. This means that each new integration test module needs to be configured explicitly in `native-tests.json` to have its integration tests run in native mode.

Setup

If you have not done so on this machine, you need to:

- Make sure you have a case-sensitive filesystem. Java development on a case-insensitive filesystem can cause headaches.
  - Linux: You're good to go.
  - macOS: Use the `Disk Utility.app` to check. It also allows you to create a case-sensitive volume to store your code projects. See this blog entry for more.
  - Windows: Enable case sensitive file names per directory
- Install Git and configure your GitHub access
  - Windows:
    - enable longpaths: `git config --global core.longpaths true`
    - avoid CRLF breaks: `git config --global core.autocrlf false`
- Install Java SDK 17+ (OpenJDK recommended)
- Install GraalVM
- Install platform C developer tools:
  - Linux
    - Make sure headers are available on your system (you'll hit 'Basic header file missing (<zlib.h>)' error if they aren't).
      - On Fedora `sudo dnf install zlib-devel`
      - Otherwise `sudo apt-get install libz-dev`
  - macOS
    - `xcode-select --install`

- Set `GRAALVM_HOME` to your GraalVM Home directory e.g. `/opt/graalvm` on Linux or `$location/JDK/GraalVM/Contents/Home` on macOS

A container engine (such as Docker or Podman) is not strictly necessary: it is used to run the container-based tests which are not enabled by default.

It is a recommended install though when working on any extension that relies on dev services or container-based tests (Hibernate ORM, MongoDB, ...):

- For Docker, check the installation guide, and the macOS installation guide. If you just install docker, be sure that your current user can run a container (no root required). On Linux, check the post-installation guide.
- For Podman, some extra configuration will be required to get testcontainers working. See the Quarkus and Podman guide for setup instructions.

IDE Config and Code Style

Quarkus has a strictly enforced code style. Code formatting is done by the Eclipse code formatter, using the config files found in the `independent-projects/ide-config` directory. By default, when you run `./mvnw install`, the code will be formatted automatically. When submitting a pull request the CI build will fail if running the formatter results in any code changes, so it is recommended that you always run a full Maven build before submitting a pull request.

If you want to run the formatting without doing a full build, you can run `./mvnw process-sources`.

Eclipse Setup

Open the *Preferences* window, and then navigate to *Java -> Code Style -> Formatter*. Click *Import* and then select the `eclipse-format.xml` file in the `independent-projects/ide-config` directory.

Next navigate to *Java -> Code Style -> Organize Imports*. Click *Import* and select the `eclipse.importorder` file.

IDEA Setup

How to work

Quarkus is a large project and IDEA will have a hard time compiling the whole of it. Before you start coding, make sure to build the project using Maven from the commandline with `./mvnw -Dquickly`.

`OutOfMemoryError` while importing

After creating an IDEA project, the first import might fail with an `OutOfMemory` error, as the size of the project requires more memory than the IDEA default settings allow.

Note In some IDEA versions the `OutOfMemory` error goes unreported. So if no error is reported but IDEA is still failing to find symbols or the dependencies are wrong in the imported project, then importing might have failed due to an unreported `OutOfMemory` exception. One can further investigate this by inspecting the `org.jetbrains.idea.maven.server.RemoteMavenServer36` process (or processes) using `JConsole`.

To fix that, open the *Preferences* window (or *Settings* depending on your edition), then navigate to *Build, Execution, Deployment > Build Tools > Maven > Importing*. In *VM options for importer*, raise the heap to at least 2 GB; some people reported needing more, e.g. `-Xmx8g`.

In recent IDEA versions (e.g. 2020.3) this might not work because *VM options for importer* get ignored when `.mvn/jdk.config` is present (see IDEA-250160) it disregards the *VM options for importer* settings. An alternative solution is to go to *Help > Edit Custom Properties...* and add the following line:

```
idea.maven.embedder.xmx=8g
```

After these configurations, you might need to run *File -> Invalidate Caches and Restart* and then trigger a `Reload all Maven projects`.

`package sun.misc does not exist` while building

You may get an error like this during the build:

```
Error:(46, 56) java: package sun.misc does not exist
```

To fix this go to *Settings > Build, Execution, Deployment > Compiler > Java Compiler* and disable *Use '--release' option for cross compilation (java 9 and later)*.

Formatting

Open the *Preferences* window (or *Settings* depending on your edition), navigate to *Plugins* and install the Adapter for Eclipse Code Formatter from the Marketplace.

Restart your IDE, open the *Preferences* (or *Settings*) window again and navigate to *Adapter for Eclipse Code Formatter* section on the left pane.

Select *Use Eclipse's Code Formatter*, then change the *Eclipse workspace/project folder or config file* to point to the `eclipse-format.xml` file in the `independent-projects/ide-config/src/main/resources` directory. Make sure the *Optimize Imports* box is ticked. Then, select *Import Order from file* and make it point to the `eclipse.importorder` file in the `independent-projects/ide-config/src/main/resources` directory.

Next, disable wildcard imports: navigate to *Editor -> Code Style -> Java -> Imports* and set *Class count to use import with '*'* to `999`. Do the same with *Names count to use static import with '*'*.

Gitpod

You can also use Gitpod to contribute without installing anything on your computer. Click here to start a workspace.

Build

- Clone the repository: `git clone https://github.com/quarkusio/quarkus.git`
- Navigate to the directory: `cd quarkus`
- Set Maven heap to 4GB `export MAVEN_OPTS="-Xmx4g"`
- Invoke `./mvnw -Dquickly` from the root directory
- *Note: On Windows, it may be necessary to run the build from an elevated shell. If you experience a failed build with the error `"A required privilege is not held by the client"`, this should fix it.*

```
git clone https://github.com/quarkusio/quarkus.git
cd quarkus
export MAVEN_OPTS="-Xmx4g"
./mvnw -Dquickly
# Wait... success!
```

This build skipped all the tests, native-image builds, documentation generation etc. and used the Maven goals `clean install` by default. For more details about `-Dquickly` have a look at the `quick-build` profile in `quarkus-parent` (root `pom.xml`).

When contributing to Quarkus, it is recommended to respect the following rules.

> **Note:** The `impsort-maven-plugin` uses the `.cache` directory on each module to speed up the build. Because we have configured the plugin to store in a versioned directory, you may notice over time that the `.cache` directory grows in size. You can safely delete the `.cache` directory in each module to reclaim the space. Running `./mvnw clean -Dclean-cache` automatically deletes that directory for you.

Contributing to an extension

When you contribute to an extension, after having applied your changes, run:

- `./mvnw -Dquickly` from the root directory to make sure you haven't broken anything obvious
- `./mvnw -f extensions/<your-extension> clean install` to run a full build of your extension including the tests
- `./mvnw -f integration-tests/<your-extension-its> clean install` to make sure ITs are still passing
- `./mvnw -f integration-tests/<your-extension-its> clean install -Dnative` to test the native build (for small changes, it might not be useful, use your own judgement)

Contributing to a core artifact

Obviously, when you contribute to a core artifact of Quarkus, a change may impact any part of Quarkus. So the rule of thumb would be to run the full test suite locally but this is clearly impractical as it takes a lot of time/resources.

Thus, it is recommended to use the following approach:

- run `./mvnw -Dquickly` from the root directory to make sure you haven't broken anything obvious
- run any build that might be useful to test the behavior you changed actually fixes the issue you had (might be an extension build, an integration test build...)
- push your work to your own fork of Quarkus to trigger CI there
- you can create a draft pull request to keep track of your work
- wait until the build is green in your fork (use your own judgement if it's not fully green) before marking your pull request as ready for review (which will trigger Quarkus CI)

Workflow tips

Due to Quarkus being a large repository, having to rebuild the entire project every time a change is made isn't very productive. The following Maven tips can vastly speed up development when working on a specific extension.

Building all modules of an extension

Let's say you want to make changes to the `Jackson` extension. This extension contains the `deployment`, `runtime` and `spi` modules which can all be built by executing following command:

```
./mvnw install -f extensions/jackson/
```

This command uses the path of the extension on the filesystem to identify it. Moreover, Maven will automatically build all modules in that path recursively.

Building a single module of an extension

Let's say you want to make changes to the `deployment` module of the Jackson extension. There are two ways to accomplish this task as shown by the following commands:

```
./mvnw install -f extensions/jackson/deployment
```

or

```
./mvnw install --projects 'io.quarkus:quarkus-jackson-deployment'
```

In this command we use the groupId and artifactId of the module to identify it.

Building with relocations

Let's say you want to make changes to an extension and try it with an existing application that uses older Quarkus with extensions that got renamed or moved recently. Quarkus maintains compatibility as much as possible and for most renamed or moved artifact, it provides a relocation artifact, allowing the build to be redirected to the new artifact. However, relocations are not built by default, and to build and install them, you need to enable the `relocations` Maven profile as follows:

```
./mvnw -Dquickly -Prelocations
```

Relocations are published with every Quarkus release, thus this is needed only when working with Quarkus main.

Running a single test

Often you need to run a single test from some Maven module. Say for example you want to run the `GreetingResourceTest` of the `resteasy-jackson` Quarkus integration test (which can be found here). One way to accomplish this is by executing the following command:

```
./mvnw test -f integration-tests/resteasy-jackson/ -Dtest=GreetingResourceTest
```

Maven Invoker tests

For testing some scenarios, Quarkus uses the Maven Invoker to run tests. For these cases, to run a single test, one needs to use the `invoker.test` property along with the name of the directory which houses the test project.

For example, in order to only run the MySQL test project of the container-image tests, the Maven command would be:

```
./mvnw verify -f integration-tests/container-image/maven-invoker-way -Dinvoker.test=container-build-jib-with-mysql
```

Note that we use the `verify` goal instead of the `test` goal because the Maven Invoker is usually bound to the integration-test phase. Furthermore, depending on the actual test being invoked, more options maybe needed (for the specific integration test mentioned above, `-Dstart-containers` and `-Dtest-containers` are needed).

Build with multiple threads

The following standard Maven option can be used to build with multiple threads to speed things up (here 0.5 threads per CPU core):

```
./mvnw install -T0.5C
```

Please note that running tests in parallel is not supported at the moment!

Don't build any test modules

To omit building currently way over 100 pure test modules, run:

```
./mvnw install -Dno-test-modules
```

This can come in handy when you are only interested in the actual "productive" artifacts, e.g. when bisecting.

Automatic incremental build

:information_source: This feature is currently in testing mode. You're invited to give it a go and please reach out via Zulip or GitHub in case something doesn't work as expected or you have ideas to improve things.

Instead of *manually* specifying the modules to build as in the previous examples, you can tell gitflow-incremental-builder (GIB) to only build the modules that have been changed or depend on modules that have been changed (downstream). E.g.:

```
./mvnw install -Dincremental
```

This will build all modules (and their downstream modules) that have been changed compared to your *local* `main`, including untracked and uncommitted changes.

If you just want to build the changes since the last commit on the current branch, you can switch off the branch comparison via `-Dgib.disableBranchComparison` (or short: `-Dgib.dbc`).

There are many more configuration options in GIB you can use to customize its behaviour: https://github.com/gitflow-incremental-builder/gitflow-incremental-builder#configuration

Parallel builds (`-T...`) should work without problems but parallel test execution is not yet supported (in general, not a GIB limitation).

Special case `bom-descriptor-json`

Without going too much into details (`devtools/bom-descriptor-json/pom.xml` has more info), you should build this module *without* `-Dincremental` *if you changed any extension "metadata"*:

- Addition/renaming/removal of an extension
- Any other changes to any `quarkus-extension.yaml`

Usage by CI

The GitHub Actions based Quarkus CI is using GIB to reduce the average build time of pull request builds and builds of branches in your fork.

CI is using a slightly different GIB config than locally:

- Special handling of "Explicitly selected projects" is deactivated
- Untracked/uncommitted changes are not considered
- Branch comparison is more complex due to distributed GitHub forks
- Certain "critical" branches like `main` are not built incrementally

For more details see the `Get GIB arguments` step in `.github/workflows/ci-actions-incremental.yml`.

Develocity build cache

Getting set up

Quarkus has a Develocity instance set up at https://ge.quarkus.io that can be used to analyze the build performance of the Quarkus project and also provides build cache services.

If you have an account on https://ge.quarkus.io, this can speed up your local builds significantly.

If you have a need or interest to share your build scans and use the build cache, you will need to get an account for the Develocity instance. It is only relevant for members of the Quarkus team and you should contact either Guillaume Smet or Max Andersen to set up your account.

When you have the account set up, from the root of your local Quarkus workspace, run:

```
./mvnw develocity:provision-access-key
```

and log in in the browser window it will open (if not already logged in). Your access key will be stored in the `~/.m2/.develocity/keys.properties` file. From then your build scans will be sent to the Develocity instance and you will be able to benefit from the build cache.

You can alternatively also generate an API key from the Develocity UI and then use an environment variable like this:

```
export DEVELOCITY_ACCESS_KEY=ge.quarkus.io=a_secret_key
```

When debugging a test (and especially flaky tests), you might want to temporarily disable the build cache. You can easily do it by adding `-Dno-build-cache` to your Maven command.

The remote cache is stored on the Develocity server and is populated by CI. To be able to benefit from the remote cache, you need to use a Java version tested on CI (at the moment, either 17 or 21) and the same Maven version (thus why it is recommended to use the Maven wrapper aka `./mvnw`). Note that the local cache alone should bring you a significant speedup.

The local cache is stored in the `~/.m2/.develocity/build-cache/` directory. If you have problems with your local cache, you can delete this directory.

-Dquickly

When using `-Dquickly` with no goals, Develocity is unable to detect that the `clean` goal is present. We worked around it but you will get the following warnings at the beginning of your build output:

```
[WARNING] Build cache entries produced by this build may be incorrect since the clean lifecycle is not part of the build invocation.
[WARNING] You must only invoke the build without the clean lifecycle if the build is started from a clean working directory.
```

You can safely ignore them.

Benchmarking the build

During the experiment phase, there might be a need to benchmark the build in a reliable manner.

For this, we can use the Gradle Profiler. It can be installed with SDKMAN! (`sdk install gradleprofiler`) or Homebrew (`brew install gradle-profiler`).

Then we can run the following commands at the root of the Quarkus project:

```
# Without cache
gradle-profiler --maven --benchmark --scenario-file build.scenario clean_install_no_cache

# With cache
gradle-profiler --maven --benchmark --scenario-file build.scenario clean_install
```

Simple HTML reports will be published in the `profile_out*` directories.

Release your own version

You might want to release your own patched version of Quarkus to an internal repository.

To do so, you will first need to update the version in the source code:

```
./update-version.sh "x.y.z-yourcompany"
```

We use a shell script as we also need to update the version in various descriptors and test files. The shell script calls `./mvnw versions:set` under the hood, among other things.

Commit the changes, then run:

```
./mvnw --settings your-maven-settings.xml \
    clean deploy \
    -DskipTests -DskipITs \
    -DperformRelease=true \
    -Prelease \
    -Ddokka \
    -Dgpg.skip
```

If your Maven settings are in your global Maven settings file located in the `.m2/` directory, you can drop the `--settings your-maven-settings.xml` part.

Documentation

The documentation is hosted in the `docs` module of the main Quarkus repository and is synced to the Quarkus.io website at release time. The Asciidoc files can be found in the `src/main/asciidoc` directory.

For more information, see the Contribute to Quarkus Documentation guide.

Building the documentation

When contributing a significant documentation change, it is highly recommended to run the build and check the output.

First build the whole Quarkus repository with the documentation build enabled:

```
./mvnw -DquicklyDocs
```

This will generate the configuration properties documentation includes in the root `target/asciidoc/generated/config/` directory and will avoid a lot of warnings when building the documentation module.

Then you can build the `docs` module specifically:

```
./mvnw -f docs clean install
```

You can check the output of the build in the `docs/target/generated-docs/` directory.

You can build the documentation this way as many times as needed, just avoid doing a `./mvnw clean` at the root level because you would lose the configuration properties documentation includes.

Referencing a new guide in the index

The Guides index page visible on the website is generated from a YAML file named `guides-latest.yaml` present in the Quarkus.io website repository . This particular file is for the latest stable version.

When adding a new guide to the `main` version of Quarkus, you need to reference the guide in the `main` guides index YAML file .

This file will later be copied to become the new `guides-latest.yaml` file when the next major or minor version is released.

Usage

After the build was successful, the artifacts are available in your local Maven repository.

To include them into your project a few things have to be changed.

With Maven

pom.xml

```
<properties>
    <quarkus-plugin.version>999-SNAPSHOT</quarkus-plugin.version>
    <quarkus.platform.artifact-id>quarkus-bom</quarkus.platform.artifact-id>
    <quarkus.platform.group-id>io.quarkus</quarkus.platform.group-id>
    <quarkus.platform.version>999-SNAPSHOT</quarkus.platform.version>
    .
    .
    .
</properties>
```

With Gradle

gradle.properties

```
quarkusPlatformArtifactId=quarkus-bom
quarkusPluginVersion=999-SNAPSHOT
quarkusPlatformVersion=999-SNAPSHOT
quarkusPlatformGroupId=io.quarkus
```

settings.gradle

```
pluginManagement {
    repositories {
        mavenLocal() // add mavenLocal() to first position
        mavenCentral()
        gradlePluginPortal()
    }
    .
    .
    .
}
```

build.gradle

```
repositories {
    mavenLocal() // add mavenLocal() to first position
    mavenCentral()
}
```

MicroProfile TCK's

Quarkus has a TCK module in `tcks` where all the MicroProfile TCK's are set up for you to run if you wish. These include tests to areas like Config, JWT Authentication, Fault Tolerance, Health Checks, Metrics, OpenAPI, OpenTracing, REST Client, Reactive Messaging and Context Propagation.

The TCK module is not part of the main Maven reactor build, but you can enable it and run the TCK tests by activating the Maven Profile `-Ptcks`. If your work is related to any of these areas, running the TCK's is highly recommended to make sure you are not breaking the project. The TCK's will also run on any Pull Request.

You can either run all the TCK's or just a subset by executing `mvn verify` in the `tcks` module root or each of the submodules. If you wish to run a particular test, you can use Maven `-Dtest=` property with the fully qualified name of the test class and optionally the method name by using `mvn verify -Dtest=fully.qualified.test.class.name#methodName`.

Test Coverage

Quarkus uses Jacoco to generate test coverage. If you would like to generate the report run `mvn install -Ptest-coverage`, then change into the `coverage-report` directory and run `mvn package`. The code coverage report will be generated in `target/site/jacoco/`.

This currently does not work on Windows as it uses a shell script to copy all the classes and files into the code coverage module.

If you just need a report for a single module, run `mvn install jacoco:report -Ptest-coverage` in that module (or with `-f ...`).

Extensions

Descriptions

Extensions descriptions (in the `runtime/pom.xml` description or in the YAML `quarkus-extension.yaml`) are used to describe the extension and are visible in https://code.quarkus.io and https://extensions.quarkus.io. Try and pay attention to it. Here are a few recommendation guidelines:

- keep it relatively short so that no hover is required to read it
- describe the function over the technology
- use an action / verb to start the sentence
- do not conjugate the action verb (`Connect foo` , not `Connects foo` nor `Connecting foo`)
- connectors (JDBC / reactive) etc. tend to start with Connect
- do not mention `Quarkus`
- do not mention `extension`
- avoid repeating the extension name

Bad examples and the corresponding good example:

- "AWS Lambda" (use "Write AWS Lambda functions")
- "Extension for building container images with Docker" (use "Build container images with Docker")
- "PostgreSQL database connector" (use "Connect to the PostgreSQL database via JDBC")
- "Asynchronous messaging for Reactive Streams" (use "Produce and consume messages and implement event driven and data streaming applications")

Update dependencies to extensions

When adding a new extension you should run `update-extension-dependencies.sh` so that special modules like `devtools/bom-descriptor-json` that are consuming this extension are built *after* the respective extension. Simply add to your commit the files that were changed by the script.

When removing an extension make sure to also remove all dependencies to it from all `pom.xml` . It's easy to miss this as long as the extension artifact is still present in your local Maven repository.

Check security vulnerabilities

When adding a new extension or updating the dependencies of an existing one, it is recommended to run in the extension directory the OWASP Dependency Check with `mvn -Dowasp-check` so that known security vulnerabilities in the extension dependencies can be detected early.

The small print

This project is an open source project, please act responsibly, be nice, polite and enjoy!

Frequently Asked Questions

- The Maven build fails with `OutOfMemoryException`

  Set Maven options to use more memory: `export MAVEN_OPTS="-Xmx4g"` .

- IntelliJ fails to import Quarkus Maven project with `java.lang.OutOfMemoryError: GC overhead limit exceeded`

  In IntelliJ IDEA if you see problems in the Maven view claiming `java.lang.OutOfMemoryError: GC overhead limit exceeded` that means the project import failed.

  See section `IDEA Setup` as there are different possible solutions described.

- IntelliJ does not recognize the project as a Java 17 project

  In the Maven pane, uncheck the `include-jdk-misc` and `compile-java8-release-flag` profiles

- Build hangs with DevMojoIT running infinitely

  ```
  ./mvnw clean install
  # Wait...
  [INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.192 s - in io.quarkus.maven.it.GenerateConfigIT
  [INFO] Running io.quarkus.maven.it.DevMojoIT
  ```

  DevMojoIT require a few minutes to run but anything more than that is not expected. Make sure that nothing is running on 8080.

- The native integration test for my extension didn't run in the CI

  In the interest of speeding up CI, the native build job `native-tests` have been split into multiple categories which are run in parallel. This means that each new integration test module needs to be configured explicitly in `native-tests.json` to have its integration tests run in native mode.

- Build aborts complaining about missing (or superfluous) `minimal *-deployment dependencies` or `illegal runtime dependencies`

  To ensure a consistent build order, even when building in parallel (`./mvnw -T...`) or building incrementally/partially (`./mvnw -pl...`), the build enforces the presence of certain dependencies. If those dependencies are not present, your local build will most likely use possibly outdated artifacts from you local repo and CI build might even fail not finding certain artifacts.

  Just do what the failing enforcer rule is telling you, and you should be fine.

- Build fails with multiple `This project has been banned from the build due to previous failures` messages

  Just scroll up, there should be an error or warning somewhere. Failing enforcer rules are known to cause such effects and in this case there'll be something like:

  ```
  [WARNING] Rule 0: ... failed with message:
  ...
  ```

- Tests fail with `Caused by: io.quarkus.runtime.QuarkusBindException: Port(s) already bound: 8080: Address already in use`

  Check that you do not have other Quarkus dev environments, apps or other web servers running on this default 8080 port.