# **What is Quarkus?**

Quarkus is a full-stack, Kubernetes-native [Java framework](https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-Java-framework) made for Java virtual machines (JVMs) and native compilation, optimizing Java specifically for containers and enabling it to become an effective platform for [serverless](https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless), [cloud](https://www.redhat.com/en/topics/cloud-computing), and [Kubernetes](https://www.redhat.com/en/topics/containers/what-is-kubernetes) environments.

**Site : quarkus.io  
  
How is quarkus more efficient than springboot?**

Quarkus is considered more efficient than Spring Boot in certain use cases, particularly for **cloud-native applications, microservices, and serverless environments.** Here are the main reasons why Quarkus might be seen as more efficient than Spring Boot in these contexts:

### **1. Faster Startup Time**

* **Quarkus**: One of its standout features is its extremely fast startup time, especially when compiled into a native image using GraalVM. This is crucial for serverless environments where quick cold-start times are important. Quarkus can start up in milliseconds, whereas Spring Boot typically has slower startup times due to the JVM's overhead and the extensive reflection and class loading it relies on.
* **Spring Boot**: While Spring Boot has been optimized for production environments, its startup times are typically slower compared to Quarkus, especially when running in JVM mode.

### **2. Lower Memory Consumption**

* **Quarkus**: Quarkus uses a combination of compile-time optimizations and the GraalVM native image technology to produce smaller, more efficient applications that consume less memory. This is especially beneficial for resource-constrained environments like microservices or Kubernetes, where resource efficiency is critical.
* **Spring Boot**: Spring Boot applications tend to be heavier, as they are optimized for flexibility and ease of use, which can lead to higher memory consumption, particularly in large applications or when running on JVM.

### **3. Native Compilation with GraalVM**

* **Quarkus**: It supports GraalVM native compilation out-of-the-box, which compiles your Java application to a native binary, significantly reducing both memory usage and startup time. This allows Quarkus to run with minimal runtime overhead, making it ideal for cloud-native, serverless, and microservice architectures.
* **Spring Boot**: Spring Boot can also be used with GraalVM for native image compilation, but it requires more work to configure and optimize, as it wasn't originally built with native compilation as a primary feature.

### **4. Optimized for Cloud-Native & Microservices**

* **Quarkus**: Quarkus was specifically designed with cloud-native and microservices architectures in mind. It provides integration with Kubernetes, Docker, and other cloud-native technologies and offers small, lightweight services that fit perfectly into the cloud-native paradigm.
* **Spring Boot**: While Spring Boot is also highly suitable for microservices, it wasn't built specifically for this purpose and can be more heavyweight when compared to Quarkus. Spring Boot offers a rich ecosystem, which makes it versatile, but sometimes that comes with overhead.

### **5. Developer Experience (Dev Mode)**

* **Quarkus**: Quarkus provides a "live reload" feature, allowing developers to make changes to the code and see the results immediately without needing to restart the application. This feature is especially useful when working on cloud-native environments where fast iteration and productivity are important.
* **Spring Boot**: Spring Boot has tools like DevTools for live reload, but its developer experience isn't as fast and efficient as Quarkus in terms of iterative development, especially when working with native images.
* **Spring Boot**: Spring Boot also supports Kubernetes and cloud-native development, but its larger size and slower startup times mean that it might not be as resource-efficient or quick to scale as Quarkus in some cases.

### **When to Choose Quarkus Over Spring Boot?**

* **Serverless Applications**: Quarkus is often preferred for serverless environments, where cold-start time and memory consumption are critical factors.
* **Microservices**: For lightweight, fast microservices that need to scale efficiently, Quarkus can provide better performance and faster startup times.
* **Cloud-Native and Kubernetes**: Quarkus's optimizations for Kubernetes and cloud-native environments, along with its smaller footprint, make it a good choice for building applications that need to be containerized and deployed in such environments.

### **When to Choose Spring Boot Over Quarkus?**

* **Enterprise Ecosystem**: If your team already has a significant investment in the Spring ecosystem (Spring Security, Spring Data, Spring Batch, etc.), Spring Boot might be more attractive due to its rich set of features and the large community behind it.
* **Larger Applications**: For applications that are not as sensitive to startup time and memory consumption, Spring Boot provides a mature and feature-rich environment with extensive documentation and tooling.

**Which is more efficient ?**

In terms of raw **efficiency**, **Quarkus** tends to be more efficient than **Spring Boot** in specific areas, particularly when you're building **cloud-native** applications, **microservices**, or working in **serverless environments.**

**What is a serverless environment ?**

A **serverless environment** refers to a cloud computing model where developers can build and deploy applications without having to manage the underlying infrastructure. Despite the name, servers are still involved, but their management is entirely handled by the cloud provider. This allows developers to focus on writing code and developing features, rather than dealing with tasks such as provisioning, scaling, and maintaining servers.

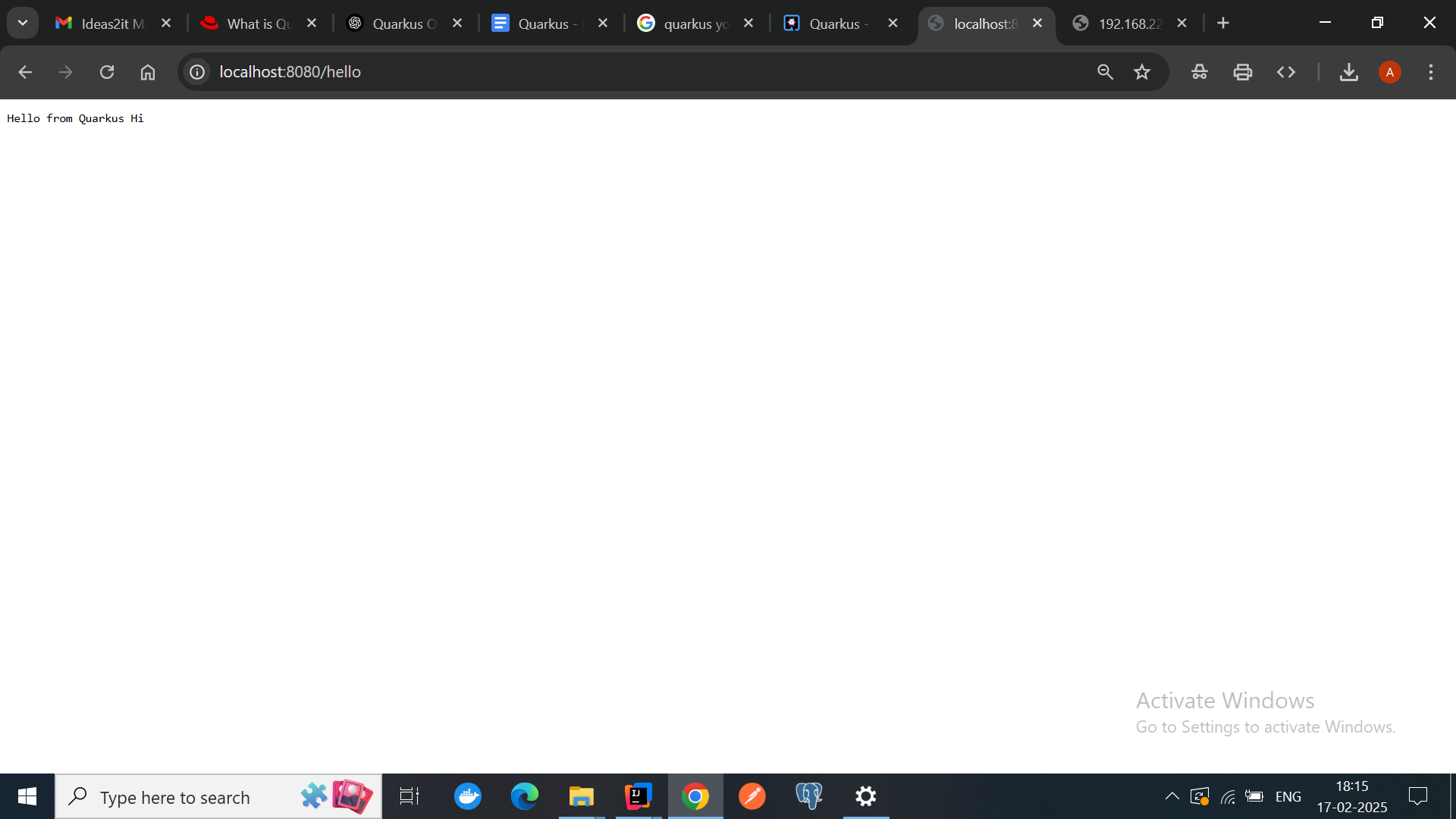
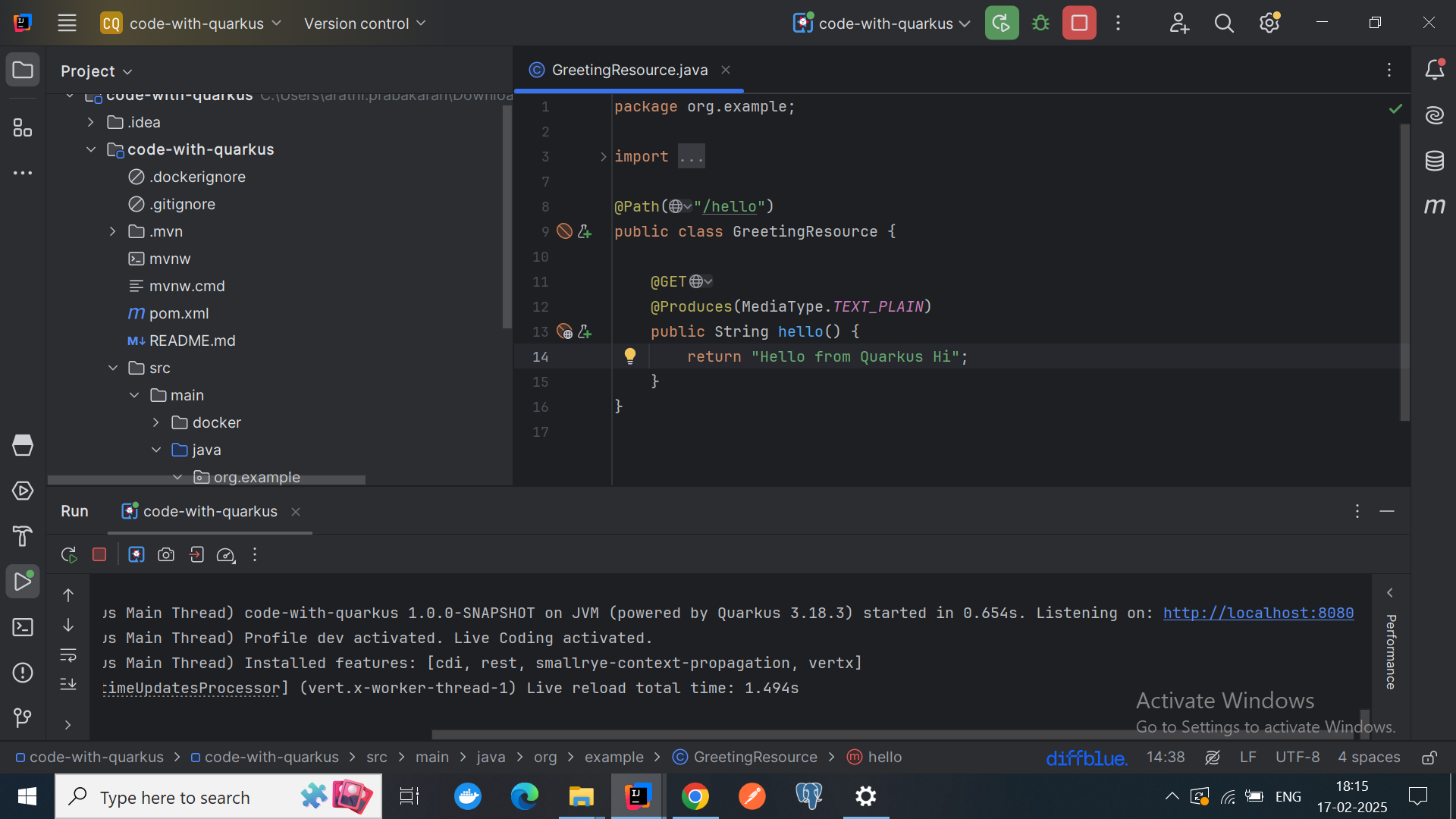
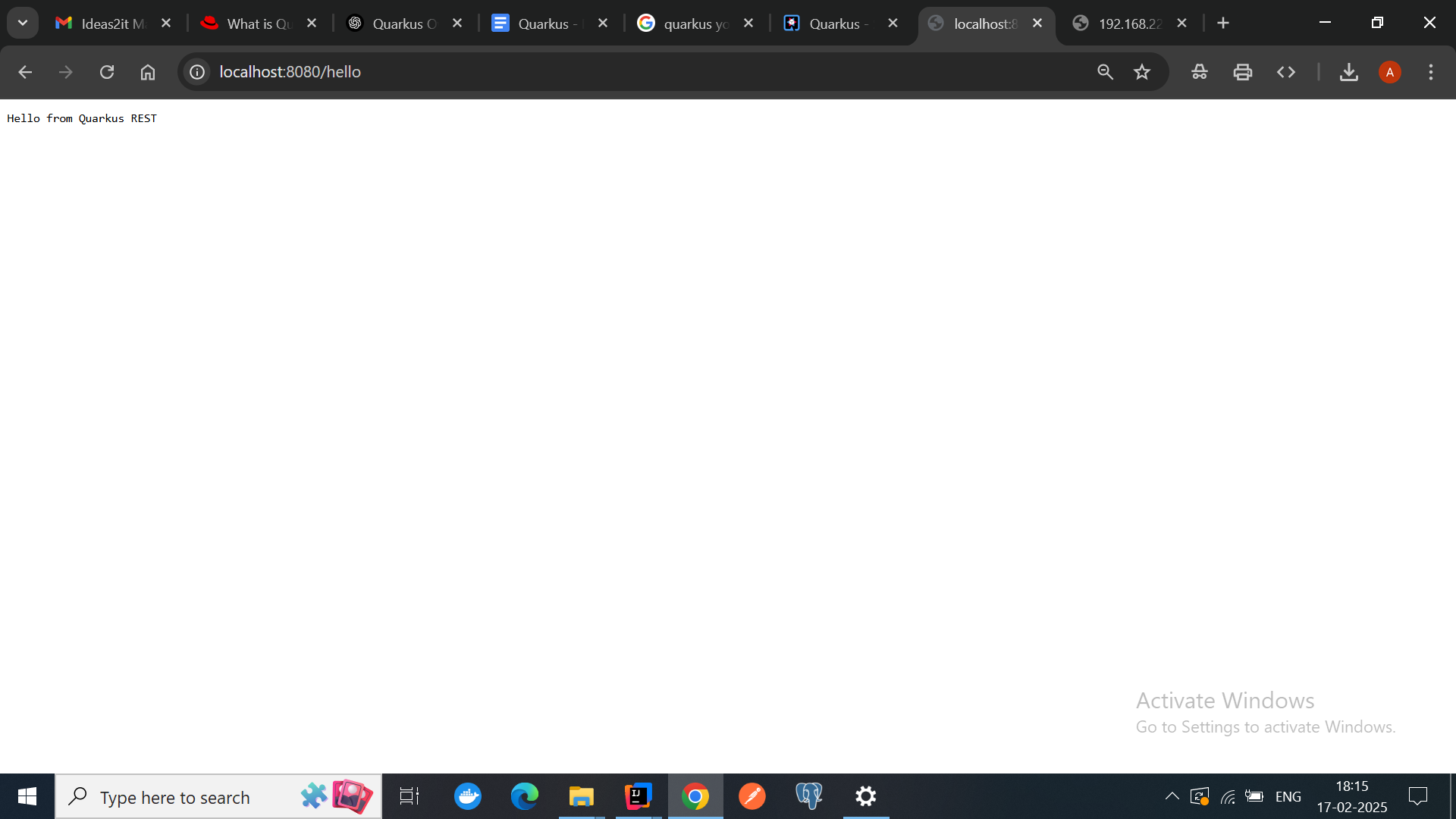
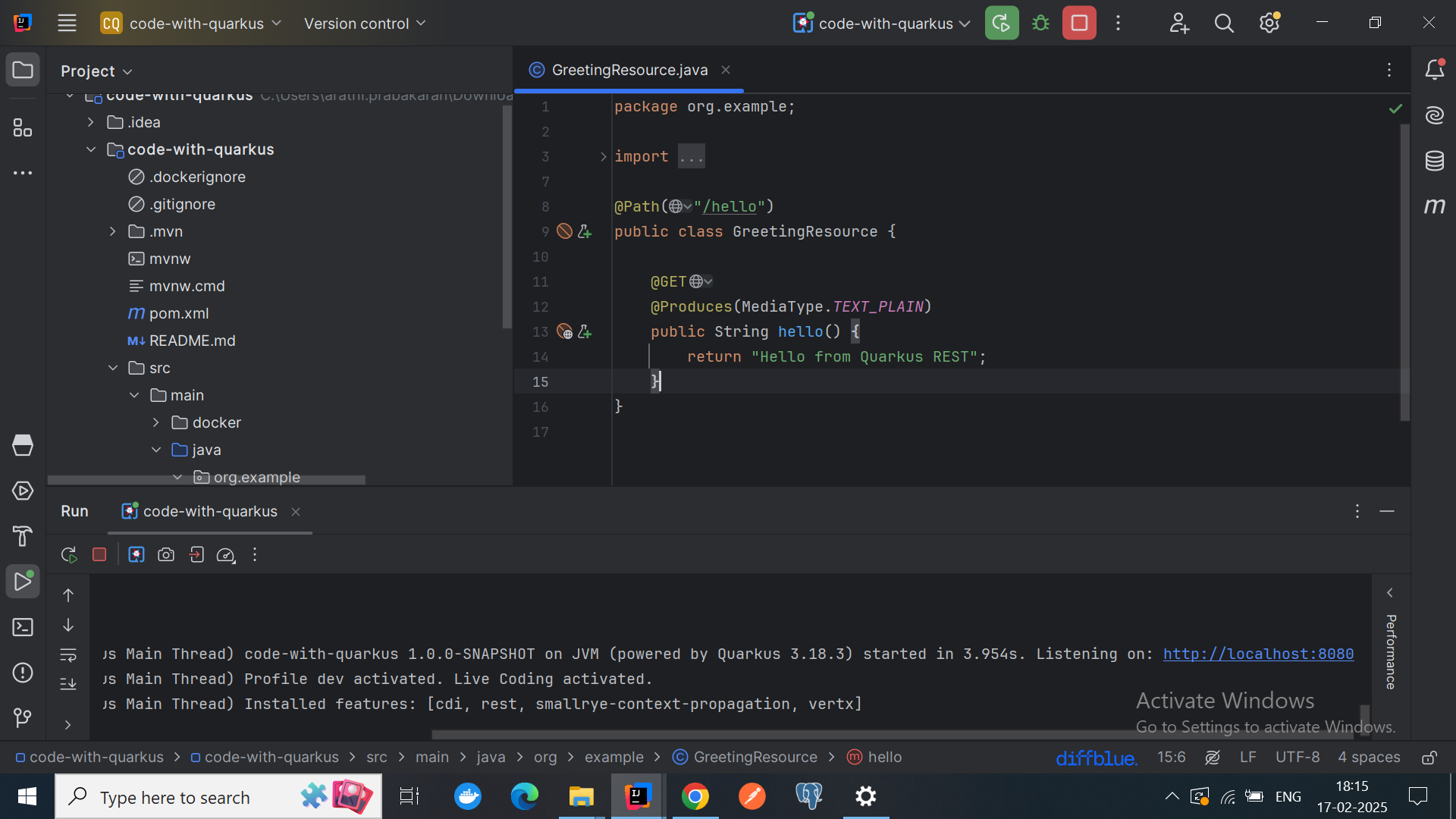
**Key Points about Quarkus Development Mode (quarkus:dev):**  
In **Quarkus**, you typically **don't need to manually rerun the application** after making changes during development, thanks to its **live reload** feature.

When you're working in **development mode** (mvn quarkus:dev or ./mvnw quarkus:dev), Quarkus automatically detects changes to your source code and **reloads the application** for you. This allows for a much faster feedback loop compared to traditional Java development, where you would normally need to stop and restart the application after every change.

### **Spring Boot vs Quarkus Reload**

* **Quarkus**: Quarkus automatically reloads and refreshes your application with live reloading, and it focuses on making the startup time fast and resource-efficient, offering **instant feedback** with minimal overhead. Quarkus is known for its **fast hot reload** and **faster feedback loop** compared to traditional Java frameworks.
* **Spring Boot**: In Spring Boot, **DevTools** is required for automatic restart and live reload, and it generally works by restarting the application. While the restart is typically fast with Spring DevTools, it's still a **full application restart**, unlike Quarkus, which is optimized to minimize restart times.

In summary, **Quarkus** offers a faster, more efficient **hot reload** experience than **Spring Boot**, which still relies on a full application restart (though relatively quick with Spring DevTools). If you are looking for minimal delay during development, Quarkus tends to be more efficient in this regard. However, **Spring Boot with DevTools** provides a very similar developer experience, albeit with a restart-based workflow.



**How does API call work in quarkus when compared to spring boot ?  
  
API Calling in Quarkus vs Spring Boot**

When building applications with Quarkus or Spring Boot, both frameworks provide convenient mechanisms for API calling—whether you're creating RESTful services or invoking APIs. However, there are differences in how each framework handles API calls, both internally and externally.

### **Creating APIs (REST APIs)**

#### **In Quarkus:**

Quarkus uses **JAX-RS** (Java API for RESTful Web Services) for creating REST APIs, along with **RESTEasy** (a JAX-RS implementation). The framework also integrates with other libraries for REST services, such as **MicroProfile REST Client** and **Reactive APIs** for asynchronous communication.

Example:

@Path("/hello")

public class HelloResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String hello() {

return "Hello from Quarkus";

}

}

**Annotations**: Quarkus uses JAX-RS annotations like @Path, @GET, and @Produces to create REST endpoints.

**Configuration**: Configuration in Quarkus is generally done via application.properties or application.yml and follows standard Quarkus conventions.

#### **In Spring Boot:**

Spring Boot uses **Spring Web** (based on **Spring MVC**) for creating REST APIs. It uses annotations like @RestController, @RequestMapping, @GetMapping, and others to define API endpoints.

Example of a REST endpoint in Spring Boot:

@RestController

public class HelloController {

@GetMapping("/hello")

public String hello() {

return "Hello from Spring Boot";

}

}

**Annotations**: Spring Boot uses annotations like @RestController, @GetMapping, @PostMapping, etc., for REST API development.

**Configuration**: Configuration in Spring Boot is usually done through application.properties or application.yml.

**Why is there no main application in quarkus ?**

In Quarkus, there is no explicit **main application** class like in traditional Java applications. This is because Quarkus follows a more **microservice-oriented** and **container-native** approach, where the application lifecycle is typically managed by the container (e.g., a Kubernetes or OpenShift environment) rather than by a manually defined entry point in code.

**What is JAX-RS ?**

JAX-RS (Java API for RESTful Web Services) is a set of Java programming language APIs to create RESTful web services. It is part of the **Java EE (Enterprise Edition)** specification, now included under the **Jakarta EE** umbrella. JAX-RS simplifies the creation of web services that follow the principles of REST (Representational State Transfer), which is a lightweight architectural style for designing networked applications.

**Annotations**: JAX-RS uses annotations to simplify the creation of RESTful services, replacing the need for complex configuration files. Common annotations include:

* @Path: Specifies the URI path for a resource.
* @GET, @POST, @PUT, @DELETE: Bind HTTP methods (GET, POST, etc.) to resource methods.
* @Produces: Specifies the media types (like JSON, XML) that a method can return.
* @Consumes: Specifies the media types that a method can accept as input.

**How Dependency Injection Works in Quarkus ?**

In Quarkus, **CDI (Contexts and Dependency Injection)** is the core mechanism used to inject dependencies. The DI system in Quarkus is based on annotations that are part of the **Jakarta EE** specification (formerly Java EE).

### **1. Defining Beans (Classes to be Injected)**

A **bean** is any class that Quarkus manages as part of its DI system. In Quarkus, beans can be defined in several scopes, such as @ApplicationScoped, @RequestScoped, @Singleton, etc. A class becomes a **bean** when it is annotated with one of the **CDI scopes**.

* **@ApplicationScoped**: This annotation tells Quarkus to create a single instance of this class during the application’s lifetime. It’s similar to the concept of a singleton.

Other common scopes:

* **@RequestScoped**: Creates a new instance for each HTTP request.
* **@Singleton**: Guarantees a single instance across the application, typically used for long-lived services.

### **4. Scopes in Quarkus**

The scope of a bean determines its lifecycle. Quarkus supports several scopes for managing the lifecycle of beans:

* **@ApplicationScoped**: The bean is created once and shared throughout the application.
* **@RequestScoped**: A new instance of the bean is created per HTTP request.
* **@Singleton**: A single instance for the whole application.
* **@Dependent**: This is the default scope and indicates that the bean’s lifecycle depends on the bean it is injected into.

**What is native compilation ?**

**Native Compilation** is the process of converting a high-level programming language (like Java) into machine code that can be executed directly by the operating system, without needing a runtime environment like the **JVM** (Java Virtual Machine). In Java, this means converting a Java application (which would typically run on the JVM) into a **native executable** that can run directly on an operating system.

### **Memory Footprint in Quarkus and Micronaut:**

* **Quarkus** is optimized for **low memory footprint**. It's designed to be lightweight, particularly when running in **native mode** (compiled with **GraalVM**). This results in faster startup times and lower memory usage compared to traditional Java applications.
* **Micronaut** also emphasizes low memory consumption. It uses **dependency injection** and **reflection-free design**, which contributes to its small memory footprint.

### **Comparing Memory Footprint: Quarkus vs. Spring Boot**

| **Aspect** | **Quarkus** | **Spring Boot** |
| --- | --- | --- |
| **Native Image Support** | **Strong**: Native compilation via **GraalVM** reduces footprint significantly. | **Weak**: Native image support via GraalVM is available, but Spring Boot is traditionally JVM-based. |
| **JVM Mode Memory Footprint** | **Low**: Typically 200-300 MB for basic applications. | **Moderate to High**: Typically 500 MB to 1 GB. |
| **Native Mode Memory Footprint** | **Very Low**: 10-50 MB. | **Higher**: Native mode applications are still typically larger than Quarkus in native mode. |
| **Startup Time** | **Fast**: Quarkus has very fast startup times due to optimizations and GraalVM native compilation. | **Slower**: Spring Boot has a slower startup time compared to Quarkus, especially in JVM mode. |
| **Optimized for Cloud/Containers** | **Yes**: Quarkus is highly optimized for containers and cloud environments. | **Yes**: Spring Boot is also widely used in cloud environments but with a larger memory footprint. |
| **Framework Size** | **Small**: Only includes necessary extensions. | **Large**: Spring Boot includes a lot of features by default. |

### 

**Console application  
Console application handling without main file  
@quarkusMain for console application**  
  
**Multiple Applications (Console vs REST)**: You can have multiple entry points (e.g., REST service and a CLI application) in a Quarkus application. These can be controlled using **Quarkus profiles**, **CLI extensions**, and **native executables**.

**Profiles**: Use @Profile to configure which parts of your application should run based on the active profile.

**Command-Line Interface (CLI)**: You can create CLI applications in Quarkus with @Command or @QuarkusMain.

**Deciding Which to Run**: You can control which part of the application runs (e.g., CLI or REST) based on the profile or configuration used at runtime. You can also write logic to specify entry points if needed.

**Define an Application Class with @QuarkusMain (Optional)**: If you'd like to make this a standalone application (without needing to invoke it via Quarkus dev mode), you can annotate a class with @QuarkusMain and define a main method.

**AOT - ahead of time**  
**AOT (Ahead-Of-Time) Compilation** is a technique used in programming where the code is compiled into a machine-readable format before it is executed, rather than being compiled at runtime (JIT - Just-In-Time). This results in a more efficient startup time and reduced runtime overhead, as the heavy lifting of compiling the code has already been done ahead of time.

### **AOT in Quarkus**

Quarkus leverages **AOT compilation** to build **native executables** using **GraalVM**. This approach improves the startup speed and reduces the memory footprint significantly.

* **GraalVM** is a high-performance runtime that provides both a **JVM** and an **AOT compilation** mode. Quarkus can compile the entire application into a **native executable** using GraalVM's **native-image** tool, enabling AOT compilation.  
  When building a Quarkus application with GraalVM, the code is compiled ahead of time into a platform-specific native executable. This native executable can be run without the need for a JVM, which significantly reduces startup time and memory usage.

**Limitations of quarkus over springBoot**

While Quarkus has many advantages, such as its speed, low memory footprint, and native compilation support, it also comes with some limitations when compared to Spring Boot. Understanding these limitations is crucial when deciding which framework to choose for your specific use case. Here are some key limitations of Quarkus compared to Spring Boot:

### **1. Smaller Ecosystem and Community:**

* Spring Boot has been around for much longer than Quarkus, and as a result, it has a much larger ecosystem, community, and support base.
* Quarkus, although growing rapidly, is still a relatively newer framework, so you may face fewer resources in terms of third-party libraries, tutorials, and community support.
* Spring Boot has a rich set of well-maintained integrations with various third-party libraries, cloud platforms, databases, and tools.

### **2. Learning Curve for Quarkus:**

* Spring Boot's "convention over configuration" approach and mature documentation make it easier for developers, especially those already familiar with the Spring ecosystem, to get started quickly.
* **Quarkus** may require developers to adapt to new paradigms, especially when dealing with **native compilation**, **GraalVM**, or working with a more minimalistic setup. There’s also a different programming model (e.g., reactive, imperative) in Quarkus that might have a steeper learning curve for developers coming from Spring Boot.

| **Aspect** | **Spring Boot** | **Quarkus** |
| --- | --- | --- |

| **Maturity** | Mature, widely adopted, established ecosystem | Newer, rapidly growing |
| --- | --- | --- |

| **Learning Curve** | Low, familiar for Spring developers | Moderate, especially for developers unfamiliar with the Quarkus style |
| --- | --- | --- |

| **Ecosystem** | Large ecosystem of libraries, tools, and integrations | Smaller ecosystem, but growing rapidly |
| --- | --- | --- |

| **Legacy Integration** | Strong support for legacy systems and Java EE features | Focused on modern, cloud-native, microservices architectures |
| --- | --- | --- |

| **Native Compilation** | Limited native compilation support (Spring Native) | Excellent native compilation support (GraalVM) |
| --- | --- | --- |

| **Community Support** | Large, well-established | Smaller but rapidly growing |
| --- | --- | --- |

| **Application Flexibility** | More flexible for various application styles (monoliths, microservices) | Primarily optimized for microservices |
| --- | --- | --- |

| **Tooling Support** | Excellent IDE support and third-party tools | Decent tooling, but not as mature as Spring Boot |
| --- | --- | --- |