Quarkus for Spring Developers

Comparison Spring V Quarkus

Feature	**Spring Boot**	**Quarkus**
Maturity & Ecosystem	Very mature, massive ecosystem, vast community support	Newer but growing fast; Red Hat-supported ecosystem
Developer Experience	Rich tooling, Spring Initializr, strong IDE support	Fast reload with Dev Mode, less boilerplate

Startup Time	Moderate (suitable for VMs, less for serverless)	Very fast (optimized for containers and serverless)
Memory Footprint	Higher	Lower
Runtime	JVM-based	JVM + Native Image (GraalVM)
Native Compilation	Supported via Spring Native (still maturing)	First-class with GraalVM out of the box
Dependency Injection	Spring DI (powerful, feature-rich)	CDI (Context and Dependency Injection - Jakarta EE standard)
Reactive Programming	WebFlux (mature, but optional)	Built-in support via Mutiny (vert.x based)
Kubernetes Readiness	Requires extensions/plugins	Kubernetes-native, with config and OpenShift support
Build Tools	Maven, Gradle	Maven, Gradle
Extensions	Starter dependencies	Quarkus Extensions (focused and optimized)
Microservice Footprint	30–80 MB typical	10–20 MB typical
Hot Reload / Dev Experience	Spring Dev Tools, JRebel, LiveReload	Quarkus Dev Mode (very fast, built-in)
Best Use Cases	Enterprise apps, wide framework support	Cloud-native apps, serverless, fast startup use cases

Hands On Lab

RESTful web service with Quarkus

1. Create Your Project

You can use the Quarkus code generator or do it via CLI:

```
mvn io.quarkus:quarkus-maven-plugin:create \
    -DprojectGroupId=com.example \
    -DprojectArtifactId=hello-quarkus \
    -DclassName="com.example.GreetingResource" \
    -Dpath="/hello" \
    -Dextensions="resteasy-reactive, jackson"

```

This creates a REST service with:

- RESTEasy Reactive (Quarkus's preferred REST API stack)
- Jackson for JSON serialization

2. Navigate to Project Directory

```
cd hello-quarkus

```

3. Create a Resource Class

This was already generated, but you can customize it:

```java
package com.example;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Greeting hello() {
        return new Greeting("Hello from Quarkus!");
    }

    public record Greeting(String message) {}
}

```

4. Run the Application

In dev mode (hot-reload included):

```
./mvnw quarkus:dev

```

5. Test the Endpoint

Open your browser or use `curl`:

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

```

Expected response:

```
{
  "message": "Hello from Quarkus!"
}

```

---

6. Optional: Add a POST Endpoint

To accept data:

```java
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Greeting echo(Greeting input) {
    return new Greeting("You said: " + input.message());
}

```

Test with:

```
curl -X POST http://localhost:8080/hello \
     -H "Content-Type: application/json" \
     -d '{"message":"Hi there!"}'

```

Add persistence with a database

Goal

- Add PostgreSQL and Hibernate Panache dependencies
- Configure database connection
- Create a JPA entity and repository
- Expose endpoints for CRUD operations

1. Add Extensions

Run:

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

```

or add manually to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jdbc-postgresql</artifactId>
</dependency>

```

---

2. Configure PostgreSQL in `application.properties`

Edit `src/main/resources/application.properties`:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=postgres
quarkus.datasource.password=postgres
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/yourdb
quarkus.hibernate-orm.database.generation=update
quarkus.hibernate-orm.log.sql=true

```

> ⚠️ Replace `yourdb`, `username`, and `password` with your actual values.

3. Create a JPA Entity

```
package com.example;

import io.quarkus.hibernate.orm.panache.PanacheEntity;
import jakarta.persistence.Entity;

```

```java
@Entity
public class Person extends PanacheEntity {
    public String name;
    public int age;
}

```

Note: `PanacheEntity` gives you an `id` field and basic CRUD methods.

4. Create a Resource (REST API)

```java
package com.example;

import jakarta.transaction.Transactional;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import java.util.List;

@Path("/people")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class PersonResource {

    @GET
    public List<Person> list() {
        return Person.listAll();
    }

    @POST
    @Transactional
    public Person create(Person person) {
        person.persist();
        return person;
    }

    @GET
    @Path("/{id}")
    public Person get(@PathParam("id") Long id) {
        return Person.findById(id);
    }

    @DELETE
    @Path("/{id}")
    @Transactional
    public void delete(@PathParam("id") Long id) {
        Person.deleteById(id);
    }
}

```

5. Run and Test

Start PostgreSQL locally and create the database (`yourdb`).

Then run:

```
./mvnw quarkus:dev

```

Test with:

```
curl -X POST http://localhost:8080/people \
     -H "Content-Type: application/json" \
     -d '{"name":"Sam","age":25}'

curl http://localhost:8080/people

```

Build and deploy

Project Structure Summary

```
hello-quarkus/
├── src/
├── Dockerfile
├── docker-compose.yml
└── target/

```

Step 1: Create `Dockerfile` for Quarkus

Add this `Dockerfile` in the root of your project:

```
FROM quay.io/quarkus/quarkus-micro-image:2.0
WORKDIR /work/
COPY target/*-runner.jar app.jar
EXPOSE 8080
CMD ["java", "-jar", "app.jar"]

```

> ⚠️ This assumes you're using **JVM mode**. You can switch to native image later if needed.

Step 2: Create `docker-compose.yml`

```
version: '3.8'

services:
  postgres:
    image: postgres:15
    container_name: quarkus-postgres
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: quarkus
      POSTGRES_PASSWORD: secret
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

  quarkus-app:
    build: .
    container_name: quarkus-app
    depends_on:
      - postgres
    environment:
      QUARKUS_DATASOURCE_JDBC_URL: jdbc:postgresql://postgres:5432/mydb
      QUARKUS_DATASOURCE_USERNAME: quarkus
      QUARKUS_DATASOURCE_PASSWORD: secret
    ports:
      - "8080:8080"

volumes:
  pgdata:

```

Step 3: Update `application.properties` for Docker

In `src/main/resources/application.properties`, use placeholders:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.jdbc.url=${QUARKUS_DATASOURCE_JDBC_URL}
quarkus.datasource.username=${QUARKUS_DATASOURCE_USERNAME}
quarkus.datasource.password=${QUARKUS_DATASOURCE_PASSWORD}
quarkus.hibernate-orm.database.generation=update
quarkus.hibernate-orm.log.sql=true

```

Step 4: Build the App

Compile your app:

```
./mvnw clean package -DskipTests

```

This creates `target/hello-quarkus-1.0.0-SNAPSHOT-runner.jar`

Step 5: Run Everything

Now launch with:

```
docker-compose up --build

```

You'll see both Postgres and Quarkus starting. Wait a few seconds and test:

```
curl http://localhost:8080/people

```

Optional: Clean Up

```
docker-compose down -v

```

Appendix

Spring Boot vs. Quarkus annotations

Spring Boot vs. Quarkus annotations across core functionalities, so you can quickly understand the differences (and similarities):

Dependency Injection / Beans

Feature	**Spring Boot**	**Quarkus (CDI/Jakarta EE)**
Define a bean/component	`@Component`, `@Service`, `@Repository`	`@ApplicationScoped`, `@Singleton`
Inject a bean	`@Autowired`, `@Inject`	`@Inject`
Qualifiers	`@Qualifier`	`@Named`, `@Qualifier`
Lifecycle callbacks	`@PostConstruct`, `@PreDestroy`	Same

REST Controllers

Feature	**Spring Boot**	**Quarkus**
REST Controller	`@RestController`	`@Path` + JAX-RS class
Map HTTP method	`@GetMapping`, `@PostMapping`, etc.	`@GET`, `@POST`, `@PUT`, etc.
Define path	`@RequestMapping("/path")`	`@Path("/path")`
Inject request param	`@RequestParam`	`@QueryParam`
Inject path variable	`@PathVariable`	`@PathParam`
Consume/Produce content	`@RequestBody`, `@ResponseBody`	`@Consumes`, `@Produces`

Persistence / JPA

Feature	**Spring Boot**	**Quarkus**
JPA Entity	`@Entity`	`@Entity`
ID field	`@Id`, `@GeneratedValue`	Same
Repository	`extends JpaRepository` + `@Repository`	Use `PanacheEntity` or `PanacheRepository`
Transactions	`@Transactional` (Spring)	`@Transactional` (Jakarta)

Validation

Feature	**Spring Boot**	**Quarkus** (Jakarta Bean Validation)
Validate fields	`@NotNull`, `@Size`, etc.	Same
Validate request body	`@Valid`	Same

Configuration

Feature	**Spring Boot**	**Quarkus**
Property injection	`@Value("${my.prop}")`	`@ConfigProperty(name = "my.prop")`
Config class	`@ConfigurationProperties`	`@ConfigMapping`

Scheduling

Feature	**Spring Boot**	**Quarkus**
Schedule method	`@Scheduled`	`@Scheduled`

Summary

- **Spring Boot** is annotation-heavy and offers its own abstraction layer (Spring annotations) on top of JSR standards.
- **Quarkus** uses **Jakarta EE / JAX-RS / CDI** standards, making it more **spec-compliant** and **lighter** in terms of abstraction.
- Quarkus uses **Panache** to simplify persistence and reduce boilerplate.