

Quarkus - Class Loading Reference

This document explains the Quarkus class loading architecture. It is intended for extension authors and advanced users who want to understand exactly how Quarkus works.

The Quarkus class loading architecture is slightly different depending on the mode that the application is run in. When running a production application everything is loaded in the system `ClassLoader`, so it is a completely flat class path. This also applies to native image mode which does not really support multiple `ClassLoaders`, and is based on a normal production Quarkus application.

For all other use cases (e.g. tests, dev mode, and building the application) Quarkus uses the class loading architecture outlined here.

Reading Class Bytecode

Whenever class bytecode needs to be read, a `@BuildStep` where class reading is done must have the `loadsApplicationClasses` parameter set to `true`. Otherwise the class will not be found.

It's also important to use the correct `ClassLoader`. The recommended approach is to get it by calling the `Thread.currentThread().getContextClassLoader()` method.

Example:

```
@BuildStep(loadsApplicationClasses = true)
GeneratedClassBuildItem instrument(final CombinedIndexBuildItem
index) {
    final String classname = "com.example.SomeClass";
    final ClassLoader cl = Thread.currentThread()
        .getContextClassLoader();
    final byte[] originalBytecode = IoUtil.readClassAsBytes(cl,
classname);
    final byte[] enhancedBytecode = ... // class instrumentation
from originalBytecode
    return new GeneratedClassBuildItem(true, classname,
enhancedBytecode));
}
```

Bootstrapping Quarkus

All Quarkus applications are created by the `QuarkusBootstrap` class in the `independent-projects/bootstrap` module. This class is used to resolve all the relevant dependencies (both deployment and runtime) that are needed for the Quarkus application. The end result of this process is a `CuratedApplication`, which contains all the class loading information for the application.

The `CuratedApplication` can then be used to create an `AugmentAction` instance, which can create production application and start/restart runtime ones. This application instance exists within an isolated `ClassLoader`, it is not necessary to have any of the Quarkus deployment classes on the class path as the curate process will resolve them for you.

This bootstrap process should be the same no matter how Quarkus is launched, just with different parameters passed in.

Current Run Modes

At the moment we have the following use cases for bootstrapping Quarkus:

- Maven creating production application
- Maven dev mode
- Gradle creating a production application
- Gradle dev mode
- QuarkusTest (Maven, Gradle and IDE)
- QuarkusUnitTest (Maven, Gradle and IDE)
- QuarkusDevModeTest (Maven, Gradle and IDE)
- Arquillian Adaptor

One of the goals of this refactor is to have all these different run modes boot Quarkus in fundamentally the same way.

Notes on Transformer Safety

A `ClassLoader` is said to be 'transformer safe' if it is safe to load classes in the class loader before the transformers are ready. Once a class has been loaded it cannot be changed, so if a class is loaded before the transformers have been prepared this will prevent the transformation from working. Loading classes in a transformer safe `ClassLoader` will not prevent the transformation, as the loaded class is not used at runtime.

ClassLoader Implementations

Quarkus has the following `ClassLoaders`:

Base ClassLoader

This is usually the normal JVM System `ClassLoader`. In some environments such as Maven it may be different. This `ClassLoader` is used to load the bootstrap classes, and other `ClassLoader` instances will delegate the loading of JDK classes to it.

Augment ClassLoader

This loads all the `-deployment` artifacts and their dependencies, as well as other user dependencies. It does not load the application root or any hot deployed code. This `ClassLoader` is persistent, even if the application restarts it will remain (which is why it cannot load application

classes that may be hot deployed). Its parent is the base ClassLoader, and it is transformer safe.

At present this can be configured to delegate to the Base ClassLoader, however the plan is for this option to go away and always have this as an isolated ClassLoader. Making this an isolated ClassLoader is complicated as it means that all the builder classes are isolated, which means that use cases that want to customise the build chains are slightly more complex.

Deployment ClassLoader

This can load all application classes, its parent is the Augment ClassLoader so it can also load all deployment classes.

This ClassLoader is non-persistent, it will be re-created when the application is started, and is isolated. This ClassLoader is the context ClassLoader that is used when running the build steps. It is also transformer safe.

Base Runtime ClassLoader

This loads all the runtime extension dependencies, as well as other user dependencies (note that this may include duplicate copies of classes also loaded by the Augment ClassLoader). It does not load the application root or any hot deployed code. This ClassLoader is persistent, even if the application restarts it will remain (which is why it cannot load application classes that may be hot deployed). Its parent is the base ClassLoader.

This loads code that is not hot-reloadable, but it does support transformation (although once the class is loaded this transformation is no longer possible). This means that only transformers registered in the first application start will take effect, however as these transformers are expected to be idempotent this should not cause problems. An example of the sort of transformation that might be required here is a Panache entity packaged in an external jar. This class needs to be transformed to have its static methods implemented, however this transformation only happens once, so restarts use the copy of the class that was created on the first start.

This ClassLoader is isolated from the Augment and Deployment ClassLoaders. This means that it is not possible to set values in a static field in the deployment side, and expect to read it at runtime. This allows dev and test applications to behave more like a production application (production applications are isolated in that they run in a whole new JVM).

This also means that the runtime version can be linked against a different set of dependencies, e.g. the hibernate version used at deployment time might want to include ByteBuddy, while the version used at runtime does not.

Runtime Class Loader

This ClassLoader is used to load the application classes and other hot deployable resources. Its parent is the base runtime ClassLoader, and it is recreated when the application is restarted.

Isolated ClassLoaders

The runtime ClassLoader is always isolated. This means that it will have its own copies of almost every class from the resolved dependency list. The exception to this are:

- JDK classes

- Classes from artifacts that extensions have marked as parent first (more on this later).

Parent First Dependencies

There are some classes that should not be loaded in an isolated manner, but that should always be loaded by the system ClassLoader (or whatever ClassLoader is responsible for bootstrapping Quarkus). Most extensions do not need to worry about this, however there are a few cases where this is necessary:

- Some logging related classes, as logging must be loaded by the system ClassLoader
- Quarkus bootstrap itself

If this is required it can be configured in the `quarkus-bootstrap-maven-plugin`. Note that if you mark a dependency as parent first then all of its dependencies must also be parent first, or a `LinkageError` can occur.

```
<plugin>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-bootstrap-maven-plugin</artifactId>
  <configuration>
    <parentFirstArtifacts>
      <parentFirstArtifact>io.quarkus:quarkus-bootstrap-
core</parentFirstArtifact>
      <parentFirstArtifact>io.quarkus:quarkus-development-
mode-spi</parentFirstArtifact>
      <parentFirstArtifact>org.jboss.logmanager:jboss-
logmanager-embedded</parentFirstArtifact>
      <parentFirstArtifact>org.jboss.logging:jboss-
logging</parentFirstArtifact>
      <parentFirstArtifact>
org.ow2.asm:asm</parentFirstArtifact>
    </parentFirstArtifacts>
  </configuration>
</plugin>
```

Banned Dependencies

There are some dependencies that we can be sure we do not want. This generally happens when a dependency has had a name change (e.g. smallrye-config changing groups from `org.smallrye` to `org.smallrye.config`, the `javax` → `jakarta` rename). This can cause problems, as if these artifacts end up in the dependency tree out of date classes can be loaded that are not compatible with Quarkus. To deal with this extensions can specify artifacts that should never be loaded. This is done by modifying the `quarkus-bootstrap-maven-plugin` config in the pom (which generates the `quarkus-extension.properties` file). Simply add an `excludedArtifacts` section as shown below:

```
<plugin>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-bootstrap-maven-plugin</artifactId>
  <configuration>
    <excludedArtifacts>
      <excludedArtifact>io.smallrye:smallrye-
config</excludedArtifact>
      <excludedArtifact>javax.enterprise:cdi-
api</excludedArtifact>
    </excludedArtifacts>
  </configuration>
</plugin>
```

This should only be done if the extension depends on a newer version of these artifacts. If the extension does not bring in a replacement artifact as a dependency then classes the application needs might end up missing.