Published by Red Hat Developer

Quarkus for Spring Developers

Eric Deandrea
with Daniel Oh and Charles Moulliard

Foreword by Martijn Verburg

Contents

Contents

Contents

Foreword

The year is 2021, and for the 21st year in a row, the Java ecosystem once more is confronted with the "Java is Dead" meme. This time, industry observers state that the Java ecosystem will not pivot to efficient runtimes and frameworks for microservices deployed on resource-constrained containers. Naturally, the Java community has ignored the meme and responded with a wave of new runtimes and frameworks.

One of the firm leaders in this new wave of "runtime plus framework" stacks is Quarkus. Quarkus combines standardized enterprise Java APIs (from Java Enterprise Edition, Jakarta Enterprise Edition, and MicroProfile) to build your services and then run as a tightly packed, precompiled, and resource-efficient native image (via GraalVM). An example of this is Eclipse Adoptium (previously AdoptOpenJDK). We moved our API to use Quarkus, and it's been brilliant both in developer productivity and performance, serving up to 300 million requests in the past two years from a single small instance.

Given the title of the book, and especially if you're a Spring developer, I suspect you might be thinking "marketing hype" right about now, right? For me, the paradigm that Quarkus brings is a major one for Java. This is one of these moments in your career when you should take stock and explore a new technology in-depth! This book makes it easy to do that, providing like-for-like examples of your favorite Spring development patterns mapped to their Quarkus equivalents (don't panic, there are many similarities) and giving you an in-depth understanding of the fundamentals at play under the hood.

—Martijn Verburg (aka "The Diabolocal Developer")
Principal Group Manager (Java), Microsoft
Eclipse Adoptium Steering Committee member
August 2021

Acknowledgments

Eric Deandrea

I've spent most of my software development and architecture career building Java applications and application frameworks. Over the last ten years, I have focused on building opinionated frameworks based on Spring and architecting DevOps solutions. My experience with Spring led my employer, Red Hat, to approach me about writing this book.

There are many similarities and differences between Quarkus and Spring. When getting started with Quarkus, I found many Quarkus books, guides, and tutorials, but not many explicitly tailored to developers familiar with Spring concepts, constructs, and conventions. I was excited to use my Spring experience to help showcase and explain the benefits of Quarkus while also showing, through examples, that Spring developers can learn and understand Quarkus with relatively little effort.

Still, I was somewhat terrified at the notion of writing a book. I had never written anything more than a blog post or short article before. I didn't even know how to get started. Luckily, I work with many talented people who were there to help along the way. Each and every one of the following people played a key role. This book would not have been possible without the contributions of each person.

First, thank you, Syed M. Shaaf, for getting me involved in this project and helping me get started. You helped me get organized and develop a table of contents. The initial table of contents was aggressive and needed to be scaled back, but it helped drive the overall topics and structure that we wanted to cover in this book. A big thank you is also needed for helping with decisions around publisher selection.

Thank you, Daniel Oh, for your initial reviews of the chapters I authored. You were always there to help me with ideas in the writing and the examples, as well as for your help testing all the examples. A huge thank you for authoring Chapter 5: Event-Driven Services. The book would not have been a success without your time and contributions.

Thank you, Charles Moulliard, for your countless hours of time and effort reviewing chapters for technical accuracy and suggesting improvements. Thank you as well for organizing reviews of all the example projects. A huge thank you for authoring Chapter 6: Building Applications for the Cloud. I don't know what I would have done without your involvement and dedication to this project, especially since all of your contributions fall outside your daily responsibilities. Your time and effort are very much appreciated!

Thank you, Georgios Andrianakis, for your help reviewing the examples used in the chapters and pushing raised GitHub issues and pull requests through the process. Thank you as well for being a sounding board for ideas and questions. I really appreciate you dealing with my constant nagging, even when it was late at night for you.

Thank you, Aurea Munoz and Gytis Trikleris, for your help reviewing and improving the quality of the examples used in the chapters. A fresh set of eyes is always a good thing. Your contributions made the examples clear and easy to follow.

Thank you, Clement Escoffier and Stephane Epardaud, for your help reviewing and improving Chapter 4. I am extremely grateful that you were willing and able to step up and help with something that was outside your normal activities.

Acknowledgments

Thank you, Thomas Qvarnström, for all your support in helping to find reviewers and the fielding of questions. This was a long project, and I could always count on you to assist in whatever capacity was needed.

Thank you, Martijn Verburg, for graciously agreeing to write the foreword for the book. Having someone of your stature within the community embracing Quarkus says a lot about where Quarkus stands today.

Thank you to the Red Hat Developer team for all of your help and support, from ideation to the production of this finished product. Thank you, Colleen Lobner, for managing the process so smoothly, and thank you, Andy Oram, for the hours of editing and help to organize the content into its current form. The book would not have been a success without both of you.

Finally, thank you to my wife, Rachel, my daughter, Emily, and my son, Ethan, for supporting me throughout this process. You might not understand what I wrote, but you were always there to listen to me talk about it. You were excited when I was excited, which helped drive me to the finish line. I couldn't have done it without your loving support.

Daniel Oh

I would like to first and foremost thank Eric Deandrea for allowing me to write Chapter 5 and review the other chapters. This opportunity enabled me to take many technical deep-dives into the Spring and Quarkus frameworks. I would also like to thank my loving and patient wife and kids for their continued support, patience, and encouragement throughout the writing of this book.

Charles Moulliard

At the beginning of this book's adventure, Eric Deandrea and Syed Shaaf contacted me to review the book content and examples. Ultimately they convinced me to write the cloud material in Chapter 6. This has been a fantastic experience. I would like to thank them for giving me this opportunity, as speaking about the cloud is not the easiest task, nor is writing a chapter of a book as a non-native English speaker.

Thank you to the Snowdrop team and my associates, who took the time to play with the different examples and made proposals to improve the code and wording. Special thanks to Aurea Munoz Hernandez and Georgios Andrianakis for their support and encouragement.

Thank you, Ioannis Canellos, for quickly releasing a new version of Dekorate to fix a blocking problem discovered when a change was made by Kubernetes to move the API of the Ingress resource.

Thank you, Dimitri and Bruno, for giving me as a manager the time needed to be part of this project and to accept some delays when my weekly reports were not ready on time.

Chapter 1

Introducing Quarkus

Eric Deandrea

As interest grows in microservices and containers, Java developers have been struggling to make applications smaller and faster to meet today's requirements. In the modern computing environment, applications must respond to requests quickly and efficiently, be suitable for running in volatile environments such as virtual machines or containers, and support rapid development. Java, as well as popular Java runtimes, are sometimes considered inferior to runtimes in other languages such as Node.js and Go. But with additions to the Java frameworks over the past few years, Java can proudly retake its role as the primary language for enterprise applications.

This book was written using Quarkus version 2.1.4.Final and Spring Boot version 2.5.4.

One of these boundary-pushing frameworks is Quarkus, an open source project introduced by Red Hat, which has taken the Java community by storm. Quarkus combines developer productivity and joy similar to Node.js with the speed and performance of Go, enabling Java developers to build solutions targeting modern platforms and architectures.

This chapter will introduce Quarkus and highlight the key drivers behind its creation. Because many Java developers already know Spring and see Quarkus as an alternative, we'll showcase fundamental differences between Quarkus and Spring while also highlighting similarities. These differences make Quarkus an ideal runtime for Java applications that target modern platforms and architectures, such as microservices architectures, event-driven architectures, serverless, functions-as-a-service, edge computing, and IoT. The remaining chapters will help developers familiar with Spring Framework learn how to do familiar things with Quarkus while highlighting key fundamental differences.

Note: We use the general term "Spring" to cover Spring Framework and any other modules within the Spring ecosystem, including Spring Boot. Source code for all examples used throughout this book is located at https://github.com/quarkus-for-spring-developers/examples.

Brief History of Java

In today's world, application architects and developers have many technology choices to solve a business or technical problem. Java remains one of the most widely used programming languages to build applications today. Within Java, there are many tools and frameworks to help developers build applications.

Java was created when the cloud, containers, and container orchestration systems such as Kubernetes [1.1] did not exist. The historical Java stack consisted of applications deployed into Java application servers. In that architecture, each application server hosted multiple applications and provided additional services, such as transaction management, caching, connection pooling, and more. These application servers focused on providing the Java 2 Enterprise Edition (J2EE) Specification implementations, many of which developers deemed "heavyweight." These application servers took minutes to start up and consumed large amounts of memory, while their response times were measured in seconds.

Some time later, other application servers supporting a smaller subset of the J2EE specification emerged. Apache Tomcat, one of the most established open source projects and web containers, catered to web applications not requiring the full J2EE specification. These web applications still needed to deliver the dynamic web capabilities made possible by the Java Servlet API and Java Server Pages (JSP) specifications, part of the larger J2EE specification. These runtimes became increasingly popular because they required fewer system resources to run.

The past 15-20 years have seen many optimizations to the Java application stack and the Java Virtual Machine (JVM) to support running large heaps and highly dynamic frameworks that make decisions at runtime, particularly at application startup.

Introduction of Spring

Spring was introduced to the open source community in 2003. At that time, the J2EE specifications moved rather slowly, sometimes with years in between releases. Developers needed a faster release cadence for features. Spring brought in another popular trend where developers could build the same enterprise applications using a lighter-weight stack based on "plain old Java objects" (POJOs) while still complimenting some of the J2EE specifications.

Such a stack did not require the overhead and complexity of many of the J2EE specifications, such as Enterprise Java Beans (EJBs) or Message Driven Beans (MDBs). This stack made Spring Framework an ideal choice for building "lighter" applications. These applications did not require the same amount of memory and CPU resources needed by a Java application server because they could be deployed into any runtime supporting the Servlet specification.

For many years after, Spring continued to evolve and grow, providing new features that supported and simplified capabilities such as access to a database, asynchronous messaging, authentication and authorization of users, and web services. Many architects and developers have promoted Spring as arguably one of the most popular frameworks to build applications.

Emergence of Microservices

Over time, large, monolithic applications became harder to scale. Many teams needed to work in parallel on the same project in the same source code repository. Any change to one part of an application affected other developers working on other parts of the application, while also forcing testing and redeployment of the entire application. A single application deployment may have consisted of tens of servers and required hours of offline maintenance. The entire application needed additional instances deployed to handle increased loads that might affect only a subset of its capabilities.

Development teams were also becoming more agile, wanting to deliver business capabilities faster. Gone were the days of waterfall design, where an entire system needed to be designed before a line of code was written. Agile development helps enable teams to deliver smaller subsets of business capabilities faster. It also allows for smaller development teams, where each team can focus on a small subset of the capabilities an overall system provides.

These technical and nontechnical issues partly influenced the rise of microservices architectures.

A microservices architecture brings together many small applications to form a larger system, communicating using HTTP, TLS, or asynchronous messaging. A microservices architecture allows for the decoupling of functionality around specific business capabilities, where each capability can be owned, maintained, tested, and deployed independently. Additionally, microservices can more efficiently consume the available CPU and memory, which is not always possible in a monolithic application. A single business capability can scale independently of other capabilities.

Spring Boot

The shift towards microservices also led to the introduction of Spring Boot in 2014. One of the most powerful features of Spring Boot was its ability to package all the needed libraries within a single JAR file. This feature allowed quick bootstrapping of an application without deploying it into a separate Servlet container. Instead, Spring Boot embeds a web container (i.e., Apache Tomcat, Eclipse Jetty, or Undertow) inside the

application's JAR. All that was required was a JVM. This packaging mechanism helped reduce the operational overhead of installing and configuring a Java application server.

Coupled with this self-encapsulated runtime was the standardization of configuration across all of the underlying Spring modules. Additionally, Spring Boot gave developers and operations teams a standard convention for providing application configuration when moving from one environment to another, such as from integration testing (IT) to quality assurance (QA) to production.

Challenges of Microservices

A microservices architecture, while solving some challenges, introduces others. Microservices create complexity for developers and operations teams alike. The distributed architecture gives rise to challenges that need to be addressed. Microservices are also harder to operate efficiently.

Challenge #1: Composition of Distributed Applications

The composition of distributed applications communicating with each other introduces a new set of challenges. New and modern applications need to be designed around scalable distributed patterns to deal with these challenges. Users today expect near-instant response times and 100% uptime. The proliferation of distributed applications communicating amongst each other to fulfill a user's request goes directly against these expectations.

A set of design principles, known as Reactive Systems or Reactive Principles, attempt to address some of these challenges.

Reactive systems

Reactive Systems are flexible, loosely-coupled, scalable, and highly responsive, making them more maintainable and extensible. Reactive Systems are also resilient when faced with failure. The Reactive Manifesto [1.2] was introduced in 2014 as a set of guidelines and principles for building Reactive Systems. It defines Reactive Systems as:

1. **Responsive** - The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.
2. **Resilient** - The system stays responsive in the face of failure. This applies not only to highly-available, mission-critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.
3. **Elastic** - The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling

algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

4. **Message Driven** - Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency. This boundary also provides the means to delegate failures as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Following the reactive guidelines in each application helps the system as a whole, since the entire system is composed of all of the applications within the system. Reactive applications can scale much more efficiently than blocking applications. As load increases on a blocking application, so does the amount of memory and CPU resources needed by the application. More details about imperative/blocking and reactive/non-blocking can be found in Appendix section A.1.

Challenge #2: Infrastructure and Operations

Another major challenge in a microservices architecture is managing an increased number of applications: there are more applications to build, deploy, and manage. The increase in the number of applications increases the pressure on the system as a whole. Each application in the system therefore must optimize compute and memory resources. Reduction in boot-up time for applications is also more of a concern than ever before. Organizations continue to optimize their existing infrastructure to support these microservices while utilizing on-premise virtualization and cloud resources to reduce operating costs.

Optimization of resources is not the only problem operations teams face. Many additional concerns arise when running applications on a bare-metal server or in a virtual machine (VM). Each server or VM needs:

- To have a supported operating system installed.
- To be patched with updates continually.
- Installation and continuous maintenance for appropriate tools, libraries, and application runtimes, such as a JVM or other operating-system-level libraries.

How do operations teams manage multiple applications sharing the same system resources? What happens when updating the version of a shared library? An update to a shared library could affect all running applications. Furthermore, if one application misbehaves and consumes all available system resources, the other applications can become starved, potentially causing a crash. A microservices architecture exacerbates these challenges further.

Containers to the Rescue

Adopting a containerized runtime can help address many of these challenges by allowing isolation between applications. A container host can place resource restrictions on the running containers, ensuring that one container cannot consume all the resources available on a system for other containers running on the same host.

All of an application's required libraries and dependencies and their configurations are encapsulated inside a container. This encapsulation eliminates the need for these depen-

dencies to be installed, maintained, and configured on the host, allowing the application to provide everything it needs to operate. Additionally, container images are portable, deployable on multiple hosts without all the necessary setup and configuration.

Deploying a new version of an application may imply changes to the application itself or other pieces of the application runtime environment, such as the JVM or any of its required runtime dependencies. This process is greatly simplified when using container images, as everything is packaged within a versioned container image.

This versioning also reinforces the concept of *immutability*: Container images should not be modified at runtime. If a running container instance is modified and the container is subsequently restarted, all changes made are lost. Each instance of the same version of a container image should be the same as every other instance.

Containers on their own aren't a complete solution to all problems. Containers introduce additional concerns, such as operating and managing many containers at scale across many environments, both on-premises and in the cloud. Kubernetes, made open source in 2015 by Google, has become an industry-standard container orchestration platform for automating deployment, scaling, and operations of application containers across clusters of hosts.

Challenges of Java in containers

Some challenges of Java in the cloud, container, and Kubernetes world stem from Java's tendency to consume lots of memory and to suffer slower startup times than other languages and runtimes.

The immutable nature of containers forces developers to rethink the design and maintenance of Java applications. Highly reconfigurable and long-running applications with large heap sizes are no longer necessary. Moreover, essential parts of an application, such as the database driver or logging framework, are known in advance. In such situations, it is unnecessary to use frameworks that rely heavily on Java reflection [1.3]. Instead, application boot times can be accelerated and runtime footprints reduced by performing as much work at build time as possible.

Reflection in Java allows the application, at runtime, to inspect, manipulate, invoke, and even create new classes, interfaces, constructors, and methods. It also allows an application to invoke methods at runtime regardless of the access specifier (i.e., `public/private/protected`) used. Reflection contributes to startup time and memory usage because these inspections and invocations are being performed at runtime rather than at compile time. All the classes' metadata is retained in the JVM for the application's lifetime.

At their core, many Java-based frameworks embrace the dynamic flexibility Java provides. A lot happens in the first few seconds of Java application startup. For example, a Spring application can alter its runtime behavior by changing a few configuration properties. The dynamic runtime binding allowed by Java enables this adaptability.

How does a Java-based framework optimize itself to cater to today's reality? These frameworks need to rethink their underlying architectures. It is certainly not an easy task, considering the number of applications built on these frameworks and the inability to introduce breaking changes.

Quarkus: The Solution to Today's Challenges

The Java language and the Java platform have been very successful over the years, preserving Java as the predominant language in current use. Analysts have estimated the global application server market size at $15.84 billion in 2020, with expectations of growing at a rate of 13.2% from 2021 to 2028 [1.4]. Additionally, tens of millions of Java developers worldwide work for organizations that run their businesses using Java. Faced with today's challenges, these organizations need to adapt and adopt new ways of building and deploying applications. Forgoing Java for other application stacks isn't a choice for many organizations, as it would involve retraining their development staff and reimplementing processes to release and monitor applications in production.

Something needed to be done to accommodate rapid development, microservices architectures, and container runtimes, given Java's market size and the number of developers and organizations invested in Java. In late 2017, the previously listed challenges prompted Red Hat, a recognized leader in the open source community, to rethink how Java applications should be developed and deployed in today's cloud and containerized world. Red Hat's goal was to design a framework centered on Java but suited to today's operational concerns while not forcing developers to learn a completely new framework.

In March of 2019, after more than a year of internal development, Quarkus was introduced to the open source community. Quarkus is a Java framework tailored for OpenJDK HotSpot and GraalVM [1.5], boasting extremely fast boot times and low memory utilization. Many organizations since the initial release have seen the immediate value of Quarkus and joined the development effort.

Quarkus also offers near-instant scale-up and high-density utilization in container orchestration platforms such as Kubernetes. Many more application instances can be run within the same hardware resources used by previous applications.

After its initial debut, Quarkus underwent many enhancements over the next few months, culminating in a 1.0 release within the open source community in October 2019. As a new framework, Quarkus doesn't need to attempt to retrofit new patterns and principles into an existing codebase. Instead, it can focus on innovation. That innovation continues even today, with a 2.0 community release in June 2021.

Red Hat offers enterprise support for Quarkus. Visit https://access.redhat.com/products/quarkus for more information.

Container First

From the beginning, Quarkus was designed around container-first and Kubernetes-native philosophies, optimizing for low memory usage and fast startup times. As much processing as possible is done at build time, including taking a closed world assumption approach to building and running applications. This optimization means that, in most cases, the resulting JVM contains only code that has an execution path at runtime.

In Quarkus, classes used only at application startup are invoked at build time and not loaded into the runtime JVM. Quarkus also avoids reflection as much as possible, instead favoring static class binding. These design principles reduce the size, and ultimately the memory footprint, of the application running on the JVM while also enabling Quarkus to be "natively native."

Quarkus's design accounted for native compilation from the onset. It was optimized for using the native image capability of GraalVM [1.8] to compile JVM bytecode to a native machine binary. GraalVM aggressively removes any unreachable code found within the application's source code as well as any of its dependencies. Combined with Linux containers and Kubernetes, a Quarkus application runs as a native Linux executable, eliminating the JVM. A Quarkus native executable starts much faster and uses far less memory than a traditional JVM.

Similar native image capabilities in Spring are still considered experimental or beta as of this writing. Spring's effort to support native compilation doesn't provide all the same compile-time optimizations and design choices that make Quarkus extremely fast and memory-efficient when running on the JVM or within a native image.

GraalVM is a Java Virtual Machine for compiling and running applications written in different languages to a native machine binary. There are three distributions of GraalVM: GraalVM Community Edition (CE), GraalVM Enterprise Edition (EE), and Mandrel. GraalVM CE and EE have varying support and licensing requirements [1.6]. Mandrel [1.7] is a downstream distribution of GraalVM CE, supporting the same capabilities to build native executables but based on the open source OpenJDK. Mandrel makes GraalVM easy to consume by Quarkus applications by including only the GraalVM CE components that Quarkus needs. Red Hat offers commercial support for using Mandrel to build native Quarkus applications since the Quarkus 1.7 release in October 2020.

As Figure 1.1 shows, Quarkus boasts incredibly fast boot times coupled with extremely low resident set size (RSS) memory usage. It is an optimal choice for building highly scalable applications that require low CPU and memory resources. IDC performed a study [1.9] confirming that Quarkus can save as much as 64% of cloud resources. Coupled with a runtime platform like Kubernetes, more Quarkus applications can be deployed within a given set of resources than other Java applications. These applications can be traditional web applications, serverless applications, or even functions as a service.

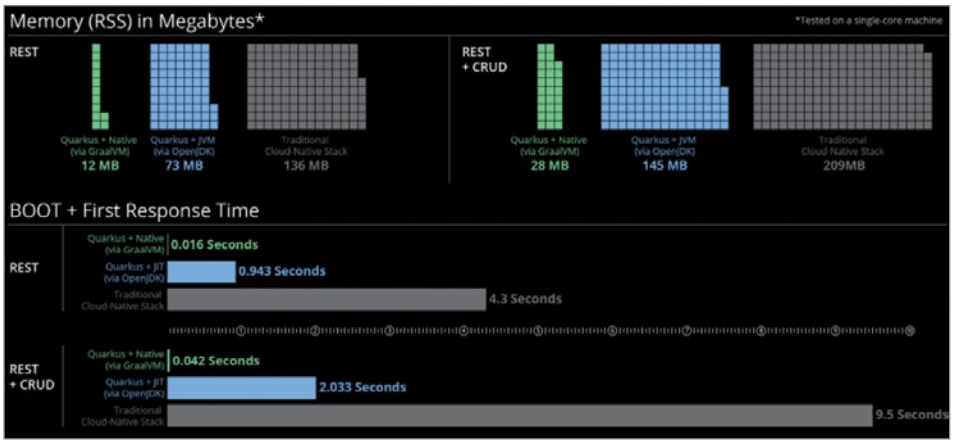

Figure 1.1: Quarkus RSS and times to first response.

Built on Standards

Quarkus rests on a vast ecosystem of technologies, standards, libraries, and APIs. Developers don't have to spend lots of time learning an entirely new set of APIs and technologies to take advantage of the benefits Quarkus brings to the JVM or native images. Among the specifications and technologies underlying Quarkus are Eclipse MicroProfile, Eclipse Vert.x, Contexts & Dependency Injection (CDI), Jakarta RESTful Web Services (JAX-RS), the Java Persistence API (JPA), the Java Transaction API (JTA), Apache Camel, and Hibernate, just to name a few.

Quarkus is also an ahead-of-time compilation (AOT) platform, optimizing code for the JVM as well as compiling to native code for improved performance. All of the underlying technologies are AOT-enabled, and Quarkus continually incorporates new AOT-enabled technologies, standards, and libraries.

All the technologies and capabilities needed for building microservices are available and optimized for Quarkus. These technologies, such as Keycloak for authentication and authorization, or Infinispan for distributed caching, take advantage of all the compile-time and runtime optimizations that have already been discussed. The technologies are optimized for use within Quarkus, regardless of whether they run in the HotSpot JVM or as a native image. Third-party framework and library developers can take advantage of this AOT platform to optimize them for Quarkus.

Eclipse MicroProfile

Eclipse MicroProfile [1.10] is an open source community specification for Enterprise Java microservices. It brings together a community of individuals, organizations, and vendors collaborating to iterate and innovate in short cycles to propose new standard APIs and functionality around capabilities that Java microservices need. Some of the organizations contributing to the MicroProfile specification process include Red Hat, IBM, Payara, Tomitribe, Oracle, Microsoft, Fujitsu, and Lightbend.

MicroProfile also incorporates concepts and efforts from other communities, such as Istio [1.11] and Netflix OSS [1.12].

Some MicroProfile capabilities [1.13] include application-level metrics, health checks, distributed tracing, standardized external configuration, fault tolerance, JSON Web Token (JWT) propagation, OpenAPI integration, and CDI.

MicroProfile is different from Java Enterprise Edition (Java EE) and Jakarta EE because it doesn't require a Java EE application server. Additionally, because MicroProfile is not a monolithic architecture, it allows developers to start small and build a minimal microservice or cut down existing Java EE applications into core components. Each core component may become its own microservice. This approach can be a useful modernization approach when moving from a monolithic architecture to microservices.

The MicroProfile specification process is lightweight to facilitate faster innovation within the community and contains no reference implementation. It is up to the community to provide implementations of the MicroProfile specification. Quarkus, Wildfly, Red Hat JBoss Enterprise Application Platform (via expansion packs), Open Liberty, Payara Micro, and Apache TomEE are some of the current MicroProfile implementations.

Various modules within the Spring and Spring Cloud ecosystems also provide many capabilities similar to those within the MicroProfile specification. These modules evolved because developers wanted higher-level abstractions than the ones provided in the different Java EE specifications. Developers familiar with these Spring modules will find many similarities in the Eclipse MicroProfile specifications.

Developer Productivity and Joy

Quarkus focuses on more than just delivering sets of features; every feature should be simple, have little-to-no configuration, and be as intuitive as possible to use. It should be trivial to do trivial things while relatively easy to do more complex things, allowing developers more time to focus on their domain expertise and business logic.

One major productivity issue that most Java developers face today is the traditional Java development workflow. For most developers, this process looks like Figure 1.2.

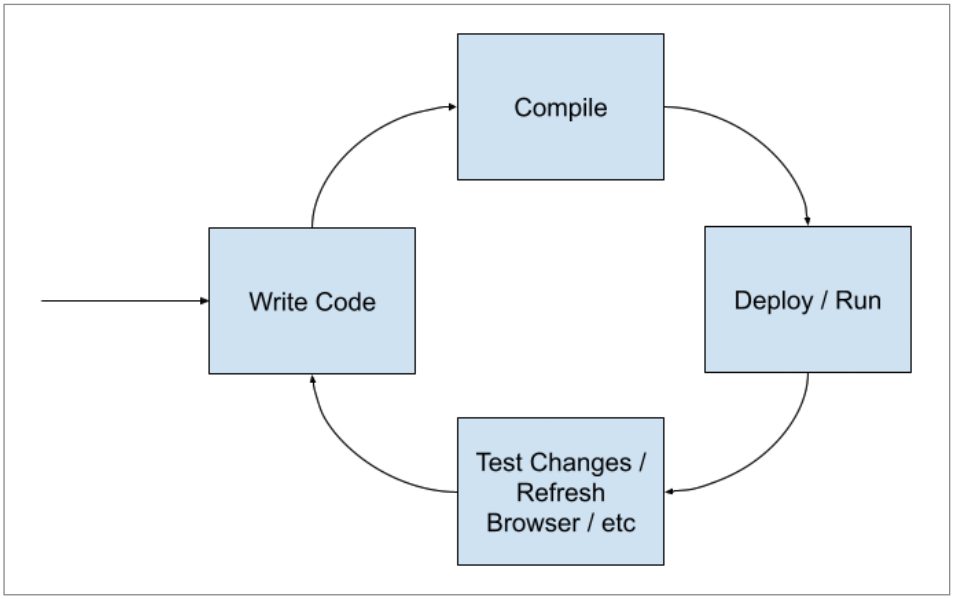

Figure 1.2: Typical developer workflow.

The issue is that the *Compile* and *Deploy/Run* cycles can take time, sometimes up to a minute or more. This delay is wasted time where a developer could be doing something productive. Quarkus's *Live Coding* feature solves this issue. Quarkus will automatically detect changes made to Java files, including class or method refactorings, application configuration, static resources, or even classpath dependency changes. When such a change is detected, Quarkus transparently recompiles and redeploys the changes. Quarkus redeployments typically happen in under a second. Using Quarkus, the development workflow now looks like Figure 1.3.

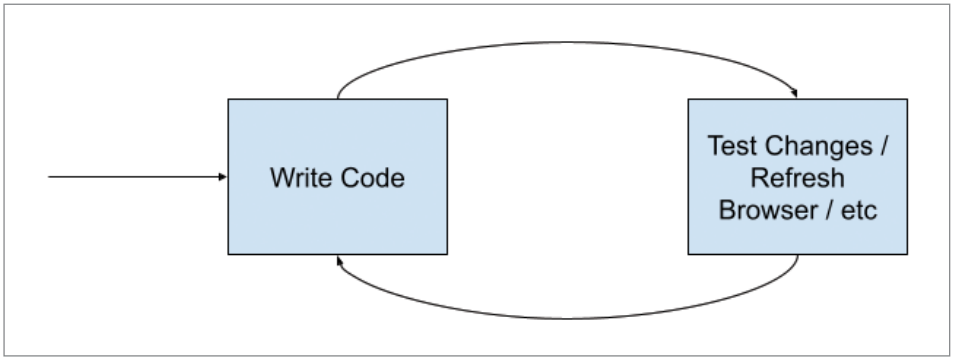

Figure 1.3: Quarkus developer workflow.

Quarkus enables this workflow out of the box for Java files, application configuration, and static resources. Any changes made are immediately reflected in the running application. Quarkus also provides its *Live Coding* feature for Quarkus applications running on remote hosts. A developer can connect their local environment to a remote application and see changes immediately reflected in the remote application.

Quarkus's Dev Mode provides an additional capability, called Dev Services, that will automatically bootstrap various containers an application might need, such as a database container or other middleware component, while also setting all of the required configuration properties for running Dev Mode or when running tests.

This capability grants developers a tight feedback loop and removes the additional overhead of having to start services manually and provide configurations while quickly iterating on code they're writing. As of this writing, Quarkus Dev Services supports database containers (discussed in Chapter 4), Apache Kafka (discussed in Chapter 5), OpenID Connect (Keycloak), Advanced Message Queuing Protocol (AMQP), Redis, HashiCorp Vault, and Apicurio Registry. Future Quarkus versions will extend this capability to include other middleware components as well.

Additionally, Quarkus takes this concept a step further by introducing continuous testing [1.14] to further facilitate test-driven development. Quarkus understands which tests are affected by classes and methods within the application. As changes are made to the application source code, Quarkus can automatically rerun affected tests in the background, giving developers instant feedback about the code they are writing or modifying.

Following the philosophy of simplicity and enhancing developer productivity, building an application into a native image is extremely simple. All the heavy-lifting and integration to consume GraalVM is done for you by the Quarkus build tools via Maven or Gradle. Developers or CI/CD systems simply need to run a build, just like any other Java build, to produce a native executable. Tests can even be run against the built artifact.

Unifying Reactive and Imperative

With Spring, a developer needs to decide up front, before writing a line of code, which architecture to follow for an application. This choice determines the entire set of libraries that a developer uses in a Spring application. In some cases, these libraries vary depending on the chosen architecture. Spring has always remained backward-compatible with all the imperative APIs built over the years. On the reactive side, Spring was able to start fresh while trying to reuse pieces of the framework where it could. As a result, Spring is left trying to maintain two versions of its APIs and its documentation.

Quarkus does not have such limitations, since it was born in the reactive era. Quarkus, at its core, is based on a fully reactive and non-blocking architecture powered by Eclipse Vert.x [1.15]. Eclipse Vert.x is a Java toolkit for building extremely resource-efficient reactive applications on the JVM. Quarkus integrates deeply with Vert.x, allowing developers to utilize both blocking (imperative) and non-blocking (reactive) libraries and APIs. In most cases, developers can use both imperative and reactive APIs within the same classes. Quarkus ensures that the imperative APIs will block appropriately while the reactive APIs remain non-blocking. The Quarkus reactive architecture [1.16] implements a proactor pattern [1.17] that switches to a worker thread when needed. Listing 1.1 illustrates this.

Listing 1.1: Unifying imperative and reactive.

```
@Path("/resources")
public class MyResourceClass {
  @Inject
  SayService say;

  @Inject
  @Stream("kafka-topic")
  Publisher<String> sseSay;

  @GET
  @Path("/blocking/{name}")
  @Produces(MediaType.TEXT_PLAIN)
  @Blocking
  public String helloBlocking(@PathParam("name") String name) {
    return say.hello(name);
  }

  @GET
  @Path("/reactive/{name}")
  @Produces(MediaType.TEXT_PLAIN)
  public Uni<String> helloReactive(@PathParam("name") String name) {
    return Uni.createFrom()
              .item(name)
              .onItem().transform(say::hello);
  }

  @GET
  @Path("/sse")
  @Produces(MediaType.SERVER_SENT_EVENTS)
  public Publisher<String> sseHello() {
    return sseSay;
  }
}
```

Note: Don't worry if you are unfamiliar with the code in this listing. It is shown merely to illustrate that you can mix and match blocking and non-blocking operations within a single Java class. Later chapters will dive deeper into these specific topics and the libraries used.

As you can see, the class `MyResourceClass` contains both blocking and non-blocking methods. The `helloBlocking` method calls a method on the injected `SayService` and returns some result as a `String`. This method blocks until the `SayService.hello()` method completes. Quarkus ensures that when the method blocks, it is moved off onto another `Thread` where it can wait for the operation to complete before returning the result on the main application `Thread`.

The `helloReactive` method returns an asynchronous reactive pipeline that will produce a single `String` value at some point in the future. Finally, the `sseHello` method returns an asynchronous pipeline that will emit `Strings` as server-sent events as new events arrive on the `kafka-topic` Apache Kafka topic.

Summary

Java has evolved since its introduction in 1995, but more importantly, the environments that host applications and the applications' requirements for responsiveness and scaling have evolved as well. Some trade-offs made in Java's design at the onset affect how Java is perceived today. Things that were important in those days aren't as important anymore. Quarkus was invented to address today's challenges and problems, and aims to solve those challenges without forcing developers to learn an entirely new programming language.

References

[1.1] "Kubernetes": https://kubernetes.io

[1.2] "The Reactive Manifesto," September 2014: https://www.reactivemanifesto.org

[1.3] "Using Java Reflection," January 1998: https://www.oracle.com/technical-resources/articles/java/javareflection.html

[1.4] "Application Server Market Size & Share Report, 2016-2028": https://www.grandviewresearch.com/industry-analysis/application-server-market

[1.5] "GraalVM": https://www.graalvm.org/docs/introduction

[1.6] "GraalVM FAQ": https://www.graalvm.org/faq

[1.7] "Mandrel: A community distribution of GraalVM for the Red Hat build of Quarkus," June 2020: https://developers.redhat.com/blog/2020/06/05/mandrel-a-community-distribution-of-graalvm-for-the-red-hat-build-of-quarkus

[1.8] "GraalVM Native Image": https://www.graalvm.org/reference-manual/native-image

[1.9] Dayaratna, Arnal. "Red Hat Quarkus Lab Validation." IDC, May 2020: https://red.ht/idc-quarkus-study

[1.10] "Eclipse MicroProfile": https://microprofile.io

[1.11] "Istio": https://istio.io

[1.12] "Netflix OSS": https://github.com/netflix

[1.13] "Eclipse MicroProfile Projects": https://microprofile.io/projects

[1.14] "Quarkus 2.0.0.Alpha1: Continuous Testing": https://quarkus.io/blog/quarkus-2-0-0-final-released/#continuous-testing

[1.15] "Eclipse Vert.x": https://vertx.io

[1.16] "Quarkus Reactive Architecture": https://quarkus.io/version/main/guides/quarkus-reactive-architecture

[1.17] "Proactor Pattern": https://en.wikipedia.org/wiki/Proactor_pattern

Chapter 2

Getting Started with Quarkus

Eric Deandrea

The tools available in the Spring ecosystem make it easy for a developer to start building applications. Additionally, Spring developers are familiar with this ecosystem's many conventions, some of which are very similar in Quarkus. A developer familiar with Spring should quickly be able to become familiar with similar concepts and conventions in Quarkus. This chapter showcases some of the tools in Quarkus, while also explaining the key differences between Quarkus and Spring capabilities.

Prerequisites

Quarkus requires a setup similar to any other typical Java application: an Integrated Development Environment (IDE), a JDK 11+ distribution, and a build tool such as Apache Maven 3.6.3+ (Maven) or Gradle. Unless otherwise stated in a specific example, all examples in the book and sample repository (https://github.com/quarkus-for-spring-developers/examples) use Java 11 and Apache Maven 3.8.1. The examples use Maven's wrapper feature, so a preinstalled version of Maven is not required. Each example project contains `mvnw` scripts for Windows and Unix/Linux/macOS.

Building a native executable requires a distribution of GraalVM. Chapter 1 discussed GraalVM and its different distributions. You need to install either a container runtime environment or one of those GraalVM distributions to run the native image examples throughout this book.

Additionally, both Quarkus and Spring support using Kotlin as the programming language to build an application.

IDEs

Quarkus tooling is available for most major IDEs, including Microsoft Visual Studio Code (VSCode), JetBrains IntelliJ IDEA (IntelliJ), and Eclipse, shown in Table 2.1.

Table 2.1: Quarkus tooling for major IDEs.

IDE	Quarkus tooling available
Microsoft VSCode (and web variants, such as Eclipse Che and Red Hat CodeReady Workspaces)	Quarkus Tools for Visual Studio Code [2.1]
JetBrains IntelliJ	Quarkus Framework support (Ultimate Edition only) [2.2] Quarkus Tools [2.3] (all editions)
Eclipse	Quarkus Tools (part of JBoss Tools) [2.4] [2.5]

Extensions

Spring Boot applications discover various types of annotated classes and methods, known as beans, at runtime. Dependency injection mechanisms inject beans into other beans. The same is true for Quarkus. Quarkus and Spring applications are made up of beans that perform various functions, such as bootstrapping an embedded HTTP server. So where do these beans come from?

In Spring, a developer provides some beans as part of the application, but other beans are part of the underlying framework's Spring Boot Starters [2.6]. Starters contain bean definitions and configurations that are added to the application's runtime classpath and are automatically discovered and applied at runtime during application startup.

In Quarkus, an extension provides similar behavior, adding new functionality or configuration to an application. An extension, similar to a Starter, is a dependency included in an application, enriching an application and enforcing opinions and sensible defaults.

The Red Hat Developer Sandbox (http://red.ht/dev-sandbox) is a Red Hat OpenShift environment and developer IDE that is available in the cloud, free to use. Once you register, navigating to https://workspaces.openshift.com/f?url=https://github.com/quarkus-for-spring-developers/examples in your browser will load an IDE with all necessary tooling installed.

Note: An IDE is not required but assists in the development of Quarkus applications. No particular IDE is assumed for any of the examples in this book. More information about the tooling can be found in Appendix A.2.

There is, however, one fundamental difference between a Spring Boot Starter and a Quarkus extension. A Quarkus extension consists of two distinct parts: build-time augmentation, referred to as the deployment module, and the runtime container, referred to as the runtime module. The majority of an extension's work is done within the deployment module when an application is built.

A Quarkus extension loads and scans the compiled application's configuration and bytecode and all of its dependencies during build time. At this stage, the extension can read configuration files, scan classes for annotations, parse descriptors, and even generate additional code. Once all metadata has been collected, the extension can preprocess bootstrap actions, such as a dependency injection framework or REST endpoint configuration. The bootstrap result is directly recorded into bytecode and made part of the final application package. The work performed by the deployment module is what makes Quarkus super fast and memory-efficient, while also making native image support trivial.

Does this mean that Quarkus applications can't supply configuration at runtime? They certainly can, but to let Quarkus best optimize the application, careful consideration needs to be taken to decide which configuration options should be read at build time versus leaving them to be overridable at runtime.

Configurations such as a database host, port, username, and password should be overridable at runtime because those settings are part of the environment and change from one deployment to another. Many other configuration properties, such as whether to enable caching or setting JDBC drivers to use, should be fixed at build time because they are design choices and are not subject to change.

Quarkus, like Spring Boot, has a vast ecosystem of extensions for many of today's commonly-used technologies. Table 2.2 lists some of the most common Quarkus extensions and the Spring Boot Starters providing similar functionality. A list of all available Quarkus extensions can be found at https://code.quarkus.io or by running the `quarkus:list-extensions` Maven goal or the `listExtensions` Gradle task. As of this writing, Quarkus has more than 400 extensions.

Table 2.2: Common Quarkus extensions.

Quarkus extension	Spring Boot Starter
`quarkus-resteasy-jackson`	`spring-boot-starter-web` `spring-boot-starter-webflux`
`quarkus-resteasy-reactive-jackson`	`spring-boot-starter-web` `spring-boot-starter-webflux`
`quarkus-hibernate-orm-panache`	`spring-boot-starter-data-jpa`
`quarkus-hibernate-orm-rest-data-panache`	`spring-boot-starter-data-rest`
`quarkus-hibernate-reactive-panache`	`spring-boot-starter-data-r2dbc`
`quarkus-mongodb-panache`	`spring-boot-starter-data-mongodb` `spring-boot-starter-data-mongodb-reactive`
`quarkus-hibernate-validator`	`spring-boot-starter-validation`
`quarkus-qpid-jms`	`spring-boot-starter-activemq`
`quarkus-artemis-jms`	`spring-boot-starter-artemis`
`quarkus-cache`	`spring-boot-starter-cache`
`quarkus-redis-client`	`spring-boot-starter-data-redis` `spring-boot-starter-data-redis-reactive`
`quarkus-mailer`	`spring-boot-starter-mail`
`quarkus-quartz`	`spring-boot-starter-quartz`
`quarkus-oidc`	`spring-boot-starter-oauth2-resource-server`
`quarkus-oidc-client`	`spring-boot-starter-oauth2-client`
`quarkus-smallrye-jwt`	`spring-boot-starter-security`

Quarkus also provides its extension framework to third-parties to build and deliver their own extensions through the Quarkiverse [2.7]. By providing a Quarkus extension, third-party technologies can be optimized for the ahead-of-time (AOT) processing Quarkus promotes.

Spring API Extensions

Quarkus provides a set of extensions for various Spring APIs. These extensions help simplify the process of learning Quarkus or migrating existing Spring applications to Quarkus, capitalizing on a developer's Spring knowledge to accelerate the learning curve to adopt Quarkus. The Quarkus Spring API extensions aren't intended to track all Spring features one-to-one. They are an adapter layer between some common Spring APIs and the underlying Quarkus runtime, adapting the Spring annotations, classes, and interfaces to supporting Quarkus technologies, such as JAX-RS or MicroProfile. Table 2.3 highlights these extensions.

Table 2.3: Quarkus Spring API extensions.

Extension name	Description
Quarkus Extension for Spring Boot Properties [2.8]	Use Spring Boot properties annotations to configure your application.
Quarkus Extension for Spring Cache [2.9]	Use Spring Cache annotations in your application.
Quarkus Extension for Spring Cloud Config Client [2.10]	Use properties from Spring Cloud Config Server at runtime.
Quarkus Extension for Spring Dependency Injection [2.11]	Define dependency injection using Spring annotations.
Quarkus Extension for Spring Data JPA [2.12]	Use Spring Data JPA annotations and repositories to create your data access layer.
Quarkus Extension for Spring Data REST [2.13]	Generate JAX-RS resources for a Spring Data application.
Quarkus Extension for Spring Scheduled [2.14]	Schedule tasks with Spring Scheduling.
Quarkus Extension for Spring Security [2.15]	Secure your application with Spring Security.
Quarkus Extension for Spring Web [2.16]	Use Spring Web annotations to create REST services.

As mentioned previously, these extensions don't implement every Spring API. However, they are useful if you have an existing Spring application you want to migrate to Quarkus. These extensions allow an application to be migrated slowly over time instead of a "rip and replace," where the entire application would have to be converted or rewritten.

While these extensions are undoubtedly useful for Spring developers wanting to try out Quarkus, they are not the focus of this book. The remainder of this book focuses on describing a pattern or solution in Spring and showcasing how it could be implemented using Quarkus. A free cheat sheet available for download [2.18] discusses these capabilities in greater detail.

Red Hat's migration toolkit for applications [2.17] is a free tool to help in such situations. It can analyze an existing Spring Boot application and offer suggestions for utilizing the Quarkus Spring Extensions best to migrate the application to Quarkus. In some cases, code changes may not be necessary. Applications using these extensions will be small, fast, and memory-efficient, just like any other Quarkus application.

Creating a New Project

Developers familiar with Spring have most likely used the Spring Initializr (https://start.spring.io) at some point. The Spring Initializr can be used directly from the website, as shown in Figure 2.1, or via plug-ins for most major IDEs.

Figure 2.1: Spring Initializr (start.spring.io).

Quarkus provides very similar capabilities at https://code.quarkus.io. The project generator can be used directly from the website, as shown in Figure 2.2, or via plug-ins for all of the IDEs mentioned previously.

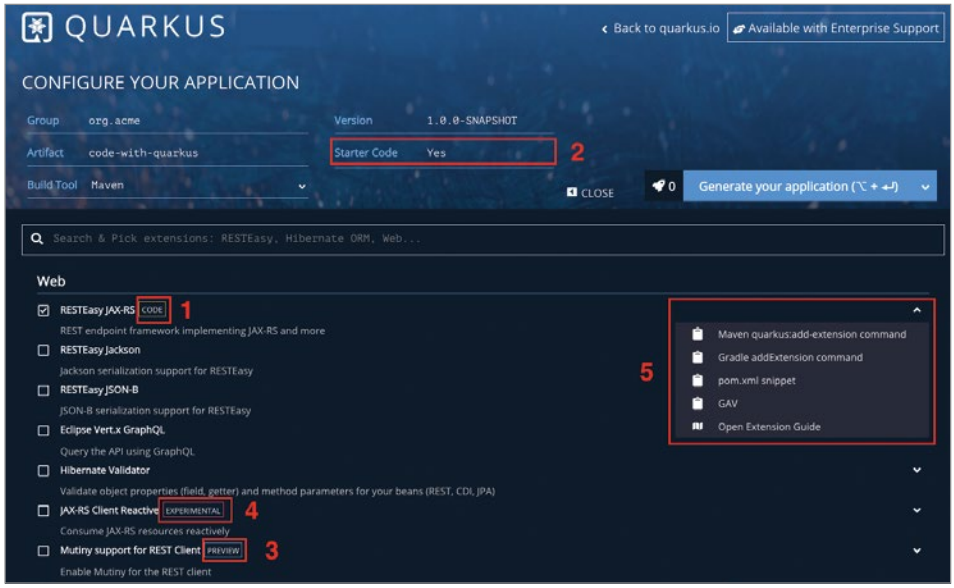

Figure 2.2: code.quarkus.io.

As shown in Figure 2.2, Red Hat offers enterprise support for Quarkus. Visit https://access.redhat.com/products/quarkus for more information.

As with the Spring Initializr, a developer can describe the type of application to build, the Maven coordinates for the project, and the build tool of choice. There is also a curated list of technologies to include in the application. These technologies are Quarkus extensions.

The Quarkus extension framework provides extension authors with additional capabilities aside from implementing the extension itself. An extension author can provide example code and a guide on how to use the extension.

Many Spring applications are built using Kotlin. To use Kotlin with Quarkus, select the Kotlin extension.

A few highlights of Figure 2.2 are:

1. This icon next to an extension indicates that the extension provides example code.
2. If the **Starter Code** selection is set to **Yes**, example code for each selected extension that provides it will be included in the generated project. This is a capability that, as of this writing, Spring Initializr does not have.

3. Some extensions are newer to the community and might be marked **PREVIEW**. This means the extension is still a work in progress. API or configuration properties may change as the extension matures.
4. Other extensions are brand new to the community and are marked **EXPERIMENTAL**. This means the extension provides no guarantee of stability or long-term presence in the platform until it matures.
5. The user interface also shows additional details about each extension, including adding the extension to an existing project with a single command. This convenience is also missing in the Spring Initializr, as of this book's writing. Additionally, there is a link to the Extension Guide for the extension, containing documentation and information about the extension and some code samples.

Generating a Project

Perform the following steps to generate a project. Once you have generated one, we will examine its structure and become familiar with some of the tooling that brings back joy to developers. The examples do not assume any particular IDE. Instead, they rely on your web browser and terminal.

1. Navigate to https://code.quarkus.io in your browser.
2. In the **Artifact** section, enter `chapter-2-simple-project`.
3. In the list of extensions under **Web**, select `RESTEasy JAX-RS`. (RESTEasy JAX-RS provides capabilities similar to the Spring Web Starter). Figure 2.3 shows what your browser should look like at this point.
4. Click the **Generate your application** button at the top.
5. You will be prompted to download the project. Save it somewhere on your computer.
6. Once downloaded, extract the contents of `chapter-2-simple-project.zip` into a folder. A directory called `chapter-2-simple-project` will be created.

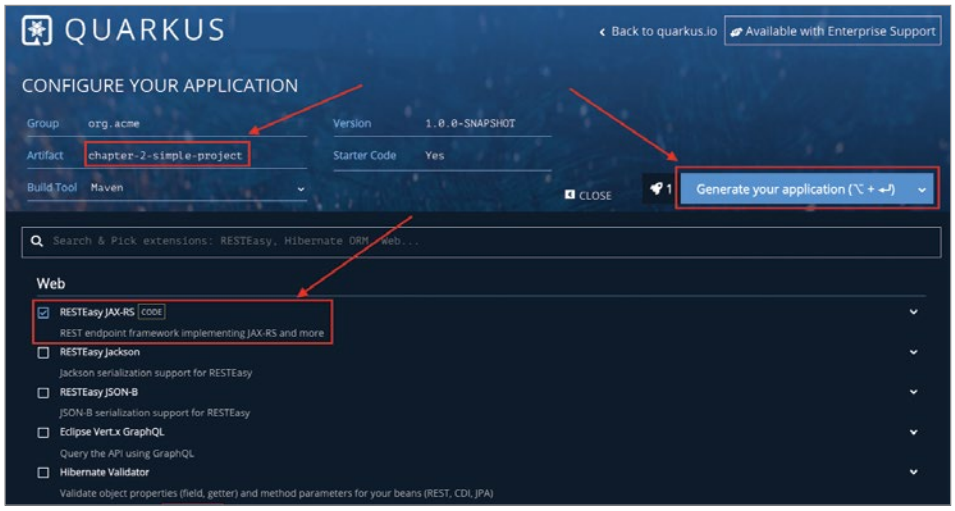

Figure 2.3: Screenshot of code.quarkus.io selections.

Project Structure

The source structure of a Quarkus application is very similar to the Spring Boot application structure. One main difference is that there is no "application" Java file containing the `main` method. Spring Boot requires such a class, but Quarkus does not. Table 2.4 details some of these files and directories.

Table 2.4: Project source files and directories.

File/directory	Description
`README.md`	README file containing instructions on how to build and run the application. It also includes instructions on how to build a native image.
`pom.xml`	Maven build file containing all repositories, build plugins, and project dependencies.
`src/main/docker`	Sample `Dockerfiles` for use in different modes (JVM, native image, etc.).
`src/main/java`	The root directory for all source code to be packaged within the application.
`src/main/resources`	The directory containing all non-source code packaged with the application.
`src/main/resources/application.properties`	Application configurations, unified in a single file. Spring Boot allows you to have multiple files, using profiles in the filenames to distinguish one from another. Profiles will be discussed in a later section. Quarkus additionally supports YAML configuration files using the Quarkus YAML Configuration extension.
`src/main/resources/META-INF/resources`	The directory containing any static content needing to be packaged with the application.
`src/test/java`	The directory containing all test code for the application.

Note: Maven was used to generate the project, so the details in Table 2.4 list some files specific to Maven. You can also use Gradle. In that case, you will see files specific to Gradle instead.

The extension we selected, RESTEasy JAX-RS, provides example code. Because of this, some corresponding sample code is inside both `src/main/java` and `src/test/java`, as well as an `index.html` file inside `src/main/resources/META-INF/resources`. Later chapters in this book discuss specifics about JAX-RS and RESTEasy.

Controlling Dependency Versions

Those familiar with Spring before Spring Boot might recall difficulty maintaining and testing the versions of the various Spring modules combined with the versions of other technologies an application may be using.

Consider an example where an application needed to use Spring MVC, Undertow, Hibernate, Simple Logging Facade for Java (SLF4J), Logback, the Spring cache abstraction, Infinispan, and a database driver. It was up to the developer to figure out which dependencies to include within the project, and which versions of those dependencies were compatible with all the other dependencies. It was not a trivial task, especially over time as a project evolved and needed to upgrade its dependencies to newer versions.

One of Spring Boot's core features was that it tested and managed the compatible dependency versions between all the technologies within the ecosystem, including the versions of Spring modules and popular community projects. An application merely had to depend on a specific version of Spring Boot. Additionally, Spring Boot provided the flexibility for applications to choose non-default versions of dependencies if required.

Today this is implemented using Maven Bill of Materials (BOM) POMs [2.19]. Spring Boot provides a parent POM [2.20] containing properties that declare versions of each module. By importing this parent POM, an application inherits the default versions of the dependencies without specifying a version. Both Maven and Gradle support importing BOM POMs into a project. Additionally, a Spring Boot user can use Maven's importing dependencies capability [2.21] rather than declaring a parent POM.

Quarkus provides this same capability with the `quarkus-universe-bom`. Open the `pom.xml` file from the `chapter-2-simple-project`. Inside you will find the section shown in Listing 2.1.

Listing 2.1: quarkus-universe-bom.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>${quarkus.platform.group-id}</groupId>
      <artifactId>${quarkus.platform.artifact-id}</artifactId>
      <version>${quarkus.platform.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The `groupId`, `artifactId`, and `version` attributes refer to properties defined further up in the `<properties>` section of the `POM`. Further down in pom.xml in the `<dependencies>` section, notice that each `<dependency>` does not include a `<version>` element. To update the version of Quarkus and all the dependencies, simply change the `quarkus.platform.version` property to a new Quarkus version.

Quarkus Dev Mode and Live Coding

As mentioned in Chapter 1, Quarkus helps facilitate faster development cycles through a feature called *Live Coding*. Let's examine that feature.

In a terminal, `cd` into the `chapter-2-simple-project` directory and execute the following command:

```
$ ./mvnw quarkus:dev
```

The application will compile and start up. Once started, it will show the Quarkus banner:

```
--  ----  --  -----   ---  -- ----  ------
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (Quarkus Main Thread) chapter-2-simple-project 1.0.0-SNAPSHOT
  on JVM (powered by Quarkus 2.xx.x.Final) started in 1.507s.
  Listening on: http://localhost:8080
INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, resteasy,
  smallrye-context-propagation]
```

Notice the following in the output:

1. `started in 1.507s`
   - Quarkus applications typically start much faster than Spring Boot applications consisting of similar capabilities.
2. `Listening on: http://localhost:8080`
   - The application creates an HTTP server listening on port 8080 of the local machine because the RESTEasy JAX-RS extension was selected.

3. `Profile dev activated`
   - Profiles are discussed in a later section in this chapter.
4. `Installed features: [cdi, resteasy, smallrye-context-propagation]`
   - Quarkus always outputs what features are enabled from the extensions used in the application. In this case, the application uses the CDI feature, enabling core dependency injection capabilities, and the RESTEasy feature, enabling RESTful or JAX-RS capabilities provided by RESTEasy. Later sections in this chapter cover dependency injection. Future chapters discuss JAX-RS and RESTEasy in more detail.

Open your browser to http://localhost:8080. Figure 2.4 shows the displayed congratulations page. This page shows you information about your project and links to the guide used by the RESTEasy JAX-RS extension.

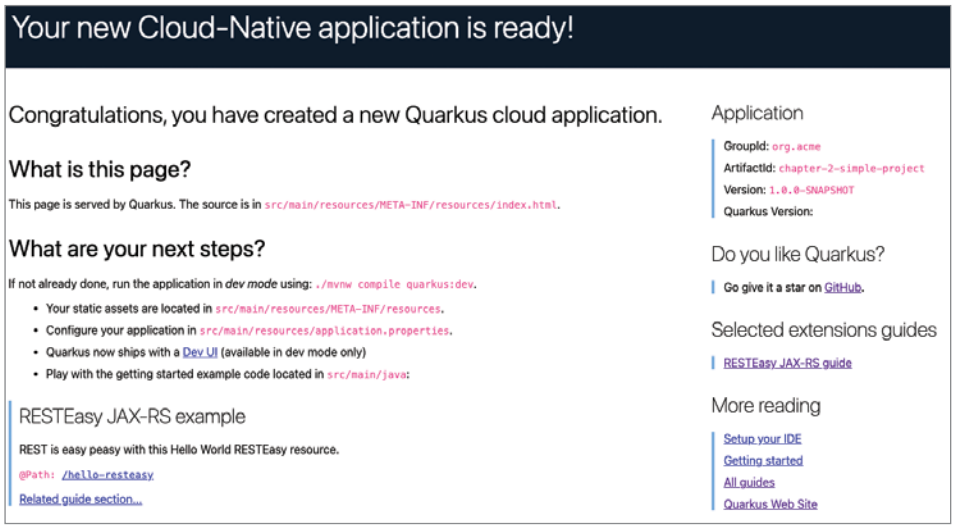

Figure 2.4: Quarkus congratulations page.

Note: Your screenshot may look slightly different based on the version of Quarkus being used at the time the project was generated.

Next, either click the `/hello-resteasy` link near the bottom of the page or append `/hello-resteasy` to the URL in your browser (http://localhost:8080/hello-resteasy). You should see the text `Hello RESTEasy` in the browser.

Now let's introduce a change and see how the Quarkus Live Coding feature works. In your favorite editor or IDE, open the `src/main/java/org/acme/GreetingResource.java` file, as shown in Listing 2.2.

Listing 2.2: GreetingResource.java.

```
@Path("/hello-resteasy")
public class GreetingResource {
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return "Hello RESTEasy";
  }
}
```

Later chapters will cover the specifics of JAX-RS, but for now, simply change the text `Hello RESTEasy` to `Hello Quarkus`. Save your editor, then go back to your browser and refresh the page. You should now see the text `Hello Quarkus`. Quarkus continues detecting changes to the project behind the scenes, recompiling the application and redeploying it to the running application runtime.

Next, return to the terminal where the `./mvnw quarkus:dev` command is running and notice some log messages about the update that was made:

```
INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] (vert.x-worker-thread-0) Changed
  source files detected, recompiling [examples/chapter-2/chapter-2-simple-project/
  src/main/java/org/acme/GreetingResource.java]
INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] (vert.x-worker-thread-0) Hot
  replace total time: 0.476s
```

The procedure in this section illustrates Quarkus's continuous development loop, which enables developers to be more productive. A developer does not have to stop, recompile, then restart the application when making changes. Everything happens in the background once Quarkus notices the source file has changed.

Dev UI

Running Quarkus in Dev Mode enables the Quarkus Dev UI. The Dev UI is a landing page for browsing endpoints offered by various extensions, conceptually similar to what a Spring Boot actuator might provide. A key difference is that a Spring Boot actuator can be enabled when running in production, whereas the Dev UI in Quarkus is enabled only when running in Dev Mode.

Open a browser to http://localhost:8080/q/dev to see the Dev UI. It will resemble Figure 2.5.

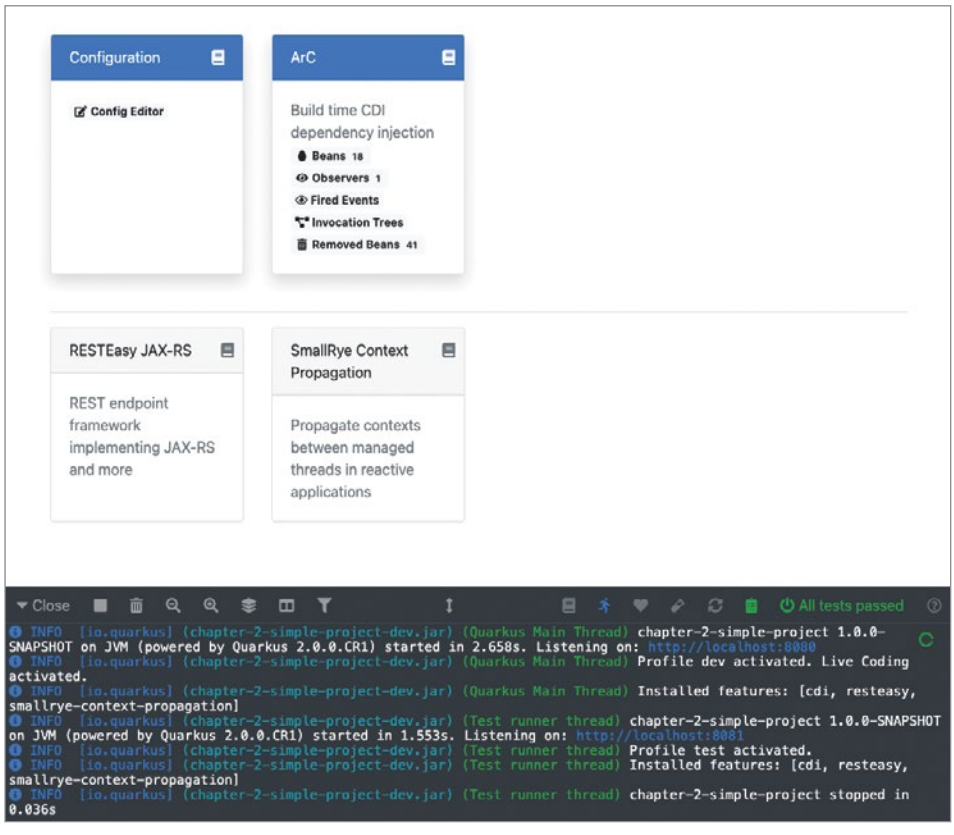

Figure 2.5: Quarkus Dev UI.

The Quarkus Dev UI allows a developer to quickly visualize all the extensions currently loaded, see their status, and go directly to their documentation. Each extension can

add custom runtime information in the overview, full custom pages, and interactive pages with custom actions.

Additionally, the Dev UI provides easy streaming access to the application's log file and quick access to the application's test suite. A developer can toggle test execution on and off, trigger test execution, and view the current test execution status within the UI.

Configuration

What if we wanted to externalize the `Hello Quarkus` message being output rather than hard-coding it in source code? This is where Quarkus configuration, similar to Spring's configuration, comes into play.

Single Property Values

We'll return now to the example for this chapter to see how to configure Quarkus. Open `src/main/resources/application.properties`. Right now, the file is empty. Add the following line to the file:

```
greeting.name=Quarkus (properties)
```

Next, return to `src/main/java/org/acme/GreetingResource.java`. Add the following block above the `hello()` method:

```
private final String greeting;

public GreetingResource(@ConfigProperty(name = "greeting.name") String greeting) {
  this.greeting = greeting;
}
```

Then, inside the `hello()` method, change

```
return "Hello Quarkus";
```

to be

```
return "Hello " + this.greeting;
```

After all the modifications, `GreetingResource.java` should resemble Listing 2.3.

Listing 2.3: GreetingResource.java with configuration property.

```
@Path("/hello-resteasy")
public class GreetingResource {
  private final String greeting;

  public GreetingResource(@ConfigProperty(name = "greeting.name") String
    greeting) {
    this.greeting = greeting;
  }

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return "Hello " + this.greeting;
  }
}
```

Return to the browser window and refresh the page. You should now see the text `Hello Quarkus (properties)`.

The `@ConfigProperty` annotation comes from the Eclipse MicroProfile Config specification [2.22] and is similar to the `@Value` annotation found in Spring. It is used to inject property values into an application. The `@ConfigProperty` annotation that you just added to your code requires the property `greeting.name` to be present in the configuration. The application will fail to start if the property is not found.

Like Spring, Quarkus reads configuration properties from several sources, in decreasing priority:

1. System properties.
2. Environment variables.
3. A file named `.env` placed in the current working directory (the directory from which the application was started).
4. `application.properties` (or a file named `application.yml` or `application.yaml`, if using the Quarkus YAML extension) placed in the `/config` directory.
5. `application.properties` (or a file named `application.yml` or `application.yaml`, if using the Quarkus YAML extension) placed in the `src/main/resources` directory.

Additional Configuration Options

To make a property optional, specify a default in the `defaultValue` attribute on the `@ConfigProperty` annotation, such as in the following example:

```
public GreetingResource(@ConfigProperty(name = "greeting.name", defaultValue =
  "Quarkus") String greeting) {
  this.greeting = greeting;
}
```

Additionally, the `String` type can be changed to `Optional<String>`. If the property isn't specified, `Optional.empty()` will be injected, such as in this example:

```
public GreetingResource(@ConfigProperty(name = "greeting.name") Optional<String>
  greeting) {
  this.greeting = greeting.orElse("Quarkus");
}
```

Quarkus also supports the use of expressions and environment variables within properties. For example, consider an `application.properties` file containing the following properties:

```
remote.host=quarkus.io
application.host=${HOST:${remote.host}}
```

This will expand the `HOST` environment variable as the value for the `application.host` property and use the value of the `remote.host` property as the default if the `HOST` environment variable is not set.

It is also possible to generate an example `application.properties` file with all known configuration properties. This example makes it easy to see what Quarkus configuration options are available. The example can be generated by running the following command:

```
$ ./mvnw quarkus:generate-config
```

This will create a `src/main/resources/application.properties.example` file containing all of the configuration options exposed by the extensions currently installed in the project. These options are commented out and are assigned their default values when applicable.

Type-Safe Configuration

Injecting a single property value is useful, but can become cumbersome if working with multiple related or hierarchical properties. Both Quarkus and Spring support the notion of type-safe configuration, where strongly typed beans are used to store custom configurations. Additionally, both Quarkus and Spring Boot support nested object configuration, useful when constructing configuration hierarchies. In both cases, these configuration classes are injected as beans into other beans when values need to be read.

Let's illustrate the process using an example. The application needs to expose the configuration properties with the requirements shown in Table 2.5.

Table 2.5: Custom configuration properties.

Property	Requirements
`greeting.message`	• Required `String` field
`greeting.suffix`	• Required `String` field • Default value `!` if not specified
`greeting.name`	• Optional `String` field • No default value if not specified
`greeting.content.prize-amount`	• Required `Integer` field • Value must be `> 0`
`greeting.content.recipients`	• Must have at least one `String` value

Listing 2.4 shows example settings in an `application.properties` file.

Listing 2.4: application.properties containing custom properties.

```
greeting.message=hello
# greeting.suffix is not defined here on purpose
# The default value is specified in code
# See Listings 2.5 and 2.6
greeting.name=quarkus
greeting.content.prize-amount=10
greeting.content.recipients=Jane,John
```

Spring Boot

In a Spring Boot configuration class, all properties are considered optional. JSR-303 annotations from the bean validation specification are needed to enforce required properties as well as the additional validations described in Table 2.5. The `spring-boot-starter-validation` starter also needs to be on the application's classpath in order to perform validation. Listing 2.5 shows how this would be built using Spring Boot.

Listing 2.5: Spring Boot GreetingConfiguration.java.

```
@ConfigurationProperties("greeting")
@Validated
public class GreetingConfiguration {
  @NotNull
  private String message;

  @NotNull
  private String suffix = "!";

  private String name;

  @Valid
  private final ContentConfig content = new ContentConfig();

  public String getMessage() { return this.message; }
  public void setMessage(String message) { this.message = message; }
  public String getSuffix() { return this.suffix; }
  public void setSuffix(String suffix) { this.suffix = suffix; }
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  public ContentConfig getContent() { return this.content; }

  public static class ContentConfig {
    @NotNull
    @Positive
    private Integer prizeAmount;

    @NotEmpty
    private List<String> recipients;
    public Integer getPrizeAmount() { return this.prizeAmount; }
    public void setPrizeAmount(Integer prizeAmount) { this.prizeAmount =
      prizeAmount; }
    public List<String> getRecipients() { return this.recipients; }
    public void setRecipients(List<String> recipients) { this.recipients =
      recipients; }
  }
}
```

Listing 2.5 shows that required fields must be annotated. Additionally, Spring's `@Validated` annotation must be placed on the class to enable any validation, and the `@Valid` annotation must exist on any nested class attributes. Additional bean validation annotations can be used for more advanced validation, such as the `@Positive` annotation placed on the `prizeAmount` field in the `ContentConfig` nested class.

Quarkus

There are a few differences in Quarkus. First, Quarkus uses interfaces rather than classes for defining type-safe configuration. Second, Quarkus uses the opposite approach when it comes to required versus optional properties. By default, all properties in a Quarkus configuration interface are required. Wrapping the return type in an `Optional` signifies the property as being optional. Lastly, Quarkus doesn't require the `@Valid` or `@Validated` annotations to perform validation. Quarkus validates a configuration if it encounters any of the specific validation annotations and the validation extension is present.

The configuration class shown in Listing 2.5 for Spring Boot would be represented in Quarkus by Listing 2.6.

Listing 2.6: Quarkus GreetingConfiguration.java.

```
@ConfigMapping(prefix = "greeting")
public interface GreetingConfiguration {
  @WithName("message")
  String getMessage();

  @WithName("suffix")
  @WithDefault("!")
  String getSuffix();

  @WithName("name")
  Optional<String> getName();

  @WithName("content")
  ContentConfig getContent();

  interface ContentConfig {
    @Positive
    @WithName("prizeAmount")
    Integer getPrizeAmount();

    @WithName("recipients")
    List<String> getRecipients();
  }
}
```

Note: For consistency with the Spring Boot example in Listing 2.5, the Quarkus example in Listing 2.6 uses method names derived from the JavaBeans specification. The `@WithName` annotations are unnecessary if the method names are the same as the property name. For example, if `getMessage()` was renamed to `message()`, the preceding `@WithName` annotation could be removed.

Profiles

The previous section showed how to inject values into properties during configuration. In practice, it is often necessary to inject different sets of configuration values at different times. For example, the configurations for testing are often different from the configurations for running the application in production. Both Quarkus and Spring support this capability by the concept of a *Profile*. However, one major difference is that Quarkus unifies all configurations for all profiles into a single configuration file.

A Quarkus application has three built-in profiles, described in Table 2.6. It is also possible to invent and use custom profiles.

Table 2.6: Default Quarkus profiles.

Profile name	Description
dev	Activated when running in development mode (i.e., `mvn quarkus:dev`).
test	Activated when running tests (i.e., `mvn verify`).
prod	The default profile when not running in development or test mode.

Table 2.7 details ways to set a custom profile.

Table 2.7: Setting a custom profile.

Mechanism	Example	Similar Spring action
`quarkus.profile` system property	`./mvnw package -Dquarkus.profile=mycustomprofile`	`spring.profiles.active=mycustomprofile` (declared within `application.properties`) or `./mvnw -Dspring.profiles.active=mycustomprofile`
`QUARKUS_PROFILE` environment variable	`export QUARKUS_PROFILE=mycustomprofile`	`export SPRING_PROFILES_ACTIVE=mycustomprofile`

If both the system property and the environment variable are set, the system property takes precedence. The syntax for defining profile-specific properties is:

```
%{profile-name}.key=value
```

To use profiles in a `.env` file, the syntax is:

```
_{PROFILE-NAME}_KEY=value
```

An example of defining a property in a `.env` file looks like:

```
_DEV_GREETING_NAME=Quarkus for Spring Developer
```

Example

Return to the `src/main/resources/application.properties` file. The only entry present is the one added previously. The value of the `greeting.name` property will be `Quarkus (properties)` for all profiles. Let's provide values for the three different profiles instead of a single value for all profiles. Add these three lines:

```
%dev.greeting.name=Quarkus for Spring Developer (dev)
%prod.greeting.name=Quarkus for Spring Developer (prod)
%test.greeting.name=RESTEasy
```

Your `application.properties` file should look like Listing 2.7.

Listing 2.7: application.properties with profile specific properties.

```
greeting.name=Quarkus (properties)
%dev.greeting.name=Quarkus for Spring Developer (dev)
%prod.greeting.name=Quarkus for Spring Developer (prod)
%test.greeting.name=RESTEasy
```

Return to the browser window and refresh the page. You should now see the text `Hello Quarkus for Spring Developer (dev)`. Return to the terminal where the `./mvnw quarkus:dev` command is running and terminate dev mode, typically using `CTRL-C` keystrokes.

Now run the application outside of dev mode to activate the `prod` profile. First, compile the application into an executable jar file. In the terminal, run the following command:

```
$ ./mvnw clean package
```

A self-executable jar file called `quarkus-run.jar` will be created inside the `target/quarkus-app` directory. Now run the application by running the following command:

```
$ java -jar target/quarkus-app/quarkus-run.jar
```

When the application starts up, it should print out `Profile prod activated`. Also, notice the startup time. In most cases, the startup time will be less than one second. This fast startup time again illustrates the compile-time efficiencies that allow for Quarkus applications to start up without the delays found in many Java applications.

Return to the browser window and refresh the page. You should now see the text `Hello Quarkus for Spring Developer (prod)`. Return to the terminal and terminate the running application in the same way as before.

Dependency Injection

A dependency injection framework allows one object to define which other objects it needs to function correctly. Dependency injection was one of the key components of Spring Framework. It became so popular that Java and the Java EE platform followed what Spring was doing. Today, Contexts and Dependency Injection (CDI) for Java 2.0 (JSR-365) [2.23] is the base for ArC [2.24], the dependency injection framework used by Quarkus.

One of the main differences between dependency injection in Quarkus and Spring is that Quarkus performs as much of the injection as possible at build time, whereas Spring performs injection at runtime. Quarkus's build-time augmentation leads to much better startup performance because ArC just needs to load all the metadata that was already computed at build time.

Quarkus and Spring both support injection via constructors, setter methods, and field attributes. Additionally, in both Spring and Quarkus, the default process of matching a bean to an injection point is done via type. This pattern is synonymous with Spring's "injection by type." In Quarkus, exactly one bean must be assignable to an injection point; otherwise, the build fails. In Spring, if there were multiple beans of the same type, the application would build but fail to start.

Catching that error at build time in Quarkus is a considerable benefit. Developers know whether the application is correct as part of their local development or CI/CD pipelines. There is no need to wait until the application is deployed into an environment and started. This benefit leads to shorter development and testing cycles and ultimately higher developer productivity and lower project costs.

Scopes

Quarkus and Spring also support the notion of bean scopes. Table 2.8 describes the scopes available for a Quarkus application, along with the Spring equivalents. Quarkus extensions can provide other custom scopes. For example, the `quarkus-narayana-jta` extension provides `@javax.transaction.TransactionScoped`.

Table 2.8: Available Quarkus bean scopes.

Quarkus scope annotation	Spring equivalent	Quarkus description
`@ApplicationScoped`	`@Scope("singleton")`	A single bean instance is used for the application and shared among all injection points. The instance is created lazily once a method is invoked on its proxy.
`@Singleton`	`@Scope("singleton")`	Just like `@ApplicationScoped` except that no proxy is used. The instance is created when an injection point resolves to an injected `@Singleton` bean.
`@RequestScoped`	`@Scope("request")`	The bean instance is associated with the current request, usually an HTTP request.
`@Dependent`	`@Scope("prototype")`	A pseudo-scope. The instances are not shared, and every injection point spawns a new instance of the dependent bean. The lifecycle of a dependent bean is bound to the bean injecting it. The bean will be created and destroyed along with the bean injecting it.
`@SessionScoped`	`@Scope("session")`	This scope is backed by a `javax.servlet.http.HttpSession` object. This annotation is available only if the `quarkus-undertow` extension is used.

@ApplicationScoped or @Singleton?

Quarkus provides two different scopes representing Spring's singleton scope: `@ApplicationScoped` and `@Singleton`. Which one to use depends on what you are trying to accomplish.

An `@Singleton` bean has no proxy. *Proxies* should be familiar to Spring users because they are the backbone behind Spring's dependency injection mechanism. When the Spring runtime injects a bean into another bean, the injection is usually a proxy to the actual bean. It's a placeholder where Spring can *wrap* the bean with some alternate implementation or delay the bean's instantiation.

Because an `@Singleton` bean lacks a proxy, the bean is created *eagerly* when an instance is first injected, leading to slightly better performance. Additionally, an `@Singleton` bean can not be mocked. By contrast, an instance of an `@ApplicationScoped` bean is created *lazily* once an instance is *acted upon* the first time. Because an `@ApplicationScoped` bean is proxied you cannot read or write fields of the bean from outside the class like you can with an `@Singleton` bean.

As a general rule of thumb, the recommendation is to use `@ApplicationScoped` by default unless there is a compelling reason to use `@Singleton`. `@ApplicationScoped` allows for more flexibility during live coding as well as when unit testing an application.

Example

Let's take the example being used so far in this chapter and change it to use dependency injection. All the logic for computing a greeting message resides in the `GreetingResource` class. Follow these steps to create a new class containing the required functionality and refactor `GreetingResource` to inject a new bean.

1. Return to the terminal and execute `./mvnw quarkus:dev` there.
2. In the `src/main/java/org/acme` directory, create a new Java class with the filename `GreetingService.java`.

3. Add the `@ApplicationScoped` annotation to the class.
4. Inject the `greeting` attribute into the `GreetingService` class exactly as in the `GreetingResource` class, by adding a `private final` attribute and a constructor injecting the `greeting.name ConfigProperty`.
5. Inside `GreetingService`, create a `getGreeting` method:

```
public String getGreeting() {
  return "Hello " + this.greeting;
}
```

6. Open the `GreetingResource` class.
7. Remove the `greeting` attribute from the class.
8. Add a new attribute to the class: `private final GreetingService greetingService;`
9. Change the constructor within the class to instead inject a `GreetingService` argument:

```
public GreetingResource(GreetingService greetingService) {
  this.greetingService = greetingService;
}
```

10. Change the return of the `hello()` method to be:

```
return this.greetingService.getGreeting();
```

Listings 2.8 and 2.9 show the completed classes.

Listing 2.8: Completed GreetingService.java.

```
@ApplicationScoped
public class GreetingService {
  private final String greeting;

  public GreetingService(@ConfigProperty(name = "greeting.name") String
greeting) {
    this.greeting = greeting;
  }

  public String getGreeting() {
    return "Hello " + this.greeting;
  }
}
```

Listing 2.9: Completed GreetingResource.java.

```
@Path("/hello-resteasy")
public class GreetingResource {
  private final GreetingService greetingService;

  public GreetingResource(GreetingService greetingService) {
    this.greetingService = greetingService;
  }

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return this.greetingService.getGreeting();
  }
}
```

Return to your browser and refresh the page. You should see the text `Hello Quarkus for Spring Developer (dev)`. Return to the terminal where the `./mvnw quarkus:dev` command is running and terminate Dev Mode.

The `GreetingResource` class uses constructor injection to inject the `GreetingService`. Spring promotes constructor injection over field injection for many reasons, one being testability. Both Quarkus and Spring additionally support field injection. All the examples in this book promote constructor injection. Listing 2.10 shows what the `GreetingResource` class would look like using field injection.

Listing 2.10: Completed GreetingResource.java using field injection.

```
@Path("/hello-resteasy")
public class GreetingResource {
  @Inject
  GreetingService greetingService;

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return this.greetingService.getGreeting();
  }
}
```

When you use field injection, Quarkus advises against using `private` fields due to the need for reflection in order to access the fields. Avoiding reflection speeds up build and application startup times and reduces an application's memory footprint.

Lifecycle Callbacks

Both Quarkus and Spring support creating lifecycle callbacks on beans, supporting the `@PostConstruct` and `@PreDestroy` annotations found in the `javax.annotation` package. Methods annotated with `@PostConstruct` are invoked before the bean instance is placed into service. It is safe to perform bean initialization within the method. Methods annotated with `@PreDestroy` are invoked before the destruction of the bean instance. It is safe to perform cleanup tasks within the method.

Native Image

Quarkus's design accounted for native compilation from the onset. A Quarkus native executable starts much faster and uses far less memory than a traditional JVM. Similar native image capabilities in Spring are still considered experimental or beta as of this writing.

To build a native image and run native image tests, you need to install either a distribution of GraalVM or a working container runtime. Additionally, Maven or Gradle has some scaffolding to facilitate the building of the native image and launching it for tests.

The following example assumes that GraalVM CE or Mandrel has been downloaded, and either the `GRAALVM_HOME` environment variable has been set to its location. Once all the requirements are met, a native image can be built using a simple command on the terminal:

```
$ ./mvnw package -Pnative
```

Native compilation can take several minutes to complete. Once complete, the native executable `chapter-2-simple-project-1.0.0-SNAPSHOT-runner` can be found in the project's `target` directory. On the terminal, execute the following:

```
$ ./target/chapter-2-simple-project-1.0.0-SNAPSHOT-runner
```

Note: Maven and Gradle scaffolding isn't discussed in this book. However, it is documented extensively in the Quarkus guide for building a native executable [2.25], along with how to use a container runtime environment instead of installing a GraalVM distribution locally. The code.quarkus.io generator provides everything needed for Maven or Gradle to function properly.

The command will produce the following output:

```
--  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (main) chapter-2-simple-project 1.0.0-SNAPSHOT native (powered
  by Quarkus 2.xx.x.Final) started in 0.015s. Listening on: http://0.0.0.0:8080
INFO  [io.quarkus] (main) Profile prod activated.
INFO  [io.quarkus] (main) Installed features: [cdi, resteasy,
  smallrye-context-propagation]
```

From this output, notice the application started in just 15 milliseconds. Return to your browser and refresh the page. You should see the text `Hello Quarkus for Spring Developer (prod)`.

Resident Set Size

Resident set size (RSS) [2.26] is the amount of memory occupied by a process. Calculating the RSS of a process is dependent on the operating system used. For macOS and Linux-based systems, the commands also vary by shell. In the bash shell on macOS or Red Hat Enterprise Linux, running the following command:

```
$ ps -o pid,rss,command -p $(pgrep -f chapter-2-simple-project-1.0.0-SNAPSHOT-runner)
```

results in the following output:

```
  PID    RSS COMMAND
13949  18604 ./target/chapter-2-simple-project-1.0.0-SNAPSHOT-runner
```

The `18604` value in the `RSS` column is the RSS size, in kilobytes, of the running process. Dividing that number by `1024` gives an RSS size of just over 18 megabytes for the running application's memory footprint.

Return to the terminal where the `./target/chapter-2-simple-project-1.0.0-SNAPSHOT-runner` command is running and terminate it the same way as before.

Note: The startup time and RSS output was gathered from a terminal running on macOS. Your startup time and RSS output may vary.

Testing

Testing is essential in any application to ascertain that the required functionality is implemented correctly. Both Quarkus and Spring include testing frameworks based on JUnit 5 [2.27] and Mockito [2.28].

Continuous Testing

As mentioned in Chapter 1, Quarkus enables test-driven development by understanding which tests are affected by classes and methods within the application. As changes are made to the application source code, Quarkus can automatically rerun affected tests in the background, giving developers instant feedback about the code they are writing.

To enable continuous testing, return to the terminal and execute `./mvnw quarkus:dev`. Once the Quarkus banner appears, notice in the terminal the following text at the very bottom: `Tests paused, press [r] to resume`, also shown in Figure 2.6.

Figure 2.6: Continuous testing output.

Press `r` in the terminal to enable continuous testing. The console should show output similar to:

```
--
All 1 tests are passing (0 skipped), 1 tests were run in 4552ms.
```

Unit Testing

Unit testing aims to isolate a single class so methods on the class can be tested. The test retains complete control over any external dependencies the class under test may have. One way to facilitate control over dependencies is to use constructor injection when creating classes, as seen in the previous examples. Bootstrapping the entire application isn't necessary in most cases when using this pattern with either Spring or Quarkus. Simply constructing a class and passing mocks or other required classes via the constructor and then asserting results is good enough.

Let's see this in action by adding a test class for the `GreetingService` class created earlier. Create a new Java class called `GreetingServiceTest` inside the `src/test/java/org/acme` folder. Inside that class, add a single test method and save the changes:

```
@Test
public void getGreetingOk() {
  Assertions.assertEquals("Hello Quarkus",
    new GreetingService("Quarkus").getGreeting());
}
```

Listing 2.11 shows the entire `GreetingServiceTest` class.

Listing 2.11: GreetingServiceTest.java.

```
public class GreetingServiceTest {
  @Test
  public void getGreetingOk() {
    Assertions.assertEquals("Hello Quarkus", new GreetingSer-
vice("Quarkus").getGreeting());
  }
}
```

Return to the terminal and notice that Quarkus automatically ran the test successfully. Also notice that only one test was run: the test that was just created. The existing test in the `GreetingResourceTest.java` class was not rerun.

```
--
All 2 tests are passing (0 skipped), 1 tests were run in 53ms.
```

There are times when you need the underlying framework to provide capabilities for you to write tests. One such example is when exposing RESTful endpoints. In that case, a test would issue an HTTP request with specific parameters, such as a request path, headers, and request parameters, and then perform assertions on some returned JSON structure. This kind of testing can't be done by instantiating a class and calling methods directly on it, either in Quarkus or in Spring.

When the `chapter-2-simple-project` project was generated earlier, a test class named `GreetingResourceTest` inside the `src/test/java/org/acme` folder was generated along with the project. The specifics of the REST-assured framework [2.29] used in this example will be discussed in detail in Chapter 3. However, the `testHelloEndpoint` method asserts that, when issuing an HTTP GET to the `/hello-resteasy` endpoint, the result is the value `Hello RESTEasy`. Listing 2.12 shows the test in its current form.

Listing 2.12: GreetingResourceTest.java.

```
@QuarkusTest
public class GreetingResourceTest {
  @Test
  public void testHelloEndpoint() {
    given()
      .when().get("/hello-resteasy")
      .then()
        .statusCode(200)
        .body(is("Hello RESTEasy"));
  }
}
```

Another thing to notice is the `@QuarkusTest` annotation at the top of the class. This annotation is similar to the `@SpringBootTest` annotation on tests in a Spring Boot application. The main difference in Quarkus is that the application is started only once and reused for all tests annotated with `@QuarkusTest`. In contrast, in Spring Boot, the application is started before each test class annotated with `@SpringBootTest` and then shut down after all tests in the class are executed. Subsequently, a Spring application restarts when the next test class annotated with `@SpringBootTest` is executed.

Additionally, classes annotated with `@QuarkusTest` can inject other beans, as we'll see in the next section. Quarkus ensures that the state of the beans in the running application matches tests or test class's requirements without having to restart the application.

Mocking

Earlier the `GreetingResource` class was refactored to inject a `GreetingService`, so the `GreetingResourceTest` class test should be refactored as well. The `GreetingResource` class needs to be tested in isolation, so it would be better for the test to mock the `GreetingService` class.

The `quarkus-junit5-mockito` library is required to accomplish this. The `quarkus-junit5-mockito` library contains many utilities, classes, and annotations that make Mockito easier to use within Quarkus. Many of these are similar to the `spring-boot-test` library provided by Spring Boot.

Open the project's `pom.xml` file and find the `<dependencies>` section. Towards the end of the the section, add the following:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-junit5-mockito</artifactId>
  <scope>test</scope>
</dependency>
```

Continuous testing will need to be re-enabled after making a change to the project's dependencies. On the terminal, press the `r` key to resume continuous testing.

Next, return to the `GreetingResourceTest` class. Add a `GreetingService` attribute to the class and place the `@InjectMock` annotation on it:

```
@InjectMock
GreetingService greetingService;
```

The `@InjectMock` annotation on the attribute injects a mock of the `GreetingService` class into the `GreetingResourceTest` class and makes it available in all test class methods. The `@InjectMock` annotation is similar to the `@MockBean` annotation provided by the Spring Boot Test framework. The Mockito framework can then be used directly to define and verify interactions on the mock, just as in a Spring Boot test.

Note: Similarly, the `@InjectSpy` annotation from Quarkus spies on a bean just like the `@SpyBean` annotation provided by the Spring Boot Test framework.

Replace the contents of the `testHelloEndpoint()` method with the contents from Listing 2.13 and save the changes.

Listing 2.13: GreetingResourceTest.java.

```
@QuarkusTest
public class GreetingResourceTest {
  @InjectMock
  GreetingService greetingService;

  @Test
  public void testHelloEndpoint() {
    Mockito.when(this.greetingService.getGreeting())
      .thenReturn("Hello Quarkus");

    given()
      .when().get("/hello-resteasy")
      .then()
        .statusCode(200)
        .body(is("Hello Quarkus"));

    Mockito.verify(this.greetingService).getGreeting();
    Mockito.verifyNoMoreInteractions(this.greetingService);
  }
}
```

The `Mockito.when(this.greetingService.getGreeting()).thenReturn("Hello Quarkus");` statement says that when the `getGreeting()` method is called on the `GreetingService` bean, the call returns the value `Hello Quarkus`. Afterwards, the test verifies that the `GreetingService.getGreeting()` method was called exactly once, and no other methods were called on the `GreetingService` class.

Return to the terminal and notice that Quarkus automatically ran the test successfully. Also notice that only one test was run: the test that was just modified. The test in the `GreetingServiceTest.java` class was not rerun.

```
--
All 2 tests are passing (0 skipped), 1 tests were run in 697ms.
```

Feel free to try out the continuous testing a bit more by making changes to the application source code. Quarkus will automatically rerun tests affected by changes made.

Return to the terminal where the `./mvnw quarkus:dev` command is running and terminate Dev Mode.

Native Image Testing

As covered in Chapter 1, one of the benefits of Quarkus is that it can compile to a native image. The native image should also be tested as part of the build process. However, keep in mind that the tests do not run within the native image; they run in the JVM. The native application image runs in a separate process, and a native image test interacts with the application as an external client. This separation prohibits injecting or mocking beans in a native image test. Instead, a native image test typically interacts with the application via HTTP, messaging, or other communication channels, just like any other external client to the application. Additionally, by default, the native image runs the `prod` profile, although this can be changed by setting the `quarkus.test.native-image-profile` property in `src/main/resources/application.properties`.

To create a native image test, create a new test class and extend an existing JVM test annotated with `@QuarkusTest`, placing the `@NativeImageTest` annotation on it. As mentioned previously, if the base test class injects or mocks beans, those tests need to be excluded from the native image test by annotating test methods with the `@DisabledOnNativeImage` annotation. The native image test class can then add additional test methods.

In the `chapter-2-simple-project` project, open the `src/test/java/org/acme/NativeGreetingResourceIt.java` file. This native image test would fail in its current state because the parent `GreetingResourceTest` class mocks the `GreetingService` bean.

To fix this, first add the `@DisabledOnNativeImage` annotation to the `testHelloEndpoint()` method in the `GreetingResourceTest` class. Then replace the `NativeGreetingResourceIT` class contents with the content in Listing 2.14.

Listing 2.14: NativeGreetingResourceIT.java.

```
@NativeImageTest
public class NativeGreetingResourceIT extends GreetingResourceTest {
  @Test
  public void testHelloEndpointNative() {
    given()
      .when().get("/hello-resteasy")
      .then()
        .statusCode(200)
        .body(is("Hello Quarkus for Spring Developer (prod)"));
  }
}
```

Assuming a proper GraalVM setup, return to the terminal and run:

```
$ ./mvnw verify -Pnative
```

This command will compile the application, run the regular tests, and build a native image. It may take a few minutes for the native image build to complete. Once completed, the native image starts and the native image tests run within the JVM. The output should show something similar to the following, indicating successful native image test execution:

```
[INFO] Running org.acme.NativeGreetingResourceIT
Executing [/examples/chapter-2/chapter-2-simple-project/target/
  chapter-2-simple-project-1.0.0-SNAPSHOT-runner, -Dquarkus.http.port=8081,
  -Dquarkus.http.ssl-port=8444, -Dtest.url=http://localhost:8081, -Dquarkus.log.
  file.path=target/target/quarkus.log]
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (main) chapter-2-simple-project 1.0.0-SNAPSHOT native (powered
  by Quarkus 2.xx.x.Final) started in 0.019s. Listening on: http://0.0.0.0:8081
INFO  [io.quarkus] (main) Profile prod activated.
INFO  [io.quarkus] (main) Installed features: [cdi, resteasy,
  smallrye-context-propagation]
[WARNING] Tests run: 2, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 2.676 s -
  in org.acme.NativeGreetingResourceIT
```

Native image tests aren't required to extend another test class. The code.quarkus.io generator uses this convention purely for convenience. There may be tests in the base class that would work fine against the native image that can be reused through inheritance. You could remove the `extends GreetingResourceTest` clause from the `NativeGreetingResourceIT` class and the native image tests would run successfully.

Additionally, the `@NativeImageTest` annotation will be deprecated at some point in the future [2.30]. The `@QuarkusIntegrationTest` annotation will replace it [2.31]. Tests annotated with `@QuarkusIntegrationTest` will run against whatever the output of the build was: an executable jar, native image, or container image. The class annotated with `@QuarkusIntegrationTest` will run in the JVM but the output of the build will be run in a separate process.

Note: `@NativeImageTest` is still used in these examples because the samples generated with code.quarkus.io at the time of this book's writing still used it.

Summary

Quarkus draws many parallels to Spring. As this chapter shows, anyone familiar with Spring should understand many of the concepts behind Quarkus and be able to become productive quickly. The main difference is and will continue to be the number of optimizations done at build time instead of runtime. This difference will continue as you move forward into further chapters in this book.

References

[2.1] "Quarkus Tools for Visual Studio Code": https://marketplace.visualstudio.com/items?itemName=redhat.vscode-quarkus

[2.2] "JetBrains IntelliJ IDEA Ultimate Edition Quarkus Tooling": https://www.jetbrains.com/help/idea/quarkus.html

[2.3] "Quarkus Tools for IntelliJ plugin": https://plugins.jetbrains.com/plugin/13234-quarkus-tools

[2.4] "Quarkus Tools JBoss Tools": https://tools.jboss.org/features/quarkus.html

[2.5] "Quarkus Tools on Eclipse Marketplace": https://marketplace.eclipse.org/content/quarkus-tools

[2.6] "Spring Boot Starters": https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter

[2.7] "Quarkiverse": https://github.com/quarkiverse/quarkiverse/wiki

[2.8] "Quarkus Extension for Spring Boot Properties": https://quarkus.io/guides/spring-boot-properties

[2.9] "Quarkus Extension for Spring Cache": https://quarkus.io/guides/spring-cache

[2.10] "Quarkus Extension for Spring Cloud Config Client": https://quarkus.io/guides/spring-cloud-config-client

[2.11] "Quarkus Extension for Spring Dependency Injection": https://quarkus.io/guides/spring-di

[2.12] "Quarkus Extension for Spring Data JPA": https://quarkus.io/guides/spring-data-jpa

[2.13] "Quarkus Extension for Spring Data REST": https://quarkus.io/guides/spring-data-rest

[2.14] "Quarkus Extension for Spring Scheduled": https://quarkus.io/guides/spring-scheduled

[2.15] "Quarkus Extension for Spring Security": https://quarkus.io/guides/spring-security

[2.16] "Quarkus Extension for Spring Web": https://quarkus.io/guides/spring-web

[2.17] "Red Hat's migration toolkit for applications": https://developers.redhat.com/products/mta

[2.18] "Quarkus + Spring Cheat Sheet": https://developers.redhat.com/cheat-sheets/quarkus-spring-cheat-sheet

[2.19] "Apache Maven Bill of Materials (BOM) POMs": https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#bill-of-materials-bom-poms

[2.20] "Apache Maven: Introduction to the POM": https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

[2.21] "Apache Maven: Importing Dependencies": https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Importing_Dependencies

[2.22] "Eclipse MicroProfile Configuration Feature": https://microprofile.io/project/eclipse/microprofile-config

[2.23] "Contexts and Dependency Injection for Java 2.0": https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html

[2.24] "ArC: Quarkus Dependency Injection": https://quarkus.io/blog/quarkus-dependency-injection

[2.25] "Quarkus: Building a Native Executable": https://quarkus.io/guides/building-native-image

[2.26] "Resident Set Size": https://en.wikipedia.org/wiki/Resident_set_size

[2.27] "JUnit 5": https://junit.org/junit5

[2.28] "Mockito": https://site.mockito.org

[2.29] "REST-assured": https://rest-assured.io

[2.30] "Quarkus Native Executable Testing Guide": https://quarkus.io/guides/getting-started-testing#native-executable-testing

[2.31] "Using `@QuarkusIntegrationTest`": https://quarkus.io/guides/getting-started-testing#quarkus-integration-test

Chapter 3

RESTful Applications

Eric Deandrea

Web applications make up a large portion of today's application landscape. Applications communicate with each other via the web, specifically over the HTTP protocol, and are designed using RESTful APIs. A RESTful web application responds to HTTP requests following standard HTTP constructs and sends responses primarily textual in representation, often JSON. This chapter examines web applications, and more specifically, RESTful web applications, highlighting similarities and differences between Quarkus and Spring when building such applications.

Both Quarkus and Spring have built-in support for building RESTful applications. Application development in each technology is similar from a conceptual standpoint. A developer in each framework constructs classes containing methods representing RESTful resources or endpoints. These methods additionally declare specific input and output parameters. Developers place annotations on these classes and methods describing how to match an incoming HTTP request to a single method within a single class. Some typical constraints used to match a request include:

- HTTP method
  - For example, GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS.
- URI path
  - For example, in the URL `https://www.somewhere.com/api/resources/1`, `/api/resources/1` is the path.
- Query parameters
  - For example, in the URL `https://www.somewhere.com/api/resources/1?param1=value&param2=value`, there are two query parameters, `param1` and `param2`.
- Media types
  - Also known as content negotiation, media type headers present in the HTTP request signify the request body's format (`Content-Type` request header) and what response formats the client can accept (`Accept` request header and `Content-Type` response header).

A RESTful Quarkus application uses JAX-RS (Jakarta RESTful Web Services [3.1], formerly known as the Java API for RESTful Web Services) for defining endpoints. This specification was introduced in Java SE 5 and later became an official part of Java EE 6. As stated, JAX-RS on its own is only a specification and requires choosing an implementation. For Quarkus, the implementation is RESTEasy [3.2]. For the most part, the internals of RESTEasy are hidden from Quarkus developers.

A Quarkus developer creates *resource classes* using APIs from the JAX-RS specification. These resource classes are analogous to Spring *controller classes*, or more specifically, Spring *restcontroller classes*, which have their own set of standards for representing RESTful concepts. Developers familiar with one can quickly become familiar with the other.

There are nevertheless a few differences when using RESTEasy in Quarkus versus within a Java EE orJakarta EE application:

- There is no need to define an `Application` class. Quarkus will automatically create and supply an `Application` subclass.
- Only a single JAX-RS application is supported within a JVM. Quarkus supports the deployment of only a single JAX-RS application, in contrast to JAX-RS running in a standard servlet container, where multiple JAX-RS applications could be deployed.
- All JAX-RS resources are treated as CDI beans by default and are scoped as CDI *Singletons*.

Underlying Runtime

One fundamental difference between Quarkus and Spring is the choice of the underlying runtime for building RESTful endpoints and its impact on development libraries and styles. For years, the Java Servlet Specification and its many synchronous and blocking APIs were the standard underlying runtime for web applications. Spring, specifically Spring MVC [3.3], adopted this runtime early on.

As mentioned in Chapter 1, reactive frameworks and servers have emerged to support the event-loop thread model. Spring had to build a mostly new web framework, called Spring WebFlux [3.4], to support the new architecture. Although WebFlux reused pieces across modules within the framework, the set of development libraries, the styles, and even the documentation for Spring MVC and Spring WebFlux are different and separated. A developer needs to decide upon project creation which framework to use and cannot mix and match within the same application, never mind within the same class. Additionally, a developer building a custom Spring Boot Starter interacting with the web layer may also need to support two different versions of the same functionality. This duplication of logic is not always trivial to implement.

Quarkus JAX-RS applications using the RESTEasy extension use the Eclipse Vert.x reactive event-loop engine as the default runtime. Quarkus manages a few I/O threads (sometimes called event-loop threads) to handle incoming HTTP requests. Additionally, Quarkus automatically moves the execution of any JAX-RS resource class methods onto a separate worker thread, enabling a developer to mix and match blocking and reactive programming styles and frameworks within the same resource class. Figure 3.1 shows a high-level overview.

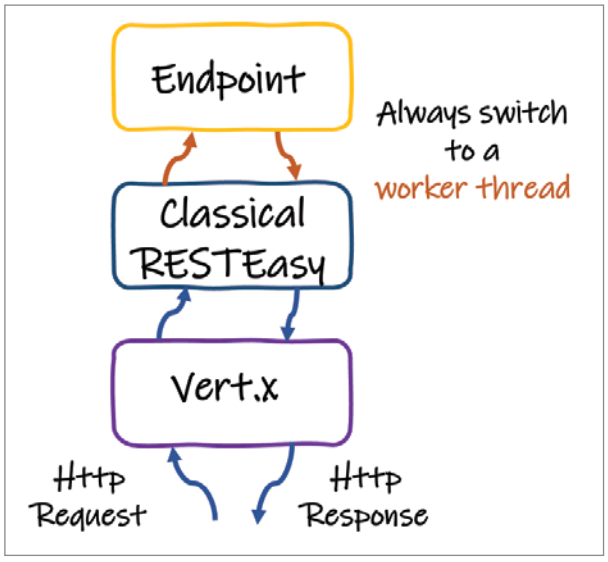

Figure 3.1: HTTP request flow with RESTEasy Classic.

The RESTEasy extension is also known as RESTEasy Classic or Classical RESTEasy.

Quarkus developers can also use the RESTEasy Reactive extension instead of the RESTEasy extension. This extension is a drop-in replacement for the RESTEasy Classic extension in most cases, achieving maximum throughput by handling each HTTP request on the I/O thread rather than moving to a worker thread. This approach is similar to how Spring WebFlux uses threads. Figure 3.2 shows a high-level overview.

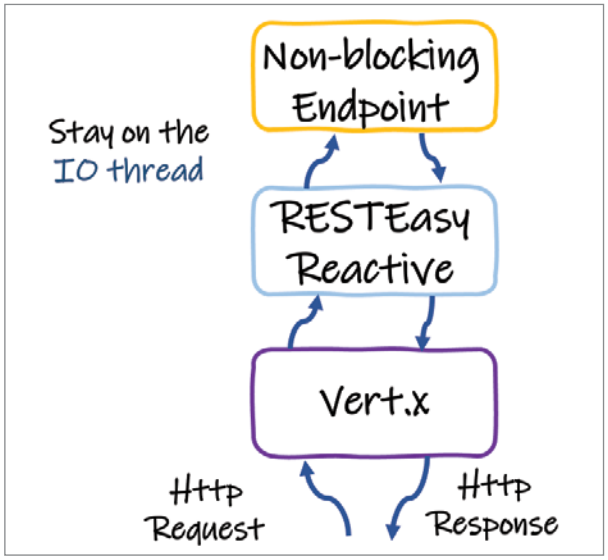

Figure 3.2: HTTP request flow with RESTEasy Reactive.

Using the RESTEasy Reactive extension, a developer needs to ensure that resource class methods do not perform blocking work, just as with Spring WebFlux. A developer can add the `@Blocking` annotation to a resource class method to signal that the method performs blocking work. In that case, Quarkus will execute the method on a separate worker thread, similar to how the RESTEasy Classic extension works.

Performance using RESTEasy Reactive is far better, even when using the `@Blocking` annotation, due to the tight integration between RESTEasy Reactive and Eclipse Vert.x. Like RESTEasy Classic, developers can mix and match blocking and reactive programming styles within the same resource class. Benchmarks have shown that a RESTEasy Reactive endpoint using `@Blocking` still achieves around 50% higher throughput than the same method using RESTEasy Classic [3.5].

Reactive Libraries

Developers building Spring WebFlux applications are most likely familiar with the `Mono` and `Flux` reactive types provided by Project Reactor [3.6]. SmallRye Mutiny [3.7] is the reactive library used by Quarkus, providing APIs conceptually similar to the ones found in Project Reactor. In fact, Mutiny and Project Reactor share a lot of common code.

Mutiny is an intuitive, event-driven, reactive programming library for Java, allowing developers to express and compose asynchronous actions. Mutiny offers two types: `Uni` for asynchronous actions providing either no result or a single value result and `Multi` to provide multi-item streams with backpressure [3.10].

Both types are lazy and follow a *subscription pattern*: Computation starts only when there is an actual need for it, known as a subscription. Both `Uni` and `Multi` expose event-driven APIs, where a developer expresses what to do upon receiving a given event (i.e., success, failure, etc.). Additionally, `Multi` implements a Reactive Streams [3.11] `Publisher` and implements the Reactive Streams backpressure mechanism. `Uni` does not implement `Publisher`, as the `Uni` subscription is enough to indicate interest in the result it may provide.

SmallRye Mutiny is part of the SmallRye project [3.8] and an implementation of the Eclipse MicroProfile Reactive Streams Operator [3.9].

Both `Uni` and `Multi` embrace the unification of reactive and imperative paradigms from Quarkus and provide bridges to imperative constructs. For example, a `Multi` can be transformed into an `Iterable` and vice-versa. A `Uni` or `Multi` can also be converted to and from RxJava [3.12] and Project Reactor types.

HTTP Method

The HTTP specification defines a set of request methods, also known as HTTP verbs, that indicate the desired action to be performed on a resource. Spring and Quarkus JAX-RS use different method annotations to define which HTTP verb should map to a given method within a class. Furthermore, the JAX-RS annotations denote only the HTTP method, whereas the Spring annotations have other metadata associated with them. Discussion of this other metadata follows in the next section.

Table 3.1 outlines some common Quarkus and Spring HTTP method annotations used in RESTful applications.

Table 3.1: HTTP method annotations in Quarkus JAX-RS and Spring.

Quarkus	Spring
`@GET`	`@GetMapping`
`@POST`	`@PostMapping`
`@PUT`	`@PutMapping`
`@DELETE`	`@DeleteMapping`
`@PATCH`	`@PatchMapping`
`@HEAD`	`@RequestMethod(method = RequestMethod.HEAD)`
`@OPTIONS`	`@RequestMethod(method = RequestMethod.OPTIONS)`

Note: The semantics around which HTTP method to use in various circumstances is outside this book's scope.

The HEAD and OPTIONS HTTP methods do not have their own Spring annotations. Spring's general-purpose `@RequestMethod` annotation handles these cases.

Routing HTTP Requests

As mentioned previously, both JAX-RS and Spring provide annotations to map an incoming HTTP request to a single method in a single class. Table 3.2 outlines some common conventions used in JAX-RS and Spring and defines where to place the annotations.

Table 3.2: Common REST annotations.

Quarkus	Spring	Annotation location	Description
`@Path`	`@RequestMapping`	Class	URI base path for all other URI paths handled by methods in the class.
`@Path`	**Preferred:** `path` attribute of the Spring HTTP method annotation from Table 3.1 **Supported:** `path` attribute of the Spring `@RequestMapping` annotation	Method	URI path for a specific endpoint, which will be appended to the class level.
`@Produces`	`produces` attribute of `@RequestMapping`	Class	Output type produced by all resource methods (i.e., the `Accept` header of the request and the `Content-Type` header of the response).

`@Produces`	`produces` attribute of the HTTP method annotation used from Table 3.1	Method	Output type produced by a resource method (i.e., the `Accept` header of the request and the `Content-Type` header of the response). Overrides anything specified at the class level.
`@Consumes`	`consumes` attribute of `@RequestMapping`	Class	Media type consumed by all resource methods (i.e., the `Content-Type` header of the request).
`@Consumes`	`consumes` attribute of HTTP method annotation used from Table 3.1	Method	Media type consumed by a resource method (i.e., the `Content-Type` header of the request). Overrides anything specified at the class level.
`@PathParam`	`@PathVariable`	Method parameter	Variable in the URI path (i.e., `/path/{variable}`).
`@QueryParam`	`@RequestParam`	Method parameter	Query parameter.
`@FormParam`	`@RequestParam`	Method parameter	Form parameter.
`@HeaderParam`	`@RequestHeader`	Method parameter	Header value.
`@CookieParam`	`@CookieValue`	Method parameter	Cookie value.
`@MatrixParam`	`@MatrixVariable`	Method parameter	Matrix parameter.
N/A. Just include the object in the method signature	`@RequestBody`	Method parameter	Request body.
`@Context` [3.13]	N/A; just include the object in the method signature [3.14] [3.15]	Method parameter	Used to inject contextual, request-specific information.

Both Quarkus and Spring [3.16] [3.17] methods have many potential return values that a developer can specify. Table 3.3 lists some common ones.

Table 3.3: Common REST method return values.

Quarkus	Spring	Description
`Response`	`HttpEntity<>`, `ResponseEntity<>`	Return a value specifying the full HTTP response, including headers, response status, and (potentially) a body.
`void`	`void`	No response body.
`Publisher<>`, `CompletionStage<>`, `CompletableFuture<>`	`Publisher<>`, `Callable<>`, `ListenableFuture<>`, `DeferredResult<>`, `CompletionStage<>`, `CompletableFuture<>`	Produce a result asynchronously.
`Publisher<>`, `Multi<>`	`SseEmitter`, `Flux<ServerSentEvent>`	Emit server-sent events.
`Uni<>`	`Mono<>`	Reactive type; provide an empty or single-value result.
`Multi<>`	`Flux<>`	Reactive type; provide multi-item streams with backpressure support.
Any other object	Any other object	Return a value converted and written to the response (i.e., marshaled to JSON).

Building RESTful Endpoints

The best way to highlight similarities and differences between Quarkus and Spring is to show a few examples in each technology. All examples showcase the following use cases:

- HTTP `GET` to `/fruits` retrieves all available fruits as JSON.
- HTTP `GET` to `/fruits/{name}` retrieves a single fruit denoted by the path variable `{name}`. Returns `404 (Not Found)` if no such fruit is found.
- HTTP `POST` to `/fruits`, passing valid JSON representing a fruit, returns all available fruits as JSON. Passing invalid JSON results in an HTTP return status of `400 (Bad Request)`.
- HTTP `DELETE` to `/fruits/{name}` deletes a single fruit denoted by the path variable `{name}`. Returns an HTTP status of `204 (No Content)`.
- HTTP `GET` to `/fruits/error` returns an HTTP status of `500 (Internal Server Error)`. This example showcases error handling.

The complete examples in the following listings can be found in this chapter's GitHub repository [3.18]:

- **Spring MVC:** Inside the `chapter-3/chapter-3-spring-rest-json` project
- **Spring WebFlux:** Inside the `chapter-3/chapter-3-spring-webflux-rest-json` project
- **Quarkus:** Inside the `chapter-3/chapter-3-quarkus-rest-json` project

Remember from earlier that a Spring developer needs to choose whether to use blocking or non-blocking/reactive architecture when creating the project. In Quarkus, a developer can mix both styles within the same class. Not shown in each snippet is the injection of a `FruitService` bean or the `Fruit` class contents. The contents of the `Fruit` class are nearly identical in all three examples.

In all the Spring examples, the class and method annotations are identical. The method return types, however, are different. The Spring MVC example returns standard objects and a `ResponseEntity`, but the Spring WebFlux version returns `Mono` and `Flux` reactive types. The `FruitService` class's implementation (not shown here but found in the examples repository) is also different in both versions. On the other hand, the Quarkus code combines reactive asynchronous and blocking synchronous methods within the same class.

Both Quarkus and Spring support Jackson [3.19] and Jakarta JSON Binding (JSON-B) [3.20] for marshaling and unmarshaling JSON to and from Java objects. Spring defaults to Jackson, whereas Quarkus forces a developer to make a conscious choice. When using Jackson, both frameworks provide a default Jackson `ObjectMapper` bean with default settings, overridable via configuration and other bean override mechanisms.

JSON serialization libraries use Java reflection to get an object's properties and serialize them. When building a Quarkus native executable, all classes using reflection need to be registered. Quarkus does most of this work automatically by analyzing the annotated resource class methods at build time. In the majority of the following examples, Quarkus knows that the `Fruit` class needs to be included in a native executable because the `Fruit` class is input to or output from the `list`, `add`, and `deleteFruit` methods.

Note: The Quarkus resource class used in the examples mixes blocking and reactive types within the same class simply to illustrate that it can be done. Besides the server-sent event endpoint, there was no compelling reason dictating which endpoints were implemented as blocking versus reactive. Additionally, none of the reactive endpoints utilize the `@Blocking` annotation because the examples are trivial and don't perform blocking work. The examples in Chapter 4 will use the `@Blocking` annotation.

RESTful Endpoint Class Structure

Listings 3.1 and 3.2 show the class structure for a class containing RESTful endpoints. The class structure for Spring MVC and WebFlux is identical.

Listing 3.1: Spring MVC and WebFlux class structure.

```
@RestController (1)
@RequestMapping(“/fruits”) (2)
public class FruitController {

}
```

Listing 3.2: Quarkus JAX-RS class structure.

```
@Path(“/fruits”) (2)
public class FruitResource {

}
```

1. Denotes the class as a handler for REST invocations. Not needed in Quarkus; the `@Path` annotation is enough.
2. Specifies the class as a handler for all subpaths beneath the `/fruits` URI.

RESTful Endpoint Examples

HTTP Path Mapping

Listings 3.3, 3.4, and 3.5 map an HTTP `GET` to the `/fruits` URI, returning all available fruits as JSON.

Listing 3.3: Spring MVC call to get all fruits.

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public Collection<Fruit> list() {
  return fruitService.getFruits();
}
```

Listing 3.4: Spring WebFlux call to get all fruits.

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public Flux<Fruit> list() {
  return fruitService.getFruits();
}
```

Listing 3.5: Quarkus call to get all fruits.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Collection<Fruit> list() {
  return fruitService.getFruits();
}
```

The Spring `@GetMapping` annotation combines the definitions of the `path` and `produces` attributes, whereas in JAX-RS separate annotations are used for each. These examples contain no `path` attribute, so the `/fruits` path used at the class level is inherited. The only other difference in the Spring versions is the return type: The Spring

MVC version returns `Collection<Fruit>` whereas the Spring WebFlux version returns the reactive type `Flux<Fruit>`.

Injecting a Path Parameter and Programmatically Determining the Response Status

Listings 3.6, 3.7, and 3.8 map an HTTP `GET` to `/fruits/{name}`, retrieving a single fruit denoted by the path variable `{name}`. They also return an HTTP status of `404 (Not Found)` if no such fruit is found. This time, there is a URI subpath (`/{name}`) that needs to be handled and injected via the method signature. Furthermore, the response status code varies based on whether or not there is something to return.

Listing 3.6: Spring MVC call to get a single fruit.

```
@GetMapping(path = "/{name}", produces = MediaType.APPLICATION_JSON_VALUE) (1)
public ResponseEntity<Fruit> getFruit(@PathVariable String name) { (2)
  return fruitService.getFruit(name)
    .map(ResponseEntity::ok) (3)
    .orElseGet(() -> ResponseEntity.notFound().build()); (4)
}
```

Listing 3.7: Spring WebFlux call to get a single fruit.

```
@GetMapping(path = "/{name}", produces = MediaType.
  APPLICATION_JSON_VALUE) (1)
public Mono<ResponseEntity<Fruit>> getFruit(@PathVariable String
  name) { (2)
  return fruitService.getFruit(name)
    .map(ResponseEntity::ok) (3)
    .defaultIfEmpty(ResponseEntity.notFound().build()); (4)
}
```

Listing 3.8: Quarkus call to get a single fruit.

```
@GET (1)
@Path("/{name}") (1)
@Produces(MediaType.APPLICATION_JSON) (1)
public Uni<Response> getFruit(@PathParam("name") String name) { (2)
  return fruitService.getFruit(name)
    .onItem().ifNotNull().transform(fruit -> Response.ok(fruit).
       build()) (3)
    .onItem().ifNull().continueWith(Response.status(Status.NOT_FOUND).
       build()); (4)
}
```

1. Map HTTP `GET` to `/fruits/{name}` to the `getFruit` method.
2. Inject the `{name}` path variable as an argument to the method and return a wrapper around the response.
3. If a fruit with the name `{name}` was found, return it as the response body with an HTTP status `200 (OK)` status.
4. If a fruit with the name `{name}` was not found, return an HTTP `404 (Not Found)`.

If this were the only method in the class, Quarkus would not be able to determine the need for the `Fruit` class based on the returned `Response` object. The `Fruit` object would need the `@RegisterForReflection` annotation in such a case. The Quarkus `@RegisterForReflection` annotation signals to the GraalVM compiler to register and retain the class during native executable compilation.

Input Validation on the Request Body

Listings 3.9, 3.10, and 3.11 map an HTTP `POST` to `/fruits`, passing valid JSON representing a fruit and returning all available fruits as JSON. Passing invalid JSON results in an HTTP status `400 (Bad Request)`.

Listing 3.9: Spring MVC call to add a fruit.

```
@PostMapping(produces = MediaType.APPLICATION_JSON_VALUE, consumes =
  MediaType.APPLICATION_JSON_VALUE)
public Collection<Fruit> add(@Valid @RequestBody Fruit fruit) {
  return fruitService.addFruit(fruit);
}
```

Listing 3.10: Spring WebFlux call to add a fruit.

```
@PostMapping(produces = MediaType.APPLICATION_JSON_VALUE, consumes =
  MediaType.APPLICATION_JSON_VALUE)
public Flux<Fruit> add(@Valid @RequestBody Fruit fruit) {
  return fruitService.addFruit(fruit);
}
```

Listing 3.11: Quarkus call to add a fruit.

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Collection<Fruit> add(@NotNull @Valid Fruit fruit) {
  return fruitService.addFruit(fruit);
}
```

The Spring `@PostMapping` annotation combines the definitions of the `path`, `produces`, and `consumes` attributes, whereas in JAX-RS separate annotations are used for each. As in Listings 3.3, 3.4, and 3.5, these examples don't contain a `path` attribute, so the `/fruits` path used at the class level is inherited. The only other difference in the Spring versions is the return type: The Spring MVC version returns `Collection<Fruit>` whereas the Spring WebFlux version returns the reactive type `Flux<Fruit>`.

The `@Valid` annotation in the method signatures in all three examples comes from the `javax.validation` package. Both Quarkus and Spring support the Jakarta Bean Validation Specification (JSR-303, JSR-349, and JSR-380) [3.21] annotations on method arguments and objects to enforce validation rules on incoming requests. The `Fruit` class (not shown in these examples) contains some of these annotations as well. Using this annotation in the method signature in front of the request body objects signals to the framework to validate the object.

Injecting Path Parameter and Statically Setting the Response Status

Listings 3.12, 3.13, and 3.14 map the HTTP `DELETE` verb to `/fruits/{name}`, deleting a single fruit denoted by the path variable `{name}`. The listings return an HTTP status of `204 (No Content)`.

Listing 3.12: Spring MVC call to delete a fruit.

```
@DeleteMapping("/{name}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteFruit(@PathVariable String name) {
  fruitService.deleteFruit(name);
}
```

Listing 3.13: Spring WebFlux call to delete a fruit.

```
@DeleteMapping("/{name}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public Mono<Void> deleteFruit(@PathVariable String name) {
  return fruitService.deleteFruit(name);
}
```

Listing 3.14: Quarkus call to delete a fruit.

```
@DELETE
@Path("/{name}")
public void deleteFruit(@PathParam("name") String name) {
  fruitService.deleteFruit(name);
}
```

The only difference between the Spring MVC and Spring WebFlux versions is the return type. Spring WebFlux needs to return a reactive type even though that type is `void`. This allows the subscription to occur within the `FruitService` class (not shown here). The Spring versions also use the `@ResponseStatus` annotation to denote the `204` response code. In Quarkus, a `void` return type automatically returns a `204 (No Content)` status.

Exception Handling

Exception handling is an important part of a RESTful interface. An application may want to return HTTP response headers or even an alternate body if some kind of error occurs. Both Quarkus and Spring have similar conventions for translating exceptions into HTTP responses. Exception handling can be implemented locally within a single controller class (in Spring) or resource class (in Quarkus) or can be implemented globally across all controller/resource classes within an application. In Quarkus, all of the binding for the error handling is performed at build time, whereas in Spring, most binding is performed at runtime.

The following example showcases how to return an alternate JSON response body when a custom `CustomRuntimeException` exception is thrown. The response will contain a custom header `X-CUSTOM-ERROR` and a body with two elements: an `errorCode` and `errorMessage`. The technique shown here is one of the more common approaches to exception handling, but there are others.

Note: Creating a custom exception class is not a requirement. It is done here merely to illustrate a simple example.

Listing 3.15 shows the exception class, `CustomRuntimeException`, used in all the examples.

Listing 3.15: CustomRuntimeException.java.

```
public class CustomRuntimeException extends RuntimeException {
  public CustomRuntimeException(String message) {
    super(message);
  }
}
```

Listing 3.16 shows the `CustomError` class, which will represent the response body to be marshalled into JSON.

Listing 3.16: CustomError.java.

```
public class CustomError {
  private int errorCode;
  private String errorMessage;

  public CustomError() {}

  public CustomError(int errorCode, String errorMessage) {
    this.errorCode = errorCode;
    this.errorMessage = errorMessage;
  }

  public int getErrorCode() { return this.errorCode; }
  public void setErrorCode(int errorCode) { this.errorCode = errorCode; }
  public String getErrorMessage() { return this.errorMessage; }
  public void setErrorMessage(String errorMessage) { this.errorMessage =
    errorMessage; }
}
```

In Quarkus, the `CustomError` class also needs the `@RegisterForReflection` annotation on the class for it to be included during native compilation.

Listings 3.17, 3.18, and 3.19 map an HTTP `GET` verb to `/fruits/error`. These examples call the `performWorkGeneratingError()` method on the `FruitService` class, which simulates a `CustomRuntimeException` being thrown.

Listing 3.17: Spring MVC simulation to generate an error.

```
@GetMapping("/error")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void doSomethingGeneratingError() {
  fruitService.performWorkGeneratingError();
}
```

Listing 3.18: Spring WebFlux simulation to generate an error.

```
@GetMapping("/error")
@ResponseStatus(HttpStatus.NO_CONTENT)
public Mono<Void> doSomethingGeneratingError() {
  return fruitService.performWorkGeneratingError();
}
```

Listing 3.19: Quarkus simulation to generate an error.

```
@GET
@Path("/error")
public void doSomethingGeneratingError() {
  fruitService.performWorkGeneratingError();
}
```

Local Exception Handling

Exception handling can be performed within a single controller/resource class. In this approach, exceptions that are thrown from within a single controller/resource class can be transformed to an HTTP response. Listings 3.20, 3.21, and 3.22 show how to transform

the thrown `CustomRuntimeException` into an HTTP response. These example snippets reside in the same controller/resource class as the methods in Listings 3.17, 3.18, and 3.19.

Listing 3.20: Spring MVC local exception mapping.

```
@ExceptionHandler(CustomRuntimeException.class) (1)
public ResponseEntity<CustomError>
  handleCustomRuntimeException(CustomRuntimeException cre) { (2)
  return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR) (3)
    .header("X-CUSTOM-ERROR", "500") (4)
    .body(new CustomError(500, cre.getMessage())); (5)
}
```

Listing 3.21: Spring WebFlux local exception mapping.

```
@ExceptionHandler(CustomRuntimeException.class) (1)
public Mono<ResponseEntity<CustomError>>
  handleCustomRuntimeException(CustomRuntimeException cre) { (2)
  return Mono.just(
    ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR) (3)
      .header("X-CUSTOM-ERROR", "500") (4)
      .body(new CustomError(500, cre.getMessage())) (5)
  );
}
```

Listing 3.22: Quarkus local exception mapping.

```
@ServerExceptionMapper(CustomRuntimeException.class) (1)
public Response handleCustomRuntimeException(CustomRuntimeException
  cre) { (2)
  return Response.serverError() (3)
    .header("X-CUSTOM-ERROR", "500") (4)
    .entity(new CustomError(500, cre.getMessage())) (5)
    .build();
}
```

1. The annotation denoting the method is an exception handler for `CustomRuntimeException` instances thrown from within the same class.
2. The method signature can include the actual exception as a parameter.
3. The handler sets an HTTP `500 (Internal Server Error)` status code on the response.
4. The handler sets the `X-CUSTOM-ERROR` header on the response.
5. The handler creates a `CustomError` POJO instance that will be marshalled to JSON as the response body.

The Spring MVC and WebFlux examples are nearly identical, except for the return type on the handler methods. Spring WebFlux requires reactive types. In Quarkus the example could use reactive types if desired. In all cases, the name of the handler method is irrelevant.

Global Exception Handling

Exception handling can also be performed globally across all controller/resource classes within an application. In this approach, exceptions thrown from any controller/resource class can be mapped to an HTTP response. Listings 3.23, 3.24, and 3.25 show how this is accomplished.

Listing 3.23: Spring MVC global exception mapping.

```
@RestControllerAdvice
public class GlobalErrorHandler {
  @ExceptionHandler(CustomRuntimeException.class)
  public ResponseEntity<CustomError>
    handleCustomRuntimeException(CustomRuntimeException cre) {
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
      .header("X-CUSTOM-ERROR", "500")
      .body(new CustomError(500, cre.getMessage()));
  }
}
```

Listing 3.24: Spring WebFlux global exception mapping.

```
@RestControllerAdvice
public class GlobalErrorHandler {
  @ExceptionHandler(CustomRuntimeException.class)
  public Mono<ResponseEntity<CustomError>>
    handleCustomRuntimeException(CustomRuntimeException cre) {
    return Mono.just(
      ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
        .header("X-CUSTOM-ERROR", "500")
        .body(new CustomError(500, cre.getMessage()))
    );
  }
}
```

Listing 3.25: Quarkus global exception mapping.

```
public class GlobalErrorHandler {
  @ServerExceptionMapper(CustomRuntimeException.class)
  public Response handleCustomRuntimeException(CustomRuntimeException cre) {
    return Response.serverError()
      .header("X-CUSTOM-ERROR", "500")
      .entity(new CustomError(500, cre.getMessage()))
      .build();
  }
}
```

In each instance, the exact same method from the local exception handling section is moved into its own class. In Spring, the class is annotated with the `@RestControllerAdvice` annotation. No annotation is needed in Quarkus.

Testing RESTful Endpoints

Developers should test all parts of an application. As mentioned in Chapter 2, both Quarkus and Spring include testing frameworks based on JUnit 5 and Mockito. How a Spring developer tests RESTful endpoints differs based on whether the application uses a reactive or blocking architecture. In contrast, a Quarkus developer tests RESTful endpoints the same way regardless of the underlying architecture. Quarkus uses the REST-assured framework [3.22] to test [3.23] and validate RESTful endpoints.

The following examples show excerpts from Spring MVC, Spring WebFlux, and Quarkus tests. Tests for all the examples shown in the previous section aren't shown here, but can be found in the examples GitHub repository [3.18].

The behavior of a test method should be the same regardless of the technology:

1. Initialize any necessary interactions on mock objects needed by the test fixture.
2. Perform the test on the test fixture.
3. Verify results on the test fixture.
4. Verify expected conditions on mock object interactions.

RESTful Endpoint Test Class Structure

Listings 3.26, 3.27, and 3.28 show the test class structure for a class containing RESTful endpoints.

Listing 3.26: Spring MVC test class structure.

```
@SpringBootTest (1)
@AutoConfigureMockMvc (2)
class FruitControllerTest {
  @Autowired
  MockMvc mockMvc; (3)

  @MockBean
  FruitService fruitService; (4)
}
```

Listing 3.27: Spring WebFlux test class structure.

```
@SpringBootTest (1)
@AutoConfigureWebTestClient (2)
class FruitControllerTest {
  @Autowired
  WebTestClient webTestClient; (3)

  @MockBean
  FruitService fruitService; (4)
}
```

Listing 3.28: Quarkus JAX-RS testcClass structure.

```
@QuarkusTest (1)
class FruitResourceTest {
  @InjectMock
  FruitService fruitService; (4)
}
```

1. Denotes the class as a test class.
2. Tells Spring to inject a test fixture (not needed in Quarkus).
3. Injects the test fixture.
4. Injects a Mockito mock of the required FruitService bean.

Unlike the controller class setup, there are different test classes for Spring MVC and Spring WebFlux. Spring MVC and WebFlux have different controller testing frame-

works, known as `MockMvc` [3.24] and `WebTestClient` [3.25], respectively. The REST-assured framework used by Quarkus doesn't require either. REST-assured can be used with Spring MVC and Spring WebFlux as well [3.26]. In those cases, the Spring MVC, Spring WebFlux, and Quarkus tests would all look nearly identical.

RESTful Endpoint Test Examples

This section shows several common testing patterns for testing HTTP path mapping and passing an invalid request body. Complete tests for all examples can be found in the example GitHub repository [3.18].

Testing HTTP Path Mapping

Listings 3.29, 3.30, and 3.31 show tests of mapping an HTTP `GET` verb to the `/fruits` URI, returning all available fruits as JSON.

Listing 3.29: Test for Spring MVC request to get all fruits.

```
@Test
public void list() throws Exception {
  Mockito.when(fruitService.getFruits())
    .thenReturn(List.of(new Fruit("Apple", "Winter fruit"))); (1)

  mockMvc.perform(get("/fruits")) (2)
    .andExpect(status().isOk()) (3a)
    .andExpect(content().contentTypeCompatibleWith(MediaType.
       APPLICATION_JSON)) (3b)

    .andExpect(jsonPath("$.size()").value(1)) (3c)
    .andExpect(jsonPath("[0].name").value("Apple")) (3d)
    .andExpect(jsonPath("[0].description").value("Winter fruit")); (3e)

  Mockito.verify(fruitService).getFruits(); (4a)
  Mockito.verifyNoMoreInteractions(fruitService); (4b)
}
```

Listing 3.30: Test for Spring WebFlux request to get all fruits.

```
@Test
public void list() {
  Mockito.when(fruitService.getFruits())
    .thenReturn(Flux.fromIterable(List.of(new Fruit("Apple",
       "Winter fruit")))); (1)

  webTestClient.get() (2)
    .uri("/fruits") (2)
    .exchange() (2)
    .expectStatus().isOk() (3a)
    .expectHeader().contentTypeCompatibleWith(MediaType.APPLICATION_
       JSON) (3b)
    .expectBody()
      .jsonPath("$.size()").isEqualTo(1) (3c)
      .jsonPath("[0].name").isEqualTo("Apple")) (3d)
      .jsonPath("[0].description").isEqualTo("Winter fruit")); (3e)

  Mockito.verify(fruitService).getFruits(); (4a)
  Mockito.verifyNoMoreInteractions(fruitService); (4b)
}
```

Listing 3.31: Test for Quarkus request to get all fruits.

```
@Test
public void list() {
  Mockito.when(fruitService.getFruits())
    .thenReturn(List.of(new Fruit("Apple", "Winter fruit"))); (1)

  given()
    .when().get("/fruits") (2)
    .then()
      .statusCode(200) (3a)
      .contentType(ContentType.JSON) (3b)
      .body(
        "$.size()", is(1), (3c)
        "[0].name", is("Apple"), (3d)
        "[0].description", is("Winter fruit") (3e)
      );

  Mockito.verify(fruitService).getFruits(); (4a)
  Mockito.verifyNoMoreInteractions(fruitService); (4b)
}
```

The test in each framework follows the previously recommended pattern:

1. Set up a mock so `fruitService.getFruits()` returns a single `Fruit` object with `name = Apple` and `description = Winter fruit`.
2. Perform the test.
3. Verify the results.
   a. Verifies that the HTTP status code returned is `200 (OK)`.
   b. Verifies that the `Content-Type` HTTP response header is compatible with `application/json`.
   c. Verifies that the returned JSON body is an array of size 1.
   d. Verifies that the first element in the returned JSON body has an attribute `name` with a value equal to `Apple`.
   e. Verifies that the first element in the returned JSON body has an attribute `description` with a value equal to `Winter fruit`.
4. Verify expected conditions on mock object interactions.
   a. Verifies that the `getFruits()` method on `fruitService` was called once.
   b. Verifies that no other method on `fruitService` was called.

Testing an Invalid Request Body

Listings 3.32, 3.33, and 3.34 show tests for part of the implementation of the `add` method: an HTTP `POST` verb to `/fruits` passing invalid JSON should result in an HTTP status of `400 (Bad Request)`.

Listing 3.32: Spring MVC test for rejecting an invalid request body when adding a fruit.

```
@Test
public void addInvalidFruit() throws Exception {
  mockMvc.perform(
    post("/fruits") (1)
      .contentType(MediaType.APPLICATION_JSON) (2)
      .content("{\"description\":\"Description\"}") (3)
    )
    .andExpect(status().isBadRequest()); (4)

  Mockito.verifyNoInteractions(fruitService); (5)
}
```

Listing 3.33: Spring WebFlux test for rejecting an invalid request body when adding a fruit.

```
@Test
public void addInvalidFruit() {
  webTestClient.post() (1)
    .uri("/fruits") (1)
    .contentType(MediaType.APPLICATION_JSON) (2)
    .bodyValue("{\"description\":\"Description\"}") (3)
    .exchange()
    .expectStatus().isBadRequest(); (4)

  Mockito.verifyNoInteractions(fruitService); (5)
}
```

Listing 3.34: Quarkus test for rejecting an invalid request body when adding a fruit.

```
@Test
public void addInvalidFruit() {
  given()
    .contentType(ContentType.JSON) (2)
    .body("{\"description\":\"Description\"}") (3)
    .when().post("/fruits") (1)
    .then()
    .statusCode(400); (4)

  Mockito.verifyNoInteractions(fruitService); (5)
}
```

Note: The test for adding a fruit successfully is not shown here. It can be found in the examples GitHub repository [3.18].

1. Performs an HTTP `POST` to `/fruits`.
2. Sends the `Content-Type` HTTP request header with the value `application/json`.
3. Sends a JSON request body with only a `description` attribute. This request should be rejected because a valid `Fruit` object (not shown in the examples) must have a non-empty `name` attribute.
4. Verifies that the HTTP status code returned is `400 (Bad Request)`.
5. Verifies that no methods on `fruitService` were called. The body of the `add` method is never executed if input validation fails.

Testing Exception Handling

Exception handling is tested the same way as other RESTful endpoints. Additionally, because testing is done for each individual endpoint, it doesn't matter whether the exception handling was implemented locally or globally. Listings 3.35, 3.36, and 3.37 show tests for the exception handling implemented earlier.

Listing 3.35: Spring MVC test for exception handling.

```
@Test
public void doSomethingGeneratingError() throws Exception {
  Mockito.doThrow(new CustomRuntimeException("Error"))
    .when(fruitService).performWorkGeneratingError(); (1)

  mockMvc.perform(get("/fruits/error")) (2)
    .andExpect(status().isInternalServerError()) (3)
    .andExpect(content().contentTypeCompatibleWith
       (MediaType.APPLICATION_JSON)) (4)
    .andExpect(header().string("X-CUSTOM-ERROR", "500")) (5)
    .andExpect(jsonPath("errorCode").value(500)) (6)
    .andExpect(jsonPath("errorMessage").value("Error")); (7)

  Mockito.verify(fruitService).performWorkGeneratingError(); (8)
  Mockito.verifyNoMoreInteractions(fruitService); (9)
}
```

Listing 3.36: Spring WebFlux test for exception handling.

```
@Test
public void doSomethingGeneratingError() {
  Mockito.when(fruitService.performWorkGeneratingError())
    .thenReturn(Mono.error(new CustomRuntimeException("Error"))); (1)

  webTestClient.get() (2)
    .uri("/fruits/error") (2)
    .exchange()
    .expectStatus().isEqualTo(HttpStatus.INTERNAL_SERVER_ERROR) (3)
    .expectHeader().contentTypeCompatibleWith(MediaType.APPLICATION_JSON) (4)
    .expectHeader().valueEquals("X-CUSTOM-ERROR", "500") (5)
    .expectBody()
      .jsonPath("errorCode").isEqualTo(500) (6)
      .jsonPath("errorMessage").isEqualTo("Error"); (7)

  Mockito.verify(fruitService).performWorkGeneratingError(); (8)
  Mockito.verifyNoMoreInteractions(fruitService); (9)
}
```

Listing 3.37: Quarkus test for exception handling.

```
@Test
public void doSomethingGeneratingError() throws Exception {
  Mockito.doThrow(new CustomRuntimeException("Error"))
    .when(fruitService).performWorkGeneratingError(); (1)

  given()
    .when().get("/fruits/error") (2)
    .then()
      .statusCode(500) (3)
      .contentType(ContentType.JSON) (4)
      .header("X-CUSTOM-ERROR", "500") (5)
      .body(
        "errorCode", is(500), (6)
        "errorMessage", is("Error") (7)
      );

  Mockito.verify(fruitService).performWorkGeneratingError(); (8)
  Mockito.verifyNoMoreInteractions(fruitService); (9)
}
```

1. Sets up a mock so `fruitService.performWorkGeneratingError()` throws a `CustomRuntimeException`.
2. Performs an HTTP `GET` to `/fruits/error`.
3. Verifies that the HTTP status code returned is `500 (Internal Server Error)`.
4. Verifies that the `Content-Type` HTTP response header is compatible with `application/json`.
5. Verifies that the `X-CUSTOM-ERROR` HTTP response header equals the value `500`.
6. Verifies that the `errorCode` attribute in the returned JSON body has a value equal to `500`.
7. Verifies that the `errorMessage` attribute in the returned JSON body has a value equal to `Error`.
8. Verifies that the `performWorkGeneratingError()` method on `fruitService` was called once.
9. Verifies that no other method on `fruitService` was called.

Server-Sent Event Endpoints

A common use case today involves a browser-based application that needs updates in real time as events occur. Updates are *pushed* to the browser rather than being *polled*. A client accomplishes these real-time updates by initiating an HTTP connection to a RESTful endpoint. That endpoint then begins pushing events, known as *server-sent events (SSE)*, back to the client. The W3C standardized these events' format into the SSE EventSource API [3.27] as part of HTML5.

Mapping an incoming HTTP request to a single method within a single class follows all of the same rules described previously in this chapter. The main difference is the `Content-Type` response header. For SSE endpoints, this header needs to specify `text/event-stream`.

Implementing a class method returning SSE events varies based on the framework used. Spring WebFlux and Quarkus are conceptually similar, whereas implementing an SSE endpoint in Spring MVC is much different.

Listing 3.38 shows a Spring MVC method emitting fruits every second until there are no more fruits to emit. Listing 3.39 shows an equivalent Spring WebFlux implementation, and Listing 3.40 shows an equivalent Quarkus implementation. Emitting SSE events in Spring MVC is somewhat cumbersome and complicated because the developer needs to explicitly implement and execute asynchronous tasks, whereas it is much simpler in Spring WebFlux and Quarkus.

Listing 3.38: Emitting SSE events in Spring MVC.

```
@GetMapping(path = "/stream", produces = MediaType.TEXT_EVENT_STREAM_VALUE) (1)
public SseEmitter streamFruits() { (2)
  SseEmitter emitter = new SseEmitter();
  AtomicInteger counter = new AtomicInteger(); (3)
  ScheduledExecutorService executor =
    Executors.newSingleThreadScheduledExecutor(); (4)

  executor.scheduleWithFixedDelay(() -> { (5)
    int tick = counter.getAndIncrement();
    List<Fruit> fruits = fruitService.getFruits()
     .stream()
     .sorted(Comparator.comparing(Fruit::getName))
     .collect(Collectors.toList());
```

Note: As mentioned previously, the complete examples in the following listings can be found in the examples GitHub repository [3.18].

```
    if (tick < fruits.size()) {
      try {
        emitter.send(fruits.get(tick), MediaType.APPLICATION_JSON); (6)
      }
      catch (IOException ex) {
        emitter.completeWithError(ex); (7)
      }
    }
    else {
      emitter.complete(); (8)
      executor.shutdown(); (9)
    }
  }, 0, 1, TimeUnit.SECONDS); (10)

  return emitter; (11)
}
```

1. Declares that an HTTP `GET` to `/fruits/stream` containing the `Accept` request header with the value `text/event-stream` should be mapped to the `streamFruits` method.
2. Declares a function that returns the `SseEmitter` type. The `SseEmitter` type in Spring MVC indicates an asynchronous stream of EventSource API events.
3. Used to keep track of the current item to emit.
4. Creates a new single-threaded thread pool `Executor` used to schedule a task to run every second.
5. A lambda function representing a `Runnable` to be executed according to a schedule.
6. Emits a `Fruit` as JSON.
7. Emits an error signal upon an error condition.
8. Completes emission of events.
9. Shuts down the scheduler upon completion.
10. Schedules the `Executor` to run a task 1 second after completion of the previous task.
11. Returns the `SseEmitter` to Spring MVC.

Listing 3.39: Emitting SSE events in Spring WebFlux.

```
@GetMapping(path = "/stream", produces = MediaType.TEXT_EVENT_STREAM_VALUE) (1)
public Flux<ServerSentEvent<Fruit>> streamFruits() { (2)
  return fruitService.streamFruits()
    .map(ServerSentEvent::builder)
    .map(Builder::build);
}

// In FruitService
public Flux<Fruit> streamFruits() {
  return Flux.interval(Duration.ofSeconds(1)) (3)
    .map(tick -> (4)
      fruits.values()
        .stream()
        .sorted(Comparator.comparing(Fruit::getName))
        .collect(Collectors.toList())
        .get(tick.intValue())
    )
    .take(fruits.size()); (5)
}
```

Listing 3.40: Emitting SSE events in Quarkus.

```
@GET (1)
@Path("/stream") (1)
@Produces(MediaType.SERVER_SENT_EVENTS) (1)
public Multi<Fruit> streamFruits() { (2)
  return fruitService.streamFruits();
}

// In FruitService
public Multi<Fruit> streamFruits() {
  return Multi.createFrom().ticks().every(Duration.ofSeconds(1)) (3)
    .map(tick -> (4)
      fruits.values()
        .stream()
        .sorted(Comparator.comparing(Fruit::getName))
        .collect(Collectors.toList())
        .get(tick.intValue())
    )
    .select().first(fruits.size()); (5)
}
```

1. Declares that an HTTP `GET` to `/fruits/stream` containing the `Accept` request header with the value `text/event-stream` should be mapped to the `streamFruits` method.
2. The `streamFruits` method simply returns reactive types indicating a multi-item stream.
3. Creates a stream with a one-second interval.
4. Emits a fruit to the stream every second.
5. Completes the stream once all fruits have been emitted.

As shown in these examples, the Spring WebFlux and Quarkus versions are very similar. Reactive types in Spring WebFlux and Quarkus allow a developer to declaratively build a data pipeline describing what to do and when to do it without needing to implement all the details. A developer familiar with these concepts in Spring WebFlux should be able to easily understand similar concepts and capabilities in Mutiny.

Testing Server-Sent Event Endpoints

Testing an SSE endpoint is nearly identical to testing any other RESTful endpoint. The main difference comes from SSE endpoints being asynchronous. Tests need to handle the response body returned in chunks rather than all at once. Writing tests for SSE endpoints is also more cumbersome in Spring MVC than Spring WebFlux and Quarkus for the same reasons as for writing the SSE endpoints themselves.

Listing 3.41 shows the test for the Spring MVC method in Listing 3.38. Listing 3.42 shows the test for the Spring WebFlux method in Listing 3.39. Listing 3.43 shows the test for the Quarkus method in Listing 3.40.

Listing 3.41: Testing a Spring MVC SSE method.

```
@Test
public void streamFruits() throws Exception {
  Mockito.when(fruitService.getFruits())
    .thenReturn(List.of(new Fruit("Apple", "Winter fruit"), new
       Fruit("Pear", "Delicious fruit"))); (1)

  MvcResult mvcResult = mockMvc.perform(get("/fruits/stream")) (2)
    .andExpect(request().asyncStarted()) (3)
    .andDo(log()) (4)
    .andReturn(); (5)

  mockMvc.perform(asyncDispatch(mvcResult)) (6)
    .andDo(log()) (7)
    .andExpect(status().isOk()) (8)
    .andExpect(content().contentTypeCompatibleWith(MediaType.TEXT_EVENT_
       STREAM)) (9)
    .andExpect(content().string("data:{\"name\":\"Apple\",
       \"description\":\"Winter fruit\"}\n\ndata:{\"name\":\"Pear\",
       \"description\":\"Delicious fruit\"}\n\n")); (10)
}
```

1. Sets up an interaction so `fruitService.getFruits()` returns two `Fruit` objects.
2. Performs an HTTP `GET` to `/fruits/stream`.
3. Verifies that asynchronous processing has started.
4. Prints details to the log upon test failure.
5. Returns a handler to the asynchronous result.
6. Starts verification of the asynchronous result.
7. Prints details to the log upon test failure.
8. Verifies that the HTTP status code returned is `200 (OK)`.
9. Verifies that the `Content-Type` HTTP response header is compatible with `text/event-stream`.
10. Verifies the complete response body after all fruits are emitted.

Listing 3.42: Testing a Spring WebFlux SSE method.

```
@Test
public void streamFruits() {
  Mockito.when(fruitService.streamFruits())
    .thenReturn(streamFruitsMock()); (1)

  webTestClient.get() (2)
    .uri("/fruits/stream") (2)
    .exchange()
    .expectStatus().isOk() (3)
    .expectHeader().contentTypeCompatibleWith
       (MediaType.TEXT_EVENT_STREAM) (4)
    .expectBody(String.class).isEqualTo("data:{\"name\":\"Apple\",
       \"description\":\"Winter fruit\"}\n\ndata:{\"name\":\"Pear\",
       \"description\":\"Delicious fruit\"}\n\n"); (5)
}
```

Listing 3.43: Testing a Quarkus SSE method.

```
@Test
public void streamFruits() {
  Mockito.when(fruitService.streamFruits())
    .thenReturn(streamFruitsMock()); (1)

  given()
    .when().get("/fruits/stream") (2)
    .then()
      .statusCode(200) (3)
      .contentType("text/event-stream") (4)
      .body(is("data:{\"name\":\"Apple\",\"description\":
        \"Winter fruit\"}\n\ndata:{\"name\":\"Pear\",
        \"description\":\"Delicious fruit\"}\n\n")); (5)
}
```

As mentioned previously, the Spring WebFlux and Quarkus tests are very similar and also resemble the tests seen earlier for other RESTful endpoints.

1. Sets up an interaction so `fruitService.getFruits()` returns two `Fruit` objects.
2. Performs an HTTP `GET` to `/fruits/stream`.
3. Verifies that the HTTP status code returned is `200 (OK)`.
4. Verifies that the `Content-Type` HTTP header returned is compatible with `text/event-stream`.
5. Verifies the complete response body after all fruits are emitted.

OpenAPI Documentation

The OpenAPI specification [3.28] is a way to describe RESTful APIs. Some common elements found in an OpenAPI document include:

- Available endpoints
- Available operations for each endpoint
- Available input parameters for each operation
- The output of each operation
- Contact information
- License information
- Terms of use

A Quarkus application can expose its API description through an OpenAPI specification. Additionally, a Quarkus application can automatically generate a Swagger UI [3.29] endpoint, allowing developers to visualize and interact with the APIs an application provides. This functionality is delivered via the SmallRye OpenAPI [3.30] extension, compliant with the Eclipse MicroProfile OpenAPI [3.31] specification.

Spring does not have Spring Boot Starters for providing similar functionality. Other open source projects, such as SpringDoc OpenAPI [3.32] and SpringFox [3.33], provide similar capabilities. These libraries, including the Quarkus SmallRye OpenAPI extension, are used in very similar ways. One fundamental difference is the build-time and ahead-of-time compilation optimizations done by the Quarkus SmallRye OpenAPI extension.

A developer can annotate resource class methods and parameters with annotations such as `@Tag`, `@Operation`, `@APIResponse`, and `@Parameter`. Listing 3.44 shows an example.

The complete examples in the following listings can be found in the examples GitHub repository [3.18].

Listing 3.44: Quarkus Resource class with OpenAPI annotations.

```
@Path("/fruits")
@Tag(name = "Fruit Resource", description = "Endpoints for fruits")
public class FruitResource {
  @GET
  @Path("/{name}")
  @Produces(MediaType.APPLICATION_JSON)
  @Operation(summary = "Get a fruit", description = "Get a fruit")
  @APIResponse(responseCode = "200", description = "Requested fruit",
    content = @Content(schema = @Schema(implementation = Fruit.class)))
  @APIResponse(responseCode = "404", description = "Fruit not found")
  public Uni<Response> getFruit(@Parameter(required = true, description =
    "Fruit name") @PathParam("name") String name) {
    return fruitService.getFruit(name)
      .onItem().ifNotNull().transform(fruit -> Response.ok(fruit).build())
      .onItem().ifNull().continueWith(Response.status(Status.NOT_FOUND).
        build());
  }
}
```

Additional metadata, such as contact and license information, can be specified in the application's `application.properties` file, as shown in Listing 3.45.

Listing 3.45: Quarkus OpenAPI metadata in application.properties.

```
quarkus.smallrye-openapi.info-title=Fruits API
quarkus.smallrye-openapi.info-version=1.0.0
quarkus.smallrye-openapi.info-description=Example Fruit Service - Quarkus
quarkus.smallrye-openapi.info-contact-email=edeandrea@redhat.com
quarkus.smallrye-openapi.info-contact-name=Eric Deandrea
quarkus.smallrye-openapi.info-contact-url=https://developers.redhat.com/
  blog/author/edeandrea
quarkus.smallrye-openapi.info-license-name=Apache 2.0
quarkus.smallrye-openapi.info-license-url=https://www.apache.org/
  licenses/LICENSE-2.0.html
quarkus.smallrye-openapi.operation-id-strategy=METHOD
```

A developer can display the Swagger UI endpoint in the browser by running the application, using `mvnw quarkus:dev`, and then navigating to `http://localhost:8080/q/swagger-ui`. The Swagger UI page is available only when Quarkus starts in Dev or test mode, but can be configured to always display by setting `quarkus.swagger-ui.always-include=true` inside `application.properties`. Additionally, the OpenAPI specification is available at `http://localhost:8080/q/openapi`.

Summary

This chapter showcased the many similarities and differences between Spring and Quarkus in building RESTful endpoints. The Eclipse MicroProfile specification also continues to provide many capabilities provided by Spring. Quarkus continues to consume these capabilities as extensions. Because of elements that the frameworks

have in common, a developer familiar with Spring MVC or Spring WebFlux should be able to become familiar with Quarkus and JAX-RS rather quickly. Again, one of the main differences is and will continue to be the number of optimizations done at build time versus runtime.

References

[3.1] "Jakarta RESTful Web Services": https://projects.eclipse.org/projects/ee4j.jaxrs

[3.2] "RESTEasy": https://resteasy.github.io

[3.3] "Spring Web on Servlet Stack": https://docs.spring.io/spring-framework/docs/current/reference/html/web.html

[3.4] "Spring Web on Reactive Stack": https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html

[3.5] "Massive performance without headaches": https://quarkus.io/blog/resteasy-reactive-faq

[3.6] "Project Reactor": https://projectreactor.io

[3.7] "SmallRye Mutiny": https://smallrye.io/smallrye-mutiny

[3.8] "SmallRye Project": https://smallrye.io

[3.9] "Eclipse MicroProfile Reactive Streams Operator": https://github.com/eclipse/microprofile-reactive-streams-operators

[3.10] "Back-Pressure": https://www.reactivemanifesto.org/glossary#Back-Pressure

[3.11] "Reactive Streams": https://www.reactive-streams.org

[3.12] "RxJava: Reactive Extensions for the JVM": https://github.com/ReactiveX/RxJava

[3.13] "JAX-RS @Context Injection Types": https://docs.jboss.org/resteasy/docs/4.5.9.Final/userguide/html/_Context.html

[3.14] "Spring MVC Method Arguments": https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-arguments

[3.15] "Spring WebFlux Method Arguments": https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-ann-arguments

[3.16] "Spring MVC Method Return Values": https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-return-types

[3.17] "Spring WebFlux Method Return Values": https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-ann-return-types

[3.18] "Examples Repository": https://github.com/quarkus-for-spring-developers/examples

[3.19] "Jackson Project Home @GitHub": https://github.com/FasterXML/jackson

[3.20] "Jakarta JSON Binding (JSON-B)": http://json-b.net/

[3.21] "Jakarta Bean Validation Specification": https://beanvalidation.org/specification

[3.22] "REST-assured": https://rest-assured.io

[3.23] "Quarkus - Testing Your Application": https://quarkus.io/guides/getting-started-testing

[3.24] "Spring Testing - MockMvc": https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#spring-mvc-test-framework

[3.25] "Spring Testing - WebTestClient": https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#webtestclient

[3.26] "REST-assured Spring Support": https://github.com/rest-assured/rest-assured/wiki/Usage#spring-support

[3.27] "W3C Server-Sent Events": https://www.w3.org/TR/eventsource

[3.28] "OpenAPI Specification": https://www.openapis.org

[3.29] "Swagger UI": https://swagger.io/tools/swagger-ui

[3.30] "SmallRye OpenAPI": https://github.com/smallrye/smallrye-open-api

[3.31] "Eclipse MicroProfile OpenAPI": https://microprofile.io/project/eclipse/microprofile-open-api

[3.32] "SpringDoc OpenAPI": https://springdoc.org

[3.33] "SpringFox": https://springfox.github.io/springfox

Chapter 4

Persistence

Eric Deandrea

Microservice applications today don't live in a bubble by themselves. Multiple microservice applications typically form a larger system, where one application exposes its data to other applications and services. One popular pattern in such cases is the "database per service" pattern [4.1]. In this pattern, each application owns and is responsible for its data. Therefore, it's common for an application to persist data in its own database. This database could be a relational database, such as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, or DB2, or a NoSQL database, such as MongoDB or Apache Cassandra. This chapter examines an application's persistence layer, highlighting similarities and differences between Quarkus and Spring.

Note: This book focuses on persistence with relational databases. Other databases are treated in a conceptually similar manner by Quarkus and Spring, but use different underlying persistence mechanisms.

Evolution of the Java Persistence API

Database connectivity in Java, known as the Java Database Connectivity API (JDBC), has been part of the Java platform since its inception. JDBC is a low-level construct containing many blocking APIs for data access. Using JDBC, a developer must handcraft SQL statements for every database interaction rather than use higher-level object-oriented constructs to map tables, rows, and columns to Java classes. An extra layer called object-relational mapping (ORM) [4.2] allowed Java applications to model the data from SQL databases in Java objects.

In 2001, the open source Hibernate project [4.3] was started. One of the goals of Hibernate was to build on JDBC and simplify the complexities involved in ORM, by utilizing lightweight Java classes. Hibernate focuses on modeling database tables and relationships as classes, leading to a higher level of abstraction that caters more to object-oriented design. A developer no longer needs to write SQL directly; Hibernate maps classes and their interactions to SQL while still providing flexibility for using SQL directly if needed.

In 2006 the Hibernate project's founders, as well as others from the community, participated in developing the Java Persistence API (JPA). JPA introduced standard APIs, part of the new `javax.persistence` package, for modeling relational entities and the relationships between them, and for performing create, read, update, and delete (CRUD) operations upon them. Developers only have to choose which certified JPA implementation to use. One such implementation choice, and perhaps the most common today, is Hibernate.

JPA Abstractions

Today JPA is one of the most widely used specifications by Java application developers. It provides many capabilities that simplify persistence in Java applications. JPA enables developers to build complex relational mappings, but it does not make common and simple mappings trivial. Both Quarkus and Spring recognized this unneeded and unwanted complexity, introducing abstractions of their own on top of the JPA APIs.

Note: Both Quarkus and Spring support using JPA APIs directly without any abstractions on top. In such cases, code in Quarkus and Spring would look nearly identical. This chapter focuses on the abstractions provided by Quarkus and Spring.

Spring Data JPA

Spring Data [4.4] evolved as an umbrella project within the Spring ecosystem. The project provides a consistent programming model familiar to Spring developers across data access technologies, including relational and non-relational databases. The Spring Data JPA subproject [4.5] is not a JPA implementation; it is an abstraction on top of JPA, providing developers an easy way to implement JPA-based repositories.

A Spring Data JPA repository is a Java interface providing standard CRUD operations upon a domain *entity*. An application defining a Spring Data JPA repository can also add additional operations that may be application-specific or domain-specific.

A *repository* is a stereotype for a class in a Spring application encapsulating data access. Such data access could involve files on a filesystem, an HTTP connection to another system, or database interactions.

Quarkus Hibernate with Panache

Quarkus Hibernate with Panache [4.6] (Panache) focuses on making entity mapping trivial. There are two patterns for using Panache with Quarkus: the repository pattern and the active record pattern.

Developers familiar with Spring Data JPA will find the Quarkus Panache repository pattern familiar. This pattern enforces a clear separation between entities and operations performed upon the entities. The active record pattern takes a somewhat different and opinionated approach, combining the entity and its logic in a single class that extends a Panache base class, providing lots of useful CRUD methods. The difference between these patterns, and their comparisons with Spring Data JPA, are better illustrated through the following examples.

JPA in Action

The complete examples throughout the remainder of this section can be found in the examples GitHub repository [4.7]:

- **Spring Data JPA:** Inside the `chapter-4/chapter-4-spring-data-jpa` project.
- **Quarkus Panache Repository Pattern:** Inside the `chapter-4/chapter-4-quarkus-panache-repository` project.
- **Quarkus Panache Active Record Pattern**: Inside the `chapter-4/chapter-4-quarkus-panache-activerecord` project.

Note: Each project in the Chapter 4 examples GitHub repository [4.7] contains a `Dockerfile.postgres` file. This file was used to construct the database image used in the examples.

Hibernate can create a database schema from all of the entities in an application. While this practice is useful in rapid application prototyping and testing, a much more common requirement is to connect to an existing database with an existing schema. A better approach, in this case, is to have Hibernate validate that the entities defined in an application can *bind* to the schema—that is, that the application can use the schema to read and write data to and from expected table structures. This validation ensures that the entities are defined correctly. This is the pattern used in all of the examples in this chapter.

Database Setup

All the examples in this chapter assume an existing PostgreSQL 13 database with an existing schema named `fruits` available at `localhost:5432`, having both username and password of `fruits`. The `fruits` schema has a single table named `fruits`. Table 4.1 details the `fruits` table.

Database versioning technologies, such as Flyway [4.8] and Liquibase [4.9], are not discussed in this book, although they can be used in both Quarkus and Spring.

Table 4.1: Fruits table.

Column name	Column type	Column constraints
id	`bigserial` [4.10]	• Primary key (implies not null and unique)
name	`varchar(255)`	• Not null • Unique
description	`varchar(255)`	• None

A PostgreSQL 13 container image [4.11] is available with the schema created and data inserted. You can start the container image with the following command:

```
$ docker run --ulimit memlock=-1:-1 -it --rm=true --memory-swappiness=0 --name
  chapter4 -p 5432:5432 quay.io/edeandrea/postgres-13-fruits:latest
```

This container image needs to be started before running the `spring-boot:run` Maven goal of the Spring Data JPA example, starting any examples using `java -jar`, or running the Quarkus examples in native image.

Application Data Source Configuration

All the examples in this chapter use YAML for configuration rather than properties files, to illustrate how YAML configuration works in Quarkus and Spring.

Listing 4.1 shows the data source configuration, located in `src/main/resources/application.yml`, for Spring Data JPA. Listing 4.2 shows the data source configuration, located in `src/main/resources/application.yml`, for Quarkus Panache. The testing configuration will be covered later in this chapter.

The configuration for the Panache repository pattern is identical to the Panache active record pattern configuration.

Listing 4.1: Spring Data JPA data source configuration.

```
spring:
  datasource:
    username: fruits (1)
    password: fruits (2)
    url: jdbc:postgresql://localhost:5432/fruits (3)
  jpa:
    hibernate:
      ddl-auto: validate (4)
```

Listing 4.2: Quarkus Panache data source configuration.

```
quarkus:
  datasource:
    username: fruits (1)
    password: fruits (2)
  hibernate-orm:
    database:
      generation: validate (4)
```

1. Sets the data source's username.
2. Sets the data source's password.
3. Sets the data source's URL, assuming a PostgreSQL database running on `localhost:5432`. Needed only in Spring for local development purposes. In Quarkus, local development via Quarkus Dev Mode, discussed in the next section, automatically starts a database container. In both Spring and Quarkus, when deploying an application into an environment, datasource configuration should be injected at runtime instead of hard-coded into a configuration file. Chapter 6 discusses deployment in more detail.
4. Instructs Hibernate to validate that the entities defined in the application can bind to the schema.

Quarkus Dev Mode

Recall from Chapter 2 Quarkus's Dev Mode and Live Coding capabilities. Quarkus's Dev Mode provides an additional capability, called Dev Services [4.12], that will automatically bootstrap a database container image and set all of the required configuration properties for running the `dev` profile via `./mvnw quarkus:dev`. Quarkus will additionally terminate the database container upon shutdown.

This capability grants developers a tight feedback loop and removes the additional overhead of having to start a database manually and provide a configuration for it. Currently, Quarkus Dev Services supports database containers, Apache Kafka (discussed in Chapter 5), OpenID Connect (Keycloak), AMQP, Redis, HashiCorp Vault, and Apicurio Registry. Future Quarkus versions will extend this capability to include other middleware components as well.

Note: A container runtime is required for Quarkus Dev Services to function properly.

Listing 4.3 shows the Quarkus Dev Services configuration inside `src/main/resources/application.yml`.

Listing 4.3: Quarkus Dev Services configuration.

```
quarkus:
  datasource:
    devservices:
      image-name: quay.io/edeandrea/postgres-13-fruits:latest
```

Providing an image name is not required. If not provided, Quarkus provides a default based on the database driver on the classpath. In such a case, the database will be schemaless.

Entity Design

The entity design is identical in all examples because all the examples use the same schema. The code, however, is slightly different in the Quarkus Panache active record pattern due to the different conventions for using the pattern. These differences will be examined further in this section.

Listing 4.4 shows the `Fruit` entity class used in both the Spring Data JPA and Quarkus Panache repository pattern examples. All constructors and the `hashCode`, `equals`, and `toString` methods have been omitted from the listing for brevity.

Listing 4.4: Spring Data JPA and Quarkus Panache repository Fruit.java entity.

```
@Entity (1)
@Table(name = "fruits") (2)
public class Fruit {
  @Id (3)
  @GeneratedValue(strategy = GenerationType.IDENTITY) (4)
  private Long id; (5)

  @Column(nullable = false, unique = true) (6)
  @NotBlank(message = "Name is mandatory") (7)
  private String name; (8)
  private String description; (9)

  public Long getId() { return this.id; }
  public void setId(Long id) { this.id = id; }
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  public String getDescription() { return this.description; }
  public void setDescription(String description) { this.description =
    description; }

  // Constructors, hashCode, equals, toString methods omitted for brevity
}
```

In all examples, the annotation imports come from the `javax.persistence` and `javax.validation` packages. The `javax.persistence` package is part of the JPA specification and the `javax.validation` package is part of the Java Bean Validation specification.

1. Annotation marking the class as a JPA entity. Hibernate will scan the application for entity classes and manage all entities' persistence lifecycle.
2. Annotation defining the name of the table that the entity represents.
3. Annotation marking the attribute as the primary key of the table.
4. Annotation marking the primary key attribute, generated according to how the underlying persistence provider generates identity columns. In this case, the PostgreSQL database uses a `bigserial` type for auto-generating new primary keys.
5. The `id` field, mapping to the `id` column in the table.
6. Annotation marking the column as non-null and unique.
7. Annotation marking the Java class field as not empty. Hibernate will perform validation on the object before attempting to persist it to the database.
8. The `name` field, mapping to the `name` column in the table.
9. The `description` field, mapping to the `description` column in the table.

One main difference between Spring and Quarkus is that in Spring, Hibernate's entity scanning is performed at application startup. In Quarkus, the entity scanning is performed at build time. This difference helps make Quarkus applications start faster.

Repository Implementation

Both Spring Data JPA and Quarkus Panache repository patterns promote a *repository* class containing an entity's operations. Both Spring Data JPA and Quarkus Panache supply many base classes and interfaces containing CRUD methods for interacting with entities. In both cases, the repositories become beans available to be injected into other beans.

One of the main differences between Quarkus Panache and Spring Data JPA is that Spring Data JPA requires just an interface definition extending one of the Spring Data JPA interfaces. Upon application startup, Spring Data JPA creates an implementation proxy for the interface and any additional method declarations a developer adds to the interface. The definition of additional methods within the interface, including the method name, return type, and method arguments, are extremely important: They influence the implementation of the proxy that Spring Data JPA dynamically creates [4.13]. This runtime proxying contributes to longer startup times and increased memory usage.

Quarkus Panache, on the other hand, implements the same pattern using a slightly different convention. In Quarkus Panache, the repository is defined as a class and injected as a CDI bean, just like any other CDI bean. The repository class implements one of Panache's base interfaces, which provides many of the same CRUD operations found in the Spring Data JPA interfaces. Additionally, when a developer adds methods to the class, the method names are unimportant in Quarkus Panache. Defining the repository as a concrete class allows Quarkus to perform build-time optimizations, including entity scanning, while also enabling native compilation from the onset.

Listing 4.5 shows a Spring Data JPA repository, and Listing 4.6 shows a Quarkus Panache repository.

Listing 4.5: Spring Data JPA repository.

```
public interface FruitRepository extends JpaRepository<Fruit, Long> { (2)
  Optional<Fruit> findByName(String name); (3)
}
```

Listing 4.6: Quarkus Panache repository.

```
@ApplicationScoped (1)
public class FruitRepository implements PanacheRepositoryBase<Fruit,
  Long> { (2)
  public Optional<Fruit> findByName(String name) { (3)
    return find("name", name).firstResultOptional();
  }
}
```

1. Annotation marking the class as CDI bean. Not present in Spring Data JPA.
2. Class/interface definition, extending a base class provided by the underlying framework. In both Quarkus and Spring Data JPA, the base class needs two pieces of information: the specific type of entity class the repository supports and the type of the entity's primary key.
3. Additional operations added to the repository.

Quarkus Panache Active Record Pattern

Quarkus Panache's active record pattern takes a slightly different approach: it combines the entity and its operations in a single class while eliminating tedious getter and setter methods within the entity class. In this pattern, an entity class needs to extend a Panache base class and define all class attributes as `public`. Quarkus generates the getter and setter methods at build time and rewrites every direct attribute access to the getter and setter methods. Additionally, any additional operations should be defined as `public static` methods within the class.

Listing 4.7 shows the `Fruit` entity class in the Quarkus Panache active record pattern. Constructors and the `hashCode`, `equals`, and `toString` methods have been omitted from the listing for brevity.

Listing 4.7: Quarkus Panache active record Fruit.java entity.

```
@Entity
@Table(name = "fruits")
public class Fruit extends PanacheEntityBase {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  public Long id;

  @Column(nullable = false, unique = true)
  @NotBlank(message = "Name is mandatory")
  public String name;
  public String description;

  public static Optional<Fruit> findByName(String name) {
    return find("name", name).firstResultOptional();
  }

  // Constructors, hashCode, equals, toString methods omitted for brevity
}
```

The attribute names and types and the JPA annotations on them in the active record pattern are identical to those in the Spring Data JPA and Quarkus Panache repository pattern entities. Some of the differences include:

- There is only a `Fruit` entity class and no repository class.
- The attributes in the `Fruit` class are `public` and the class has no getter or setter methods.
- The `Fruit` class is not a bean managed by the dependency injection infrastructure. Instead, consumers directly use the `public` attributes and `public static` methods.
- The `Fruit` class extends Panache's `PanacheEntityBase`. Panache also provides a `PanacheEntity` base class. The `PanacheEntity` class extends `PanacheEntityBase` and represents an entity with a generated primary key `ID` field of type `Long`, using an auto-generation strategy. A custom primary key generation strategy is needed in this example, so extending `PanacheEntityBase` instead of `PanacheEntity` required defining the primary key and corresponding strategy.

Using the Persistence Layer

Other classes in an application utilize the persistence layer. The examples [4.7] show a set of RESTful endpoints with the following use cases:

- An HTTP `GET` to `/fruits` retrieves all available fruits in the database as JSON.
- An HTTP `GET` to `/fruits/{name}` retrieves a single fruit denoted by the path variable `{name}`. The response returns `404 (Not Found)` if no such fruit is found.
- An HTTP `POST` to `/fruits`, passing valid JSON representing a fruit, returns the full fruit, with database generated primary key, as JSON. Passing invalid JSON results in an HTTP status of `400 (Bad Request)`.

Listing 4.8 shows a Spring MVC controller class with the HTTP `POST` operation. Listing 4.9 shows a Quarkus JAX-RS resource class implementing the same operation using the Panache repository pattern, and Listing 4.10 shows the same operation implemented in a Quarkus JAX-RS resource class using the Panache active record pattern.

The implementations for all the operations in all examples can be found in the examples GitHub repository [4.7].

Note: The details for creating Spring MVC REST controllers and Quarkus JAX-RS resource classes are not discussed in this chapter. They are discussed in detail in Chapter 3.

Listing 4.8: Spring MVC controller using JPA repository.

```
@RestController
@RequestMapping("/fruits")
public class FruitController {
  private final FruitRepository fruitRepository; (1)

  public FruitController(FruitRepository fruitRepository) {
    this.fruitRepository = fruitRepository;
}

  @PostMapping(produces = MediaType.APPLICATION_JSON_VALUE, consumes =
    MediaType.APPLICATION_JSON_VALUE)
  @Transactional (3)
  public Fruit addFruit(@Valid @RequestBody Fruit fruit) {
    return fruitRepository.save(fruit); (4) (5)
  }
}
```

Listing 4.9: Quarkus resource class using Panache repository.

```
@Path("/fruits")
@Blocking (2)
public class FruitResource {
  private final FruitRepository fruitRepository; (1)

  public FruitResource(FruitRepository fruitRepository) {
    this.fruitRepository = fruitRepository;
  }

  @POST
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  @Transactional (3)
  public Fruit addFruit(@Valid Fruit fruit) {
    fruitRepository.persist(fruit); (4)
    return fruit; (5)
  }
}
```

Listing 4.10: Quarkus resource class using Panache active record.

```
@Path("/fruits")
@Blocking (2)
public class FruitResource {
  @POST
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  @Transactional (3)
  public Fruit addFruit(@Valid Fruit fruit) {
    Fruit.persist(fruit); (4)
    return fruit; (5)
  }
}
```

The implementations of each of the methods, outside of the REST framework specific annotations, are nearly identical.

1. Injection of the `FruitRepository`. Not needed in the Quarkus Panache active record pattern because there is no repository class.
2. Annotation denoting all the methods in the class perform blocking operations.
   - Not needed in Spring MVC because the underlying runtime is based on the blocking Servlet API and therefore all controller methods are blocking.
   - In Quarkus, the underlying runtime (RESTEasy Reactive) sits on top of the Eclipse Vert.x reactive engine. Resource class methods that perform blocking work need to be moved off of the I/O thread onto a worker thread. The `@Blocking` annotation accomplishes this.
3. An annotation indicating that the method should execute in the context of a transaction. The `@Transactional` annotation in Spring and Quarkus have the same name but are found in different packages. In Spring, the annotation comes from Spring's `org.springframework.transaction.annotation` package, whereas in Quarkus it comes from the `javax.transaction` package, part of the Java Transaction API.

4. Persists a fruit to the database.
5. Returns the persisted fruit. In Quarkus, the `persist` method returns `void`. In a Spring Data JPA repository, the `save` method returns the persisted entity.

JPA Testing

Testing is essential in all applications. Any additional or custom operations created on a Spring Data JPA repository, Quarkus Panache repository, or Quarkus Panache active record entity need to be tested. Additionally, tests for classes consuming the persistence layer need to mock the Spring Data JPA repository, Quarkus Panache repository, or Quarkus Panache active record entity.

Test Database Setup

There are several approaches to testing a data access layer. One popular approach uses an in-memory database, such as H2 [4.14], during test execution. One downside to this approach is that there could be differences in how the database implements certain features, such as auto-generation of primary keys.

A better approach would be to use an actual database of the same type the application expects in production. A container runtime, such as Docker, can bootstrap a database during test execution. Both Quarkus and Spring support using Testcontainers [4.15] in conjunction with JUnit 5 for executing tests that need different kinds of external services, such as databases, message brokers, or other cloud-provider native services.

The Quarkus Dev Services mentioned previously uses Testcontainers internally to bootstrap the database when running Quarkus Dev Mode or executing tests. No additional setup or configuration for Testcontainers within a Quarkus application is needed. If a specific image is not provided, Quarkus will choose one based on the database driver found on the classpath.

Database Schema

Additionally, a database schema matching the application's expected schema should be defined and maintained.

The `src/test/resources/db` folder within each example project contains a `schema.sql` file, shown in Listing 4.12. This file defines the `fruits` table and inserts some records into the table. This file is the same file that is used in the `quay.io/edeandrea/postgres-13-fruits:latest` database container image used in the examples. This configuration allows the tests to validate that the application is compatible with the same schema used in production.

There are libraries, such as Flyway [4.8] and Liquibase [4.9], that can assist with database schema creation and migration. They are not the focus of this book, mainly because, at the time of this writing, they don't yet support reactive data sources. This book intends to showcase the same functionality in both Quarkus and Spring using both blocking and reactive architectures, while keeping the implementations as close as possible.

Listing 4.12:- Database schema file.

```
CREATE TABLE fruits
(
  id bigserial NOT NULL,
  name varchar(255) NOT NULL UNIQUE,
  description varchar(255),
  PRIMARY KEY (id)
);

INSERT INTO fruits(name, description) VALUES
  ('Apple', 'Hearty fruit'),
  ('Pear', 'Juicy fruit');
```

Application Test Configuration

The application test configuration can be created once the schema is defined. Test configuration is needed only for Spring. Quarkus will automatically reuse the Dev Services setup described previously when running tests. Listing 4.13 shows the Spring Data JPA configuration in `src/test/resources/application.yml`.

Listing 4.13: Spring Data JPA test configuration.

```
spring:
  datasource:
    url: jdbc:tc:postgresql:13:///fruits?TC_INITSCRIPT=db/schema.sql (1)
```

1. The database URL includes the `tc` prefix, which is a Testcontainers-specific JDBC URL [4.16]. In this case, a `postgres:13` container image will be started, and a schema named `fruits` will be created in the running container instance. Additionally, the `TC_INITSCRIPT` URL parameter [4.17] instructs Testcontainers to run the `db/schema.sql` script initialization script after the container is started but before the test is given a connection to it.

Testcontainers makes sure the database container with the schema and initialization runs when any JUnit tests requiring a JDBC `DataSource` are encountered. In Spring Boot, a container image is started for each `@SpringBootTest` class encountered, whereas in Quarkus the container image is started only once and reused for all `@QuarkusTest` tests.

One benefit of this approach is that because the application's Hibernate configuration is set to `validate`, the tests also assert that the application's JPA entities can properly bind to the same expected database type and schema as a production environment.

Testing Additional Data Access Operations

Listings 4.5, 4.6, and 4.7 introduce a single additional operation named `findByName`. This operation optionally returns a `Fruit` given a `Fruit`'s `name`. Listing 4.14 shows a test class for `findByName` method using Spring Data JPA. Listing 4.15 shows a test class for the same method using the Quarkus Panache repository pattern. Listing 4.16 shows a test class for the method using the Quarkus Panache active record pattern.

The flow of each test is identical in each example:

1. Persist a new `Fruit` into the database.
2. Retrieve the persisted `Fruit` from the database using the `findByName` method.
3. Perform assertions on the retrieved `Fruit` to validate it is the correct `Fruit`.

Listing 4.14: Spring Data JPA test class for findByName operation.

```
@SpringBootTest (1)
@Transactional (2)
class FruitRepositoryTests {
  @Autowired
  FruitRepository fruitRepository; (3)

  @Test
  public void findByName() { (4)
    fruitRepository.save(new Fruit(null, "Grapefruit", "Summer fruit")); (5)

    Optional<Fruit> fruit = fruitRepository.findByName("Grapefruit"); (6)
    assertThat(fruit)
      .isNotNull() (7)
      .isPresent() (8)
      .get()
      .extracting(Fruit::getName, Fruit::getDescription) (9)
      .containsExactly("Grapefruit", "Summer fruit"); (9)

    assertThat(fruit.get().getId())
      .isNotNull() (10)
      .isGreaterThan(2L); (11)
  }
}
```

Note: All the examples use the AssertJ assertions library [4.18].

Listing 4.15: Quarkus Panache repository pattern test class for findByName operation.

```
@QuarkusTest (1)
@TestTransaction (2)
class FruitRepositoryTests {
  @Inject
  FruitRepository fruitRepository; (3)

  @Test
  public void findByName() { (4)
    fruitRepository.persist(new Fruit(null, “Grapefruit”, “Summer fruit”)); (5)

    Optional<Fruit> fruit = fruitRepository.findByName(“Grapefruit”); (6)
    assertThat(fruit)
      .isNotNull() (7)
      .isPresent() (8)
      .get()
      .extracting(Fruit::getName, Fruit::getDescription) (9)
      .containsExactly(“Grapefruit”, “Summer fruit”); (9)

      assertThat(fruit.get().getId())
        .isNotNull() (10)
        .isGreaterThan(2L); (11)
  }
}
```

Listing 4.16: Quarkus Panache active record pattern test class for findByName operation.

```
@QuarkusTest (1)
@TestTransaction (2)
class FruitTests {
  @Test
  public void findByName() { (4)
    Fruit.persist(new Fruit(null, “Grapefruit”, “Summer fruit”)); (5)

    Optional<Fruit> fruit = Fruit.findByName(“Grapefruit”); (6)
    assertThat(fruit)
      .isNotNull() (7)
      .isPresent() (8)
      .get()
      .extracting(“name”, “description”) (9)
      .containsExactly(“Grapefruit”, “Summer fruit”); (9)

      assertThat(fruit.get().id)
        .isNotNull() (10)
        .isGreaterThan(2L); (11)
  }
}
```

The tests are all nearly identical, aside from the annotations on the test classes.

1. Annotation denoting the class as a test.
2. Annotation denoting that every test method should run in an isolated transaction, which is rolled back automatically after each test method is executed. In the Spring example, this annotation comes from the `org.springframework.transaction.annotation` package, not the `javax.transaction` package.

3. Injection of the repository class. Not needed in the Quarkus Panache active record pattern because there is no repository class.
4. Test method for the `findByName` operation.
5. Persists a `Fruit` object into the database.
6. Retrieves the `Fruit` object using the `findByName` method.
7. Asserts that the retrieved `Fruit` is not `null`.
8. Asserts that the retrieved `Fruit` is present (not `Optional.empty()`).
9. Extracts the name and description attributes from the `Fruit` object and asserts that they contain the correct values. The Spring Data JPA and Quarkus Panache repository pattern can use method references to the getter methods. The Quarkus Panache active record pattern needs to use the `String` names of the attributes instead because the `Fruit` object attributes are `public`, and there are no getter methods.
10. Asserts that the `id` of the `Fruit` object is not `null`. The `Fruit` object was persisted without a defined `id`, so the database generated an ID.
11. Asserts that the `id` of the `Fruit` is greater than the value `2`. The new `Fruit` should have an `id` greater than `2` because the database table already contained two entries having IDs `1` and `2`.

Mocking Persistence Layer

Testing classes that consume the JPA repositories and entities involves mocking the interactions with the repositories and entities. Additionally, this style of testing does not require an in-memory or external database to be present. The examples here are very similar to the examples from Chapter 3 in the *Testing RESTful Endpoints* section and follow the same pattern:

1. Initialize any necessary interactions on mock objects needed by the test fixture.
2. Perform the test on the test fixture.
3. Verify results on the test fixture.
4. Verify expected conditions on mock object interactions.

The Spring MVC controller test class and the Quarkus resource test class using the Panache repository pattern setup are nearly identical: The `FruitRepository` bean is injected into the test as a mock. Each test method sets an interaction on the repository and then executes a test of the REST endpoint while performing assertions. There is no repository class in the Quarkus Panache active record pattern, therefore there is nothing to inject into the test. Additionally, the tests of the resource class in that case will simply mock the interactions on the `Fruit` class.

Listings 4.17. 4.18, and 4.19 show the tests for the `addFruit` operation.

Listing 4.17: Spring MVC Test for addFruit operation.

```
@SpringBootTest
@AutoConfigureMockMvc
class FruitControllerTests {
  @Autowired
  MockMvc mockMvc;

  @MockBean
  FruitRepository fruitRepository;

  @Test
  public void addFruit() throws Exception {
    Mockito.when(fruitRepository.save(Mockito.any(Fruit.class)))
      .thenReturn(new Fruit(1L, “Grapefruit”, “Summer fruit”)); (1)

    mockMvc.perform( (2)
      post(“/fruits”)
        .contentType(MediaType.APPLICATION_JSON)
        .content(“{\”name\”:\”Grapefruit\”,\”description\”:\”Summer
fruit\”}”)
      )
      .andExpect(status().isOk())
      .andExpect(content().contentTypeCompatibleWith(MediaType.
         APPLICATION_JSON))
      .andExpect(jsonPath(“id”).value(1))
      .andExpect(jsonPath(“name”).value(“Grapefruit”))
      .andExpect(jsonPath(“description”).value(“Summer fruit”));

    Mockito.verify(fruitRepository).save(Mockito.any(Fruit.class)); (3)
    Mockito.verifyNoMoreInteractions(fruitRepository); (4)
  }
}
```

Note: Only a single test is shown here. All the tests for all operations in all examples can be found in the examples GitHub repository [4.7].

Listing 4.18: Quarkus Test for addFruit operation using Panache repository pattern.

```
@QuarkusTest
class FruitResourceTests {
  @InjectMock
  FruitRepository fruitRepository;

  @Test
  public void addFruit() {
    Mockito.doNothing()
      .when(fruitRepository).persist(Mockito.any(Fruit.class)); (1)

    given() (2)
      .when()
        .contentType(ContentType.JSON)
        .body(“{\”id\”:1,\”name\”:\”Grapefruit\”,\”description\”:
           \”Summer fruit\”}”)
        .post(“/fruits”)
      .then()
        .statusCode(200)
        .contentType(ContentType.JSON)
        .body(
          “id”, is(1),
          “name”, is(“Grapefruit”),
          “description”, is(“Summer fruit”)
        );

    Mockito.verify(fruitRepository).persist(Mockito.any(Fruit.class)); (3)
    Mockito.verifyNoMoreInteractions(fruitRepository); (4)
  }
}
```

Listing 4.19: Quarkus Test for addFruit operation using Panache active record pattern.

```
@QuarkusTest
class FruitResourceTests {
  @Test
  public void addFruit() {
    PanacheMock.mock(Fruit.class); (5)
    PanacheMock.doNothing()
      .when(Fruit.class)
      .persist(Mockito.any(Fruit.class), Mockito.any()); (1)

    given() (2)
      .when()
        .contentType(ContentType.JSON)
        .body("{\"id\":1,\"name\":\"Grapefruit\",\"description\":
           \"Summer fruit\"}")
        .post("/fruits")
      .then()
        .statusCode(200)
        .contentType(ContentType.JSON)
        .body(
          "id", is(1),
          "name", is("Grapefruit"),
          "description", is("Summer fruit")
        );

    PanacheMock.verify(Fruit.class).persist(Mockito.any(Fruit.class),
Mockito.any()); (3)
    PanacheMock.verifyNoMoreInteractions(Fruit.class); (4)
  }
}
```

Again, the tests in each example are very similar.

1. Defines an interaction to return an appropriate result from the JPA repository/entity. The Spring Data JPA and Quarkus Panache repository pattern examples directly use the Mockito library for mocking. The Quarkus Panache active record pattern can't use Mockito because the mocked methods are `static`. Instead, Quarkus introduces the `PanacheMock` class containing useful methods for mocking active record entities, mirroring those provided by Mockito.
2. Performs the test and verify the results.
3. Verifies that the expected method on the mock was called once.
4. Verifies that no other methods on the mock were called.
5. Creates a mock of the `Fruit` class. Needed only in the Quarkus Panache active record pattern because there is no repository. Instead, the `static` methods on the entity class need to be mocked. This is conceptually similar to injecting a mock of the repository into the test class in the other examples.

Reactive Data Access

As application architectures shift towards microservice and reactive architectures, existing relational databases simply can't be thrown away. Today's modern applications need to provide low latency and high throughput to data. Existing data access technologies based on JDBC and its blocking APIs force applications to bring lots of data into memory before being pushed or pulled out to the application's consumer(s).

Making data access reactive requires an entirely new set of APIs. Unfortunately, there currently is no standard defining non-blocking relational data access. The lack of a standard has spawned the creation of two open source projects attempting to provide full non-blocking support for relational databases: the Reactive Relational Database Connectivity (R2DBC) project [4.19] and the Eclipse Vert.x reactive database clients [4.20], part of the larger Eclipse Vert.x project [4.21] for building reactive applications on the JVM.

R2DBC is a specification defining a non-blocking Service Provider Interface (SPI) [4.22] for driver vendors to implement and clients to consume. The design is similar to JDBC, except that JDBC is aimed at both database driver writers and application developers. R2DBC is not intended to be used directly within application code. Vendor-specific R2DBC drivers implement the database wire protocol on a nonblocking I/O layer. It provides a minimal SPI and does not target a data store's specific functionality.

The Vert.x reactive clients are similar in that they provide a common interface that each vendor can implement. These clients also allow vendors to provide functionality specific to the capabilities of a particular data store, similar to clients provided by existing nonrelational data stores, such as MongoDB, Apache Cassandra, and Redis.

One fundamental difference, however, is that the Vert.x reactive clients are based on the request-and-reply protocol used by databases and do not abuse the protocol by keeping connections, and sometimes transactions, open until the last record is read. This is why the Vert.x reactive clients do not stream result sets from the database by default. Streaming from a database would dramatically reduce concurrency, one of the core reactive principles. The Vert.x reactive clients, using a specific API and not by default, can stream using cursors.

Spring Data R2DBC

The Spring Data R2DBC subproject [4.23], part of the larger Spring Data umbrella project, brings common Spring opinions, abstractions, and repository support for reactive relational data access to Spring applications using R2DBC as the underlying data access API. Spring Data R2DBC is a simple, limited, and opinionated object mapper. Many ORM framework features, such as caching, lazy loading, or write-behind, are not offered.

Quarkus Hibernate Reactive with Panache

Hibernate Reactive [4.24] is a reactive API for Hibernate ORM, a true ORM implementation supporting non-blocking database drivers and clients. Out of the box, Hibernate Reactive uses the Vert.x reactive clients to provide the underlying database connection. Additionally, Hibernate Reactive reuses many of the same concepts and code from JPA, especially when it comes to mapping entity classes.

Quarkus Hibernate Reactive with Panache is the same Panache with all the same features discussed previously in this chapter, but using Hibernate Reactive as the ORM engine. Both the repository and active record patterns are available.

Reactive Data Access in Action

The complete examples throughout the remainder of this section can be found in the examples GitHub repository [4.7]:

- **Spring Data R2DBC:** Inside the `chapter-4/chapter-4-spring-data-r2dbc` project.
- **Quarkus Panache Reactive Repository Pattern:** Inside the `chapter-4/chapter-4-quarkus-panache-reactive-repository` project.

- **Quarkus Panache Reactive Active Record Pattern**: Inside the `chapter-4/chapter-4-quarkus-panache-reactive-activerecord` project.

Like Hibernate with JPA, Hibernate Reactive can also create a database schema from all the application's entities. Hibernate Reactive does not yet support verification of an existing schema as of this writing.

Note: Database versioning technologies, such as Flyway [4.8] and Liquibase [4.9], are not discussed in this book. As of this writing, they support only blocking JDBC drivers and therefore would require an application to include both reactive and JDBC driver libraries and configuration.

Database Setup

The database setup for reactive data access is identical to the JPA section's database setup discussed previously: an existing PostgreSQL 13 database with an existing schema. The same PostgreSQL container image [4.11] is reused.

The container needs to be started before running the `spring-boot:run` Maven goal of the Spring Data R2DBC example, starting any of the examples using `java -jar`, or running the Quarkus examples in a native image.

Application Data Source Configuration

Listing 4.20 shows the data source configuration, located in `src/main/resources/application.yml`, for Spring Data R2DBC. Listing 4.21 shows the data source configuration, located in `src/main/resources/application.yml`, for Quarkus Panache. The testing configuration will be covered later in this chapter.

The configuration for the Panache repository pattern is identical to the configuration for the active record pattern.

Listing 4.20: Spring Data R2DBC data source configuration.

```
spring:
  r2dbc:
    username: fruits (1)
    password: fruits (2)
    url: r2dbc:postgresql://localhost:5432/fruits (3)
```

Listing 4.21: Quarkus Panache data source configuration.

```
quarkus:
  datasource:
    username: fruits (1)
    password: fruits (2)
```

1. Sets the data source's username.
2. Sets the data source's password.
3. Sets the data source's URL, assuming a PostgreSQL database running on `localhost:5432`. Similar to the JPA section, this is needed only in Spring to support local development.

Quarkus Dev Mode

The Quarkus Dev Services [4.12] capability shown previously also applies to reactive data sources. Its usage and configuration is identical when working with reactive data sources. Listing 4.3 from the JPA section shows how to configure Quarkus Dev Services inside `src/main/resources/application.yml`.

Entity Design

As in the JPA section, the entity design is identical in all the examples because they all use the same schema. The code, however, is different in each case. Spring Data R2DBC is not a full ORM implementation, whereas Hibernate Reactive is. In Spring Data R2DBC, an entity class needs to be annotated with Spring Data specific annotations, whereas an entity class in Quarkus reuses the same JPA annotations. The entity classes in the Quarkus Panache Reactive examples are identical to the Quarkus Panache ones in the previous section.

Listing 4.22 shows the `Fruit` entity class used in Spring Data R2DBC. Listing 4.4 from the JPA section shows the entity class in the Quarkus Panache Reactive repository pattern.

Listing 4.22: Spring Data R2DBC Fruit.java entity.

```
@Table("fruits") (1)
public class Fruit {
  @Id (2)
  private Long id;

  @NotBlank(message = "Name is mandatory") (3)
  private String name;
  private String description;

  public Long getId() { return this.id; }
  public void setId(Long id) { this.id = id; }
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  public String getDescription() { return this.description; }
  public void setDescription(String description) { this.description =
    description; }

  // Constructors, hashCode, equals, toString methods omitted for brevity
}
```

Note: The `@Table` and `@Id` annotations come from the `org.springframework.data` package, not the `javax.persistence` package as in the Spring Data JPA example. Additionally, no `@GeneratedValue` or `@Column` annotations are provided, because Spring Data R2DBC is not a complete ORM implementation.

1. Annotation marking the class as a mapper for the `fruits` table.
2. Annotation denoting a mapping column identifier.
3. Annotation marking the Java class field as not empty. Spring Data R2DBC performs validation on the object before mapping it to the database.

Repository Implementation

Both the Spring Data R2DBC and the Quarkus Panache Reactive repository patterns promote a *repository* class containing an entity's operations. The pattern and concept is identical to the Spring Data JPA and Quarkus Panache repository pattern. Listing 4.23 shows a Spring Data R2DBC repository and Listing 4.24 shows a Quarkus Panache Reactive repository.

Listing 4.23: Spring Data R2DBC repository.

```
public interface FruitRepository extends ReactiveCrudRepository
  <Fruit, Long> {
  Mono<Fruit> findByName(String name);
}
```

Listing 4.24: Quarkus Panache Reactive repository.

```
@ApplicationScoped
public class FruitRepository implements PanacheRepositoryBase<Fruit,
Long> {
  public Uni<Fruit> findByName(String name) {
    return find("name", name).firstResult();
  }
}
```

The main way in which Listings 4.23 and 4.24 here differ from Listings 4.5 and 4.6 in the JPA section is the use of reactive types. Methods on the Spring Data R2DBC `ReactiveSortingRepository` use the `Mono` and `Flux` reactive types, whereas methods on the Quarkus Panache Reactive `PanacheRepositoryBase` use the `Uni` and `Multi` reactive types.

Note: Reactive types are discussed in detail in Chapter 3.

Quarkus Panache Active Record Pattern

Quarkus Panache Reactive also supports the active record pattern. The pattern is exactly the same in Quarkus Panache Reactive as in Quarkus Panache. The only difference is that methods in the `PanacheEntityBase` class use the `Uni` and `Multi` reactive types. Listing 4.25 shows the same `findByName` method as Listing 4.7, but defined in the Panache Reactive active record pattern.

Listing 4.25: Panache Reactive active record pattern findByName method.

```
public static Uni<Fruit> findByName(String name) {
  return find("name", name).firstResult();
}
```

Using the Persistence Layer

Using the persistence layer is the same in reactive data access as in the JPA section. The same examples [4.7] used in the JPA section are used in this section.

The implementations for all the operations in all examples can be found in the examples GitHub repository [4.7].

Listing 4.26 shows a Spring WebFlux controller class with an HTTP `POST`. The main difference between Listings 4.8 and 4.26 is the `Mono` reactive type. Listing 4.27 shows a Quarkus JAX-RS resource class implementing the same operation using the Panache Reactive repository pattern, and Listing 4.28 shows the same operation implemented in a Quarkus JAX-RS resource class using the Panache Reactive active record pattern. These listings are also very similar to Listings 4.9 and 4.10, respectively.

Note: The details for creating Spring WebFlux controllers and Quarkus JAX-RS resource classes are not discussed in this chapter. They are discussed in detail in Chapter 3.

Listing 4.26: Spring WebFlux controller using R2DBC repository.

```
@RestController
@RequestMapping("/fruits")
public class FruitController {
  private final FruitRepository fruitRepository; (1)

  public FruitController(FruitRepository fruitRepository) {
    this.fruitRepository = fruitRepository;
}

  @PostMapping(produces = MediaType.APPLICATION_JSON_VALUE, consumes =
    MediaType.APPLICATION_JSON_VALUE)
  @Transactional (2)
  public Mono<Fruit> addFruit(@Valid @RequestBody Fruit fruit) {
    return fruitRepository.save(fruit); (3) (4)
  }
}
```

Listing 4.27: Quarkus resource class using Panache Reactive repository.

```
@Path("/fruits")
public class FruitResource {
  private final FruitRepository fruitRepository; (1)

  public FruitResource(FruitRepository fruitRepository) {
    this.fruitRepository = fruitRepository;
  }

  @POST
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  @ReactiveTransactional (2)
  public Uni<Fruit> addFruit(@Valid Fruit fruit) {
    return fruitRepository.persist(fruit); (3) (4)
  }
}
```

Listing 4.28: Quarkus resource class using Panache Reactive active record.

```
@Path("/fruits")
public class FruitResource {
  @POST
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  @ReactiveTransactional (2)
  public Uni<Fruit> addFruit(@Valid Fruit fruit) {
    return Fruit.persist(fruit) (3)
                .replaceWith(fruit); (4)
  }
}
```

1. Injection of the `FruitRepository`. Not needed in the Quarkus Panache active record pattern because there is no repository class.
2. Denotes that the operation executes in the context of a transaction. Spring WebFlux, like Spring MVC, can use the `@Transactional` annotation (from the `org.springframework.transaction.annotation` package) on a method to signify a transaction boundary. In Quarkus Panache Reactive, the transaction boundary is defined using the `@ReactiveTransactional` annotation.
3. Persists a fruit to the database.
4. Returns the persisted fruit.

Reactive Data Access Testing

The same patterns used for testing a JPA-based application also apply to reactive data sources. Any additional or custom operations created on a Spring Data R2DBC repository, Quarkus Panache Reactive repository, or Quarkus Panache Reactive active record entity need to be tested. Additionally, tests for classes consuming the persistence layer need to mock the Spring Data R2DBC repository, Quarkus Panache Reactive repository, or Quarkus Panache Reactive active record entity.

Test Database Setup

The same Testcontainers [4.15] testing structure used previously in the JPA examples is reused here.

Database Schema

The database schema in the reactive examples is identical to the one in the JPA examples, and the examples use the same schema setup inside `src/test/resources/db/schema.sql`.

Application Test Configuration

Tests for reactive data sources use the same testing structure as the JPA examples. In Spring, it needs to be set up differently for reactive data sources. Simply adding a data source URL to the properties doesn't work. Instead, a custom class utilizing the Testcontainers API needs to be created for Spring. This class bootstraps Testcontainers in a reusable way within the tests. In Quarkus, the same Dev Services setup used in the JPA section is reused here. There is nothing else a developer needs to do for Quarkus to bootstrap a PostgreSQL container when executing tests.

For Spring, this class will be an `abstract` base class, extended by any tests needing Testcontainers support. One main difference between the Spring and Quarkus implementations is that in Spring, the Testcontainer is restarted before each test class, whereas in Quarkus, it is started only once and reused for all tests.

Listing 4.30 shows the Spring base class, which is extended by any test class needing a database container.

Listing 4.30: Spring TestContainerBase.java.

```
@Testcontainers (1)
@DirtiesContext (2)
public abstract class TestContainerBase {
  @Container (3)
  static final PostgreSQLContainer<?> DB =
    new PostgreSQLContainer<>("postgres:13") (4)
      .withDatabaseName("fruits") (5)
      .withUsername("fruits") (6)
      .withPassword("fruits") (7)
      .withInitScript("db/schema.sql"); (8)

  @DynamicPropertySource
  static void registerDynamicProperties(DynamicPropertyRegistry regis-
try) { (9)
    registry.add("spring.r2dbc.url", TestContainerBase::getDbUrl);
    registry.add("spring.r2dbc.username", DB::getUsername);
    registry.add("spring.r2dbc.password", DB::getPassword);
  }

  private static String getDbUrl() {
    return String.format("r2dbc:postgresql://%s:%d/%s", DB.getHost(),
DB.getFirstMappedPort(), DB.getDatabaseName());
  }
}
```

1. JUnit Jupiter extension to activate automatic startup and stopping of containers used in a test case.

2. Ensures that every subclass gets a fresh `ApplicationContext` with updated properties.
3. Annotation used in conjunction with the `@Testcontainers` annotation to mark containers that the Testcontainers extension should manage.
4. Creates a `postgres:13` container image.
5. Sets the database name on the database to `fruits`.
6. Sets the username on the database to `fruits`.
7. Sets the password on the database to `fruits`.
8. Sets the database initialization script to create the table and populate it with data.
9. Register a set of datasource properties dynamically to inject into the application tests.

Testing Additional Data Access Operations

Listings 4.24, 4.25, and 4.26 introduce a single additional operation named `findByName`. Listing 4.31 shows a test class for `findByName` using Spring Data R2DBC. Listing 4.32 shows a test class for the same method using the Quarkus Panache Reactive repository pattern. Listing 4.33 shows a test class for the method using the Quarkus Panache Reactive active record pattern.

The flow of each test is identical in each example as well as in the JPA examples shown earlier in this chapter:

1. Persist a new `Fruit` into the database.
2. Retrieve the persisted `Fruit` from the database using the `findByName` method.
3. Perform assertions on the retrieved `Fruit` to validate that it is the correct `Fruit`.

Note: All the examples use the AssertJ assertions library [4.18].

Listing 4.31: Spring Data R2DBC test class for findByName operation.

```
@SpringBootTest
class FruitRepositoryTests extends TestContainerBase { (1)
  @Autowired
  FruitRepository fruitRepository; (2)

  @Autowired
  TestTransaction testTransaction; (3)

  @Test
  public void findByName() {
    Fruit fruit = TestTransaction.withRollback(() -> (4)
      fruitRepository
        .save(new Fruit(null, “Grapefruit”, “Summer fruit”)) (5)
        .then(fruitRepository.findByName(“Grapefruit”)) (6)
    )
      .block(Duration.ofSeconds(10)); (7)

    assertThat(fruit)
      .isNotNull() (8)
      .extracting(Fruit::getName, Fruit::getDescription) (9)
      .containsExactly(“Grapefruit”, “Summer fruit”); (9)

    assertThat(fruit.getId())
      .isNotNull() (10)
      .isGreaterThan(2L); (11)
  }
}
```

Listing 4.32: Quarkus Panache Reactive repository pattern test class for findByName operation.

```
@QuarkusTest
class FruitRepositoryTests {
  @Inject
  FruitRepository fruitRepository; (2)

  @Test
  public void findByName() {
    Fruit fruit = TestTransaction.withRollback(() -> (4)
      fruitRepository
        .persist(new Fruit(null, “Grapefruit”, “Summer fruit”)) (5)
        .replaceWith(fruitRepository.findByName(“Grapefruit”)) (6)
    )
      .await() (7)
      .atMost(Duration.ofSeconds(10)); (7)

    assertThat(fruit)
      .isNotNull() (8)
      .extracting(Fruit::getName, Fruit::getDescription) (9)
      .containsExactly(“Grapefruit”, “Summer fruit”); (9)

    assertThat(fruit.getId())
      .isNotNull() (10)
      .isGreaterThan(2L); (11)
  }
}
```

Listing 4.33: Quarkus Panache Reactive active record pattern test class for findByName operation.

```
@QuarkusTest
class FruitTests {
  @Test
  public void findByName() {
    Fruit fruit = TestTransaction.withRollback(() -> (4)
      Fruit
        .persist(new Fruit(null, “Grapefruit”, “Summer fruit”)) (5)
        .replaceWith(Fruit.findByName(“Grapefruit”)) (6)
    )
      .await() (7)
      .atMost(Duration.ofSeconds(10)); (7)

    assertThat(fruit)
      .isNotNull() (8)
      .extracting(“name”, “description”) (9)
      .containsExactly(“Grapefruit”, “Summer fruit”); (9)

    assertThat(fruit.id)
      .isNotNull() (10)
      .isGreaterThan(2L); (11)
  }
}
```

1. Wiring the Testcontainers support from the previous section. In Spring, the `TestContainerBase` class is extended. Nothing is needed in Quarkus, as Quarkus Dev Services will automatically bootstrap the database container instance.
2. Injection of the repository class. Not needed in the Quarkus Panache Reactive active record pattern, because there is no repository class.
3. Injection of a `TestTransaction` bean. Needed only in the Spring example because a Spring bean is required. Transaction support in tests are discussed next in this section.
4. Sets the transaction to roll back automatically at the end. Unlike the JPA versions at the beginning of this chapter, there are no annotations in Spring or Quarkus to automatically mark each test method as a transaction and perform an automatic rollback. Rollback must be performed manually within each reactive pipeline requiring it. The examples here use a custom class, `TestTransaction`, for managing automatic rollback. The `TestTransaction` class for Spring and Quarkus are examined in the next section.
5. Persists a `Fruit` object into the database.
6. Retrieves the `Fruit` object using the `findByName` method.
7. Blocks and waits for the reactive pipeline to complete, waiting at most 10 seconds before failing.
8. Asserts that the retrieved `Fruit` is not `null`.
9. Extracts the name and description attributes from the `Fruit` object and asserts that they contain the correct values. The Spring Data R2DBC and Quarkus Panache Reactive repository pattern can use method references to the getter methods. The Quarkus Panache Reactive active record pattern needs to use the `String` names of the attributes instead, because the `Fruit` object attributes are `public` and there are no getter methods.
10. Asserts that the `id` of the `Fruit` object is not `null`. The `Fruit` object was persisted without a defined `id`, so the database generated an ID.
11. Asserts that the `id` of the `Fruit` is greater than the value `2`. The new `Fruit` should have an `id` greater than `2` because the database table already contained two entries having IDs `1` and `2`.

Test Transactions

As mentioned in the previous section, transaction management and automatic rollback in tests require manual coding within the reactive pipeline. The Spring and Quarkus examples use a custom `TestTransaction` class for managing automatic rollback. Listing 4.34 shows the Spring version of this class, and Listing 4.35 shows the Quarkus version.

Listing 4.34: Spring Data R2DBC TestTransaction.java.

```
@Component (1)
public class TestTransaction {
  private final TransactionalOperator rxtx; (2)

  public TestTransaction(TransactionalOperator rxtx) {
    this.rxtx = rxtx;
  }

  public <T> Mono<T> withRollback(Supplier<Mono<T>> mono) {
    return rxtx.execute(tx -> { (3)
      tx.setRollbackOnly(); (4)
      return mono.get(); (5)
    }).next(); (6)
  }
}
```

Listing 4.35: Quarkus TestTransaction.java.

```
public class TestTransaction {
  public static <T> Uni<T> withRollback(Supplier<Uni<T>> uni) {
    return Panache.getSession()
      .flatMap(sesssion -> session.withTransaction(tx -> { (3)
        tx.markForRollback(); (4)
        return uni.get(); (5)
    }));
  }
}
```

1. Marks the class as a Spring `@Component` bean. Needed only in Spring so that the Spring `TransactionalOperator` bean can be injected as the transaction manager from the Spring test `ApplicationContext`.
2. Injection of Spring's `TransactionalOperator` bean. Needed only in Spring.
3. Executes the reactive pipeline within the context of a transaction.
4. Tells the transaction to roll back when complete.
5. Returns the reactive pipeline.
6. Returns the reactive pipeline as a `Mono`. Needed only in Spring because `TransactionalOperator.execute` returns a `Flux`.

Mocking Persistence Layer

As in the JPA section, testing classes that consume database repositories and entities involves mocking the interactions with the repositories and entities. The pattern followed for reactive data sources is identical to blocking data sources:

1. Initialize any necessary interactions on mock objects needed by the test fixture.
2. Perform the test on the test fixture.
3. Verify results on the test fixture.
4. Verify expected conditions on mock object interactions.

Listings 4.36 shows the test for the `addFruit` operation in the Spring WebFlux controller class. Listings 4.37 and 4.38 show the test for the `addFruit` operation in the Quarkus JAX-RS resource classes.

All the tests for all operations in all examples can be found in the examples GitHub repository [4.7]

Listing 4.36: Spring WebFlux Test for addFruit operation.

```
@SpringBootTest
@AutoConfigureWebTestClient
class FruitControllerTests extends TestContainerBase {
  @Autowired
  WebTestClient webTestClient;

  @MockBean
  FruitRepository fruitRepository;

  @Test
  public void addFruit() {
    Mockito.when(fruitRepository.save(Mockito.any(Fruit.class)))
      .thenReturn(Mono.just(new Fruit(1L, "Grapefruit", "Summer
        fruit"))); (1)
```

```
    webTestClient.post() (2)
      .uri("/fruits")
      .contentType(MediaType.APPLICATION_JSON)
      .bodyValue("{\"name\":\"Grapefruit\",\"description\":
         \"Summer fruit\"}"))
      .exchange()
      .expectStatus().isOk()
      .expectHeader().contentTypeCompatibleWith(MediaType.APPLICATION_
JSON)
      .expectBody()
        .jsonPath("id").isEqualTo(1)
        .jsonPath("name").isEqualTo("Grapefruit")
        .jsonPath("description").isEqualTo("Summer fruit"));

    Mockito.verify(fruitRepository).save(Mockito.any(Fruit.class)); (3)
    Mockito.verifyNoMoreInteractions(fruitRepository); (4)
  }
}
```

Listing 4.37: Quarkus Test for addFruit operation using Panache Reactive repository pattern.

```
@QuarkusTest
class FruitResourceTests {
  @InjectMock
  FruitRepository fruitRepository;

  @Test
  public void addFruit() {
    Mockito.when(fruitRepository.persist(Mockito.any(Fruit.class)))
      .thenReturn(Uni.createFrom().item(new Fruit(1L, "Grapefruit",
         "Summer fruit"))); (1)

    given() (2)
      .when()
        .contentType(ContentType.JSON)
        .body("{\"name\":\"Grapefruit\",\"description\":\"Summer
fruit\"}")
        .post("/fruits")
      .then()
        .statusCode(200)
        .contentType(ContentType.JSON)
        .body(
          "id", is(1),
          "name", is("Grapefruit"),
          "description", is("Summer fruit")
        );

    Mockito.verify(fruitRepository).persist(Mockito.any(Fruit.class)); (3)
    Mockito.verifyNoMoreInteractions(fruitRepository); (4)
  }
}
```

Listing 4.38: Quarkus Test for addFruit operation using Panache Reactive active record pattern.

```
@QuarkusTest
class FruitResourceTests {
  @Test
  public void addFruit() {
    PanacheMock.mock(Fruit.class); (5)
    PanacheMock.doReturn(Uni.createFrom().voidItem())
      .when(Fruit.class).persist(Mockito.any(Fruit.class),
         Mockito.any()); (1)

    given() (2)
      .when()
        .contentType(ContentType.JSON)
        .body("{\"id\":1,\"name\":\"Grapefruit\",\"description\":
           \"Summer fruit\"}")
        .post("/fruits")
      .then()
        .statusCode(200)
        .contentType(ContentType.JSON)
        .body(
          "id", is(1),
          "name", is("Grapefruit"),
          "description", is("Summer fruit")
        );

    PanacheMock.verify(Fruit.class).persist(Mockito.any(Fruit.class),
      Mockito.any()); (3)
    PanacheMock.verifyNoMoreInteractions(Fruit.class); (4)
  }
}
```

The tests in each example are very similar to each other and to the JPA tests at the beginning of the chapter.

1. Defines an interaction to return an appropriate result from the repository or entity. The Spring Data R2DBC and Quarkus Panache Reactive repository pattern examples directly use the Mockito library for mocking. The Quarkus Panache Reactive active record pattern can't use Mockito because the mocked methods are `static`. Instead, Quarkus uses the same `PanacheMock` class from the JPA section. The `PanacheMock` class contains many useful methods for mocking active record entities.
2. Performs the test and verify the results.
3. Verifies that the expected method on the mock was called once.
4. Verifies that no other methods on the mock were called.
5. Creates a mock of the `Fruit` class. Needed only in the Quarkus Panache Reactive active record pattern, because there is no repository. Instead, the `static` methods on the entity class need to be mocked. This is conceptually similar to injecting a mock of the repository into the test class in the other examples.

Summary

As shown in this chapter, Spring Data JPA and Quarkus Panache are parallel in many ways. Spring Data R2DBC and Quarkus Panache Reactive also demonstrate many parallels for reactive data access. However, one main difference is that Quarkus Panache Reactive is based on a complete ORM implementation, Hibernate Reactive. In contrast, Spring Data R2DBC is a more lightweight object mapper and not a complete ORM implementation. A developer familiar with many of the Spring Data projects should become familiar with persistence in Quarkus fairly quickly. Again, one of the main differences is and will continue to be the number of optimizations done at build time versus runtime.

References

[4.1] "Microservice Architecture: Database per Service Pattern": https://microservices.io/patterns/data/database-per-service.html

[4.2] "Object-relational mapping": https://en.wikipedia.org/wiki/Object-relational_mapping

[4.3] "Hibernate Project": https://hibernate.org

[4.4] "Spring Data": https://spring.io/projects/spring-data

[4.5] "Spring Data JPA": https://spring.io/projects/spring-data-jpa

[4.6] "Quarkus - Simplified Hibernate ORM with Panache": https://quarkus.io/guides/hibernate-orm-panache

[4.7] "Examples Repository": https://github.com/quarkus-for-spring-developers/examples

[4.8] "Flyway": https://flywaydb.org

[4.9] "Liquibase": https://www.liquibase.org

[4.10] "PostgreSQL Serial Types": https://www.postgresql.org/docs/13/datatype-numeric.html#DATATYPE-SERIAL

[4.11] "Examples PostgreSQL Container Image": https://quay.io/repository/edeandrea/postgres-13-fruits?tag=latest&tab=tags

[4.12] "Quarkus Dev Services: Configuration-free Databases": https://quarkus.io/guides/datasource#dev-services-configuration-free-databases

[4.13] "Spring Data JPA Reference - Query Methods": https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

[4.14] "H2 Database Engine": https://www.h2database.com

[4.15] "Testcontainers": https://www.testcontainers.org

[4.16] "Testcontainers: JDBC URL Examples": https://www.testcontainers.org/modules/databases/jdbc/#using-postgresql

[4.17] "Testcontainers: Using a Classpath Init Script": https://www.testcontainers.org/modules/databases/jdbc/#using-a-classpath-init-script

[4.18] "AssertJ - Fluent Assertions Java Library": https://assertj.github.io/doc

[4.19] "Reactive Relational Database Connectivity (R2DBC) Project": http://r2dbc.io

[4.20] "Eclipse Vert.x Reactive Database Clients": https://vertx.io/docs/#databases

[4.21] "Eclipse Vert.x": https://vertx.io

[4.22] "Service Provider Interface": https://en.wikipedia.org/wiki/Service_provider_interface

[4.23] "Spring Data R2DBC": https://spring.io/projects/spring-data-r2dbc

[4.24] "Hibernate Reactive": http://hibernate.org/reactive

Chapter 5

Event-Driven Services

Daniel Oh

Asynchronous communication protocols decouple applications, letting a caller send a message to a recipient and then continue processing other requests instead of waiting idly for a reply. Asynchronous behavior can improve performance, security, and scalability. With asynchronous communication growing in importance, message queue frameworks have become critical to decouple applications. For example, Apache Kafka (Kafka) [5.1] and Flume are popular platforms to process messages asynchronously nowadays.

Traditional message queuing technologies face challenges when processing messages in event-driven scenarios with high data volumes. For example, the messaging systems for complex event processing (CEP) [5.2] can't respond to messages quickly enough during a high volume of data.

Distributed messaging and publish-subscribe platforms were introduced into the messaging world to help solve some of these critical issues. Pub-sub messaging can also enable event-driven architectures [5.3] for high performance, reliability, and scalability.

Apache Kafka is one of the most popular distributed pub-sub message platforms for handling real-time data feeds while providing high throughput, low latency, and scalability. Event streaming and analytic platforms in many enterprises use Kafka to provide high-volume data access. Key capabilities of Apache Kafka include:

- High throughput: Kafka can reduce the latency of message handling to milliseconds, allowing applications at scale to handle millions of data points per second. This capability is suited for big data processing applications.
- Scalability: Clusters can grow transparently and elastically without downtime by minimizing point-to-point integrations for data sharing in applications.
- Storage: Programmers can build streaming data pipelines and store the data records in a fault-tolerant, durable way.
- High availability: Duplication of resources allows faster access to real-time data over multiple availability zones or across geographical regions by users.

A Kafka cluster divides data streams by criteria known as *topics* and delivers data in each topic to multiple subscribing processes, known as *partitions,* that receive and store the data. Figure 5.1 shows an architectural overview of Kafka.

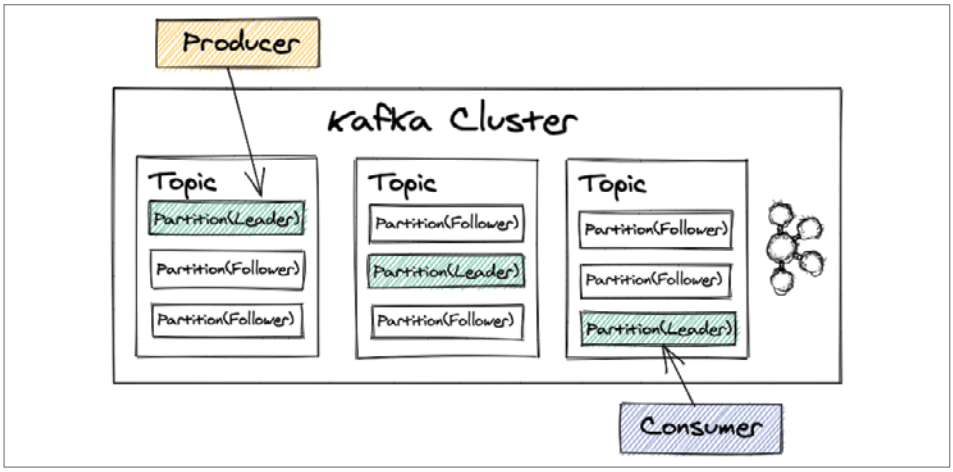

Figure 5.1: Kafka architecture overview.

This chapter explains the benefits of Quarkus for implementing event-driven services using publish-subscribe events, reactive streams with Kafka, and Knative events. As usual, we show Quarkus' similarities with and differences from Spring.

Event Message Handling

In event-driven architectures, message brokers facilitate asynchronous delivery of event messages between multiple applications. The event processors consume events, process them, and potentially publish other events at the end.

Spring Events and Integration

Spring supports event publishing and listening via its built-in `ApplicationEventPublisher`. This event publishing mechanism is "fire-and-forget," regardless of whether the listener is synchronous or asynchronous. The publisher emits an event (or message), and the consumer consumes the event, but there is no mechanism for the consumer to emit a response back to the publisher.

Spring Integration [5.4] enables developers to overcome these limitations with channel adapters for one-way and bidirectional integration and gateways for communication. Spring Integration supports enterprise integration solutions using lightweight messaging, remoting, scheduling with external systems, and "wiring" the application events into Spring Integration channels.

Quarkus Event Bus

Quarkus is built on Eclipse Vert.x [5.5] to implement network-related features, reactive programming, and integration with the Vert.x event bus [5.6]. The event bus enables Quarkus applications to pass messages asynchronously, one-way or bidirectionally, among event processors (e.g., reactive clients), as shown in Figure 5.2.

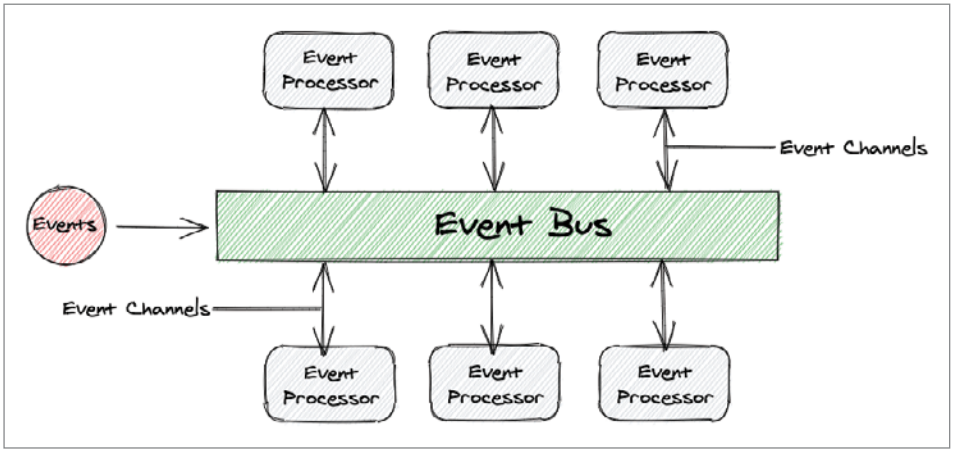

Figure 5.2: Quarkus event bus.

The Vert.x event bus is distributed, enabling developers to write reactive, non-blocking, asynchronous applications that run on the JVM. Thus, applications written in different languages can communicate over the event bus. For example, the event bus can be bridged to allow client-side JavaScript running in a browser to communicate on the same event bus as other applications. The event bus supports three delivery mechanisms, described in Table 5.1.

Table 5.1- Vert.x Delivery Mechanisms

Delivery	Description
Point-to-point	Messages are sent to a virtual address. One of the consumers receives each message. If there are more than one consumer registered at the address, one will be chosen by a round-robin algorithm.
Publish-subscribe	Messages are published to a virtual address. All consumers listening to the address receive the messages.
Request-reply	A message is received by a single consumer, who needs to reply to the message asynchronously.

Event Message Handling in Action

The complete examples throughout the remainder of this section can be found in the examples GitHub repository [5.7]:

- **Spring Event Handling:** Inside the `chapter-5/chapter-5-spring-eventbus` project.
- **Quarkus Event Handling:** Inside the `chapter-5/chapter-5-quarkus-vertx-eventbus` project.

Consuming Events

In both Spring and Quarkus, events are processed by non-blocking asynchronous I/O threads or blocking I/O threads. However, one of the main differences is that Spring has no core component to implement the event bus capabilities that Quarkus supports out of the box with the Vert.x event bus. Instead, Spring Integration needs to be used to achieve similar functionality to the Quarkus event bus.

Listing 5.1 shows a Spring `GreetingService` class using Spring Integration and Listing 5.2 shows a Quarkus `GreetingService` class using the Quarkus event bus.

Listing 5.1: Spring GreetingService class using ServiceActivator.

```
@MessageEndpoint (1)
public class GreetingService {
  @ServiceActivator(inputChannel = "greeting", async = "true") (2)
  public Mono<String> consume(Mono<String> name) {
    return name.map(String::toUpperCase);
  }

  @ServiceActivator(inputChannel = "blocking-greeting") (3)
  public String consumeBlocking(String message) {
    return "Processing Blocking I/O: " + message;
  }
}
```

Listing 5.2: Quarkus GreetingService class using ConsumeEvent.

```
@ApplicationScoped
public class GreetingService {
  @ConsumeEvent("greeting") (2)
  public String consume(String name) {
    return name.toUpperCase();
  }

  @ConsumeEvent("blocking-consumer")
  @Blocking (3)
  public String consumeBlocking(String message) {
    return "Processing Blocking I/O: " + message;
  }
}
```

1. Annotation marking the class as a message endpoint by Spring Integration.
2. Annotation indicating that the method receives the messages sent to the `greeting` address (or input channel). Spring Integration processes the event message synchronously, so the service activator should set the `async` value to `true` for asynchronous processing. Message processing in Quarkus is asynchronous by default.
3. Annotation indicating that the method processes the event using blocking I/O.

Another benefit of the `ConsumeEvent` annotation in Quarkus is that developers can use "fire and forget" interaction when the messages are consumed and the sender does not expect a response. To implement this, the consumer method just needs to return `void`.

Publishing Events

Spring and Quarkus support publishing events in a slightly different way. Spring Integration allows developers to implement a `MessagingGateway` [5.8] that creates a publish-subscribe channel to send messages and return responses. A `MessagingGateway` hides the messaging API provided by Spring Integration. It lets an application's business logic be unaware of the Spring Integration API. By using a generic `Gateway`, application code interacts with only a simple interface.

Quarkus, on the other hand, allows developers to simply inject the `EventBus` bean into other beans to publish events and await responses. To enable the Vert.x Event Bus capabilities, Quarkus provides the `vertx` extension [5.9] that can be added to an existing Quarkus application.

Listing 5.3 shows a Spring `GreetingGateway` interface to implement the Spring Integration Gateway interface and Listing 5.4 shows a Spring `GreetingController` class, demonstrating how to map HTTP requests via REST APIs to the gateway. Listing 5.5 shows a Quarkus `GreetingResource` class, demonstrating how to inject the event bus for mapping HTTP requests to the bus.

Listing 5.3: Spring GreetingGateway implementation.

```
@MessagingGateway (1)
public interface GreetingGateway {
  @Gateway(requestChannel = "greeting") (2)
  Mono<String> greeting(Mono<String> input);

  @Gateway(requestChannel = "blocking-greeting")
  String blockingGreeting(String input);
}
```

Listing 5.4: Spring GreetingController using GreetingGateway.

```
@RestController
@RequestMapping("/async")
public class GreetingController {
  private final GreetingGateway greetingGateway; (3)

  public GreetingController(GreetingGateway greetingGateway) {
    this.greetingGateway = greetingGateway;
  }

  @GetMapping(path = "/{name}", produces = MediaType.TEXT_PLAIN_VALUE) (4)
  public Mono<String> greeting(@PathVariable String name) {
    return greetingGateway.greeting(Mono.justOrEmpty(name)); (5) (6)
  }

  @GetMapping(path = "/block/{message}", produces = MediaType.TEXT_PLAIN_
VALUE)
  public Mono<String> blockingConsumer(@PathVariable String message) {
    return Mono.justOrEmpty(greetingGateway.
      blockingGreeting(message)); (5) (6)
  }
}
```

Listing 5.5: Quarkus GreetingResource using EventBus.

```
@Path("/async")
public class GreetingResource {
  private final EventBus bus; (3)

  public GreetingResource(EventBus bus) {
    this.bus = bus;
  }

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  @Path("{name}") (4)
  public Uni<String> greeting(@PathParam("name") String name) {
    return bus.<String>request("greeting", name) (5)
      .map(Message::body); (6)
  }

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  @Path("block/{message}")
  public Uni<String> blockingConsumer(String message) {
    return bus.<String>request("blocking-consumer", message) (5)
      .map(Message::body); (6)
  }
}
```

1. Annotation marking the class as a `MessagingGateway` by Spring Integration.
2. Annotation indicating that Spring Integration provides a proxy for the gateway.
3. Injection of the Gateway in Spring and the EventBus in Quarkus.
4. Maps between a REST API request and a method to publish events.

5. An event message is sent to the address or request channel (e.g., `greeting`).
6. An asynchronous pipeline response is returned to the consumer. The pipeline will produce a response once the message is replied to.

Testing Event Message Handling

Testing ensures that an application is functioning correctly for each use case. Both Spring and Quarkus support testing frameworks based on JUnit 5 and Mockito.

The behavior of a test method should be the same regardless of the technology:

1. Initialize any necessary interactions on mock objects needed by the test fixture.
2. Perform the test on the test fixture.
2. Verify results on the test fixture.
4. Verify expected conditions on mock object interactions.

Mocking Event Bus

In both Spring and Quarkus, tests need to show that when making an HTTP request, the REST layer of the application successfully passes a message through the bus to the message listener, and that the message listener responds with an appropriate response. The tests are implemented differently in Spring and Quarkus due to the underlying implementation details of Spring Integration versus the Quarkus event bus.

Listing 5.6 shows the test for consuming the events from the `greeting` channel in the Spring controller class. Listing 5.7 shows the test for receiving the events from the `greeting` address in the Quarkus resource class.

All the tests for all consuming events in all examples can be found in the examples GitHub repository [5.7].

Listing 5.6: Spring GreetingControllerTests using SpyBean.

```
@SpringBootTest
@AutoConfigureWebTestClient
public class GreetingControllerTests {
  @Autowired
  WebTestClient webTestClient;

  @SpyBean
  GreetingService greetingService; (1)

  @Test
  public void async() {
    webTestClient (3)
      .get()
      .uri("/async/hi")
      .exchange()
      .expectStatus().isOk()
      .expectHeader().contentTypeCompatibleWith(MediaType.TEXT_PLAIN)
      .expectBody(String.class).isEqualTo("HI");

    Mockito.verify(greetingService).consume(Mockito.argThat
      (mono -> "hi".equals(mono.block(Duration.ofSeconds(5))))); (4)
    Mockito.verifyNoMoreInteractions(this.greetingService); (5)
  }
}
```

Listing 5.7: Quarkus GreetingResourceTest using InjectMock.

```
@QuarkusTest
public class GreetingResourceTest {
  @InjectMock
  GreetingService greetingService; (1)

  @Test
  public void testGreeting() {
    Mockito.when(greetingService.consume(Mockito.eq("events")))
      .thenReturn("EVENTS"); (2)

    given() (3)
      .when().get("/async/events")
      .then()
        .statusCode(200)
        .contentType(ContentType.TEXT)
        .body(is("EVENTS"));

    Mockito.verify(greetingService).consume(Mockito.eq("events")); (4)
    Mockito.verifyNoMoreInteractions(this.greetingService); (5)
  }
}
```

1. Inject the required `GreetingService` bean.
   - In Spring, the bean is a spy because Spring Integration needs a real bean in order for the gateway to start properly. Spying on the real bean allows the test to watch and verify interactions on the bean.
   - In Quarkus, the bean is a mock where the test can create its own interactions.
2. Create an interaction on the `GreetingService`. Needed only in Quarkus. In Spring, the real `GreetingService` bean will handle the request.
3. Perform an HTTP request and verify the results.
4. Verify that the expected method on the `GreetingService` was called once with the expected arguments.
5. Verify that no other methods on the `GreetingService` were called.

Reactive Messaging and Streams

To utilize reactive messaging streams that can exchange massive amounts of data between different applications at scale, Spring and Quarkus developers need to implement stream processing applications based on Apache Kafka. Figure 5.3 illustrates a reactive streams topology that shows two messaging components using Kafka. The first component generates random prices that are stored in a Kafka topic. The second component consumes these prices from the Kafka topic, applying a conversion algorithm before the converted price results are sent to an in-memory stream. The in-memory stream is then consumed by a browser client using a server-sent event stream.

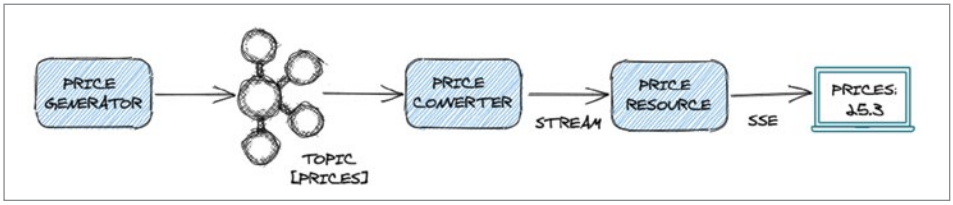

Figure 5.3: Kafka with reactive streams.

Reactive Messaging and Streams in Action

Spring Cloud Stream [5.10] is a framework for creating event-driven microservices on top of Spring Boot and Spring Integration. Spring Cloud Stream supports an Apache

Kafka binder for binding between the Kafka cluster and the application code (e.g., producer, consumer) provided by the developer.

Quarkus provides a CDI extension based on the Eclipse MicroProfile Reactive Messaging specification [5.11] to build event-driven microservices and data streaming applications. This extension also supports a Kafka connector based on the Vert.x Kafka Client to consume messages from and write messages to Kafka.

The complete examples throughout the remainder of this section can be found in the examples GitHub repository [5.7]:

- **Spring Reactive Streams:** Inside the `chapter-5/chapter-5-spring-kafka-streams` project.
- **Quarkus Reactive Streams:** Inside the `chapter-5/chapter-5-quarkus-kafka-streams` project.

Kafka Cluster Setup

There are several ways to set up a Kafka cluster:

- Run a local installation [5.12].
- Run Kafka in containers.
- Consume a managed Kafka service.

Listing 5.8 shows the second method, using Docker Compose [5.13] to spin up a Kafka cluster easily. Quarkus Dev Services for Kafka [5.14] can automatically start a Kafka container when running in Dev Mode as well as when running tests. Quarkus developers don't have to run a Kafka broker manually using Docker Compose. Starting up a Kafka cluster manually is needed only in Spring.

Listing 5.8: Docker Compose configuration.

```
version: '2'

services:
  zookeeper:
    image: strimzi/kafka:0.20.1-kafka-2.6.0
    command: [
      "sh", "-c",
      "bin/zookeeper-server-start.sh config/zookeeper.properties"
    ]
    ports:
      - "2181:2181"
    environment:
      LOG_DIR: /tmp/logs

  kafka:
    image: strimzi/kafka:0.20.1-kafka-2.5.0
    command: [
      "sh", "-c",
      "bin/kafka-server-start.sh config/server.properties
         --override listeners=$${KAFKA_LISTENERS}
         --override advertised.listeners=$${KAFKA_ADVERTISED_LISTENERS}
         --override zookeeper.connect=$${KAFKA_ZOOKEEPER_CONNECT}"
    ]
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      LOG_DIR: "/tmp/logs"
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

Note: This is a development cluster, not ready to be used in production. The example doesn't include critical Kubernetes manifests such as replicas, load balancing, or deployments.

When developers run the Kafka cluster using Docker Compose, they need to use the `kafka-topics` [5.15] command-line tool that should be installed in the local development environment. Listing 5.9 shows how to create a topic and list it using the `kafka-topics` tool.

Listing 5.9: Create Kafka topic.

```
$ kafka-topics --bootstrap-server localhost:9092 --create --partitions 4
  --replication-factor 1 --topic prices
$ kafka-topics --bootstrap-server localhost:9092 --list
```

Note: The example projects include a `create-topic.sh` script utilizing this tool to create the necessary topics.

Application Kafka Configuration

Listings 5.10 and 5.11 show the Kafka configuration, located in `src/main/resources/application.properties`, for both Spring and Quarkus. The configuration shows how to map logical channel names to Kafka topics and how to configure serialization and deserialization.

Listing 5.10: Spring Kafka configuration.

```
spring.cloud.stream.function.definition=generateprice;priceconverter (1)

# Configure the Kafka sink (2)
spring.cloud.stream.bindings.generateprice-out-0.destination=prices
spring.cloud.stream.bindings.generateprice-out-0.producer.
  use-native-encoding=true
spring.cloud.stream.kafka.bindings.generateprice-out-0.producer.configuration.
  value.serializer=org.apache.kafka.common.serialization.IntegerSerializer

# Configure the Kafka source (3)
spring.cloud.stream.bindings.priceconverter-in-0.destination=prices
spring.cloud.stream.bindings.priceconverter-in-0.consumer.
  use-native-decoding=true
spring.cloud.stream.bindings.priceconverter-in-0.group=priceConsumer
spring.cloud.stream.kafka.bindings.priceconverter-in-0.consumer.
  configuration.value.deserializer=org.apache.kafka.common.serialization.
  IntegerDeserializer
```

Listing 5.11: Quarkus Kafka configuration.

```
# Configure the Kafka sink (2)
mp.messaging.outgoing.generated-price.connector=smallrye-kafka
mp.messaging.outgoing.generated-price.topic=prices
mp.messaging.outgoing.generated-price.value.serializer=org.apache.kafka.
   common.serialization.IntegerSerializer

# Configure the Kafka source (3)
mp.messaging.incoming.prices.connector=smallrye-kafka
mp.messaging.incoming.prices.health-readiness-enabled=false
mp.messaging.incoming.prices.topic=prices
mp.messaging.incoming.prices.value.deserializer=org.apache.kafka.common.
   serialization.IntegerDeserializer
```

1. Defines multiple functions (`generateprice` and `priceconverter`) for Kafka sink and source. Needed only in Spring Cloud Stream.
2. Configures how to write messages to Kafka with proper serialization.
3. Configures how to read messages from Kafka with proper deserialization.

Price Generator Design

Listing 5.12 shows how developers can build a bean (e.g., "Price Generator" in Figure 5.3) to produce random prices every five seconds using Spring Cloud Stream, Spring WebFlux, and Reactive Streams Publisher. Listing 5.13 shows how Quarkus enables developers to implement the same functionality using MicroProfile Reactive Messaging.

Listing 5.12: Spring PriceGenerator class using Reactive Streams Publisher.

```
@Component("generateprice")
public class PriceGenerator implements Supplier<Flux<Integer>> { (1)
  private final Random random = new Random();

  @Override
  public Flux<Integer> get() { (2)
    return Flux.interval(Duration.ofSeconds(5))
      .onBackpressureDrop()
      .map(tick -> this.random.nextInt(100));
  }
}
```

Listing 5.13: Quarkus PriceGenerator class using MicroProfile Reactive Messaging.

```
@ApplicationScoped
public class PriceGenerator {
  private final Random random = new Random();

  @Outgoing("generated-price") (1)
  public Multi<Integer> generate() { (2)
    return Multi.createFrom().ticks().every(Duration.ofSeconds(5))
      .onOverflow().drop()
      .map(tick -> this.random.nextInt(100));
  }
}
```

1. Instructs Reactive Messaging to dispatch the items from the returned stream to the `generated-price channel`.
2. The method returns a stream (e.g., `Flux` in Spring and `Multi` in Quarkus) emitting a random price every five seconds.

The main difference between Spring and Quarkus is that the `@Outgoing` annotation in Quarkus allows developers to instruct reactive stream messages to dispatch directly to the Kafka topic (e.g., `prices`) without having to configure the stream function definition as in the Spring application. The method return type is specified as a multivalue stream using the Mutiny reactive library [5.16].

Price Converter Design

The "Price Converter" in Figure 5.3 needs to consume the generated random prices from the `prices` Kafka topic. Then, after performing some business logic, every result should be sent to the `my-data-stream` in-memory stream. To design these functionalities, Listings 5.14 and 5.15 show a Spring `PriceConverter` class using Reactive Streams Consumer and custom-built `InMemoryChannel` to consume the random price and convert it with a specific conversion rate.

Listing 5.16 shows a Quarkus `PriceConverter` class using MicroProfile Reactive Messaging for implementing the same functionality.

Spring on its own doesn't have the concept of an in-memory channel like Quarkus does, so a custom implementation needed to be built.

Listing 5.14: Spring PriceConverter class using Reactive Streams Consumer.

```
@Component(“priceconverter”)
public class PriceConverter implements Consumer<Flux<Integer>> {
  static final double CONVERSION_RATE = 0.88;
  private final InMemoryChannel<Double> inMemoryChannel;

  public PriceConverter(InMemoryChannel<Double> inMemoryChannel) {
    this.inMemoryChannel = inMemoryChannel;
  }

  @Override
  public void accept(Flux<Integer> priceInUsd) {
    priceInUsd
      .map(price -> price * CONVERSION_RATE) (4)
      .subscribe(inMemoryChannel::emitValue); (2)(3)
  }
}
```

Listing 5.15: Spring InMemoryChannel implementation.

```
public class InMemoryChannel<T> {
  private final Sinks.Many<T> sink;

  public InMemoryChannel(Sinks.Many<T> sink) {
    this.sink = sink;
  }

  public InMemoryChannel() {
    this(Sinks.many().multicast().onBackpressureBuffer(1, false));
  }

  public Publisher<T> getPublisher() {
    return sink.asFlux();
  }

  public EmitResult emitValue(T value) { (2)(3)
    return sink.tryEmitNext(value);
  }

  public EmitResult emitComplete() {
    return sink.tryEmitComplete();
  }

  public EmitResult emitError(Throwable error) {
    return sink.tryEmitError(error);
  }
}
```

Listing 5.16: Quarkus PriceConverter class using MicroProfile Reactive Messaging.

```
@ApplicationScoped
public class PriceConverter {
  static final double CONVERSION_RATE = 0.88;

  @Incoming(“prices”) (1)
  @Outgoing(“my-data-stream”) (2)
  @Broadcast (3)
  public double process(int priceInUsd) {
    return priceInUsd * CONVERSION_RATE; (4)
  }
}
```

1. Indicates that the method consumes the items from the `prices` topic.
2. Indicates that the objects returned by the method are sent to the in-memory stream (e.g., `my-data-stream` in Quarkus, `InMemoryChannel` in Spring).
3. Indicates that the item is dispatched to all subscribers.
4. Converts the generated random prices with the `CONVERSION_RATE`.

Price Resource Implementation

The code for binding the message stream to a REST API resource (e.g., `/prices/stream`) is almost identical in Spring and Quarkus, except for handling the in-memory channel. As in the "Price Converter" shown earlier, Spring continues to use a custom-built in-memory channel, but Quarkus provides an `@Channel` qualifier from the MicroProfile Reactive Message specification.

Listing 5.17 and 5.18 show how developers can access the REST API to read a server-sent event message stream from the in-memory channel in both Spring and Quarkus.

Listing 5.17: Spring PriceController class using InMemoryChannel service.

```
@RestController
@RequestMapping("/prices")
public class PriceController {
  private final InMemoryChannel<Double> inMemoryChannel;

  public PriceController(InMemoryChannel<Double> inMemoryChannel) { (1)
    this.inMemoryChannel = inMemoryChannel;
  }

  @GetMapping(path = "/stream", produces =
    MediaType.TEXT_EVENT_STREAM_VALUE) (2)
  public Flux<ServerSentEvent<Double>> stream() { (3)
    return Flux.from(inMemoryChannel.getPublisher())
      .map(price -> ServerSentEvent.builder(price).build());
  }
}
```

Listing 5.18: Quarkus PriceResource class using MicroProfile Reactive Messaging Channel.

```
@Path("/prices")
public class PriceResource {
  private final Publisher<Double> prices;

  public PriceResource(@Channel("my-data-stream") Publisher<Double>
    prices) { (1)
    this.prices = prices;
  }

  @GET
  @Path("/stream")
  @Produces(MediaType.SERVER_SENT_EVENTS) (2)
  public Publisher<Double> stream() { (3)
    return prices;
  }
}
```

1. Injection of an in-memory channel by a custom service (e.g, `InMemoryChannel`) in Spring and specifying the built-in `@Channel` qualifier to use the `my-data-stream` channel in Quarkus.
2. Indicates that the content is sent using `ServerSentEvent`.
3. Returns the reactive stream.

In summary, Table 5.2 lists the differences between Quarkus and Spring Boot in the implementation of reactive streams with Kafka.

Table 5.2: Common REST method return values.

Kafka integrations	Quarkus	Spring
Dependencies	`smallrye-reactive-messaging-kafka`	`spring-cloud-starter-stream-kafka`
Producers (Kafka Sink)	`@Outgoing` annotation	Spring Cloud Stream Function and Binding (e.g., `generateprice`)
Consumers (Kafka Source)	`@Incoming` annotation	Spring Cloud Stream Function and Binding (e.g., `priceconverter`)
Channels	`@Channel` qualifier	Custom in-memory channel (e.g., `InMemoryChannel`)
Serialization and deserialization	MicroProfile configurations (e.g., `mp.messaging.outgoing.generated-price.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer` `mp.messaging.incoming.prices.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer`)	Spring Cloud Stream configurations (e.g., `spring.cloud.stream.kafka.bindings.generateprice-out-0.producer.configuration.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer` `spring.cloud.stream.kafka.bindings.priceconverter-in-0.consumer.configuration.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer`)

Reactive Messaging and Streams Testing

The test patterns used for testing event message handling also apply to reactive stream testing. Testing is done to ensure that all use cases within an application are functioning correctly. Both Spring and Quarkus support testing frameworks based on JUnit 5, Mockito, and Testcontainers [5.17].

The behavior of a test method should be the same regardless of the technology:

1. Start the Kafka cluster using Testcontainers in Spring Boot. In Quarkus, Dev Services starts the Kafka cluster automatically.
2. Initialize any necessary interactions on mock objects needed by the test fixture.
3. Perform the test on the test fixture.
4. Verify results on the test fixture.
5. Verify expected conditions on mock object interactions.

Test Kafka Cluster Setup

Testcontainers provides a Docker Compose module [5.18] to run multiple services specified in the `docker-compose.yaml` file. This module also allows developers to test Spring applications locally with the Kafka and Zookeeper versions, configurations, and environments specified by the same file they would use when running `docker compose up`. In Quarkus, no custom class is required since the Kafka cluster is bootstrapped automatically by Quarkus Dev Services.

Listing 5.19: Spring DockerComposeBase implementation.

```
public class DockerComposeBase { (1)
  static final DockerComposeContainer<?> DOCKER_COMPOSE =
    new DockerComposeContainer<>(new File("docker-compose.yaml"))
      .withExposedService("zookeeper", 1, 2181, Wait.forListeningPort())
      .withExposedService("kafka", 1, 9092, Wait.forListeningPort()); (2)

  static {
    DOCKER_COMPOSE.start(); (3)
  }
}
```

1. Implements a custom base class that knows how to bootstrap the docker-compose engine.
2. Creates a `DockerComposeContainer` using the `docker-compose.yaml` file in each project's root directory.
3. Starts the docker-compose process.

Testing the Prices Event Stream

Listing 5.20 shows a Spring test case for receiving the `prices` event stream from the in-memory channel where the stream data is published in the Kafka topic. Listing 5.21 shows a Quarkus test case for receiving the server-sent events from the local Kafka broker. Both Spring and Quarkus use a web client to access the REST API (e.g., `/prices/stream`) to receive the event stream data.

All the tests for all reactive streams in all examples can be found in the examples GitHub repository [5.7].

Listing 5.20: Spring PriceControllerTests using DockerComposeBase.

```
@SpringBootTest
@AutoConfigureWebTestClient
class PriceControllerTests extends DockerComposeBase { (1)
  @Autowired
  WebTestClient webTestClient; (3)

  @MockBean
  InMemoryChannel<Double> inMemoryChannel; (2)

  @Test
  public void stream() {
    Mockito.when(inMemoryChannel.getPublisher())
      .thenReturn(Flux.just(1.1, 2.2).delayElements
        (Duration.ofSeconds(1))); (2)

    webTestClient.get() (3)
      .uri("/prices/stream") (4)
      .exchange()
      .expectStatus().isOk()
      .expectHeader().contentTypeCompatibleWith(MediaType.TEXT_EVENT_STREAM)
      .expectBody(String.class).isEqualTo("data:1.1\n\ndata:2.2\n\n"); (5) (6)

    Mockito.verify(this.inMemoryChannel).getPublisher(); (6)
    Mockito.verifyNoMoreInteractions(this.inMemoryChannel);
  }
}
```

Listing 5.21: Quarkus PriceResourceTest.

```
@QuarkusTest
class PriceResourceTest {
  @TestHTTPEndpoint(PriceResource.class)
  @TestHTTPResource("/stream") (4)
  URI uri;

  @Test
  public void sseEventStream() {
    List<Double> received = ClientBuilder.newClient() (3)
      .target(uri) (4)
      .request(MediaType.SERVER_SENT_EVENTS)
      .rx(MultiInvoker.class)
      .get(Double.class) (3)
      .select().first(3)
      .subscribe().withSubscriber(AssertSubscriber.create(3))
      .assertSubscribed()
      .awaitItems(3, Duration.ofSeconds(20))
      .assertCompleted()
      .getItems(); (5)

    assertThat(received) (6)
      .hasSize(3) (6)
      .allMatch(value -> (value >= 0) && (value < 100)); (6)
  }
}
```

1. Utilizes the test Kafka cluster setup in Spring Boot. In Quarkus, the Kafka cluster will be automatically bootstrapped.
2. Adds a mock to use the `InMemoryChannel` in Spring. Quarkus uses a Kafka broker directly to consume the reactive streams.
3. Creates a web client to access the in-memory channel in Spring or a consumer using server-sent events in Quarkus.
4. Sets the target endpoint (e.g., `/prices/stream`).
5. Receives the reactive streams from the publisher.
6. Verifies the expected interactions.

Knative Events Binding

As part of increasingly popular serverless architectures, developers need to decouple relationships between event producers and consumers. Knative [5.19] is an open source project enabling developers to deploy, run, and manage serverless applications on Kubernetes. Knative is composed of three primary components: Build, Serving, and Eventing.

Knative Eventing is consistent with the CloudEvents [5.20] specification, providing common formats for creating, parsing, sending, and receiving events in any programming language. Knative Eventing also enables developers to late-bind event sources and event consumers. Listing 5.22 shows an example of what a `CloudEvent` looks like in JSON format.

Listing 5.22: CloudEvent example in JSON format.

```
{
  "specversion" : "1.0", (1)
  "id" : "342342-88ddd03432", (2)
  "source" : "http://localhost:8080/cloudevents", (3)
  "type" : "knative-events-binding", (4)
  "subject" : "cloudevents", (5)
  "time" : "2021-05-25T09:00:00Z", (6)
  "datacontenttype" : "application/json", (7)
  "data" : "{\"message\": \"Knative Events\"}", (8)
}
```

1. Indicates which version of the Cloud Event specification to use.
2. ID field for a specific event. Combining the `id` and the `source` provides a unique identifier.
3. URI to identify the event source in terms of the context in which an event happened or the application that emitted that specific event.
4. Indicates the type of the event with an arbitrary chosen string.
5. (Optional) Additional details about the event.
6. (Optional) Indicates the creation time of the event.
7. (Optional) Defines the content type of the data attribute
8. Contains the business data for the specific event.

Knative Events Source and Sink Patterns

We have seen that asynchronous, decoupled applications can operate without the knowledge of the corresponding producers or consumers. For example, a developer can publish events without considering who or when they are consumed. In Knative, a Sink Binding [5.21] decouples the destination information in the serverless application when the event is sent to a broker (e.g., Kafka, InMemoryChannel) over the HTTP POST method.

Furthermore, a developer can consume the `CloudEvents` without the publisher's configuration in the serverless application because the broker's trigger already subscribes to the events. More information on how to install Knative components on a Kubernetes cluster is available [5.22].

There are three primary use patterns for implementing event sources and sinks with `CloudEvents`:

- **Source to Sink**: A single sink controls the received event sources without filtering, queueing, or backpressure. In this pattern, the source doesn't expect a response from the sink. This pattern is also known as the "fire-and-forget" messaging pattern, as shown in Figure 5.4.

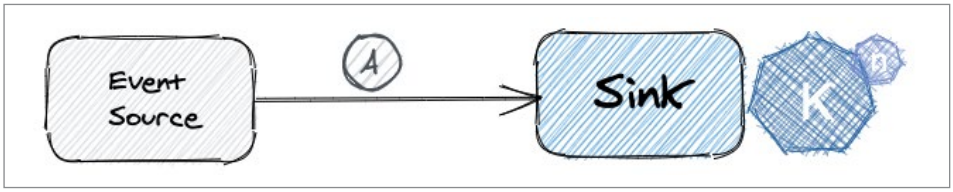

Figure 5.4: Source to Sink pattern.

- **Channel and Subscription**: A channel receives events from multiple event sources or producers. Then the events are sent to several backend services, such as Knative services and sinks, by subscriptions, as shown in Figure 5.5.

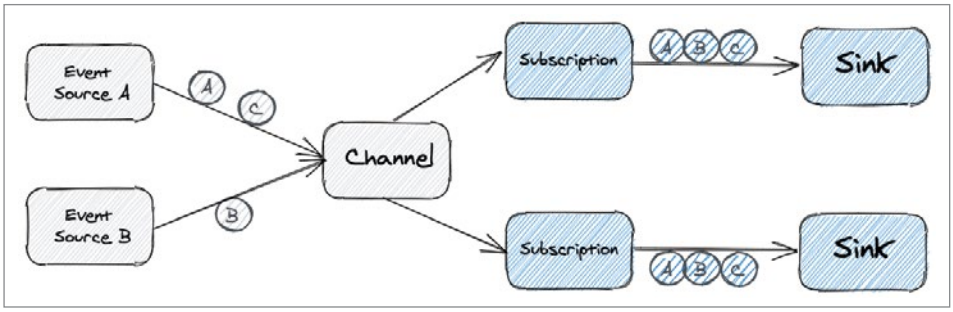

Figure 5.5: Channel and Subscription pattern.

- **Broker and Trigger**: Multiple event sources send events to a broker over HTTP POST requests. Then the events are filtered by CloudEvent triggers to be sent to an event sink, as shown in Figure 5.6.

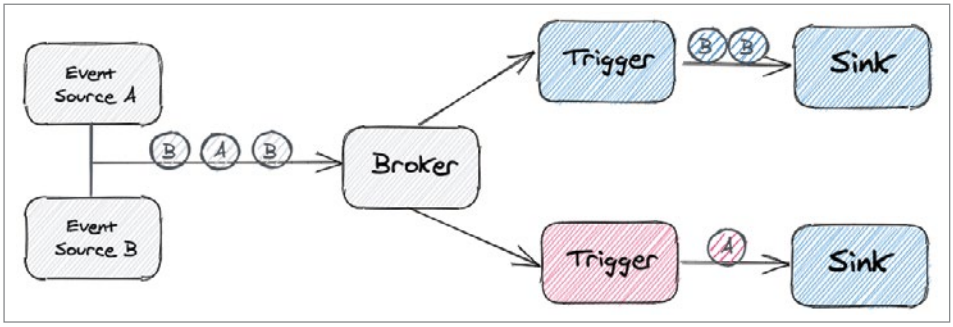

Figure 5.6: Broker and Trigger pattern.

Knative Events Binding in Action

One main difference between Spring and Quarkus is that Quarkus provides the Funqy extension [5.23] to write deployable functions based on portable Java APIs across various function-as-a-service (FaaS) environments, such as AWS Lambda, Azure Functions, Knative, and Knative Events (Cloud Events), as well as a stand-alone service. Spring has similar abstractions with Spring Cloud Function [5.24], but vendor-specific APIs need to be implemented before deploying to any particular cloud provider.

The Funqy API is designed to be a simple abstraction that can be distributed to multiple different cloud function providers and protocols. To use the Funqy API, developers must annotate a method with `@Funq`. The method can have input parameters and return a response like normal HTTP methods for a RESTful service. Only one function application based on the Quarkus Funqy extension can be exposed at once while testing locally or deploying to a Kubernetes cluster.

Listing 5.23 shows the Quarkus application property exposing a specific function.

Listing 5.23: Quarkus configurations.

```
quarkus.funqy.export=uppercase
```

Listing 5.24 shows how a Spring application responds to the input events with the output event to convert lowercase to uppercase using the Knative Eventing Broker pattern and Spring Cloud Function. Listing 5.25 shows how developers can process the `CloudEvent` using the `@Funq` annotation in cloud-native microservices with the `quarkus-funqy-knative-events` extension.

Quarkus Funqy also supports the SmallRye Mutiny `Uni` reactive type as a return type.

The complete examples throughout the remainder of this section can be found in the examples GitHub repository [5.7]:

- **Spring Cloud Events:** Inside the `chapter-5/chapter-5-spring-cloud-events` project.
- **Quarkus Cloud Events:** Inside the `chapter-5/chapter-5-quarkus-cloud-events` project.

Listing 5.24: Spring Cloud Function.

```
@Component("uppercase") (1)
public class ToUppercaseFunction implements Function<Message<Input>, Output> {
  private static final Logger LOGGER = LoggerFactory.getLogger
    (ToUppercaseFunction.class);

  @Override
  public Output apply(Message<Input> inputMessage) { (2)
    HttpHeaders httpHeaders = HeaderUtils.fromMessage
      (inputMessage.getHeaders());

    LOGGER.info("Input CE Id: {}", httpHeaders.getFirst(ID));
    LOGGER.info("Input CE Spec Version: {}", httpHeaders.getFirst(SPECVERSION));
    LOGGER.info("Input CE Source: {}", httpHeaders.getFirst(SOURCE));
    LOGGER.info("Input CE Subject: {}", httpHeaders.getFirst(SUBJECT));

    Input input = inputMessage.getPayload();
    LOGGER.info("Input: {}", input);

    String outputStr = Optional.ofNullable(input.getInput())
      .map(String::toUpperCase) (3)
      .orElse("NO DATA");

    LOGGER.info("Output CE: {}", outputStr);

    return new Output(inputStr, httpHeaders.getFirst(SUBJECT), outputStr, null); (4)
  }
}
```

Listing 5.25: Quarkus Funqy CloudEvent.

```
public class ToUppercaseFunction {
  private static final Logger LOGGER = LoggerFactory.getLogger
    (ToUppercaseFunction.class);

  @Funq("uppercase") (1)
  public Output function(Input input, @Context CloudEvent<Input>
    cloudEvent) { (2)
    LOGGER.info("Input CE Id: {}", cloudEvent.id());
    LOGGER.info("Input CE Spec Version: {}", cloudEvent.specVersion());
    LOGGER.info("Input CE Source: {}", cloudEvent.source());
    LOGGER.info("Input CE Subject: {}", cloudEvent.subject());
    LOGGER.info("Input: {}", input);

    String inputStr = input.getInput();
    String outputStr = Optional.ofNullable(inputStr)
      .map(String::toUpperCase) (3)
      .orElse("NO DATA");

    LOGGER.info("Output CE: {}", outputStr);
    return new Output(inputStr, cloudEvent.subject(), outputStr, null); (4)
  }
}
```

1. Annotation denoting a function name (e.g., `uppercase`). In Quarkus, if a developer doesn't specify the function name explicitly, the function name is equivalent to the method name. In Spring, in the same situation, the function name is the name of the bean if it is exposed as a `@Bean` and the name of the class that implements the Functional interface.

2. Defines a method accepting the input parameters. In Spring, the input object is wrapped in a `Message` object. In Quarkus, the input object can be injected directly. Additional information about the `CloudEvent` can be injected via the `@Context` annotation.
3. Converts the input message to uppercase.
4. Returns the output message.

Knative Events Binding Testing

Listing 5.26 and 5.27 show how developers can verify the `uppercase` function triggered by a `CloudEvent` message in both Spring and Quarkus. There's no significant difference in testing the functionality except for how the `CloudEvent` input message is created and injected into the test case. The tests are created using JUnit Jupiter parameterized tests because the function can potentially return multiple results (e.g., the uppercase of the input if an input exists, or `NO DATA` if no input exists). Each of these cases needs to be tested.

Listing 5.26: Spring ToUppercaseFunctionHttpTests.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class ToUppercaseFunctionHttpTests {
  @Autowired
  TestRestTemplate rest;

  @ParameterizedTest(name = ParameterizedTest.DISPLAY_NAME_PLACEHOLDER
    + "[" + ParameterizedTest.INDEX_PLACEHOLDER + "] (" + Parameterized-
    Test.ARGUMENTS_WITH_NAMES_PLACEHOLDER + ")")
  @MethodSource("toUppercaseFunctionArguments") (1)
  public void toUppercase(String inputText, String expectedOutputText) {
    HttpHeaders ceHeaders = new HttpHeaders();
    ceHeaders.add(SPECVERSION, "1.0");
    ceHeaders.add(ID, UUID.randomUUID().toString());
    ceHeaders.add(TYPE, "com.redhat.faas.springboot.uppercase.test");
    ceHeaders.add(SOURCE, "http://localhost:8080/uppercase");
    ceHeaders.add(SUBJECT, "Convert to UpperCase");
    ceHeaders.add(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_
      VALUE);

    HttpEntity<Input> request = new HttpEntity<>(new Input(inputText),
      ceHeaders); (1)
    ResponseEntity<Output> response = rest.postForEntity("/", request,
      Output.class); (2)

    assertThat(response) (3)
      .isNotNull()
      .extracting(
        ResponseEntity::getStatusCode,
        re -> re.getHeaders().getContentType()
      )
      .containsExactly(
        HttpStatus.OK,
        MediaType.APPLICATION_JSON
      );

    assertThat(response.getBody()) (3)
      .isNotNull()
      .extracting(
        Output::getInput,
        Output::getOperation,
        Output::getOutput,
        Output::getError
      )
```

Quarkus Funqy is not a replacement for REST services over HTTP because Quarkus Funqy doesn't support cache-control and conditional GETs, which are provided by common REST services. Quarkus enables the developers to create portable functions that can be deployed across serverless providers, including Red Hat OpenShift Serverless, AWS Lambda, Azure Functions, and Google Cloud Functions.

All the tests for all examples can be found in the examples GitHub repository [5.7].

```
      .containsExactly(
        inputText,
        “Convert to UpperCase”,
        expectedOutputText,
        null
      );
  }

  static Stream<Arguments> toUppercaseFunctionArguments() { (1)
    return Stream.of(
      Arguments.of(“hello”, “HELLO”),
      Arguments.of(null, “NO DATA”)
    );
  }
}
```

Listing 5.27: Quarkus ToUppercaseFunctionHttpTest.

```
@QuarkusTest
public class ToUppercaseFunctionHttpTest {
  @ParameterizedTest(name = ParameterizedTest.DISPLAY_NAME_PLACEHOLDER
    + “[“ + ParameterizedTest.INDEX_PLACEHOLDER + “] (“ + Parameterized-
    Test.ARGUMENTS_WITH_NAMES_PLACEHOLDER + “)”)
  @MethodSource(“toUppercaseFunctionArguments”) (1)
  public void toUppercase(String inputText, String expectedOutputText) {
    Output output = given()
      .when()
        .contentType(ContentType.JSON)
        .body(new Input(inputText)) (1)
        .header(“ce-specversion”, “1.0”)
        .header(“ce-id”, UUID.randomUUID().toString())
        .header(“ce-type”, “com.redhat.faas.quarkus.uppercase.test”)
        .header(“ce-source”, “http://localhost:8080/uppercase”)
        .header(“ce-subject”, “Convert to UpperCase”)
        .post(“/”) (2)
      .then() (3)
        .statusCode(200)
        .contentType(ContentType.JSON)
        .extract().body().as(Output.class);

    assertThat(output) (3)
      .isNotNull()
      .extracting(
        Output::getInput,
        Output::getOperation,
        Output::getOutput,
        Output::getError
      )
      .containsExactly(
        inputText,
        “Convert to UpperCase”,
        expectedOutputText,
        null
      );
  }

  static Stream<Arguments> toUppercaseFunctionArguments() { (1)
    return Stream.of(
      Arguments.of(“hello”, “HELLO”),
      Arguments.of(null, “NO DATA”)
    );
  }
}
```

1. Specifies the input message to send.
2. Performs the HTTP request.
3. Verifies the expected result.

Deploying Knative Events to Kubernetes

To deploy the previous functions, Knative components should be installed in a Kubernetes cluster.

Several steps are required, from building a container image for the function application to pushing the image to an external container registry (e.g, DockerHub, quay.io), and deploying it to the Kubernetes cluster.

The `func` CLI [5.28] is a new open source client library and command-line interface tool for developing platform-agnostic functions on any Kubernetes cluster.

Using the `func` CLI to deploy to Kubernetes requires creating a deployment configuration file. Listings 5.28 and 5.29 show the `func.yaml` for both the Spring and Quarkus projects. This file is located in the root directory of each project.

Note: Knative installation is beyond the scope of this book. Comprehensive Knative tutorials exist to help you set up Knative on various Kubernetes environments such as Kubernetes in Docker (Kind) [5.25], Minikube [5.26], and Red Hat OpenShift [5.27].

Listing 5.28: Spring function YAML specification.

```
name: chapter-5-spring-cloud-events (1)
namespace: "" (2)
runtime: springboot (3)
image: "" (4)
imageDigest: "" (5)
trigger: events (6)
builder: quay.io/boson/faas-springboot-builder (7)
builderMap: (8)
  default: quay.io/boson/faas-springboot-builder
env: {} (9)
annotations: {}
```

Note: Install the `func` tool [5.29] on your OS (e.g. Linux, macOS, and Windows). Then make sure to log in to a Kubernetes cluster where you need to deploy the function.

Listing 5.29: Quarkus function YAML specification.

```
name: chapter-5-quarkus-cloud-events (1)
namespace: "" (2)
runtime: quarkus (3)
image: "" (4)
imageDigest: "" (5)
trigger: events (6)
builder: quay.io/boson/faas-quarkus-jvm-builder (7)
builderMap: (8)
  default: quay.io/boson/faas-quarkus-jvm-builder (a)
  jvm: quay.io/boson/faas-quarkus-jvm-builder (b)
  native: quay.io/boson/faas-quarkus-native-builder (c)
env: {} (9)
annotations: {}
```

1. Indicates a deployment name for the function in Kubernetes.
2. Indicates a namespace in Kubernetes for the function deployment. When a developer uses the "`-n`" option in the `func` CLI, this value will be replaced with the value of the `namespace` in `func.yaml` automatically.
3. Indicates the runtime (e.g., `springboot`, `quarkus`).
4. Indicates a full image name (e.g., `quay.io/myuser`) to push the container image. The full image name is automatically determined based on the local directory

name. If not provided, the registry will be taken from `func.yaml` or from the `FUNC_REGISTRY` environment variable. When a developer uses the "`-r`" option in the `func` CLI, this value will be updated in the `func.yaml` automatically.

5. Indicates an `imageDigest`, an immutable identifier for a specific image, that is updated automatically after a container image is built on the function application.
6. Indicates the type of the function trigger (e.g., `http` or `cloudevent`).
7. Specifies the builder image for the application build.
8. Indicates where to get the builder image for the runtime.
   a. Default builder image (JVM builder image)
   b. JVM builder image to build the application (function) as a JAR file and then run it on Open JDK HotSpot
   c. Native builder image to build a native executable file then run it on GraalVM
9. (Optional) Specifies the environment variables on the Kubernetes Serving service.

Listing 5.30 shows how to run the YAML file. Make sure to run the following command in the same path where the `func.yaml` file exists. Replace *`IMAGE_URI`* and *`NAMESPACE`* with your own values.

Listing 5.30: Deploy Spring or Quarkus function using func CLI.

```
$ func deploy -r IMAGE_URI -n NAMESPACE -v
```

Note: If you use the Docker CLI and a container engine, make sure to use the 20.10.6 or later version for both the client and server.

Listing 5.31 shows how developers can request a Knative Eventing message over HTTP to the Knative broker using the `curl` command line. The resulting message triggers the startup of the function for processing the event. The same command can be used for both the Spring and Quarkus examples.

Listing 5.31: Send a Knative Event.

```
$ curl \
  -X POST -v "http://<Knative_Serving_URL>" \
  -H "content-type: application/json"  \
  -H "ce-specversion: 1.0"  \
  -H "ce-source: curl"  \
  -H "ce-type: knative-events-binding"  \
  -H "Ce-id: 342342-88ddd03432"  \
  -d '{"input":"Knative Events"}'

//Output should be
{"input":"Knative Events","operation":null,"output":"KNATIVE EVENTS",
  "error":null}
```

Enabling Knative Events to Kafka Event Source

Developers can also create a Kafka event source to enable Knative Eventing, whether using Quarkus or Spring Boot. Apache Kafka event sources can be specified by applying a Kubernetes custom resource (CR) that performs a link between an event producer and an event sink. Listing 5.32 shows an example of a `KafkaSource` resource that connects to a serverless function (e.g., a `payment` service).

Listing 5.32: Create KafkaSource for Spring or Quarkus function.

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: kafka-source
spec:
  consumerGroup: knative-group
  bootstrapServers:
    - my-cluster-kafka-bootstrap.kafka:9092
  topics:
    - orders
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: payment
```

For example, suppose you use Red Hat OpenShift Serverless [5.30] as a Kubernetes variant for deploying a Quarkus serverless function with Knative Eventing and an Apache Kafka Source. In that case, the OpenShift Developer console shows the deployment topology in Figure 5.7.

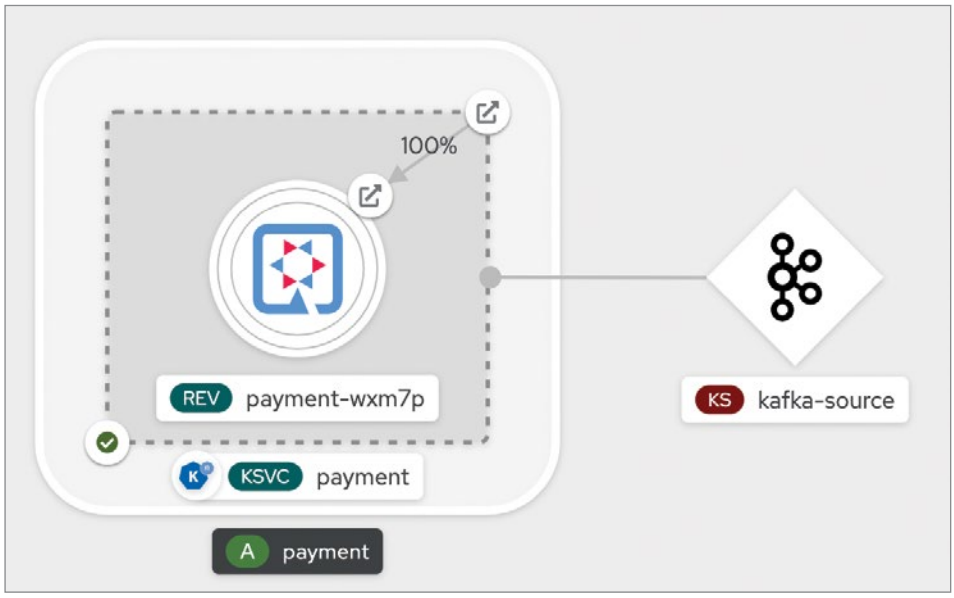

Figure 5.7: Quarkus with Apache Kafka Source.

Summary

This chapter showcased the many differences between Spring and Quarkus in developing event-driven services. We looked at producing and consuming event messages, processing reactive streams with Apache Kafka cluster, and Knative Event binding with serverless functions. Additionally, we showed how Quarkus can simplify the processes used by Spring for implementing the same event processing features and for integrating with Knative, Kubernetes, and Apache Kafka using unified configuration and simple annotations.

References

[5.1] "Apache Kafka": https://www.redhat.com/en/topics/integration/what-is-apache-kafka

[5.2] "Complex event processing": https://en.wikipedia.org/wiki/Complex_event_processing

[5.3] "Event-driven architecture" : https://en.wikipedia.org/wiki/Event-driven_architecture

[5.4] "Spring Integration": https://spring.io/projects/spring-integration

[5.5] "Eclipse Vert.x": https://vertx.io/

[5.6] "Vert.x event bus": https://vertx.io/docs/vertx-core/java/#event_bus

[5.7] "Examples Repository": https://github.com/quarkus-for-spring-developers/examples

[5.8] "MessagingGateway": https://docs.spring.io/spring-integration/reference/html/gateway.html

[5.9] "Quarkus Vertx extension": https://quarkus.io/guides/reactive-event-bus

[5.10] "Spring Cloud Stream": https://spring.io/projects/spring-cloud-stream

[5.11] "MicroProfile Reactive Messaging Specification": https://download.eclipse.org/microprofile/microprofile-reactive-messaging-1.0/microprofile-reactive-messaging-spec.html

[5.12] "Kafka Installation": https://kafka.apache.org/quickstart

[5.13] "Docker Compose": https://github.com/docker/compose

[5.14] "Dev Services for Kafka": https://quarkus.io/guides/kafka-dev-services

[5.15] "kafka-topics tool download": https://kafka.apache.org/downloads

[5.16] "Mutiny Reactive Library": https://smallrye.io/smallrye-mutiny/

[5.17] "Testcontainers": https://www.testcontainers.org

[5.18] "Docker compose module": https://www.testcontainers.org/modules/docker_compose/

[5.19] "What is Knative": https://www.redhat.com/en/topics/microservices/what-is-knative

[5.20] "CloudEvents": https://github.com/cloudevents/spec

[5.21] "Sink binding": https://knative.dev/docs/developer/eventing/sources/sinkbinding/

[5.22] "Installing Knative": https://knative.dev/docs/admin/install/

[5.23] "Quarkus Funqy": https://quarkus.io/guides/funqy

[5.24] "Spring Cloud Function": https://docs.spring.io/spring-cloud-function/docs/current/reference/html/spring-cloud-function.html

[5.25] "Kind (Kubernetes in Docker)": https://kind.sigs.k8s.io/docs/user/quick-start

[5.26] "Installing Minikube": https://v1-18.docs.kubernetes.io/docs/tasks/tools/install-minikube/

[5.27] "What is Red Hat OpenShift": https://www.openshift.com/

[5.28] "func": https://github.com/boson-project/func

[5.29] "Installing func": https://github.com/boson-project/func/blob/main/docs/installing_cli.md

[5.30] "Getting started with OpenShift Serverless": https://docs.openshift.com/container-platform/4.7/serverless/serverless-getting-started.html

Chapter 6

Building Applications for the Cloud

Charles Moulliard

Applications designed today as microservices are often deployed on a cloud platform such as Kubernetes [6.1] or Red Hat OpenShift [6.2], which can orchestrate the deployment and management of your containers. Because of the growing dominance of microservices in the cloud, it is very important to adopt technologies and frameworks that support deployment in the cloud. Microservices development calls for some sophisticated patterns (health checks, fault tolerance, load balancing, remote debugging and development, and distributed tracing) and tools to help you build and deploy your application.

This chapter guides you through the process of building cloud applications and explains the capabilities offered by Spring Boot and Quarkus. The examples developed for this chapter are part of this GitHub repository [6.3].

Prerequisites

The examples in this chapter require a Kubernetes platform and related tools described in Table 6.1.

Table 6.1: Cloud tooling and cluster.

Tool	Description
Kubernetes client	Kubernetes version >= 1.16 [6.4] or OpenShift version >= 4.5 [6.5]
Container Registry account	Have an account on one of the following container registries: docker.io, quay.io, or gcr.io.
Kubernetes cluster (e.g., Minikube, Kind, CodeReady Containers, Red Hat Developer Sandbox)	Have access on your machine or remotely to a Kubernetes or OpenShift cluster.

Preparing Your Application for the Cloud

Preparing an application that will run in a container in the cloud requires numerous steps that aren't required for local deployment. Developers are used to executing a Maven goal or Gradle task to launch the application locally. Running the application in a container requires several tools and concepts, such as a Kubernetes manifest [6.6], that may be new to developers. This section shows you some simple ways to prepare the application for the cloud through the following steps:

- Building a container image
- Creating a manifest that describes the deployment of the pod and its containers
- Deploying and running your application

Building a Container Image

To deploy your application on Kubernetes or Red Hat OpenShift, you have to package your application's binary and resources as a container image and configure it to pass the arguments needed to launch the application [6.8]. We will use tools offered by the Quarkus framework out of the box to make this process as smooth as possible and avoid installing extra tools.

Note: Instead of populating a container image, which consumes a lot of CPU, memory, and time, you can use the `odo` CLI tool [6.7] to push the code from your local machine to the pod's container.

Several key components are needed to run a Java application in a container: an operating system (or base image), the JVM, and the application. Different approaches exist to assemble these into a container image, where each component will become part of a layer. Table 6.2 summarizes the container building strategies that Spring Boot or Quarkus support. This table is not at all exhaustive because additional tools exist, such as kaniko, buildah, and skaffold.

Table 6.2: Building strategies.

Strategy	Build performed	Quarkus	Spring Boot
Google JIB [6.9]	Locally (*)	Using the `container-image-jib` extension [6.10]	Not supported by Spring Boot directly, but you can use the JIB Maven or Gradle plugin instead [6.9]
Buildpacks [6.11]	Locally or Kubernetes (**)	Not yet available, but scheduled in the next release	Spring Boot using `mvn spring-boot:build-image` [6.12] [6.13]
Docker	Locally or Kubernetes/ OpenShift using Docker build	Using the `container-image-docker` extension [6.10]	Done through a `docker build` command
OpenShift Source-to-Image (S2I) [6.14]	OpenShift	Using the `container-image-s2i` extension [6.10]	Not supported

* Locally means that the user can execute the build process on their laptop without using a Kubernetes platform.
** Kubernetes/OpenShift means that a component such as OpenShift S2I, Tekton, or Buildpacks is added to the Kubernetes platform to perform the build.

Later in this chapter, the `chapter-6-quarkus-rest` example builds a Quarkus container image using the JIB tool [6.9], packaged as part of the `quarkus-container-image-jib` extension.

To build and push the image to a container registry, execute:

```
$ ./mvnw clean package \
  -Dquarkus.container-image.build=true \
  -Dquarkus.container-image.push=true
```

Notes:

- If no registry is set as a parameter/property, the `docker.io` registry will be used as the default.
- Use the following properties, described within the Quarkus guide [6.15], to use a different registry, credentials, and image name:
  - `-Dquarkus.container-image.registry` (`docker.io`, `quay.io`, etc.)
  - `-Dquarkus.container-image.username`
  - `-Dquarkus.container-image.password`
  - `-Dquarkus.container-image.image` (Specifies the entire image string, such as `docker.io/<group>/<name>:<version>`)

The process to build an image using Spring Boot is similar, except:

- It supports only the `Buildpacks` strategy.
- The Maven `build-image` goal is needed for building the image.
- A container daemon (Docker or Podman) must be running locally or be accessible from a remote machine.

To build the image for an existing Spring Boot application, where the framework version is >= 2.4, execute:

```
$ ./mvnw spring-boot:build-image
```

The image will be built using the Paketo Spring Boot Buildpacks. This project can package the application, install and configure a JVM, and do additional tasks described in the Paketo documentation [6.12].

The Spring or Quarkus containerized application can be run locally using the Docker or Podman `run` command, specifying the container image that was created and pushed to a container registry:

```
$ docker run -it localhost:5000/quarkus/chapter-6-quarkus-rest:1.0
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (main) chapter-6-quarkus-rest 1.0.0-SNAPSHOT on JVM (powered
  by Quarkus) started in 1.162s. Listening on: http://0.0.0.0:8080
INFO  [io.quarkus] (main) Profile prod activated.
INFO  [io.quarkus] (main) Installed features: [cdi, hibernate-validator,
  kubernetes, resteasy-reactive, resteasy-reactive-jackson,
  smallrye-context-propagation, smallrye-health]
```

Note: The `build-image` goal of the Spring Boot Maven plugin can be configured in various ways [6.16].

Generation of Kubernetes Manifests

A cloud platform like Kubernetes is an orchestration engine that can start and stop containers, scale them, and manage different capabilities such as permissions, CPU and memory resources, volumes, etc. The Kubernetes project defines different resources (aka objects) to configure such features. These resources are called *manifests* [6.17] and allow containers to be deployed using the `Pod` resource [6.18]. A `Pod` represents the smallest deployable unit of compute that Kubernetes manages. A `Pod` can contain one or more containers, like the one that was built in the previous section as a container image.

The manifest resource files can be described using the YAML or JSON format. Resources share the following sections:

- Type of the object or resource (e.g. `Pod`, `Deployment`) and API version (e.g., `v1`, `apps/v1`)
- `metadata`, such as resource name, labels, and annotations
- A `spec`, describing the parameters of the resource
- Status and events

Listing 6.1 illustrates a YAML configuration for a `Pod`.

Listing 6.1: Pod example.

```
apiVersion: v1 (1)
kind: Pod (2)
metadata: (3)
  name: backend-fruit
  namespace: jpa-project
  labels:
    role: my-quarkus-app
spec: (4)
  containers:
    - name: app
      image: quay.io/group/name:1.0
```

1. Version of the API exposed by the resource: `v1` for the `Pod`
2. The type or `kind` of the object or resource
3. The metadata, allowing enrichment and description of the resource
4. The spec, including the fields and parameters of the resource

A Kubernetes API server [6.19] exposes services to perform different operations on the resources, such as create, read, delete, list, and watch. The Kubernetes resources are deployed into a *namespace* [6.20]. Namespaces isolate groups of containers from one another. The command-line tools `kubectl` and the OpenShift `oc` interact with the Kubernetes API server to perform requested actions on deployed resources.

The deployment of an application on Kubernetes requires the creation of several resources. Table 6.3 summarizes some of the most important resource types for deployment and access to an application container.

Table 6.3: Kubernetes manifests.

Type	Description
`Deployment` [6.21]	Manages the application resources volumes, properties, environment variables, replicas, etc., to be deployed.
`Service`	Endpoint provicing access to the application port of the `Pods` hosting an application. The `Service` performs load balancing on incoming requests to `Pods`.
`Namespace`	Virtual space isolating the `Pods`.
`Ingress` (Kubernetes) `Route` (OpenShift)	Manages external access to the `Service`.

The Quarkus Kubernetes extension [6.22] frees a developer from manually creating the required Kubernetes manifests. Under the covers, the Dekorate tool/library [6.23] generates the Kubernetes resources when the application is built. Additionally, the generated resources can be customized using well-defined properties.

The `quarkus-kubernetes` extension has been included as a dependency within the `pom.xml` file of the `chapter-6/chapter-6-quarkus-rest` example project.

To create the manifests, execute simply the following Maven goals:

```
$ ./mvnw clean package
```

The build will generate the required Kubernetes manifests, which can be found inside the local `target` directory, relative to the project's root directory.

```
$ ls -la ./target/kubernetes
total 16
drwxr-xr-x   4 cmoullia  staff   128 May 19 12:24 .
drwxr-xr-x  10 cmoullia  staff   320 May 19 12:24 ..
-rw-r--r--   1 cmoullia  staff  2030 May 19 12:24 kubernetes.json
-rw-r--r--   1 cmoullia  staff  1444 May 19 12:24 kubernetes.yml
```

By default, only the `Deployment` and `Service` resources are generated. The `Deployment` is responsible for telling Kubernetes to create each `Pod` using a `ReplicaSet` resource. The advantage of the `Deployment` over a `Pod` resource is that you will be able to scale up or down the number of running containers.

Note: This type of developer experience doesn't exist in the Spring Boot project unless you manually integrate the Dekorate Spring Boot library part of your project [6.24]. The Dekorate project contains several examples to guide you.

On OpenShift, use the `quarkus-openshift` extension instead of the `quarkus-kubernetes` extension. Be careful not to include both within the `pom.xml` file, which would cause the build to fail.

Using the `quarkus-openshift` extension will create OpenShift-specific resources, such as a `DeploymentConfig` or `Route`. Additionally, you will need to set the `quarkus.container-image.builder=jib` property in order to use JIB to build the container image. The `quarkus-openshift` extension uses the `container-image-s2i` extension out of the box [6.25].

Deployment

The application can be deployed to a Kubernetes cluster once the YAML/JSON manifest file containing the resources is generated. The `Deployment` manifest's `spec` contains a template defining how each `Pod` should be created to run the containerized application (path of the image, port to be exposed, image pull policy, security context, etc.). The `ENTRYPOINT` has been declared within the image, so it doesn't need to be configured within the `Deployment`.

Listing 6.2 shows an example spec.

Listing 6.2: Specification of the container.

```
spec:
  containers:
    - image: quay.io/quarkus/chapter-6-quarkus-rest:1.0
      imagePullPolicy: Always
      name: code-with-quarkus
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Execute the following commands to deploy the application into the `quarkus-demo` namespace:

```
$ kubectl create ns quarkus-demo
$ kubectl apply -f target/kubernetes/kubernetes.yml -n quarkus-demo
```

The application can also be deployed on OpenShift using the following commands:

```
$ oc new-project quarkus-demo
$ oc apply -f target/kubernetes/kubernetes.json
```

You have two options to deploy a Spring Boot application:

- "Dekorate" your application and use the `dekorate-spring-starter`.
- Use the `kubectl create` command as documented in the Spring Boot Kubernetes guide [6.27] to generate the `Deployment` and `Service` resources.

When you create a RESTful web application, you expect to issue a request through a tool such as `curl` or a programming call to the application's HTTP endpoint and get a response. For an application running locally, the request to be used using the `chapter-6-quarkus-rest` example would be:

```
$ ./mvnw quarkus:dev
$ curl http://localhost:8080/fruits
[{"name":"Apple","description":"Winter fruit"},{"name":"Pineapple",
  "description":"Tropical fruit"}]
```

Great! But what should we do to access our endpoint on the cloud platform?

First, check whether the `Pod` has been created and is running. In this case, the `status` value should be `Running` and `1/1` instances should be ready:

```
$ kubectl get pod -n quarkus-demo
NAME                                    READY   STATUS    RESTARTS   AGE
chapter-6-quarkus-rest-5876896cf5-n5dvp   1/1     Running   0          76s
```

Note: The `quarkus-openshift` extension generates two additional files in the directory: `openshift.json` and `openshift.yml`, both of which contain OpenShift-specific resources.

Note: We can rely on the Quarkus extension (Kubernetes or OpenShift) to deploy the resources directly using the following parameter "`-Dquarkus.kubernetes.deploy=true`" without the help of the "`kubectl`" tool.

An OpenShift project is an alternative representation of a Kubernetes namespace. A project has one or more members, with a quota on the resources that the project may consume and security controls on the resources in the project [6.26].

The events of the Pod should report that the image that packages your application has been pulled successfully and the Pod has started (see the Reason column):

```
$ kubectl describe pod -n quarkus-demo
 Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  66s   default-scheduler  Successfully assigned quarkus-demo/
                                              chapter-6-quarkus-rest-5876896cf5-
                                              n5dvp to kind-control-plane
  Normal  Pulling    66s   kubelet            Pulling image "quay.io/cmoulliard/
                                              example:1.0"
  Normal  Pulled     29s   kubelet            Successfully pulled image "quay.io/
                                              cmoulliard/example:1.0" in
                                              37.180625002s
  Normal  Created    28s   kubelet            Created container chapter-6-
                                              quarkus-rest
  Normal  Started    28s   kubelet            Started container chapter-6-
                                              quarkus-rest
```

The Started container message tells you that deployment was successful and that your application is alive. But how do you access the service's HTTP endpoint? Because the cluster runs inside a VM when deployed locally and uses a private IP address, you cannot access it directly from the local machine. Instead, the Pod's traffic needs to be forwarded to the local machine:

```
$ kubectl port-forward -n quarkus-demo service/chapter-6-quarkus-rest 8888:8080
Forwarding from 127.0.0.1:8888 -> 8080
Forwarding from [::1]:8888 -> 8080
```

Now, you can query the HTTP endpoint from the localhost using port 8888 as you would do for a Java application not deployed on Kubernetes or Red Hat OpenShift:

```
$ curl http://localhost:8888/fruits
[{"name":"Apple","description":"Winter fruit"},{"name":"Pineapple",
  "description":"Tropical fruit"}]
```

Although the port-forwarding mechanism is useful during development or as a workaround, it's not a good long-term approach to accessing services in test or production environments. There might be many microservices deployed on the cluster that you need access to. Port-forwarding all of them to different ports on your local machine is not a scalable solution. We'll look at a better approach in the next section.

Routing

As a large-scale solution to getting access to container endpoints, use a proxy server deployed on the Kubernetes cluster. A proxy server forwards traffic from the endpoints exposed on the cluster to the corresponding services backed by their Pod instances. Different proxy controllers can be deployed [6.28] on the cloud platform.

Quarkus can generate the corresponding Ingress Kubernetes resource by using the following two properties in the project's src/main/resources/application.properties file:

```
quarkus.kubernetes.ingress.expose=true (1)
quarkus.kubernetes.ingress.host=chapter-6-quarkus-rest.127.0.0.1.nip.io (2)
```

OpenShift 4.x natively packages an IngressController that works with HAProxy and supports Ingress and Route Kubernetes objects [6.29].

- Boolean property that, when `true`, causes the Quarkus Kubernetes extension to generate the `Ingress` resource.
- The fully-qualified DNS entry to access the endpoint exposed on your cluster. We use the domain name `nip.io` to map the IP address `127.0.0.1` to the domain without the need to edit the `/etc/hosts` file.

To generate the new resource and redeploy the project, execute the following commands:

```
$ ./mvnw clean package
$ kubectl delete -f target/kubernetes/kubernetes.yml -n quarkus-demo
$ kubectl apply -f target/kubernetes/kubernetes.yml -n quarkus-demo
```

Next, open the URL (`http://chapter-6-quarkus-rest.127.0.0.1.nip.io` in the example) in a browser to access the deployed service, as shown in Figure 6.1.

Figure 6.1: Simple Quarkus application.

Health Checks

Distributed applications deployed on a cloud platform can be hard to manage because they are composed of multiple systems. If some of the Pods are not accessible for various reasons (the Pod not starting because its JVM crashed, a service is not reachable, etc.), you need a way to control and be informed of the situation. Such a mechanism, also known as the HealthCheck cloud microservice pattern [6.30], is supported by the Kubernetes Spec [6.31] and has been defined under the MicroProfile Health specification [6.32]. Quarkus is based upon MicroProfile specifications.

To determine whether the application is alive and ready to serve requests, or to expose an endpoint or service, some health controls should be exposed by your application. The kubelet engine will then use them to query your Pod to figure out what it should do. Some of the health checks offered by Kubernetes platform are:

- Liveness
- Readiness
- Startup

These are used for different purposes, such as to restart a Pod if the Liveness probe fails, or to remove the IP address and disable access to the service if the Pod is not ready. The startup probe could be used to deal with legacy applications that require an unusually long time to start up, to avoid a failure during a Liveness probe that would try to restart the Pod.

Note: If you use the `quarkus-openshift` extension, the corresponding OpenShift `Route` resource can be created using the `quarkus.openshift.route.expose` parameter.

Note: Uncomment the two `ingress` properties in the `application.properties` file of the example before regenerating the resources.

Note: Check the file `./target/kubernetes/kubernetes.yml` after issuing the previous commands to verify whether the `Ingress` resource, or the `./target/kubernetes/openshift.yml` file for the OpenShift `Route`, has been generated.

Kubernetes currently supports three types of health probes:

- HTTP GET, to call an HTTP endpoint
- TCP Socket, to execute a TCP socket call on a specific port
- Exec, to execute a command

To add one or more of these probes to your application, follow these guidelines:

- The `Deployment/spec/container` sections covering the `liveness` and `readiness` probes should always be present.
- The application should implement a Health endpoint. If using an HTTP endpoint, a response code in the `2xx` or `3xx` range indicates a healthy application. If using a command, the execution of this command should return a `0` exit code to indicate a healthy application.

The HealthCheck pattern is supported in Quarkus using the SmallRye Health extension [6.33]. SmallRye Health is an implementation of the Eclipse MicroProfile Health specification. Spring Boot applications can use the Kubernetes Probes included in the Spring Boot Actuator starter [6.34] to implement similar functionality.

Spring Boot Actuator only provides the endpoints and doesn't generate the necessary configuration within the required Kubernetes manifests. You will need to create the configuration manually or use a tool such as Dekorate [6.35].

It is time now to play and experiment with the HTTP Liveness and Readiness probes using the `chapter-6/chapter-6-quarkus-rest` example. First, compile the project to create the Kubernetes manifest file:

```
$ ./mvnw clean package
```

You can now open the `./target/kubernetes/kubernetes.yml` file to see that the `container` definition (part of the `spec` section) includes the Liveness and Readiness probes:

```
apiVersion: apps/v1
kind: Deployment
spec:
 template:
   spec:
     containers:
      - name:
        image: quay.io/group/application:1.0
        imagePullPolicy: Always
        livenessProbe: (1)
          failureThreshold: 3
          httpGet: (2)
            path: /q/health/live
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 0
          periodSeconds: 30
          successThreshold: 1
          timeoutSeconds: 10
         readinessProbe: (3)
           httpGet: (4)
            path: /q/health/ready
            port: 8080
            scheme: HTTP
           ...
```

1. Section declaring and defining the `livenessProbe`.
2. Configuration of the probe, issuing an HTTP GET request to the `/q/health/live` path on port `8080` of the container.
3. Section declaring and defining the `readinessProbe`.
4. Configuration of the probe, issuing an HTTP GET request to the `/q/health/ready` path on port `8080` of the container.

Because we have already previously deployed the application and the probes are already there, you can use your browser or a tool such as `curl` or `httpie` [6.36], to call the endpoints. For our example, the Liveness endpoint is:

```
http://chapter-6-quarkus-rest.127.0.0.1.nip.io/q/health/live
```

The Health endpoint is:

```
http://chapter-6-quarkus-rest.127.0.0.1.nip.io/q/health/ready
```

```
$ http chapter-6-quarkus-rest.127.0.0.1.nip.io/q/health/ready
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 136
Content-Type: application/json; charset=UTF-8

{
  “checks”: [
    {
      “checks”: [],
      “status”: “UP”
    }
  ],
  “status”: “UP”
}
```

You can easily customize the Liveness and Readiness endpoints. This can be achieved by adding a Java class implementing the `HealthCheck` interface and adding the `@Liveness` or `@Readiness` annotation. The `HealthCheck` interface has a single method, `call`, that you need to implement.

Listing 6.3 illustrates such an example of customization. You can retrieve the example from the code of the `chapter-6-quarkus-rest` example in the Java file `src/main/java/org/acme/service/SimpleHealthCheck.java`.

Listing 6.3: Quarkus HealthCheck custom example.

```
@Liveness
@ApplicationScoped
public class SimpleHealthCheck implements HealthCheck {
  @Override
  public HealthCheckResponse call() {
    return HealthCheckResponse.up(“Simple health check”);
  }
}
```

Spring Boot also supports the HealthCheck pattern. You can customize it using the `HealthIndicator` interface, as shown in Listing 6.4.

Listing 6.4: Spring Boot HealthCheck custom example.

```
@Component
public class SimpleHealthCheck implements HealthIndicator {
  @Override
  public Health health() {
    return Health.up().withDetail("Simple health check").build();
  }
}
```

If the project has already been deployed on the cloud cluster, open your browser at the URL `http://chapter-6-quarkus-rest.127.0.0.1.nip.io/q/health/live` or call it using `httpie`:

```
http chapter-6-quarkus-rest.127.0.0.1.nip.io/q/health/live
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 136
Content-Type: application/json; charset=UTF-8

{
  "checks": [
    {
      "name": "Simple health check",
      "status": "UP"
    }
  ],
  "status": "UP"
}
```

Several Quarkus extensions, such as gRPC, Mongo, Neo4j, Redis, Vault, and Kafka, have implemented `HealthCheck` endpoints and are enabled by default when they are included.

The health indicators can also provide additional information. Such information could report CPU usage, memory usage, or other business information, as shown in Listing 6.5.

Listing 6.5: Adding additional Information to a Quarkus health probe.

```
@Override
public HealthCheckResponse call() {
  return HealthCheckResponse.named("Simple health check")
    .withData("foo", "fooValue")
    .withData("bar","barValue")
    .up()
    .build();
}
```

You can disable each extension's health check via the `quarkus.health.extensions.enabled` property so none will be automatically registered.

Service Binding

A modern application comprises integrated microservices that access backend systems, such as a database or messaging broker. Applications do not know in advance the address or port of a service to be reached, access parameters such as: username and password, etc. This information needs to be discovered dynamically at runtime.

This process, called service binding, automatically injects the service information from a Kubernetes `Secret` upon container startup. The Kubernetes Service Binding specification [6.37] addresses this process.

The benefit of this specification is that all of the actors, whether an application developer, service provider, or platform supporting the installation of the service, contribute as if in a chain to providing the information required.

The good news for application developers is that such a specification is currently supported by Quarkus and Spring (through the Spring Cloud Service Binding [6.38]). The most common services discoverable by these projects are summarized in Table 6.4. Refer to the Quarkus or Spring Cloud documentation to get the most up-to-date information, including a complete list of supported services.

Table 6.4: Service binding.

Service	Spring Boot	Quarkus
MariaDB	Yes	Yes
MySQL	Yes	Yes
Microsoft SQL Server	Yes	Yes
Oracle	Yes	Yes
Kafka	Yes	Yes
MongoDB	Yes	No
PostgreSQL	Yes	Yes

The way that the different frameworks create objects, such as a JMS connection, data source, etc. and how they discover the data from the filesystem under the `SERVICE_BINDING_ROOT` path are specific to the framework.

Both Spring and Quarkus offer the same process to consume the Service Binding:

1. Select the Spring Boot Starter or Quarkus Extension supporting the service binding and the service to bind to.
2. Create a `Secret` containing the properties to access the service.
3. Mount the `Secret` as a volume within the `Pod` according to the Application projection [6.39].
4. Set the `SERVICE_BINDING_ROOT` environment variable specifying where the service information has been mounted under the local filesystem.

Note: Steps 2 through 4 can be automated by installing the Service Binding Operator [6.40] on the Kubernetes cluster. Cluster admin privileges are required in order to install this operator and its Custom Resources Definitions.

To play with a Quarkus microservice backed by a database, we created the `chapter-6/chapter-6-quarkus-rest-database` project in the examples GitHub repository [6.3].

Now that we have explained the theory, it is time to see in action what you must do to bind a service such as a PostgreSQL database to your Quarkus JPA application. The `chapter-6/chapter-6-quarkus-rest-database` project already contains the `quarkus-kubernetes-service-binding` extension.

Install the PostgreSQL database on the Kubernetes cluster using the image built for this book, as it includes the SQL schema needed to create the `fruits` table used by our example. Execute the following commands from the project's root directory:

```
$ kubectl create ns quarkus-demo-db
$ kubectl apply -n quarkus-demo-db -f ./k8s/postgresql.yaml
```

Because Spring Boot doesn't use Dekorate natively, you will have to manually create the different Kubernetes manifest files to configure your Spring Boot project to use Dekorate.

The Kubernetes `ConfigMap` (part of the example at this path `./k8s/postgresql.yaml` file) includes the following properties to configure the `fruits` database:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-configuration
  labels:
    app: postgresql
data:
```

```
  POSTGRES_DB: fruits
  POSTGRES_USER: fruits
  POSTGRES_PASSWORD: fruits
```

Check that the database's Pod is running before continuing:

```
$ kubectl get pod -l name=postgresql -n quarkus-demo-db
NAME                          READY   STATUS    RESTARTS   AGE
postgresql-f695ccb5c-6dk2b    1/1     Running   0          58s
```

You now need to create a `Secret` containing the datasource properties for the application. This `Secret` will be injected into the application using the Service Binding.

The `fruit-db Secret` that we will install follows the convention defined within the Service Binding Specification [6.37] and includes the following properties:

```
host: postgresql # This is the Kubernetes Service name of the DB
port: 5342
database: fruits # Name of the database
username: fruits
password: fruits
type: postgresql
```

The name of the `Secret` should be unique per service and backend type. This name will be used as the key of the service to be consumed by the Application, and for which we need to bind the properties as declared within the `Secret`:

To create the `Secret`, execute the following command:

```
$ kubectl apply -n quarkus-demo-db -f ./k8s/service-binding-secret.yaml
```

We can now revisit the configuration of the `application.properties` file, located in `src/main/resources`, to see what parameters we will use to configure the Service Binding.

```
quarkus.kubernetes-service-binding.enabled=true (1)
quarkus.kubernetes-service-binding.root=k8s-sb (2)
```

1. Enable the service binding, as it is disabled by default.
2. Specifies the root path to access the service under the filesystem mounted. This root path will allow the Service Binding library to scan the files created for each service (e.g., `<root>/fruits-db`). Each service's folder contains a list of files, where each file corresponds to a parameter (e.g., `<root>/fruits-db/username`) and stores the value as the file's content.

As we did before, to simplify your life, we will use the Quarkus Kubernetes extension to extend the definition of the Pod specification to mount a volume from the information in the `Secret` created previously:

```
quarkus.kubernetes.mounts.service-binding-volume.path=/work/k8s-sb/fruits-db (3)
quarkus.kubernetes.secret-volumes.service-binding-volume.secret-name=fruit-db (4)
```

3. Path of the volume to be mounted within the pod under the volume name `service-binding`.
4. Name of the secret `service-binding-volume` to be searched for and to be linked to the volume.

Note: We need this step only because we are manually binding our application to the database. For production Kubernetes clusters, the Service Binding Operator should be deployed to automate such an installation process.

Note: The current implementation of the Service Binding spec, part of the `quarkus-kubernetes-service-binding` extension, supports only one service type per binding. Therefore, it is not possible to use the Service Binding Spec to bind a microservice to several databases.

To let your application bind to the database, execute the following command:

```
$ ./mvnw clean package
$ kubectl apply -f target/kubernetes/kubernetes.yml -n quarkus-demo-db
```

Open your favorite browser and visit the `Ingress` route of your application. If you use Kind [6.41] to start a local Kubernetes cluster, the URL will be `http://chapter-6-quarkus-rest-database.127.0.0.1.nip.io/`.

Figure 6.2 illustrates the running application.

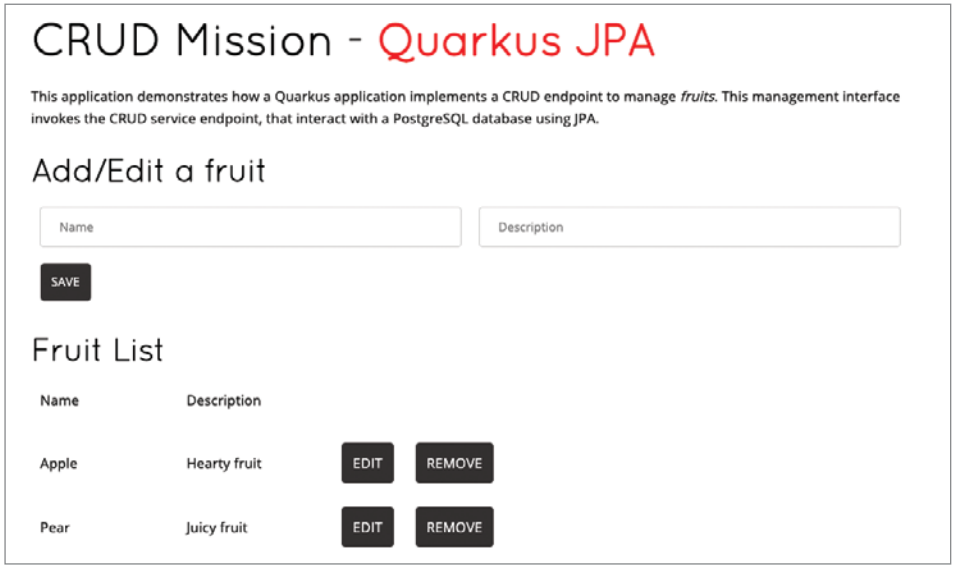

Figure 6.2: Application using the service binding.

Remote Coding and Debugging

When a microservice runs in the cloud, on a machine where you don't have easy access, you need technology for remote live coding and debugging more than ever. It avoids the need to execute repetitive processes locally, which can consume a lot of CPU, memory, and networking resources whenever a new image needs to be redeployed and tested on a cluster. Local testing can also be hampered by differences between your local machine and the runtime environment, such as JDK versions. Remote live coding and debugging alleviates the "works on my machine" problem. Furthermore, your application may need additional services, such as a database or messaging infrastructure, that are difficult or unfeasible to run on a local developer machine.

Such features are supported out of the box by Spring and Quarkus, but they need to be configured differently, as shown in Table 6.5.

Table 6.5: Remote coding and debugging.

Service	Spring Boot	Quarkus
Remote LiveCoding	Use `mvn spring-boot:run` and Spring Boot DevTools [6.42] with a socket opened to access the `Pod` remotely.	Use `mvn quarkus:remote-dev` remotely and the application's URL to push the changes to the `Pod`. See the example for details on use and configuration.
Remote debugging	Build the image using `BP_DEBUG_ENABLED=true` parameter. Enable it using a Kubernetes environment variable, `BPL_DEBUG_ENABLED=true`. Port defined by default is `8000`. Socket should be opened for remote access to the `Pod` using, for example, a Kubernetes `Service` configured as `NodePort`.	No need to define a specific parameter to build the container image. Enable it using a Kubernetes environment variable, `JAVA_TOOL_OPTIONS=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=0.0.0.0:5005` Socket should be opened for remote access to the `Pod` using, for example, a Kubernetes `Service` configured as `NodePort`.

Remote Live Coding

The example `chapter-6/chapter-6-quarkus-rest-debug` example has been designed to showcase how to do remote live coding and remote debugging. Feel free to use it to experiment with the steps that we will describe later.

The remote live coding feature of Quarkus, covered by this excellent developer article [6.43], requires some modifications to the example:

- Change the package type to embed additional JAR libraries in the container image.
- Configure the Live Reload feature so you can push the changes to the remote application.
- Pass a parameter to the JVM in the running container when Quarkus starts to enable remote development mode.

We can now review the `src/main/resources/application.properties` file to see what to change:

```
quarkus.kubernetes.env.vars.quarkus-launch-devmode=true (1)
quarkus.package.type=mutable-jar (2)
quarkus.live-reload.password=changeit (3)
quarkus.live-reload.url=http://chapter-6-quarkus-rest-debug.127.0.0.1.nip.io/ (4)
```

1. Environment variable used to enable and launch the remote application in development mode.
2. Package type that includes the deployment time parts of Quarkus to facilitate the remote Live Reload capabilities.
3. Password used between the client and server to secure the communication between the developer's machine and the remote application.
4. URL of the Kubernetes `Ingress` or OpenShift `Route` resource. Replace the value shown with the actual URL according to your platform where the application is deployed.

Build the container image and deploy it on the Kubernetes cluster:

```
$ ./mvnw clean package
$ kubectl create ns quarkus-demo-debug
$ kubectl apply -f target/kubernetes/kubernetes.yml -n quarkus-demo-debug
```

Once the application is deployed and running on the Kubernetes cluster, launch your Quarkus application locally:

```
$ ./mvnw quarkus:remote-dev
```

Next, open the URL of the `hello` endpoint within your browser (e.g., `http://chapter-6-quarkus-rest-debug.127.0.0.1.nip.io/hello`) to see whether your remote application is alive and replies to a request.

It is time now to make some changes within the code of the application imported within an IDE. To do so, open the `chapter-6-quarkus-rest-debug/src/main/java/org/acme/GreeterResource.java` file and modify the response message (e.g., append the text `from Kubernetes/OpenShift` after `Hello RESTEasy`):

```
@Path("/hello")
public class GreeterResource {
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return "Hello RESTEasy from Kubernetes/OpenShift";
  }
}
```

Call the endpoint of the service again and check the log of your local Quarkus application locally. It should include the following text, which is reported as soon as Quarkus detects a change and recompiles the project.

```
INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] (Remote dev client thread)
  Changed source files detected, recompiling [quarkus-for-spring-developers/
  chapter-6/chapter-6-quarkus-rest-debug/src/main/java/org/acme/GreeterResource.java]
```

As soon as the code has been recompiled, it will be pushed to the remote application:

```
INFO  [io.qua.ver.htt.dep.dev.HttpRemoteDevClient] (Remote dev client thread)
  Sending dev/app/org/acme/GreeterResource.class
```

Check the response again using the URL within your browser: `http://chapter-6-quarkus-rest-debug.127.0.0.1.nip.io/hello`. You should see the updated text.

Remote Debugging

Debugging an application is a task that most developers perform locally using their favorite IDE. However, there are certain situations where attaching a remote debugger to a `Pod` running on a remote cluster is needed to investigate an issue. As microservices make up a more complex project or application, it might not be possible to run the different services locally. This is why, in this case, you will use a remote debugger.

Such a feature is supported natively by Quarkus. An environment variable is needed to tell the Java agent of the JVM to launch a socket able to accept remote connections to debug:

```
quarkus.kubernetes.env.vars.JAVA_TOOL_OPTIONS=-agentlib:jdwp=transport=
  dt_socket,server=y,suspend=n,address=0.0.0.0:5005
```

After changing the property, execute the following command again to regenerate a Kubernetes `Deployment` manifest and install it in the Kubernetes cluster:

```
./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

Note: If the remote application also supports TLS, the local client will be able to encrypt the data using the public key of the HTTPS certificate.

Note: As explained before, the application's URL depends on the Kubernetes distribution where it has been deployed and how access to it has been provided from outside the cluster.

Note: Instead of redeploying the application, you can edit the `Deployment` resource of your remote application as follows to add the needed environment variable:

```
spec:
  containers:
    - env:
      - name: JAVA_TOOL_OPTIONS
        value: -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,
               address=0.0.0.0:5005
```

A `NodePort` Service will be needed to allow your IDE tool to access the port exposed remotely on the cluster. An example of such a configuration has been added to `application.properties` file of the `chapter-6/chapter-6-quarkus-rest-debug` example, mapping the port `5005` to the external port `31000`.

Now that remote debugging is enabled on port `5005`, the remote debugger of your favorite IDE can be set to debug the remote application on that port.

Configuration Management

Managing the properties and resources needed to configure microservices properly is critical to project success. Adopting the most appropriate strategy for configuring and managing them is just as critical. The key point here concerns how to externalize such properties without the need to package them within the application. As discussed in detail in Chapter 2, the following options [6.44] are available to configure an application for local development:

1. System properties
2. Environment variables
3. A file named `.env` placed in the current working directory
4. An `application.properties` file placed in the `$PWD/config/` directory
5. An application configuration file, i.e., `src/main/resources/application.properties`

For cloud deployment, option 2 using environment variables is one of the easiest to use, as we will demonstrate. Options 3 through 5 imply that you have access to the filesystem mounted within the `Pod`. While not impossible, these options are more complex to configure and use in production. Option 1 requires you to pass the system properties in the `JAVA_OPTS` environment variable.

Environment Variables

A property defined within the `application.properties` file can be overridden using an environment variable, which is passed from the command line to the JVM when the application starts. For local development, this is really easy to configure, as follows:

```
export GREETING_MESSAGE=Hello Quarkus developers; java -jar
  target/quarkus-app/quarkus-run.jar
```

A Spring Boot application deployed on Kubernetes is also configured this way, declaring and passing environment variables to the container definition.

To make this method work in a container, you have to define the environment variable that the `Pod` will use when it starts the container. In this case, we use the `env` field, part of the `spec/containers` setting within the Kubernetes `Deployment` manifest [6.45].

To configure the `greeting.message` property in the example `chapter-6/chapter-6-quarkus-rest-config`, for instance, define the property using the `env` field of the YAML file:

```
spec:
  containers:
    - env:
        - name: GREETING_MESSAGE
          value: Hello from Kubernetes to
        - name: GREETING_NAME
          value: Quarkus developers
      image: quay.io/<group>/chapter-6-quarkus-rest-config:1.0
      imagePullPolicy: Always
      name: chapter-6-quarkus-rest-config
```

It is not necessary to edit the `kubernetes.yml or kubernetes.json` files manually to add such environment variables. Instead, you can use some new properties in the `application.properties` file:

```
quarkus.kubernetes.env.vars."greeting.message"=Hello from Kubernetes to
quarkus.kubernetes.env.vars."greeting.name"=Quarkus developers
```

The name of the environment variable appears between `env.vars.` and the = sign, and the value is declared after the = sign.

Variables that contain multiple words separated by periods need to be double quoted. Alternatively, you can specify variables in uppercase text separated by underscores (e.g., `quarkus.kubernetes.env.vars.GREETING_NAME=Quarkus developers` instead of `quarkus.kubernetes.env.vars."greeting.name"=Quarkus developers`).

This way of working is certainly not the most productive, as it requires you to declare the properties within the `application.properties` file twice; one time for the property itself and a second time to tell the Quarkus Kubernetes or OpenShift extension to generate the `Deployment` manifest:

```
greeting.message=Hello
quarkus.kubernetes.env.vars."greeting.message"=Hello from Kubernetes
```

For this reason, in the next section, we will show you a better way to externalize the variables.

ConfigMaps and Secrets

One of the native Kubernetes resources, called a `ConfigMap` [6.46], allows you to configure nonconfidential data in key/value pairs. This feature externalizes the configuration, therefore making your microservice application portable.

Both Quarkus and Spring [6.47] support the `ConfigMap` feature according to one of the following approaches:

- The `ConfigMap` data is mounted as environment variables within the `Pod` and recognized by the application, or;
- The application, at startup, will lookup from a `ConfigMap`, key/value pairs or a file named either `application.yml` or `application.properties`, which is used as input.

Let's have a look at a Quarkus project using a `ConfigMap`. Check the content of the `chapter-6/chapter-6-quarkus-rest-config` example project. The greeting parameters, which were previously part of the `application.properties` file, have been moved into a file named `src/main/configs/greeting-env`:

```
greeting.message=Hello configmap
greeting.name=quarkus
```

To create a ConfigMap with the content of this file on the Kubernetes cluster within the quarkus-demo-config namespace, execute the following commands:

```
$ kubectl create ns quarkus-demo-config
$ kubectl create -n quarkus-demo-config configmap greeting-map
  --from-env-file=src/main/configs/greeting-env
```

Note: In a "real" scenario, these properties would be maintained for each environment (e.g., dev/test/qa/stage/prod/etc) under different ConfigMaps or in different namespaces, and would be configured through CI/CD pipelines.

To ask the Quarkus Kubernetes extension to create an environment variable under the spec/container section, where the content is injected using the greeting-map ConfigMap, declare the following property within application.properties:

```
quarkus.kubernetes.env.configmaps=greeting-map
```

You can generate the Kubernetes resources and check the content of the generated Deployment manifest, making sure that Kubernetes has bound the data from the ConfigMap as environment variables:

```
envFrom:
  - configMapRef:
      name: greeting-map
```

Then redeploy a new microservice within the quarkus-demo-config namespace:

```
$ ./mvnw clean package
$ kubectl apply -f target/kubernetes/kubernetes.yml -n quarkus-demo-config
```

Note: If you change the contents of the properties declared within the ConfigMap, you must recreate the Pod to reload the ConfigMap data.

Everything we have explained using a ConfigMap can also be applied to a Kubernetes Secret [6.48]. The only difference is that values are stored as unencrypted base64-encoded strings, whereas in a ConfigMap they are stored in plain text. The properties using the Quarkus Kubernetes extension will also be different [6.49]:

```
envFrom:
  - secretRef:
      name: my-secret
```

The Spring Cloud Kubernetes project also supports mapping ConfigMaps and Secrets to a Spring Boot application [6.50]. The default behavior recommended in this case is to mount the Secret as a volume in the running Pod. While still supported, the lookup option is not enabled by default for security reasons, and it is recommended that you restrict access to the Secrets using some authorization policy.

Note: Even though the Kubernetes object is called a Secret, it has nothing to do with data protection or encryption. If data confidentiality is needed, you need to perform this encryption yourself or with the help of a product such as HashiCorp Vault [6.51].

In Quarkus, you can also map the content of a key part of a ConfigMap or Secret to a specific Quarkus environment variable within the application.properties using the with-key parameter:

```
quarkus.kubernetes.env.mapping.foo.from-secret=my-secret
quarkus.kubernetes.env.mapping.foo.with-key=keyName
```

Spring Cloud Config

The Kubernetes ConfigMap and Secret cover many use cases but require additional work by the release management team responsible for application configuration. Different configurations (ConfigMap or Secret) are needed to cover the different environments where the applications will be deployed. The security officers are also responsible for designing different authorization policies (aka role-based access control, or RBAC) to ensure that the Pod's service account will not be granted access to consume a Secret or ConfigMap from other namespaces.

Such extra work and the lack of a centralized place to manage all the configurations for the different projects can easily be a burden for organizations dealing with hundreds or thousands of applications. A centralized approach might be preferred in this case.

For centralized configuration management, you can rely on a configuration server such as the Spring Cloud Config Server (SCCS) [6.52]. Quarkus supports SCCS if you install, as part of your project, the client that can communicate with this server.

To get Quarkus working with SCCS, include the `spring-cloud-config-client` extension in your project [6.53]. The SCCS server can be installed on the Kubernetes cluster or anywhere else that is accessible from your microservices over HTTP.

The steps for setting up SCCS on a Kubernetes cluster are beyond the scope of this book. You can follow the instructions shared within the documentation in the `chapter-6/chapter-6-quarkus-rest-cloud-config/README.md` example project for more information.

The example's `src/main/application.properties` file includes the necessary information for the application to communicate with the remote SCCS server:

```
quarkus.application.name=greeting-application (1)
quarkus.spring-cloud-config.enabled=true (2)
quarkus.spring-cloud-config.url=http://spring-cloud-config-server.config-storage:80 (3)
```

1. The `quarkus.application.name` property corresponds to the property file that the SCCS server will fetch from the Git repository. The property follows the Spring Cloud Config naming convention [6.52].
2. The client is not enabled by default even if the extension is added to the path, so it must be enabled explicitly.
3. This setting specifies the URL and port to access the SCCS server. The URL uses the Kubernetes DNS service address.

The Spring Boot Client configuration is nearly identical to a Quarkus application and is defined within the `bootstrap.properties`. More information is available in this article [6.54].

To install the application on the Kubernetes cluster, execute the following commands:

```
$ ./mvnw clean package
$ kubectl create ns quarkus-demo-cloud-config
$ kubectl apply -f target/kubernetes/kubernetes.yml -n quarkus-demo-cloud-config
```

Once the application is deployed and can access the SCCS server, you can call the endpoint using the `httpie` tool (or `curl`):

```
http http://chapter-6-quarkus-rest-cloud-config.127.0.0.1.nip.io/hello
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 32
Content-Type: text/plain;charset=UTF-8

Hello cloud config prod quarkus!
```

Note: The Quarkus client can read and set the properties only when the application is started, not during the build. This means that Quarkus properties that are read and fixed at build time cannot be stored within SCCS.

Note: If your application supports different profiles (such as `dev`, `prod`, etc), you can create different files in the Git repository backing the SCC server, one for each profile, such as *`applicationName`*`-dev.properties` and *`applicationName`*`-prod.properties`.

Monitoring

Microservices running in the cloud must be managed carefully to optimize production runs, anticipate CPU and memory issues, and collect usage statistics. We have already covered the `HealthCheck` monitoring feature, which can provide information on the status of your `Pods` using a heartbeat mechanism. This is not enough, though. You also need to collect metrics, logs, or even distributed traces of the applications in order to make appropriate decisions about the architecture of the deployed solution.

All these technologies are available on top of Quarkus and can be easily implemented, as shown in the following subsections.

Metrics

Several projects exist to collect metrics from microservices. Micrometer [6.55] is the standard adopted by Spring and Quarkus. This framework allows you to instrument your code with dimensional metrics using a vendor-neutral interface. In Spring, Micrometer is supported by the Spring Boot Actuator starter if the Micrometer libraries are included on the classpath of the application. Quarkus has a Micrometer extension [6.56]. As mentioned within the Micrometer documentation, you can think about this project as equivalent of the Simple Logging Facade for Java (SLF4J) framework, but to support the collection and exposure of metrics for different monitoring systems, such as Prometheus, StatsD, Influx, DataDog, etc.

The code of the example that we will show in detail is part of the `chapter-6/chapter-6-quarkus-rest-monitoring` project within the GitHub examples repository [6.3].

To collect metrics, you need to configure a `MeterRegistry` object. The `MeterRegistry` is configured and maintained by the Micrometer extension, and can be injected into your application as follows:

```
@Path("/hello")
public class GreetingsResource {
  private final MeterRegistry registry;

  public GreetingResource(MeterRegistry registry) {
    this.registry = registry;
  }
}
```

To increase a counter every time the `/hello` endpoint is called, simply call the `counter` method on the `MeterRegistry` object:

```
@GET
@Path("/{name}")
@Counted
public String sayHello(@PathParam(value = "name") String name) {
  registry.counter("greeting_counter", Tags.of("name", name)).increment();
}
```

The REST endpoint methods are counted and timed by the Micrometer extension out of the box. Micrometer also defines two annotations, `@Counted` and `@Timed`, that can be added to methods.

Note:

With metrics enabled, compile and launch your Quarkus application:

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

Now you should be able to call your service:

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

It should return `Hello!`. You can then view the metrics via the metrics endpoint exposed at `http://localhost:8080/q/metrics`. There, you should see your `greeting_counter` counter:

```
# HELP greeting_counter_total
# TYPE greeting_counter_total counter
greeting_counter_total{name="quarkus",} 1.0
```

Note: To collect metrics using Prometheus as a back end and a Grafana dashboard, we recommend that you run your containers on Red Hat OpenShift, which supports these tools natively [6.57]. Prometheus and Grafana can also be deployed on a vanilla Kubernetes cluster [6.58].

Distributed Tracing and Logs

To complete our coverage of important cloud behavior, we must speak about the "Distributed tracing" pattern that is used to collect information from the different microservices in order to follow, like on a pathway, the places where your application has been called and has invoked various microservices.

Distributed tracing plays an important role in helping teams monitor applications. Traces help developers figure out where there may be resource contention or bad response times between microservices, back ends, or other middleware components, such as message brokers and databases while Ops engineers will be able to determine more easily where problems occurred on the platform.

Like the metrics we covered in the previous section, different standards have been designed and implemented. Two popular open source options are OpenTracing [6.59], which will cover here, and OpenTelemetry [6.60], a newer player.

Note: At the time of writing this book, the implementation of the Quarkus OpenTelemetry extension was not yet finalized. This is why we cannot yet cover it, but you can follow its development [6.61].

A trace tracks the progress of a single request as it passes through services. Distributed tracing is a form of tracing that traverses process, network, and security boundaries. Each unit of work in a trace is called a *span*, which is an object that represents the work being done by individual services or components involved in a request as it flows through a system. A *trace* is a tree of spans. Think of a distributed trace as like a Java stack trace, capturing every component of a system that a request flows through while tracking the amount of time spent in and between each component.

All the Quarkus REST endpoints are automatically traced, but you can customize this default behavior by using the `@Traced` annotation. When you declare it at the class level, all methods are traced. At the method level, it registers just that method to the tracing chain.

The following code, part of the `chapter-6/chapter-6-quarkus-rest-monitoring` example, shows such a declaration of the `@Traced` annotation at the method level:

```
@ApplicationScoped
public class FrancophoneService {
  @Traced
  public String bonjour() {
    return "bonjour";
  }
}
```

To configure Jaeger [6.62], the framework responsible for collecting traces, you need some additional properties in the `src/main/application.properties`:

```
quarkus.jaeger.service-name=myservice (1)
quarkus.jaeger.sampler-type=const (2)
quarkus.jaeger.sampler-param=1 (3)
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId},
  parentId=%X{parentId}, spanId=%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t)
  %s%e%n (4)
```

1. If the `quarkus.jaeger.service-name` property (or `JAEGER_SERVICE_NAME` environment variable) is not provided, a "no-op" tracer will be configured, resulting in no tracing data being reported to the back end.
2. Sets up a sampler that uses a constant sampling strategy. Strategies supported are: constant, probabilistic, rate limiting, and remote.
3. Samples all requests. Set `sampler-param` to somewhere between 0 and 1 (e.g., 0.50), if you do not wish to sample all requests.
4. Adds trace IDs to log messages.

Next, start the tracing system to collect and display the captured traces:

```
$ docker run -p 5775:5775/udp -p 6831:6831/udp -p 6832:6832/udp -p 5778:5778
  -p 16686:16686 -p 14268:14268 jaegertracing/all-in-one:latest
```

The tracing system is composed of a centralized collector able to collect from the Jaeger agents running next to the application.

Now you are ready to run your application:

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

Once both the application and tracing system are started, you can make a request to the provided endpoint:

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

When the first request is submitted, the Jaeger tracer within the application is initialized:

```
INFO  traceId=4f72bd578d154b25, parentId=0, spanId=4f72bd578d154b25,
  sampled=true [or.ac.TracedResource] (vert.x-eventloop-thread-21) hello
```

Visit the Jaeger user interface, running at `http://localhost:16686/` and shown in Figure 6.3, to see the tracing information.

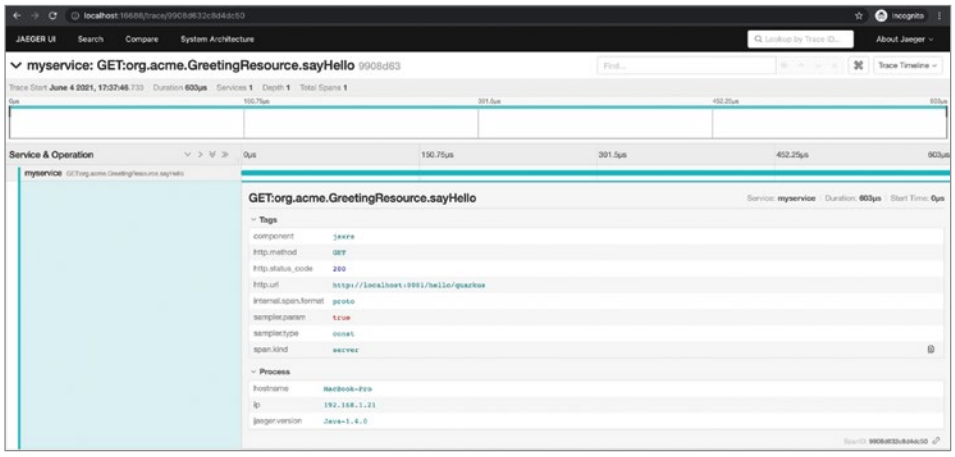

Figure 6.3: Jaeger tracing information.

Press `CTRL+C` to stop the application.

Summary

This chapter covered several cloud patterns and features that Quarkus supports natively, which improve the developer experience when you need to build and deploy microservices on a Kubernetes cluster. When applicable, we have also explained what needs to be done to deploy Spring Boot microservices in the cloud and to generate the Kubernetes manifests properly.

We have covered only a few patterns, such as HealthCheck, externalization of the configurations, service binding, distributed logging, metrics collection, and remote coding. These patterns and features will help you to manage better, maintain, and troubleshoot your applications in the cloud, as well as help you figure out how to bind services of the cluster with your microservices.

References

[6.1] "Kubernetes": https://kubernetes.io/

[6.2] "OpenShift Kubernetes Distribution": https://www.okd.io/

[6.3] "Quarkus for Spring Developers - GitHub Repository of the examples": https://github.com/quarkus-for-spring-developers/examples.git

[6.4] "Kubernetes client - kubectl": https://kubernetes.io/docs/tasks/tools/

[6.5] "OpenShift client - oc": https://docs.openshift.com/container-platform/4.7/cli_reference/openshift_cli/getting-started-cli.html#installing-openshift-cli

[6.6] "Kubernetes manifest": https://kubernetes.io/docs/reference/glossary/?all=true#term-manifest

[6.7] "odo Client tool": https://odo.dev/docs/getting-started/features

[6.8] "Docker ENTRYPOINT": https://docs.docker.com/glossary/#entrypoint

[6.9] "Google JIB Image building tool": https://github.com/GoogleContainerTools/jib

[6.10] "Quarkus Container image - JIB" : https://quarkus.io/guides/container-image#jib

[6.11] "CNCF Buildpacks" : https://buildpacks.io/

[6.12] "Paketo Buildpacks for Spring Boot" : https://paketo.io/docs/buildpacks/language-family-buildpacks/java/#components

[6.13] "Build a Spring Boot container image": https://docs.spring.io/spring-boot/docs/2.5.2/reference/htmlsingle/#features.container-images.building.buildpacks

[6.14] "OpenShift S2i": https://docs.openshift.com/container-platform/4.7/openshift_images/create-images.html#images-create-s2i-build_create-images

[6.15] "Quarkus parameters to configure the container image, registry": https://quarkus.io/guides/container-image#container-image-options

[6.16] "Spring Boot Maven plugin parameters to configure the image": https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/#build-image.docker-registry

[6.17] "Kubernetes manifest" : https://kubernetes.io/docs/reference/glossary/?all=true#term-manifest

[6.18] "Pod resource" : https://kubernetes.io/docs/concepts/workloads/pods/

[6.19] "Kubernetes API Server": https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[6.20] "Kubernetes namespace" : https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[6.21] "Kubernetes Deployment": https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[6.22] "Quarkus Kubernetes guide": https://quarkus.io/guides/deploying-to-kubernetes

[6.23] "Dekorate": https://github.com/dekorateio/dekorate

[6.24] "Dekorate and Spring Boot": https://github.com/dekorateio/dekorate/tree/master/examples

[6.25] "Quarkus and Openshift S2I Build": https://quarkus.io/guides/deploying-to-openshift

[6.26] "OpenShift Project": https://docs.openshift.com/container-platform/4.7/rest_api/project_apis/project-project-openshift-io-v1.html

[6.27] "Spring Boot Kubernetes Guide": https://spring.io/guides/gs/spring-boot-kubernetes/

[6.28] "Ingress controllers": https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

[6.29] "OpenShift Ingress Controller and HAProxy": https://docs.openshift.com/container-platform/4.7/networking/ingress-operator.html

[6.30] "Health Check microservice pattern": https://microservices.io/patterns/observability/health-check-api.html

[6.31] "Kubernetes Health Check": https://kubernetes.io/docs/reference/using-api/health-checks/

[6.32] "MicroProfile Health": https://github.com/eclipse/microprofile-health/

[6.33] "SmallRye Health": https://quarkus.io/guides/smallrye-health

[6.34] "Spring Boot Health Check": https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.kubernetes-probes

[6.35] "Dekorate Health Check probes" : https://github.com/dekorateio/dekorate/blob/master/assets/config.md#probe

[6.36] "Httpie tool": https://httpie.io/

[6.37] "Kubernetes Service Binding": https://github.com/k8s-service-bindings/spec

[6.38] "Spring Cloud Service Binding": https://github.com/spring-cloud/spring-cloud-bindings

[6.39] "Application projection" : https://github.com/k8s-service-bindings/spec#application-projection

[6.40] "Service Binding Operator" : https://github.com/redhat-developer/service-binding-operator

[6.41] "Kind - Tool for running local Kubernetes clusters using Docker desktop": https://kind.sigs.k8s.io/docs/user/quick-start/

[6.42] "Spring Boot DevTools": https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.devtools

[6.43] "Remote live coding": https://developers.redhat.com/blog/2021/02/11/enhancing-the-development-loop-with-quarkus-remote-development/

[6.44] "Configuration sources": https://quarkus.io/guides/config-reference#configuration_sources

[6.45] "Define environment variables for a container": https://kubernetes.io/docs/tasks/inject-data-application/define-environment-variable-container/

[6.46] "Storing Kubernetes configuration - ConfigMap" : https://kubernetes.io/docs/concepts/configuration/configmap/

[6.47] "Spring Cloud Kubernetes and ConfigMap": https://docs.spring.io/spring-cloud-kubernetes/docs/current/reference/html/#kubernetes-propertysource-implementations

[6.48] "Storing sensitive Kubernetes configuration - Secret" : https://kubernetes.io/docs/concepts/configuration/secret/

[6.49] "Quarkus Environment variables from Secrets": https://quarkus.io/guides/deploying-to-kubernetes#environment-variables-from-secret

[6.50] "Spring Cloud Kubernetes and Secrets": https://docs.spring.io/spring-cloud-kubernetes/docs/current/reference/html/#secrets-propertysource

[6.51] "Hashicorp Vault on Kubernetes": https://www.vaultproject.io/docs/platform/k8s

[6.52] "Spring Cloud Config Server": https://cloud.spring.io/spring-cloud-config

[6.53] "Quarkus and Spring Cloud Config guide": https://quarkus.io/guides/spring-cloud-config-client

[6.54] "Spring Cloud Config client": https://cloud.spring.io/spring-cloud-config/reference/html/#_spring_cloud_config_client

[6.55] "Micrometer Application Monitoring framework": https://micrometer.io/

[6.56] "Quarkus and Micrometer": https://quarkus.io/guides/micrometer

[6.57] "Micrometer and Prometheus on OpenShift": https://quarkus.io/blog/micrometer-prometheus-openshift/

[6.58] "Start monitoring your Kubernetes cluster with Prometheus and Grafana": https://opensource.com/article/21/6/chaos-grafana-prometheus

[6.59] "Quarkus OpenTracing guide": https://quarkus.io/guides/opentracing

[6.60] "OpenTelemetry CNCF project": https://opentelemetry.io/

[6.61] "Quarkus OpenTelemetry extension": https://github.com/quarkusio/quarkus/tree/main/extensions/opentelemetry

[6.62] "Jaeger Opentracer": https://www.jaegertracing.io/

Appendix

A.1 Imperative / Blocking versus Reactive / Non-Blocking

A.1.1 Imperative / Blocking

An application that serves HTTP requests, such as one using the Java Servlet API, is synchronous and blocking. Classes such as `Filter` and `Servlet` are at the core of these well-known blocking APIs. When a request arrives, a `Thread` is assigned to handle the request from start to finish. That `Thread` may be created on-demand or come from a preconfigured pool of available `Threads`.

Upon completion of the request, the `Thread` is discarded or returned to the pool. The flow through the application code consists of chained method calls, where a parent call blocks and waits for all of its child method calls to complete. This style is also known as *imperative programming*.

This design is wasteful because, during a request's lifetime, there is usually only a short amount of time when a `Thread` actively consumes CPU cycles. The majority of the time, the `Thread` is waiting on work to be completed. This work could be network I/O, where perhaps the application is waiting for a network call or database operation to complete, or anything else where some latency is involved and expected.

A servlet container can create and manage only a limited number of threads. If many requests arrive simultaneously, the servlet container may start queuing those requests. This queue takes up valuable memory resources. If a request's duration is long, the application may eventually run out of available memory, causing the application to crash.

A.1.2 Reactive / Non-Blocking

A reactive engine, such as Netty [A.1], minimizes the number of `Threads` that a system uses to handle requests to a small fixed number, resulting in more efficient use of hardware resources. Any task waiting for work to complete is moved off onto a separate worker `Thread`.

When one of those waiting `Threads` needs to resume, its work is moved back onto the main `Thread` and completed. This architecture enables a reactive, or functional, programming style where developers can compose asynchronous logic via higher-level functions. This architecture doesn't necessarily make an application faster; instead, it makes an application more scalable and resilient. An increased load caused by a spike in requests does not increase the overhead of creating and releasing threads, as it does in a servlet-based application. This architecture is also known as the *event-loop model*.

Drawbacks with the reactive model are that developers need to learn how to write non-blocking code correctly, compose asynchronous processing, and use non-blocking libraries. An application could become unresponsive quickly if code is not written correctly. It's a paradigm shift for many developers today because reactive code isn't as simple as imperative code to read, write, or test. Changing the development model is not simple. It requires relearning and restructuring code.

A.2 IDE Support

A.2.1 Microsoft VSCode

The Quarkus Tools for Visual Studio Code [A.2] extension supports Quarkus and MicroProfile development by extending the Visual Studio Code extension for MicroProfile with Quarkus features. Additionally, this plug-in can be installed in other web-based IDEs, such as Eclipse Che and Red Hat CodeReady Workspaces. The plug-in introduces

Appendix

various commands that can generate or interact with a Quarkus project. It also enables completion and context support for Quarkus and MicroProfile properties and many other capabilities while building Quarkus applications.

A.2.2 JetBrains IntelliJ

JetBrains IntelliJ is available in two different editions: Community and Ultimate editions [A.3]. IntelliJ Community Edition is free and supports JVM development. The Ultimate edition requires a paid subscription, but it contains additional tooling and features that aren't available in the Community edition. Two such features in the Ultimate edition include Spring Framework integration and support and Quarkus integration [A.4]. These capabilities are provided out of the box and are supported directly by JetBrains. The Quarkus plug-in provides capabilities such as code assistance specific to Quarkus, integration with the Bean Validation, CDI, and Endpoints tool windows, a dedicated project creation wizard based on code.quarkus.io, and dedicated run configurations for Quarkus applications.

Additionally, Red Hat provides a Quarkus plug-in for the Community or Ultimate editions [A.5]. This plug-in provides many capabilities beyond the one included in the Ultimate edition, including codestart options when creating new Quarkus projects, completion and context support for Quarkus and MicroProfile properties, hovers, and additional context-sensitive tooling within the editors, as well as many other capabilities.

A.2.3 Eclipse

Eclipse remains one of the most widely used IDEs by Java developers today. There are a few different ways to get all the same expected Quarkus tooling and capabilities into your Eclipse IDE.

Quarkus Tools is part of JBoss Tools [A.6]. If you are already a JBoss Tools user, Quarkus Tools is already installed in your Eclipse IDE. Quarkus Tools is also part of Red Hat CodeReady Studio [A.7], a Red Hat-supported version of JBoss Tools, starting at version 12.14. If you are already using CodeReady Studio, Quarkus Tools is already installed. Additionally, Quarkus Tools can be installed directly from the Eclipse Marketplace into any other Eclipse installation [A.8].

References

[A.1] "Netty": https://netty.io

[A.2] "Quarkus Tools for Visual Studio Code": https://marketplace.visualstudio.com/items?itemName=redhat.vscode-quarkus

[A.3] "JetBrains IntelliJ IDEA Edition Comparison Matrix": https://www.jetbrains.com/idea/features/editions_comparison_matrix.html

[A.4] "JetBrains IntelliJ IDEA Ultimate Edition Quarkus Tooling": https://www.jetbrains.com/help/idea/quarkus.html

[A.5] "Quarkus Tools for IntelliJ plugin": https://plugins.jetbrains.com/plugin/13234-quarkus-tools

[A.6] "Quarkus Tools JBoss Tools": https://tools.jboss.org/features/quarkus.html

[A.7] "Red Hat CodeReady Studio": https://www.redhat.com/en/technologies/jboss-middleware/codeready-studio

[A.8] "Quarkus Tools on Eclipse Marketplace": https://marketplace.eclipse.org/content/quarkus-tools

About the Authors

Eric Deandrea is a Senior Principal Developer Advocate at Red Hat, focusing on application development technologies such as Quarkus and Spring Boot, as well as the rest of Red Hat's portfolio. He has spent most of his 22+-year software development and architecture career in the financial services and insurance industries building Java applications and application frameworks, with most of the last ten years within these organizations building opinionated frameworks based on Spring and architecting DevOps solutions. He has been responsible for creating and delivering developer training programs based on Spring and Spring Boot. He has also contributed to Spring projects, including Spring Boot and Spring Security. Outside of work, Eric enjoys ice hockey and martial arts. He holds a black belt in Kempo Karate. You can find him on Twitter @edeandrea and on LinkedIn.

Daniel Oh is a Senior Principal Technical Marketing Manager at Red Hat, a well-known public speaker, open source contributor, published author, and developer advocate. He has more than 20 years of experience solving real-world enterprise problems in production environments using cloud-native technologies such as Quarkus, Spring Boot, Node.js, and Kubernetes. He is also a Cloud Native Computing Foundation (CNCF) and DevOps Institute ambassador for evangelizing developers and operators to develop cloud-native microservices, design serverless functions, and deploy them to multi and hybrid clouds in flexible, easy-to-use, cost-effective, open, and collaborative ways. You can find Daniel on Twitter @danieloh30 and on LinkedIn.

Charles Moulliard is a Software Engineer Manager at Red Hat, an open source Apache contributor, PMC member, and middleware and Kubernetes technology advocate. He currently leads the Spring team at Red Hat to integrate the Spring technology part of the Red Hat Middleware offering and Quarkus runtime. He has more than 25 years of experience as an architect, project delivery manager, technical leader, and manager focused on Java EE, Spring, Spring Boot technology, and middleware enterprise integration patterns and products.

Over the last few years, Charles has participated in the development of different cloud-native projects to enhance the developer experience on Kubernetes and Red Hat OpenShift. He is a mountain biker, hiker, passionate homebrewer, and gardener. You can follow him on Twitter @cmoulliard and on LinkedIn.