Quarkus - Application Initialization and Termination

You often need to execute custom actions when the application starts and clean up everything when the application stops. This guide explains how to:

- Write a Quarkus application with a main method

- Write command mode applications that run a task and then terminate

- Be notified when the application starts

- Be notified when the application stops

Prerequisites

To complete this guide, you need:

- less than 10 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.6.3

Solution

We recommend that you follow the instructions in the next sections and create the application 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 `lifecycle-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=lifecycle-quickstart \
    -DclassName="org.acme.lifecycle.GreetingResource" \
    -Dpath="/hello"
cd lifecycle-quickstart
```

It generates:

- the Maven structure

- a landing page accessible on `http://localhost:8080`

- example `Dockerfile` files for both `native` and `jvm` modes

- the application configuration file

- an `org.acme.lifecycle.GreetingResource` resource

- an associated test

The main method

By default Quarkus will automatically generate a main method, that will bootstrap Quarkus and then just wait for shutdown to be initiated. Let's provide our own main method:

```
package om.acme;

import io.quarkus.runtime.annotations.QuarkusMain;
import io.quarkus.runtime.Quarkus;

@QuarkusMain    ①
public class Main {

    public static void main(String ... args) {
        System.out.println("Running main method");
        Quarkus.run(args);  ②
    }
}
```

① This annotation tells Quarkus to use this as the main method, unless it is overridden in the config

② This launches Quarkus

This main class will bootstrap Quarkus and run it until it stops. This is no different to the automatically generated main class, but has the advantage that you can just launch it directly from the IDE without needing to run a Maven or Gradle command.

> ⚠️ It is not recommended to do any business logic in this main method, as Quarkus has not been set up yet, and Quarkus may run in a different ClassLoader. If you want to perform logic on startup use an `io.quarkus.runtime.QuarkusApplication` as described below.

If we want to actually perform business logic on startup (or write applications that complete a task and then exit) we need to supply a `io.quarkus.runtime.QuarkusApplication` class to the run method. After Quarkus has been started the `run` method of the application will be invoked. When this method returns the Quarkus application will exit.

If you want to perform logic on startup you should call `Quarkus.waitForExit()`, this method will wait until a shutdown is requested (either from an external signal like when you press `Ctrl+C` or because a thread has called `Quarkus.asyncExit()`).

An example of what this looks like is below:

```
package com.acme;

import io.quarkus.runtime.Quarkus;
import io.quarkus.runtime.QuarkusApplication;
import io.quarkus.runtime.annotations.QuarkusMain;

@QuarkusMain
public class Main {
    public static void main(String... args) {
        Quarkus.run(MyApp.class, args);
    }

    public static class MyApp implements QuarkusApplication {

        @Override
        public int run(String... args) throws Exception {
            System.out.println("Do startup logic here");
            Quarkus.waitForExit();
            return 0;
        }
    }
}
```

Injecting the command line arguments

It is possible to inject the arguments that were passed in on the command line:

```
@Inject
@CommandLineArguments
String[] args;
```

Listening for startup and shutdown events

Create a new class named `AppLifecycleBean` (or pick another name) in the `org.acme.lifecycle` package, and copy the following content:

```
package org.acme.lifecycle;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;
import org.jboss.logging.Logger;

@ApplicationScoped
public class AppLifecycleBean {

    private static final Logger LOGGER =
Logger.getLogger("ListenerBean");

    void onStart(@Observes StartupEvent ev) {               ①
        LOGGER.info("The application is starting...");
    }

    void onStop(@Observes ShutdownEvent ev) {               ②
        LOGGER.info("The application is stopping...");
    }

}
```

1. Method called when the application is starting

2. Method called when the application is terminating

> The events are also called in *dev mode* between each redeployment.

> The methods can access injected beans. Check the AppLifecycleBean.java class for details.

What is the difference from @Initialized(ApplicationScoped.class) and @Destroyed(ApplicationScoped.class)

In the JVM mode, there is no real difference, except that StartupEvent is always fired **after** @Initialized(ApplicationScoped.class) and ShutdownEvent is fired **before** @Destroyed(ApplicationScoped.class). For a native executable build, however, @Initialized(ApplicationScoped.class) is fired as **part of the native build process**, whereas StartupEvent is fired when the native image is executed. See Three Phases of Bootstrap and Quarkus Philosophy for more details.

> ℹ️ In CDI applications, an event with qualifier `@Initialized(ApplicationScoped.class)` is fired when the application context is initialized. See the spec for more info.

Using `@Startup` to initialize a CDI bean at application startup

A bean represented by a class, producer method or field annotated with `@Startup` is initialized at application startup:

```
package org.acme.lifecycle;

import javax.enterprise.context.ApplicationScoped;

@Startup ①
@ApplicationScoped
public class EagerAppBean {

    private final String name;

    EagerAppBean(NameGenerator generator) { ②
        this.name = generator.createName();
    }
}
```

1. For each bean annotated with `@Startup` a synthetic observer of `StartupEvent` is generated. The default priority is used.

2. The bean constructor is called when the application starts and the resulting contextual instance is stored in the application context.

> ℹ️ `@Dependent` beans are destroyed immediately afterwards to follow the behavior of observers declared on `@Dependent` beans.

> 💡 If a class is annotated with `@Startup` but with no scope annotation then `@ApplicationScoped` is added automatically.

Package and run the application

Run the application with: `./mvnw compile quarkus:dev`, the logged message is printed. When the application is stopped, the second log message is printed.

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file. You can also generate the native executable using `./mvnw clean package -Pnative`.

Launch Modes

Quarkus has 3 different launch modes, `NORMAL` (i.e. production), `DEVELOPMENT` and `TEST`. If you are running `quarkus:dev` then the mode will be `DEVELOPMENT`, if you are running a JUnit test it will be `TEST`, otherwise it will be `NORMAL`.

Your application can get the launch mode by injecting the `io.quarkus.runtime.LaunchMode` enum into a CDI bean, or by invoking the static method `io.quarkus.runtime.LaunchMode.current()`.

Graceful Shutdown

Quarkus includes support for graceful shutdown, this allows Quarkus to wait for running requests to finish, up till a set timeout. By default this is disabled, however you can configure this by setting the `quarkus.shutdown.timeout` config property. When this is set shutdown will not happen until all running requests have completed, or until this timeout has elapsed. This config property is a duration, and can be set using the standard `java.time.Duration` format, if only a number is specified it is interpreted as seconds.

Extensions that accept requests need to add support for this on an individual basis. At the moment only the HTTP extension supports this, so shutdown may still happen when messaging requests are active.