Cheat Sheet

Quarkus – the Kubernetes Native Java Framework

by Alex Soto

Start-Up

What is Quarkus?

Quarkus is a Kubernetes Native Java stack tailored for GraalVM & OpenJDK HotSpot that makes Java programs run 10X faster and 100X smaller. It also improves the developer experience by adding features like live reloading/debugging and persistence with Panache.

Requirements

- Java 8
- Maven 3.5.3+ (recommended Maven 3.6.X)
- GraalVM 19 or Docker (optional for native compilation)

Scaffolding a Quarkus application

You can create a minimal Quarkus application using a Maven goal. This goal creates a minimal endpoint, a test and Docker resources:

```
mvn io.quarkus:quarkus-maven-plugin:0.19.1:create \
    -DprojectGroupId=io.alexsoto \
    -DprojectArtifactId=qa \
    -DclassName="org.alexsoto.quickstart.GreetingResource" \
    -Dpath="/hello"
```

Start Quarkus in live reload (dev) mode:

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

You can request the provided endpoint:

```
curl http://localhost:8080/hello
```

Live Reload

You can make any change on any (java) resource and, if the Quarkus application was started in live reload mode, the change will be automatically reflected without having to recompile or repackage the application. Try it by changing the output value of the `src/main/java/org/alexsoto/quickstart/GreetingResource.java` class.

Packaging

Run `./mvnw package` to generate a `jar` file called `qa-1.0-SNAPSHOT-runner.jar` that contains the application and its directory containing required dependencies for running the application. Quarkus, unlike other solutions, does not create a fat jar so as to be Docker layers friendly. Both jar file and `lib` folder should be placed together in the same directory.

To create a native executable, you can either install GraalVM locally and produce an executable for the installed platform, or you can use Docker to produce a Docker container running a native executable of the application:

```
./mvnw package -Pnative -Dnative-image.docker-build=true
```

Then you can build the Docker container and run it:

```
docker build -f src/main/docker/Dockerfile.native -t quarkus/jax .
docker run -i --rm -p 8080:8080 quarkus/jax
```


Basics

JAX-RS

Quarkus uses JAX-RS to define REST-ful web APIs:

```
@Path("/book")
public class BookResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> getAllBooks() {
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response createBook(Book book) {
    }

    @DELETE
    @Path("/{isbn}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteBook(@PathParam("isbn") String isbn) {
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/search")
    public Response searchBook(@QueryParam("description") String description) {
    }

}
```

Valid HTTP method annotations are: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@PATCH`, `@HEAD`, and `@HTTPDEL`.

Valid annotations to get information from request: `@PathParam`, `@QueryParam`, `@FormParam`, `@MatrixParam`, `@CookieParam`, and `@HeaderParam`.

You can inject request information as parameter, method or field by using `@Context` annotation.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public Response getAllBooks(UriContext UriInfo uriInfo) {
    return Response.ok(uriInfo.getBaseUri().toString());
}
```

Classes that can be injected with `@Context`: SecurityContext, Request, Application, Configuration, Providers, ResourceContext, ServletConfig, ServletContext, HttpServlet^{Request}, HttpServlet^{Response}, HttpHeaders, UriInfo, SecurityEvent and `Sse`.

JSON Marshaling/Unmarshaling

To marshal and unmarshal a java object to JSON format, you need to register the JSON-B extension.

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

Configuration

Quarkus centralizes all the configuration into a single file at `src/main/resources/application.properties` where you can configure Quarkus extensions or add custom configuration properties for the application.

```
greetings.message=Hello World!
```

```
@ConfigProperty(name = "greetings.message")
String message;
```

You can have multiple configurations in the same file with a prefix for the environment. The syntax for this is `%{profile}.config.key=value`

```
quarkus.http.port=9090
%dev.quarkus.http.port=8181
```

HTTP port will be 9090, unless the 'dev' profile is active. Default profiles are `dev` (when running `quarkus:dev` goal), `test` (when running tests) and `prod` (when not the others).

You can create a custom profile, which you can set either in the `quarkus-profile` system property or the `QUARKUS_PROFILE` environment variable (i.e. `quarkus.profile=staging`).

```
quarkus.http.port=9090
%staging.quarkus.http.port=9999
```

Properties can be overridden by setting values as an environment variable (`QUARKUS_HTTP_PORT`), system property (`-Dquarkus.http.port`) or by creating an `application.properties` file in the `config` directory under the current working directory.

CDI

Quarkus is based on CDI 2.0 to implement injection of code. Although most used features are implemented, the whole specification is not supported.

```
@ApplicationScoped
public class GreetingService {
    public String message(String msg) {
        return msg.toUpperCase();
    }
}
```

Scope annotation is mandatory to make the bean discoverable.

```
@Inject
GreetingService greetingService;
```

Testing

Quarkus works with JUnit 4 and 5 and integrates smoothly to Rest-Assured. By default, the project is scaffolded with JUnit 5 dependencies. Annotate a test with `@QuarkusTest` to start the application before the test is executed.

```
@QuarkusTest
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
          .when().get("/hello")
          .then()
             .statusCode(200)
             .body("hello");
    }
}
```

The `quarkus.http.test.port` property can be used to set a listening port for an application deployed on the testing phase.

Logging

You can configure how Quarkus logs:

```
quarkus.log.console.enable=true
quarkus.log.console.level=DEBUG
quarkus.log.console.color=false
quarkus.log.category."com.jordirh.tests".level=DEBUG
```

Advanced Use Cases

Asynchronous

You can run asynchronous code by returning `CompletionStage` as return type:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public CompletionStage<String> hello() {
    return CompletableFuture.supplyAsync(() -> {
        return "Hello";
    });
}
```

Reactive

To start using reactive streams (MicroProfile Reactive Streams and Rxjava 2) you need to add the reactive extension:

```
./mvnw quarkus:add-extension -Dextensions="smallrye-reactive-streams-operators"
```

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public CompletionStage<String> hello() {
    return ReactiveStreams.of("A", "B", "C", "D", "E")
        .map(String::toUpperCase)
        .flatMap(s -> {
            int[] ints = {1, 2, 3};
            return ReactiveStreams.fromIterable(Arrays.asList(ints));
        })
        .map(i -> Integer.toString(i));
}
```

You can stream continuously by using `org.reactivestreams.Publisher`.

```
@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
public Publisher<String> getAllBooks(@Context Sse sseInfo) {
    return Flowable
        .interval(500, TimeUnit.MILLISECONDS)
        .map(s -> atomicInteger.getAndIncrement())
        .map(i -> Integer.toString(i));
}
```

Hibernate and Panache

Quarkus works with JPA (Hibernate) as a persistence solution, but also provides an Active Record and DAO pattern implementation under the Panache project, so writing persistence code is a simple task.

```
./mvnw quarkus:add-extension -Dextensions="jdbc-postgresql, resteasy-jsonb, hibernate-orm-panache, hibernate-validator"
```

Configure JDBC parameters in `src/main/resources/application.properties`:

```
quarkus.datasource.url=jdbc:postgresql://localhost:5432/mydatabase
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=developer
quarkus.datasource.password=developer
quarkus.hibernate-orm.database.generation=update
```

Create `Entity` object.

```
@Entity
public class Book extends PanacheEntity {

    @NotNull
    @Column(unique = true)
    public String isbn;

    @Column
    public String title;

    @Column
    public String description;

}
```

And finally, a resource object using `Repository` methods.
Some are methods per instance, and others are static methods:

```
@Path("/book")
public class BookResource {

    @Context UriInfo uriInfo;

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Transactional
    public Response createBook(Book book) {

        book.persist();

        return Response
            .created(uriInfo.getBaseUriBuilder()
                    .path(BookResource.class)
                    .path(Long.toString(book.id))
                    .build())
            .build();

    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{isbn}")
    public Response findBookByPathParam(@Path("isbn") String isbn) {
        Book book = Book.find("isbn", isbn).firstResult();

        return Response.ok(book).build();
    }

}
```

Kubernetes

Quarkus can use the project `Dekorate` to generate an opinionated base Kubernetes resource.

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

When running the `./mvnw package` the Kubernetes resources are created in the `target/wiring-classes/META-INF/kubernetes/` directory.

Alex Soto

Alex is a Software Engineer at Red Hat in the developers group. He is passionate about the Java world and software automation, and he believes in the open source software model. Alex is the creator of the NoSQLUnit project, a member of the JSR374 Java API for JSON Processing) Expert Group, co-author of the Testing Java Microservices book, and has been contributing to many open source projects. A Java Champion since 2017 and international speaker, he has talked about new testing techniques for microservices and continuous delivery in the 21st century.

Illustrations: © Software & Support Media

jax LONDON

www.jaxlondon.com