

Quarkus - Scripting with Quarkus

Quarkus provides integration with [jbang](#) which allows you to write Java scripts/applications requiring no Maven nor Gradle to get running.

In this guide, we will see how you can write a REST application using just a single Java file.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

Prerequisites

To complete this guide, you need:

- less than 5 minutes
- [jbang v0.40.3+](#)
- an IDE
- GraalVM installed if you want to run in native mode

Solution

Normally we would link to a Git repository to clone but in this case there is no additional files than the following:

```

//usr/bin/env jbang "$0" "$@" ; exit $?
//DEPS io.quarkus:quarkus-resteasy:1.8.2.Final
//JAVAC_OPTIONS -parameters
//JAVA_OPTIONS
-Djava.util.logging.manager=org.jboss.logmanager.LogManager

import io.quarkus.runtime.Quarkus;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.jboss.resteasy.annotations.jaxrs.PathParam;
import org.jboss.logging.Logger;

@Path("/hello")
@ApplicationScoped
public class quarkusapp {

    @GET
    public String sayHello() {
        return "hello";
    }

    public static void main(String[] args) {
        Quarkus.run(args);
    }

    @Inject
    GreetingService service;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    public String greeting(@PathParam String name) {
        return service.greeting(name);
    }

    @ApplicationScoped
    static public class GreetingService {

        public String greeting(String name) {
            return "hello " + name;
        }
    }
}

```

Architecture

In this guide, we create a straightforward application serving a **hello** endpoint with a single source file, no additional build files like **pom.xml** or **build.gradle** needed. To demonstrate dependency injection, this endpoint uses a **greeting** bean.



Creating the initial file

First, we need a Java file. jbang lets you create an initial version using:

```
jbang init scripting/quarkusapp.java  
cd scripting
```

This command generates a .java file that you can directly run on Linux and macOS, i.e. **./quarkusapp.java** - on Windows you need to use **jbang quarkusapp.java**.

This initial version will print **Hello World** when run.

Once generated, look at the **quarkusapp.java**.

You will find at the top a line looking like this:

```
//usr/bin/env jbang "$0" "$@" ; exit $?
```

This line is what on Linux and macOS allows you to run it as a script. On Windows this line is ignored.

The next line

```
// //DEPS <dependency1> <dependency2>
```

Is illustrating how you add dependencies to this script. This is a feature of **jbang**.

Go ahead and update this line to include the **quarkus-resteasy** dependency like so:

```
//DEPS io.quarkus:quarkus-resteasy:1.8.2.Final
```

Now, run **jbang quarkusapp.java** and you will see **jbang** resolving this dependency and building the jar with help from Quarkus' jbang integration.

```
$ jbang quarkusapp.java

[jbang] Resolving dependencies...
[jbang]     Resolving io.quarkus:quarkus-
resteasy:1.8.2.Final...Done
[jbang] Dependencies resolved
[jbang] Building jar...
[jbang] Post build with io.quarkus.launcher.JBangIntegration
Aug 30, 2020 5:40:55 AM org.jboss.threads.Version <clinit>
INFO: JBoss Threads version 3.1.1.Final
Aug 30, 2020 5:40:56 AM io.quarkus.deployment.QuarkusAugmentor run
INFO: Quarkus augmentation completed in 722ms
Hello World
```

For now the application does nothing new.



How do I edit this file and get content assist?

As there is nothing but a `.java` file, most IDE's don't handle content assist well. To work around that you can run `jbang edit quarkusapp.java`, this will print out a directory that will have a temporary project setup you can use in your IDE.

On Linux/macOS you can run `<idecommand> `jbang edit quarkusapp.java``.

If you add dependencies while editing you can get jbang to automatically refresh the IDE project using `jbang edit --live=<idecommand> quarkusapp.java`.

The JAX-RS resources

Now let us replace the class with one that uses Quarkus features:

```
import io.quarkus.runtime.Quarkus;
import javax.enterprise.context.ApplicationScoped;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/hello")
@ApplicationScoped
public class quarkusapp {

    @GET
    public String sayHello() {
        return "hello";
    }

    public static void main(String[] args) {
        Quarkus.run(args);
    }
}
```

It's a very simple class with a main method that starts Quarkus with a REST endpoint, returning "hello" to requests on "/hello".



*Why is the **main** method there?*

A **main** method is currently needed for the **jbang** integration to work - we might remove this requirement in the future.

Running the application

Now when you run the application you will see Quarkus start up.

Use: **jbang quarkusapp.java**:

```
[jbang] Building jar...
[jbang] Post build with io.quarkus.launcher.JBangIntegration
Aug 30, 2020 5:49:01 AM org.jboss.threads.Version <clinit>
INFO: JBoss Threads version 3.1.1.Final
Aug 30, 2020 5:49:02 AM io.quarkus.deployment.QuarkusAugmentor run
INFO: Quarkus augmentation completed in 681ms

--  -----
--/  _  \ / / / / _ | / _ \ / / _ / / / _ /
-/ / _ / / / / / _ | / , _ / , < / / _ / \ \
--\ _ \ \ _ \ _ \ / _ | _ / | _ / | _ | \ _ \ _ \
2020-08-30 05:49:03,255 INFO    [io.quarkus] (main) Quarkus
1.8.2.Final on JVM started in 0.638s. Listening on:
http://0.0.0.0:8080
2020-08-30 05:49:03,272 INFO    [io.quarkus] (main) Profile prod
activated.
2020-08-30 05:49:03,272 INFO    [io.quarkus] (main) Installed
features: [cdi, resteasy]
```

```
$ curl -w "\n" http://localhost:8080/hello
hello
```



We are using `curl -w "\n"` in this example to avoid your terminal printing a '%' or put both result and next command prompt on the same line.



In this second run you should not see a line saying it is resolving `quarkus-resteasy` as jbang caches the dependency resolution between runs. If you want to clear the caches to force resolution use `jbang cache clear`.

Let's modify the application and add a companion bean.

Normally you would add a separate class, but as we are aiming to have it all in one file you will add a nested class.

Add the following **inside** the `quarkusapp` class body.

```
@ApplicationScoped
static public class GreetingService {

    public String greeting(String name) {
        return "hello " + name;
    }

}
```

Use of nested static public classes

We are using a nested static public class instead of a top level class for two reasons:



1. jbang currently does not support multiple source files.
2. All Java frameworks relying on introspection have challenges using top level classes as they are not as visible as public classes; and in Java there can only be one top level public class in a file.

Edit the `quarkusapp` class to inject the `GreetingService` and create a new endpoint using it, you should end up with something like:

```

//usr/bin/env jbang "$0" "$@" ; exit $?
//DEPS io.quarkus:quarkus-resteasy:1.8.2.Final

import io.quarkus.runtime.Quarkus;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

@Path("/hello")
@ApplicationScoped
public class quarkusapp {

    @GET
    public String sayHello() {
        return "hello from Quarkus with jbang.dev";
    }

    public static void main(String[] args) {
        Quarkus.run(args);
    }

    @Inject
    GreetingService service;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    public String greeting(@PathParam String name) {
        return service.greeting(name);
    }

    @ApplicationScoped
    static public class GreetingService {

        public String greeting(String name) {
            return "hello " + name;
        }
    }
}

```

Now when you run `jbang quarkusapp.java` you can check what the new end point returns:


```
$ curl -w "\n" http://localhost:8080/hello/greeting/quarkus
hello null
```

Now that is unexpected, why is it returning `hello null` and not `hello quarkus`?

The reason is that JAX-RS `@PathParam` relies on the `-parameters` compiler flag to be set to be able to map `{name}` to the `name` parameter.

We fix that by adding the following comment instruction to the file:

```
//JAVAC_OPTIONS -parameters
```

Now when you run with `jbang quarkusapp.java` the end point should return what you expect:

```
$ curl -w "\n" http://localhost:8080/hello/greeting/quarkus
hello quarkus
```

Debugging

To debug the application you use `jbang --debug quarkusapp.java` and you can use your IDE to connect on port 4004; if you want to use the more traditional Quarkus debug port you can use `jbang --debug=5005 quarkusapp.java`.

Note: `jbang` debugging always suspends thus you need to connect the debugger to have the application run.

Logging

To use logging in Quarkus scripting with `jbang` you do as usual, with configuring a logger, i.e.

```
public static final Logger LOG =
    Logger.getLogger(quarkusapp.class);
```

To get it to work you need to add a Java option to ensure the logging is initialized properly, i.e.

```
//JAVA_OPTIONS
-Djava.util.logging.manager=org.jboss.logmanager.LogManager
```

With that in place running `jbang quarkusapp.java` will log and render as expected.

Configuring Application

You can use `//Q:CONFIG <property>=<value>` to set up static configuration for your application.

I.e. if you wanted to add the `smallrye-openapi` and `swagger-ui` extensions and have the Swagger UI always show up you would add the following:

```
//DEPS io.quarkus:quarkus-smallrye-openapi:1.8.2.Final
//DEPS io.quarkus:quarkus-swagger-ui:1.8.2.Final
//Q:CONFIG quarkus.swagger-ui.always-include=true
```

Now during build the `quarkus.swagger-ui.always-include` will be generated into the resulting jar and <http://0.0.0.0:8080/swagger-ui> will be available when run.

Running as a native application

If you have the `native-image` binary installed and `GRAALVM_HOME` set, you can get the native executable built and run using `jbang --native quarkusapp.java`:

```
$ jbang --native quarkusapp

[jbang] Building jar...
[jbang] Post build with io.quarkus.launcher.JBangIntegration
Aug 30, 2020 6:21:15 AM org.jboss.threads.Version <clinit>
INFO: JBoss Threads version 3.1.1.Final
Aug 30, 2020 6:21:16 AM
io.quarkus.deployment.pkg.steps.JarResultBuildStep
buildNativeImageThinJar
INFO: Building native image source jar:
/var/folders/yb/sytszfld4sg8vwr1h0w20jlw0000gn/T/quarkus-
jbang3291688251685023074/quarkus-application-native-image-source-
jar/quarkus-application-runner.jar
Aug 30, 2020 6:21:16 AM
io.quarkus.deployment.pkg.steps.NativeImageBuildStep build
INFO: Building native image from
/var/folders/yb/sytszfld4sg8vwr1h0w20jlw0000gn/T/quarkus-
jbang3291688251685023074/quarkus-application-native-image-source-
jar/quarkus-application-runner.jar
Aug 30, 2020 6:21:16 AM
io.quarkus.deployment.pkg.steps.NativeImageBuildStep
checkGraalVMVersion
INFO: Running Quarkus native-image plugin on GraalVM Version 20.1.0
(Java Version 11.0.7)
Aug 30, 2020 6:21:16 AM
io.quarkus.deployment.pkg.steps.NativeImageBuildStep build
```

```
INFO: /Users/max/.sdkman/candidates/java/20.1.0.r11-gr1/bin/native-
image -J-Djava.util.logging.manager=org.jboss.logmanager.LogManager
-J-Dsun.nio.ch.maxUpdateArraySize=100 -J-Dvertx.logger-delegate
-factory-class
-name=io.quarkus.vertx.core.runtime.VertxLogDelegateFactory -J
-Dvertx.disableDnsResolver=true -J
-Dio.netty.leakDetection.level=DISABLED -J
-Dio.netty allocator.maxOrder=1 -J-Duser.language=en -J
-Dfile.encoding=UTF-8 --initialize-at-build-time=
-H:InitialCollectionPolicy=com.oracle.svm.core.genscavenge.Collecti
onPolicy\$$BySpaceAndTime -H:+JNI -jar quarkus-application-
runner.jar -H:FallbackThreshold=0 -H:+ReportExceptionStackTraces
-H:-AddAllCharsets -H:EnableURLProtocols=http --no-server -H:
-UseServiceLoaderFeature -H:+StackTrace quarkus-application-runner
```

```
Aug 30, 2020 6:22:31 AM io.quarkus.deployment.QuarkusAugmentor run
INFO: Quarkus augmentation completed in 76010ms
```

```
--  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
--/  _  \ / / / / _ | / _ \ / / / / / / _ /
-/  / / / / / / _ | / , _ / , < / / / / \ \
--\ _ \ \ _ \ / / | _ / | _ / | _ | \ _ \ / _ \
2020-08-30 06:22:32,012 INFO  [io.quarkus] (main) Quarkus
1.8.2.Final native started in 0.017s. Listening on:
http://0.0.0.0:8080
2020-08-30 06:22:32,013 INFO  [io.quarkus] (main) Profile prod
activated.
2020-08-30 06:22:32,013 INFO  [io.quarkus] (main) Installed
features: [cdi, resteasy]
```

This native build will take some time on first run but any subsequent runs (without changing `quarkusapp.java`) will be close to instant thanks to jbang cache:

```
$ jbang --native quarkusapp.java

--  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
--/  _  \ / / / / _ | / _ \ / / / / / / _ /
-/  / / / / / / _ | / , _ / , < / / / / \ \
--\ _ \ \ _ \ / / | _ / | _ / | _ | \ _ \ / _ \
2020-08-30 06:23:36,846 INFO  [io.quarkus] (main) Quarkus
1.8.2.Final native started in 0.015s. Listening on:
http://0.0.0.0:8080
2020-08-30 06:23:36,846 INFO  [io.quarkus] (main) Profile prod
activated.
2020-08-30 06:23:36,846 INFO  [io.quarkus] (main) Installed
features: [cdi, resteasy]
```

Conclusion

If you want to get started with Quarkus or write something quickly, Quarkus Scripting with jbang lets you do that. No Maven, no Gradle - just a Java file. In this guide we outlined the very basics on using Quarkus with jbang; if you want to learn more about what jbang can do, go see <https://jbang.dev>.