QUARKUS VS SPRING BOOT

The Ultimate Framework Comparison

Cloud-Native • Microservices • Performance

HISTORY AND CONTEXT

Spring Boot: A Pillar of Java Development

Spring Boot was launched in **2014** as an extension of the Spring Framework. It quickly gained popularity by simplifying the configuration and development of Java applications, offering a **convention-over-configuration** programming model. The goal was clear: reduce the complexity of creating Spring applications by providing a production-ready framework.

Spring Boot is widely adopted in enterprise projects due to its **robust ecosystem** and the Java community's familiarity with the Spring Framework. Large and small companies alike rely on Spring Boot for business-critical applications due to its maturity and rich library of integrations.

Quarkus: Optimized for Cloud and Containers

Quarkus was launched in **2019** by Red Hat with a specific goal: to be the most efficient Java framework for the **cloud-native era**. It was designed to be extremely fast, lightweight, and ideal for microservices and container environments. With the advent of Kubernetes-based and serverless architectures, Quarkus stood out for its focus on performance and reduced memory footprint.

Unlike Spring Boot, which evolved from an older framework, Quarkus was **built from the ground up with modernity in mind**. It introduces concepts like **build-time processing** and deep integration with **GraalVM**, enabling the compilation of applications into native binaries, promising extremely fast startup times and efficient resource usage.

> **Key Timeline**
>
> **2014** — Spring Boot 1.0 released (Spring Framework maturity)
> **2019** — Quarkus 1.0 released (Cloud-native focus)
> **2020** — Spring Native experimental (response to Quarkus)
> **2022** — Quarkus 2.x (production-grade native compilation)
> **2023** — Spring Boot 3.x (native support with GraalVM)

ARCHITECTURE AND DESIGN

Spring Boot

Spring Boot is based on the architecture of the Spring Framework, which means it inherits all the features of **dependency injection**, **aspect-oriented programming (AOP)**, and **reactive programming**, along with strong integration with patterns like MVC, REST, and many others.

- **Auto-configuration:** Spring Boot attempts to automatically configure application components based on the dependencies present on the classpath. This allows developers to focus more on business logic.
- **Spring Actuator:** Provides ready-to-use monitoring and metrics, giving a clear view of application performance and health.
- **Profiles:** Allows different configurations for different environments (development, testing, production), making it easier to migrate applications across these environments.
- **Spring Cloud:** An extension of the Spring ecosystem that simplifies microservices development with well-defined patterns like distributed configuration, service discovery, and API gateways.

Quarkus

Quarkus adopts a much more **modular and lightweight architecture**, using technologies like **MicroProfile**, a specification offering APIs for microservices development, and integrating deeply with **GraalVM**.

- **Build-Time Processing:** Quarkus performs most of the work at compile time rather than at runtime, meaning the application is pre-processed and optimized before it runs.
- **Quarkus Extensions:** Modularity is one of Quarkus's strengths. Extensions can be added as needed without loading unnecessary components, keeping the application lightweight.
- **Native Compilation:** With native support for GraalVM, Quarkus can compile Java applications into native executables, resulting in extremely fast startup times and low memory usage.
- **Reactive Programming:** Quarkus is oriented towards reactive programming, making it especially useful in applications that need to handle a large number of simultaneous connections or require high I/O performance.

⚡ PERFORMANCE COMPARISON

Startup Time

Spring Boot

The startup time of a Spring Boot application tends to be longer due to its traditional approach of runtime configuration. In a typical application, Spring Boot needs to configure all components, perform dependency injections, and process auto-configurations.

Typical: 2-10 seconds
Large apps: 30+ seconds

Quarkus

Quarkus does most of the work at compile time. This means the application is already optimized and ready to run as soon as it is launched. In environments where startup time is critical (serverless, scaling microservices), this difference is decisive.

JVM mode: 0.5-2 seconds
Native: 0.01-0.1 seconds

Memory Consumption

Spring Boot

A Spring Boot application generally consumes more memory, especially in scenarios where many features are loaded automatically, even if they are not needed. This can be a problem in resource-limited environments, such as low-memory containers.

Typical: 200-500 MB
With features: 500 MB-1 GB+

Quarkus

Due to compile-time processing and the ability to create native binaries, Quarkus can significantly reduce memory consumption. Quarkus applications are designed to be lightweight, ideal for container and cloud environments where resource efficiency is a priority.

JVM mode: 70-150 MB
Native: 20-50 MB

Runtime Performance

Spring Boot

During runtime, Spring Boot is quite efficient, but it carries some overhead due to its flexibility and support for a wide range of features and libraries. In large-scale applications, performance may suffer unless specific optimizations are made.

Quarkus

Quarkus excels in runtime performance, especially in scenarios requiring high concurrency and low response times. Its reactive nature and ability to eliminate runtime overhead make it a strong choice for applications demanding extreme performance.

DEVELOPMENT EASE

Tools and Ecosystem

Spring Boot

One of Spring Boot's strengths is its **vast ecosystem** and support for development tools. IDEs like IntelliJ IDEA and Eclipse have robust support for Spring, with plugins that make development, debugging, and testing easier.

Spring Boot offers a rich collection of **starters** and libraries, allowing developers to add features like security, database integration, and messaging with ease.

Quarkus

Although Quarkus is newer, it also offers good support for development, with extensions for various IDEs and a CLI that simplifies the creation of new projects and the addition of extensions.

However, due to its youth, the ecosystem is **not as vast** as Spring Boot's, which may limit options for developers needing very specific features.

Learning Curve

Spring Boot

For developers already familiar with the Spring Framework, the learning curve for Spring Boot is relatively low.

However, for **new developers**, the complexity of Spring can be intimidating, and the initial configuration may require a good understanding of the Spring ecosystem.

Quarkus

Quarkus presents a steeper learning curve for developers coming from traditional Java environments. The need to understand concepts like **build-time processing** and **reactive programming** may require additional learning effort.

But for developers focused on cloud-native environments, the investment may be worth it.

IDE Support Comparison

Spring Boot: Excellent support in IntelliJ IDEA Ultimate, Eclipse STS, VS Code (Spring Tools)
Quarkus: Good support in IntelliJ IDEA, VS Code (Quarkus Tools), Eclipse with Red Hat plugins

INTEGRATION WITH EXTERNAL TECHNOLOGIES

Database

Spring Boot

Spring Boot has built-in support for a wide range of databases, both relational and NoSQL. With **Spring Data**, database integration and manipulation become simpler and more powerful.

- ✓ Spring Data JPA
- ✓ Spring Data MongoDB
- ✓ Spring Data Redis
- ✓ Spring Data Elasticsearch
- ✓ R2DBC (reactive)

Quarkus

Quarkus offers database support through **Hibernate ORM** and **Panache** but may not have as many out-of-the-box integrations as Spring Boot.

- ✓ Hibernate ORM with Panache
- ✓ Hibernate Reactive
- ✓ MongoDB client
- ✓ Neo4j client

However, Quarkus extensions are designed to be **lightweight and efficient**.

Microservices and Distributed Architectures

Spring Boot

Spring Boot is widely used in microservices architectures, especially when combined with **Spring Cloud**. It offers tools for:

- Distributed configuration
- Service discovery (Eureka)
- Circuit breakers (Resilience4j)
- API Gateway (Spring Cloud Gateway)
- Load balancing (Ribbon)

Making it easier to build robust and scalable systems.

Quarkus

Quarkus, with its focus on being cloud-native, is also a strong choice for microservices. It is compatible with **MicroProfile**.

- Config (MicroProfile Config)
- Health checks (MicroProfile Health)
- Metrics (MicroProfile Metrics)
- Fault tolerance (SmallRye Fault Tolerance)
- REST Client (MicroProfile REST Client)

Extremely resource-efficient, ideal for Kubernetes.

USE CASES

When to Choose Spring Boot

Complex Enterprise Applications: If you're building an application with complex enterprise requirements that need a wide range of out-of-the-box features, Spring Boot is a natural choice. The maturity of the ecosystem and support for deep integrations with enterprise technologies make Spring Boot ideal for these scenarios.

Projects with Experienced Spring Teams: If your team already has experience with the Spring Framework, using Spring Boot can significantly reduce development time and avoid a steep learning curve.

Need for Tool and Plugin Support: If you rely on tools and plugins with robust support for Spring Boot, this may be the most pragmatic choice.

Rich Ecosystem Requirements: When you need deep integration with various third-party libraries and frameworks that have mature Spring Boot starters.

Long-running Applications: Applications that run for extended periods where startup time is less critical than runtime stability and ecosystem maturity.

When to Choose Quarkus

Cloud-Native Applications: If you're developing for cloud-native environments or serverless architectures, Quarkus offers the best performance in terms of startup time and memory usage.

Microservices and Containers: Quarkus is ideal for microservices that need to scale quickly, especially in Kubernetes environments. The resource efficiency and ability to compile into native binaries make it the ideal choice for lightweight containers.

Performance-Focused Projects: If performance is a priority, especially in terms of response time and the ability to handle high concurrency, Quarkus offers significant advantages.

Cost Optimization: Lower memory footprint means lower cloud infrastructure costs, especially when running hundreds or thousands of container instances.

Reactive Applications: Applications that need to handle thousands of concurrent connections with non-blocking I/O.

🚀 OPTIMIZATION STRATEGIES

Spring Boot Optimization

Lazy Initialization: Enable lazy initialization of beans to improve startup time by loading beans only when necessary.

Configuration

```
spring.main.lazy-initialization=true
```

Customizing Auto-Configuration: Disable unnecessary auto-configurations to reduce memory usage and improve performance.

Example

```
@SpringBootApplication(exclude = {
  DataSourceAutoConfiguration.class,
  HibernateJpaAutoConfiguration.class
})
```

Spring Native: Consider using Spring Native to compile the application into native binaries, reducing startup time and memory usage.

Quarkus Optimization

Quarkus Extensions: Use Quarkus extensions sparingly to ensure only what's needed is included in the application, keeping it lightweight and efficient.

Add extensions selectively

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

Native Compilation with GraalVM: Compile the application into a native binary using GraalVM to take full advantage of runtime optimizations.

Build native image

```
./mvnw package -Pnative
# Result: 20-50 MB binary, 0.01s startup, 30-50 MB RSS
```

Reactive Programming: Leverage Quarkus's reactive nature to build applications that can scale efficiently and handle a large number of simultaneous requests.

PERFORMANCE METRICS

Real-World Benchmark: Simple REST API

Spring Boot 3.x (JVM):
- Startup: 2.5 seconds
- Memory (RSS): 250 MB
- JAR size: 35 MB
- First request: ~50ms

Spring Boot 3.x (Native):
- Startup: 0.08 seconds
- Memory (RSS): 80 MB
- Binary size: 75 MB
- First request: ~10ms

Quarkus (JVM):
- Startup: 1.2 seconds
- Memory (RSS): 120 MB
- JAR size: 15 MB
- First request: ~20ms

Quarkus (Native):
- Startup: 0.016 seconds
- Memory (RSS): 35 MB
- Binary size: 45 MB
- First request: ~5ms

Cost Analysis (Cloud Deployment)

Assuming 100 microservice instances running 24/7 on AWS ECS Fargate:

Spring Boot (JVM):
- Memory per instance: 512 MB
- Monthly cost: ~$1,500

Quarkus (Native):
- Memory per instance: 128 MB (0.25 vCPU)
- Monthly cost: ~$400

💰 **Savings: ~$1,100/month (~73% reduction)**

Serverless Comparison (AWS Lambda)

Spring Boot:
- Cold start: 5-15 seconds
- Memory required: 1024 MB minimum
- Cost impact: High (billed for cold start duration)

FEATURE COMPARISON MATRIX

Feature	Spring Boot	Quarkus
Maturity	Excellent (10+ years)	Good (5+ years)
Startup Time (Native)	~80ms	~16ms
Memory (Native)	~80 MB	~35 MB
Native Compilation	Experimental (3.x)	Production-ready
Developer Experience	Excellent	Excellent
Hot Reload	DevTools	Dev Mode
Community Size	Very Large	Growing
Third-party Integrations	Extensive	Good
Cloud-native Focus	Good	Excellent
Kubernetes Integration	Spring Cloud K8s	Native K8s support
Reactive Support	Spring WebFlux	Mutiny (built-in)
Database Support	Extensive	Good
Testing Support	Excellent	Excellent
Documentation	Comprehensive	Comprehensive
Learning Curve	Moderate	Moderate-Steep
Best For	Enterprise apps, Spring teams	Cloud-native, microservices

MIGRATION CONSIDERATIONS

Spring Boot to Quarkus Migration

Migrating from Spring Boot to Quarkus requires understanding the differences in architecture and replacing Spring-specific components with Quarkus equivalents.

Common Replacements

Spring Boot	Quarkus Equivalent
@SpringBootApplication	@QuarkusMain (if needed)
@RestController	@Path + @ApplicationScoped
@Autowired / @Inject	@Inject (CDI)
Spring Data JPA	Hibernate Panache
@ConfigurationProperties	@ConfigProperty
Spring Security	Quarkus Security + OIDC
Spring WebFlux	Mutiny (reactive streams)
Spring Cloud Config	MicroProfile Config
Spring Actuator	MicroProfile Health/Metrics

Migration Challenges

Reflection Limitations: Native compilation requires explicit reflection configuration for classes used reflectively.

Dynamic Class Loading: Features relying on runtime class loading need build-time registration.

Third-party Library Compatibility: Not all Spring-compatible libraries work with Quarkus native mode.

Testing Adjustments: Test frameworks and mock libraries may need updates for native compilation.

Migration Best Practices

Start Small: Migrate one microservice at a time, not the entire monolith.

JVM Mode First: Test in JVM mode before attempting native compilation.

Use Quarkus Extensions: Leverage official extensions instead of raw dependencies when possible.

Test Native Mode: Build and test native binaries early to catch compatibility issues.

🎯 FINAL CONSIDERATIONS

The choice between Spring Boot and Quarkus should be guided by the **specific needs of the project** and the goals for performance and scalability.

Choose Spring Boot If:

- ✓ You need the **most mature and extensive ecosystem** in Java
- ✓ Your team has **existing Spring Framework experience**
- ✓ You require deep integration with **enterprise technologies**
- ✓ Startup time and memory footprint are **not critical**
- ✓ You need the widest selection of **third-party integrations**
- ✓ You're building traditional **monolithic or modular applications**

Choose Quarkus If:

- ✓ You're building **cloud-native or serverless applications**
- ✓ **Startup time and memory usage** are critical requirements
- ✓ You need to **optimize cloud infrastructure costs**
- ✓ Your architecture is focused on **Kubernetes and containers**
- ✓ You want **reactive programming** as a first-class citizen
- ✓ You're willing to invest in **learning new patterns** for better performance

> **The Hybrid Approach**
>
> Many organizations use **both frameworks** strategically:
>
> • **Spring Boot** for complex, long-running services with rich business logic
> • **Quarkus** for edge services, API gateways, and frequently-scaling microservices
>
> This allows teams to leverage the strengths of each framework where they matter most.

Regardless of which framework you choose, it is essential to **understand its capabilities and limitations** and adopt optimization strategies that maximize your application's performance.

READY TO MASTER QUARKUS?

Build a Real-World Indoor Tracking Platform

10 Microservices • Event-Driven Architecture

Native Compilation • Native Startup

Kubernetes Deployment • Production Observability

Learn Quarkus the Right Way

Crash course

Real hardware • Production patterns • Expert guidance

quarkus.suddo.io