

Compliments of



Quarkus

IN ACTION

Martin Štefanko
Jan Martiška



A detailed illustration of a woman from the 18th or 19th century, wearing a large white bonnet, a red shawl over a blue dress, and a patterned shawl. She is carrying a large red banner with the letters "MEAP" written in white, which is partially visible at the bottom of the page.

MEAP

This book is in MEAP - Manning Early Access Program

What is MEAP?

A book can take a year or more to write, so how do you learn that hot new technology today? The answer is MEAP, the Manning Early Access Program. In MEAP, you read a book chapter-by-chapter while it's being written and get the final eBook as soon as it's finished. In MEAP, you get the book before it's finished, and we commit to sending it to you immediately when it is published. The content you get is not finished and will evolve, sometimes dramatically, before it is good enough for us to publish. But you get the chapter drafts when you need the information. And you get a chance to let the author know what you think, which can really help us both make a better book.

MEAP offers several benefits over the traditional "wait to read" model.

- Get started now. You can read early versions of the chapters before the book is finished.
- Regular updates. We'll let you know when updates are available.
- Contribute to the writing process. Your feedback in the [liveBook Discussion Forum](#) makes the book better.

To learn more about MEAP, visit <https://www.manning.com/meap-program>.





MEAP Edition
Manning Early Access Program

Quarkus in Action
Version 9

Copyright 2024 Manning Publications

For more information on this and other Manning titles go to manning.com.

Welcome

Thanks for purchasing the MEAP for *Quarkus in Action*!

Quarkus is a full-stack framework for building cloud-native Java applications. This book focuses on explaining how Quarkus functions and the extensive overview of different technologies that integrate with Quarkus to provide various functionalities integral to your applications. Whether your focus is on microservices or you target serverless deployments, this book will teach you how to write them in the productive manner that Quarkus provides. We assume that you've worked with Java or a similar JVM language in the past and possibly that you also might have some experience with enterprise application development.

Throughout the book, we create a Car rental application simulating a real-world example that consists of several Quarkus microservices. Whether you choose to code this application with us is totally up to you, but it can be an engaging hands-on experience that you might not find elsewhere. The application is intentionally demonstrating multiple different technologies to cover as many options that Quarkus supports as possible.

Quarkus in Action is divided into three parts. Part 1 focuses on the description of Quarkus and the design decisions that drive the development of Quarkus applications. In this part, we also create our first Quarkus REST-based application and compile it into a GraalVM native image. You also learn about the features that make application development with Quarkus different from alternative Java frameworks.

In Part 2, we start developing the Car rental system. Step by step, we cover different requirements for modern cloud-native applications, including multiple communication, databases, or security options. Each description of technology provides an example of integration into Car rental Quarkus microservices. The last Part 3 focuses on the application deployment into the cloud. We learn about the ease of deployment to Kubernetes and the Quarkus development of serverless applications.

Audience feedback is a fundamental benefit of any creative activity. Your feedback is essential for us to create the best final version of the book. We hope you will consider using the [liveBook's Discussion Forum](#) if you have any comments or questions about the content you read in this book. But also if you have any ideas about the areas you would like to see covered.

-Martin Štefanko and Jan Martiška

brief contents

PART 1: GETTING STARTED WITH QUARKUS

- 1 What is Quarkus?*
- 2 Your first Quarkus application*
- 3 Enhancing developer productivity with Quarkus*

PART 2: DEVELOPING QUARKUS APPLICATIONS

- 4 Handling communications*
- 5 Testing Quarkus applications*
- 6 Exposing and securing web applications*
- 7 Database access*
- 8 Reactive programming*
- 9 Reactive messaging*
- 10 Cloud-native application patterns*
- 11 Quarkus applications in the cloud*

PART 3: QUARKUS IN THE CLOUD

- 12 Quarkus applications in cloud*
- 13 Serverless Quarkus applications*

Appendix A. Alternatives for developing frontend applications with Quarkus

1

What is Quarkus?

This chapter covers

- Introducing Quarkus
- Understanding the Quarkus principles
- Analyzing Quarkus architecture
- Evaluating Quarkus alternatives

Java is one of the most popular programming languages utilized for developing enterprise systems. With its vast ecosystem of libraries, frameworks, standards, runtimes, and most importantly us, the developers, Java represents a genuine choice for building modern, robust, and scalable software.

However, many systems often solve similar challenges. This is why they often rely on some underlying technology that provides solutions to these problems. Whether the composition of the system consists of a set of JARs, WARs, or EARs (Java, Web, or Enterprise Archives) deployed on an application server, or whether it follows the more recent microservices architecture, there are multiple choices of Java frameworks and libraries that the system can utilize.

Quarkus is a Java framework primarily targeting microservices and serverless applications. It aims to provide an application framework that delivers an unmatched performance benefits while still providing a development model utilizing the *APIs* (Application-Programming interfaces) of popular libraries and Java standards that the Java ecosystem has been practicing for years. Quarkus also puts a strong focus on developer productivity because a technology's productivity and usability are what developers appreciate the most.

In this book, we focus on delivering a concise learning experience of the Quarkus framework assuming no prior Quarkus experience and gradually building towards a completely developed enterprise system consisting of multiple Quarkus microservices. Our priority is to explain the main concepts, tools, and features of Quarkus, so by the end of the book you understand the values that Quarkus provides and can also assess the Quarkus applicability for your projects.

1.1 Why Quarkus?

Especially with the move to the cloud environments, application startup times (as restarts are managed by the deployment platform) and memory/CPU utilization (representing cloud costs) became the prominent application metrics. However, Java was originally designed for big, long-running applications, which meant it didn't particularly fit into these environments.

Quarkus was created to address these issues. It defined a very lightweight framework that splits the application processing into the build-time and runtime phases. Any Java stack needs to do a lot of processing once the application starts, e.g. reading and processing of the application's code, annotations, configuration, injection points (inversion of control), or generating dynamic code to properly initialize the application within the framework. And all of this processing repeats with every application start.

Quarkus, on the other hand, splits the processing into build-time and runtime phases. It moves as much of the processing as possible to the build time phase (application build/packaging) to save resources at runtime. The packaged application carries only the results of these initial framework operations, which makes it not only faster to start since they don't need to be executed at startup, but also smaller in size, because any code that is needed only in this initialization phase is never included in the final packaged artifact.

While performance benefits are very important, Quarkus also puts enormous effort into the developer experience when using the framework. It provides the so-called *Dev mode* which is a continuous execution of the Quarkus application allowing live reloads of (not only) code changes without the application restart or compilation. By just saving our changes in the editor and invoking the application, Quarkus dynamically applies our changes, so we can see the results of our modifications in mere milliseconds. The development flow doesn't need to stop to recompile and rerun the application which closes the reevaluation loop in a continuous experience similar to scripting languages while we still work with JVM. Dev mode bundles a lot of productivity boosting features that we dive into in this book.

Quarkus also exposes all these features for integrations which makes it very popular also among library developers. With its over 670 *extensions* (pluggable pieces of functionalities) that integrate different popular libraries (e.g., Hibernate, Jackson, or LangChain4j), Quarkus proves that it stands out as a framework where the Java ecosystem wants to be.

1.2 Who is this book for?

This book is for all software developers with basic Java enterprise application development knowledge. Other JVM (Java Virtual Machine) languages are also a good starting point since Quarkus also allows you to develop with, for instance, Kotlin (stable) or Scala (preview). However, we focus on Java in this book. The individual chapters of this book dive into different technologies, some of which are quite new. You will find that Quarkus is a great learning tool for such purposes. So if we encounter a more recent technology that you might not know yet, we also explain the basic concepts in addition to its utilization in Quarkus to provide the complete picture.

1.3 Introducing Quarkus

The name Quarkus consists of two parts—quark and us. A quark is an elementary particle that constitutes matter aligning with the very small resource footprint that Quarkus aims to provide. The second part, "us", comprises us developers, engineers, and operations who utilize various software development processes where Quarkus creates as useful environments as possible.

Quarkus.io website describes Quarkus as a "Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards" (<https://quarkus.io>). But what does this really mean? Let's break down the definition and describe in depth what Quarkus is, what problems it tries to solve, and why you should care about learning it.

1.3.1 A Kubernetes native stack

Let's start with *Kubernetes* native. This warrants a quick Kubernetes (<https://kubernetes.io>) introduction. As enterprise applications and services started to move from physical servers to the cloud, the need to encapsulate the application with its execution environment (i.e., the operating system and dependencies) became prominent. This led to the evolution of container technologies made mainstream by Docker (<https://docker.io>). With containers becoming increasingly popular, the need for container management appeared, and that's precisely what Kubernetes is: the de-facto standard container management platform.

[Figure 1.1](#) visualizes the relationships between applications, containers, and the management platform. The user-provided containers lifecycle and resourcing are delegated to the Kubernetes platform. The technologies that run inside the containers might differ. Kubernetes treats the container as a unit without knowing the container's internals.

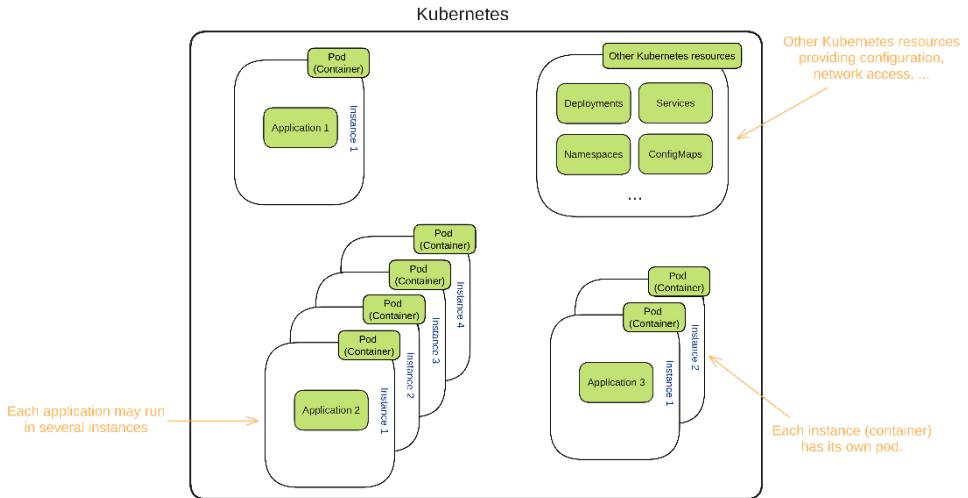


Figure 1.1 Kubernetes architecture

Kubernetes operates containers in the units called pods. Each pods consists of usually one or sometimes even more containers (e.g., sidecars with the *service mesh*). [Figure 1.2](#) depicts a standard JVM container in a pod which, except for the user application, also packages the actual JVM on which the application runs. In turn, the JVM requires some other dependencies, for instance, `glibc`, which the container also needs to include.

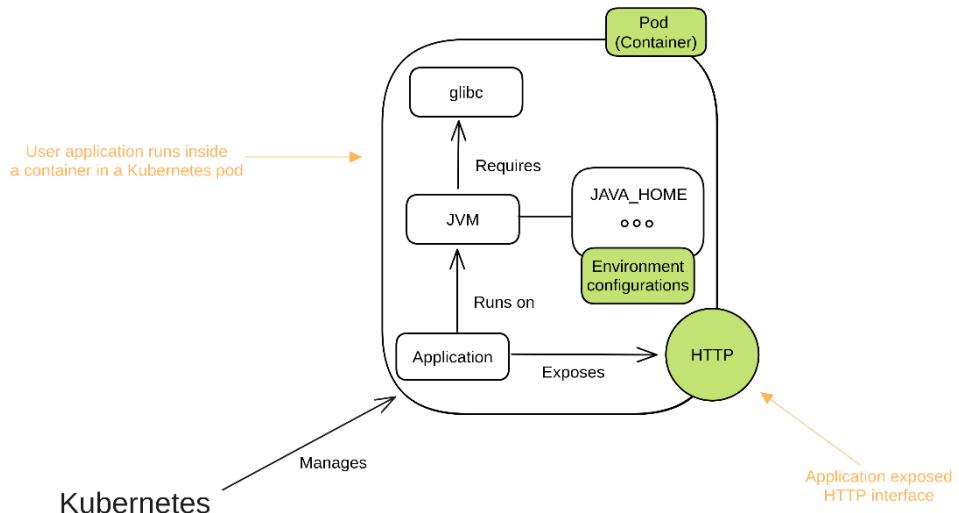


Figure 1.2 Visualizing the relationships between the application, container, and Kubernetes

Delegating management of the application lifecycle to Kubernetes also requires a way to make dynamic decisions in terms of configuring our application. The configuration of the individual containers might be different even for the same application replicated in multiple containers (e.g., a unique identifier). For this reason, Docker containers and consecutively, the Kubernetes platform allows us to override the configuration values inside the respective containers with environment variables. These can be implicitly deduced from the container specification as, for instance, the `JAVA_HOME` shown in [Figure 1.2](#) or we can manually pass them to the container (pod) when it's starting. Changing the configuration manually passed in the environment variables thus also means that the Kubernetes restarts the container with new values.

We use the term *Kubernetes native* to describe tools and frameworks specifically targeting Kubernetes as the target deployment platform. Does this label imply that the stack of these tools that Quarkus provides is exclusively related to Kubernetes? No, it's mostly a way to communicate that the development team has taken the extra effort to ensure a positive user experience on Kubernetes (or cloud in general).

So, in what ways is Quarkus Kubernetes native? By being designed for containers as the main packaging format utilized in cloud and by providing a toolset that allows you to build and deploy containers in a single step. Additionally, Quarkus also supports a series of features that promote integration with the cloud platforms (e.g., exposing health-related information, externalizing configuration directly in the platform, etc.), which are integral for a positive user experience in the cloud environment.

DESIGNED FOR CONTAINERS

The software industry has been skeptical about the use of Java inside containers. Containers need to start quickly, consume few resources (CPU, memory), and be small because these performance metrics directly translate to the cost of applications in the cloud. Java generally doesn't have a good reputation in this area. But this is no longer true with Quarkus.

Quarkus has been designed to create applications that start quickly and have excellent efficiency and performance. It uses a concept of build time processing to move as much processing as possible from runtime to compile time. This is an approach popularized by Google Dagger, a dependency injection framework for Java and Android. For mobile applications, runtime performance and resource utilization are really important as they directly impact the user experience. So, Dagger had to push as much processing possible from runtime to compile time, to reduce response times and battery consumption. In enterprise applications, battery consumption may be irrelevant. Still, the memory footprint is not as it can directly impact the production costs since it is one of the critical factors affecting cloud computing pricing.

SINGLE STEP DEPLOYMENTS

Many developers despise writing configuration files, deployment manifests, or anything else expressed in a markup language. However, Kubernetes deployments do require writing such manifests. Some feel it's not part of their role, and others find it boring. Still, all of these developers might be relieved to learn that Quarkus comes with its own tools that allow your application to be packaged into a container which can then be shipped to Kubernetes as part of your application build without requiring developers to compose the manifests themselves. This feature saves time and guarantees that the experience is optimized for each application. In other words, the framework understands the needs of your application that it requires from its platform and expresses them by tuning the deployment process accordingly.

1.3.2 OpenJDK HotSpot and GraalVM

OpenJDK HotSpot (<https://openjdk.org/groups/hotspot/>) refers to the Java Virtual Machine (JVM), the most common runtime for Java applications. GraalVM (<https://www.graalvm.org/>) is a JVM and development kit distribution that brings improved performance, support for multiple languages, and native image (binary) compilation to the table. GraalVM is composed of a set of different layers which compared to the traditional JVM, provides a way of also supporting non-JVM languages (e.g., JavaScript or Python). The complete overview of GraalVM functionality is beyond the scope of this book. We focus on the GraalVM compiler and the native image compilation which are the main parts of GraalVM that Quarkus utilizes.

Native compilation allows users to build a standalone binary executable that runs without requiring a JVM. It represents simplified application distribution and provides an execution environment where the application no longer has to wait for the JVM to start, which decreases the application startup time. Such native executables start in tens of milliseconds which is incredibly fast for any Java application (we will get more into this in a later part of the book). So why don't we compile all our Java code into native binaries? The main problem is that the process of creating a native image is often tedious as it requires a lot of configuration and tuning. Quarkus helps significantly on this front, as we demonstrate throughout this book.

The important decision applications developers need to make is whether they should stick to JVM (OpenJDK) or make a move to the GraalVM native compilation. The correct answer is that there is no "one size fits all" solution. Services that need to optimize on startup time (e.g., serverless) prefer native compilation, while the ones that need to optimize the throughput (microservices) might perform better on JVM.

For Quarkus, it is essential to provide a framework that can work equally well in both scenarios. It provides almost all required configurations for the successful GraalVM compilation of your application which makes it very easy to switch between the JAR and native. The build of native executable is then trivial as it only requires using a flag at build time without needing additional code changes or configuration files.

1.3.3 Libraries and standards

The last needed part of the definition is the best of breed Java libraries and standards, which refers to the wide range of popular libraries and enterprise Java standards supported by Quarkus both in JVM and native mode. As we mentioned before, compiling applications into native binaries is a tedious process. Adding third-party libraries to your application makes it even more complex. Quarkus provides native compilation support for the most popular libraries and standards implementations available in the Java ecosystem. Furthermore, this support is not limited just to the native mode. Even in the JVM mode, Quarkus defines sensible configuration defaults (convention over configuration) and provides innovative features improving the manipulation and ease of use of the library in your code that you'll discover in the chapters to come.

Quarkus uses standards like MicroProfile (<https://microprofile.io/>) and Jakarta EE (Enterprise Edition, <https://jakarta.ee/>) and popular open-source frameworks such as Hibernate, Vertx, Apache Camel, or RESTEasy (<https://hibernate.org/>, <https://vertx.io/>, <https://camel.apache.org>, <https://resteasy.dev/>). This allows developers to reuse their expertise and years of practice with these libraries when they start working with Quarkus.

Open-source standards alleviate the need for applications to be tightly coupled to a single vendor since multiple vendors implement the standard. Practically, this means that teams can move from other frameworks supporting MicroProfile or Jakarta EE standards to Quarkus without the need to re-implement everything from scratch. This is not just because the users utilize the same APIs they already know, but also because it presents easier migration paths of existing modules. For instance, if the application already uses standards like MicroProfile, the migration might not even require code changes.

What if we told you that you could even port chunks of code written using Spring Boot APIs? Quarkus also provides limited support for a few Spring APIs (e.g., dependency injection, data, or web) to ease the migration paths for developers that need to adjust to the MicroProfile and Jakarta EE APIs, which might be new to them.

1.4 Principles of Quarkus

Now that we've broken down the definition of Quarkus, let's discuss a few more important principles that Quarkus builds upon:

- Imperative and reactive programming seamlessly connected together
- Making developers' lives easier

1.4.1 Imperative and reactive programming seamlessly connected together

Most Java developers are used to writing code in an imperative model, which involves writing a sequence of statements that describe how the program operates. Lately, an alternative paradigm called reactive programming has been gaining popularity. Reactive programming is a paradigm that utilizes asynchronous data streams that allow your software to be performant, scale dynamically, sustain heavy load, and react to events declaratively.

Quarkus has excellent support for both imperative and reactive programming. Under the hood, Quarkus has a reactive engine. However, it doesn't force you to use the reactive programming model. Users can still program imperatively or even combine the two paradigms (even in the same application). Using "just enough reactive" is helpful for developers and teams that are new to the reactive programming model and need to ease into it.

Reactive programming has become an important alternative to the imperative paradigm that needs to be considered when the developed application expects to be responsive and scalable also under heavy user traffic. This is why we demonstrate the use of reactive programming in the example application developed throughout the book.

1.4.2 Making developers' lives easier

We can devote whole chapters to talking about performance characteristics, programming paradigms, and standards, but what good are they for if you, the developer, can't enjoy what you are doing? People may argue about the importance of developer experience. Still, if you take a moment to think about it, you'll realize that many software stacks and even programming languages have been created with the sole purpose of the developer experience in mind.

Quarkus brings to the table a pleasant development model which boosts developers' productivity with features that make tedious repeating software development tasks obsolete. Quarkus provides a feedback loop that allows developers to test their code as they develop it without having to restart the actual application or perform any other kind of ceremonial tasks. Throughout this book, we refer to this loop as the "Development mode" (aka Dev mode). This feedback loop not only makes the development processes faster, but it's also an essential learning tool. Developers get to see what works and what doesn't very fast, free of repetitiveness, which helps them to learn things quickly.

This kind of feedback loop is pretty common among interpreted languages (e.g., JavaScript). It allows developers to write code and see their changes take immediate effect without rebuilding or reloading. However, it is not common in compiled languages like Java. With all Quarkus features encompassed in this development loop, Quarkus provides a unique development experience similar to scripting languages. But Quarkus does this with compiled JVM languages like Java or Kotlin.

1.5 The Quarkus architecture

Let's take a closer look at the key components of the Quarkus architecture that is visualized in [Figure 1.3](#).

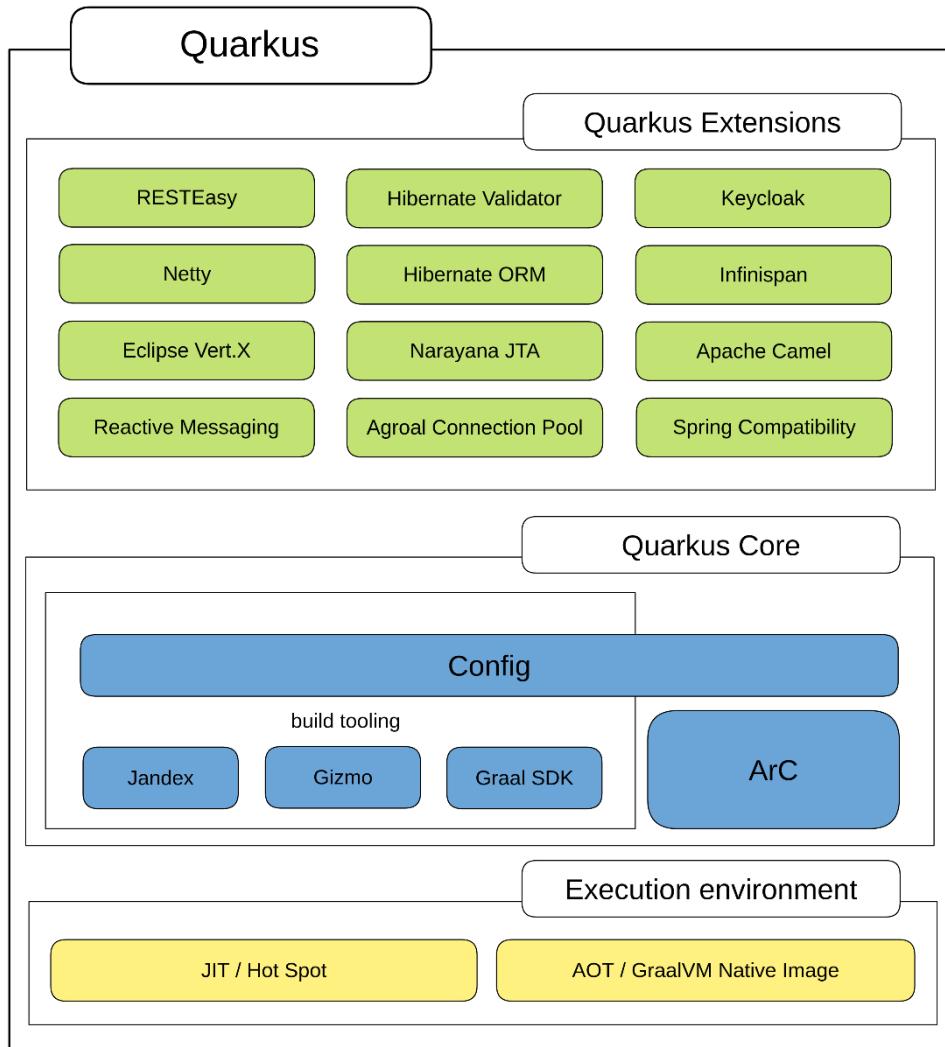


Figure 1.3 The overview of the Quarkus architecture

At the base of the figure are the components representing the execution environment. HotSpot refers to the Java Virtual Machine, and GraalVM to the tooling responsible for compiling Java code into native binaries.

On top of this layer are the critical components used in the Quarkus build tooling. *Jandex* (<https://github.com/smallrye/jandex>) is a library that provides the indexing of Java classes to a class model representation that is memory efficient and fast to search. It also allows the build tool to do all the heavy lifting related to code metadata extraction so that it doesn't have to happen at runtime.

Gizmo (<https://github.com/quarkusio/gizmo>) is a bytecode manipulation library. Bytecode is the output of the Java compiler the JVM executes. Tools and frameworks that need to generate code either generate source code that is then compiled into bytecode or generate the bytecode directly later during runtime. *Gizmo* is the library used by the Quarkus build tools to generate bytecode during build time. An example use of *Gizmo* is to produce code needed by *ArC* (Quarkus extension, see later), the dependency injection framework of Quarkus. *ArC* implements the *CDI* specification (<https://www.cdi-spec.org/>), but all the dependency resolution and wiring are processed at compile time, following the general philosophy of Quarkus tooling.

The included Graal SDK (Software Development Kit) provides integration for the native image builds. The last remaining component of the core layer is related to the configuration. Quarkus presents a unified configuration model for everything that the application might require, including both the platform and user-specific configuration properties.

The extensions are pluggable pieces of functionality that can extend either the build time or the runtime features of Quarkus and cover a wide area of different functionalities, including integration, automation, security, and more. There are hundreds of Quarkus extensions available. Individual applications are free to pick and choose only the functionalities that they require, which has a direct impact on the application size and processing speed.

1.6 Alternative frameworks

This section covers high level overview of the alternative options to Quarkus, which helps you not only better understand the philosophy of modern Java application stacks but also highlights why Quarkus is the right choice for you.

Doing a side-by-side comparison and bench-marking is beyond the scope of this section, but it's safe to claim that Quarkus has top performance characteristics, both in JVM and native mode.

1.6.1 Spring Boot

Spring Boot (<https://spring.io/projects/spring-boot>) is a framework for creating standalone Java applications. It extends and modernizes the Spring Framework by moving from the XML-based configuration model to an annotation-based one. On top of that, it comes with many sensible defaults out of the box, further reducing the amount of the needed boilerplate code (convention over configuration), which allows users to override only the required functionality provided by the framework. This is often referred to as an opinionated approach. This approach affects not only the configuration but also the selection of APIs that Spring supports. For example, Spring doesn't support many MicroProfile specifications, and it usually isn't the recommended approach for the few that it does. The most prominent example is REST support. While JAX-RS, a specification under MicroProfile, is optionally supported by Spring, many users choose to use the Spring REST controller because it is a part of the Spring ecosystem.

Another aspect of the Spring modernization introduced by Spring Boot and adopted by pretty much all frameworks covered in this section is the concept of the executable JAR. An executable JAR (aka fat JAR or über JAR) is a Java archive that contains everything the application needs, including classes, resources, and dependencies. This allows us to execute it like a regular binary (`java -jar`).

The Spring ecosystem has been around for over 20 years now. This comes with its pros and cons. It has matured, and it has a great and vibrant community around it. On the other hand, it has been designed around techniques that take a toll on runtime performance. So, compared to the new generation of frameworks like Quarkus and others discussed in this section, it is sometimes harder to align to recent trends such as native compilation for Spring Boot.

Spring provides support for native images with GraalVM since Spring Framework 6 and Spring Boot 3. The native compilation can, however, be problematic. Quarkus was created as a build time processing framework from the start, so the problems that Spring still faces regarding GraalVM compilations are often resolved in Quarkus (e.g., reflection). Additionally, Quarkus extensions are also able to provide any GraalVM native compilation specific information because they know what kind of code the user writes. This eliminates the need for users to provide this information themselves.

As already mentioned above, Quarkus also provides a set of extensions with Spring APIs like web, data, or security. These extensions significantly help with the transition from Spring Boot to Quarkus.

1.6.2 Micronaut

Micronaut (<http://micronaut.io>) is a modern Java-based application framework that emphasizes microservice architectures and serverless deployments.

Traditional Java frameworks heavily use reflection and classpath scanning, which adds overhead to memory footprint and startup time. This is not the case for frameworks like Quarkus or Micronaut which take the more modern approach called ahead-of-time compilation, which offers noticeable performance gains. This approach also optimizes the application startup time. Why is startup time important? In some deployment scenarios, for instance, with serverless deployments, the startup time is one of the most important metrics as it can be potentially added to the request overhead. So, a multi-second startup time that used to be the norm a few years back is not acceptable for such applications.

In terms of native support, Micronaut has decent support for native images. The image can be created by specifying a build flag or a separate Gradle task, similar to Quarkus. It provides configuration options that users can specify to tweak the native compilation. However, as of this writing, Micronaut doesn't guarantee native image compatibility for the third-party libraries, which Quarkus does if the library is integrated as an extension (we will learn about extensions in the following chapter).

1.6.3 Helidon

Helidon (<https://helidon.io/>) is a project for building Java-based microservices. Helidon feels more like a library and less like a framework. This stems from the fact that developers can use it without the inversion of control. Still, inversion of control is possible if the user decides to use it. This creates two distinct flavors of the Helidon: the standard edition (SE) and the MicroProfile edition (MP), with the former being more of a library and the latter more of a framework. The user has to decide on project creation as the two flavors are quite different in terms of dependencies and style.

Likewise, as with Quarkus or Micronaut, the native compilation can be enabled by the build time property, and it is available with both Helidon SE and MP. Creating native images with third-party libraries is supported for the libraries for which Helidon provides integrations.

1.6.4 Framework summary

The direction and goals that all modern frameworks take are pretty clear: helping developers write resource-efficient applications with fast startup times optimized for cloud deployment. Compared to the competition, Quarkus delivers top performance and the most concise story around the native image compilation (authors opinion), making it as transparent as possible to take care of its own optimization and the optimization of supported libraries. Additionally, it's faster to develop and explore Quarkus features due to its Dev mode. Quarkus brings a sense of fulfillment and confidence for any productive developers because they can quickly see the fruit of their labor free of the overhead of traditional Java development workflows.

1.7 Building the Acme Car Rental application using Quarkus

Throughout this book, we build a real-life application called Acme Car Rental. Most chapters of this book will contribute some parts of the functionality, bringing everything together by the end. An application like car rental is realistic and broad enough to cover all the different technologies explained in this book. And it is a change from the to-do apps, blogs, and shops, which software stacks often use for their examples. It can also be architected using microservices.

Microservice architecture is not a panacea, but it simplifies the teaching process as it cleanly separates different parts of the system. You can also argue that refactoring microservices to monolith (if we would need to do so) is more straightforward than the other way around.

You can find the code of the Acme Car Rental application at <https://github.com/xstefank/quarkus-in-action>. The repository subdivides the content into directories per chapter. Each directory contains the final version of the system by the end of the respective chapter. You might also analyze the commit history, where each commit provides even more granular code additions (per individual sections).

The provided code is excellent reference material. However, we chose to write this book so that you, the reader, can follow the development of the car rental system with us. It isn't required to code with us, but it can give you a very different experience. You may decide for yourself what is your preferred practice.

1.7.1 Use cases

To get an idea of the car rental application's essential features, let's evaluate the primary use cases that cover user management, billing, fleet management, and integration areas. It is important to understand the problems we are solving first before diving into the actual solutions.

The car rental system requirements include an interface from which its employees can look up inventory and perform bookings. A modern one must also expose this functionality to its customers via the web or mobile so that customers can browse, book, and pay for cars. This is visually represented in [Figure 1.4](#).

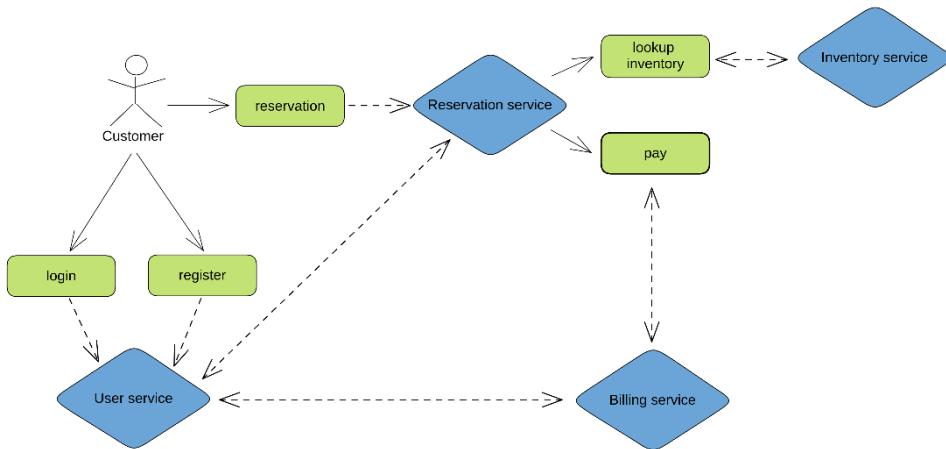


Figure 1.4 Modeling actions performed by customers

By analyzing how the customer utilizes the system, we can differentiate two separate use cases: user management and reservation making. User management relates to user login and register options provided by the User service. This service acts as the user entry API that separates the rest of the services. Making the reservation is a more complicated process for which the customers need to log into the system. The Reservation service then provides cars received from the Inventory service filtered to only the available cars for the customer-selected time period. If the user decides to make the reservation, the Reservation service calls the Billing service in order to proceed with the payment.

What about employees? What is their interaction with the system now that most of the functionality is passed directly to the customers? [Figure 1.5](#) provides a visual representation of the employee use cases and how they tie into the core system services.

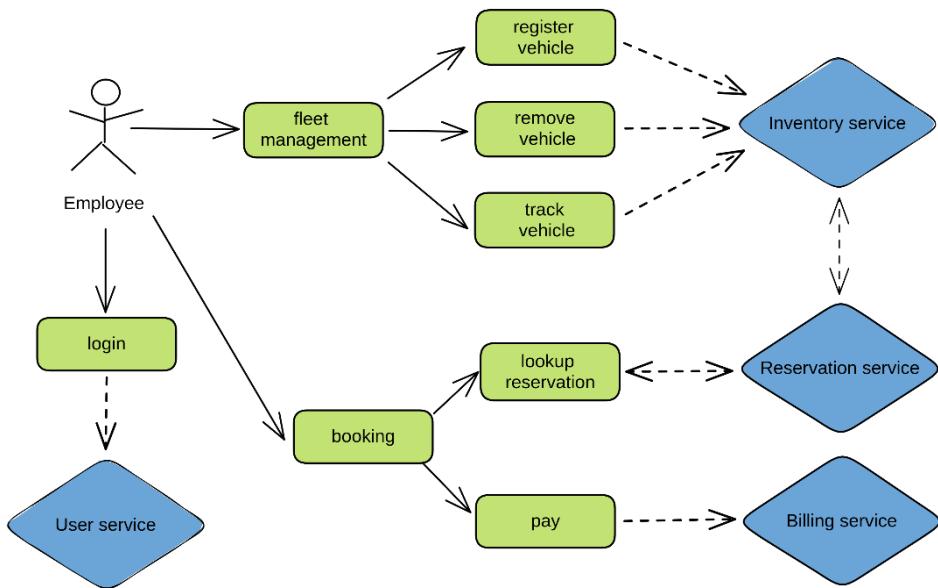


Figure 1.5 Modeling actions performed by employees

Since renting a car requires interaction with an employee to verify the validity of the reservation, employees must have access to the system. An employee also performs other tasks, such as fleet management. Employees can register and remove vehicles through the Inventory service. They can also perform bookings, a multi-step process that involves looking up the reservation and paying through the Billing service.

The Acme Car Rental system is relatively simple on purpose. It represents an adequate complexity problem to solve. We want to focus on explaining the Quarkus features without the need to provide genuine business value. However, this surely doesn't mean that such a system wouldn't make it in the real world.

1.7.2 Architecture

The previous section identified core actors, use cases, and potential services. If we are to embrace the microservices architecture, we compose the car rental system as a set of standalone applications. Each application represents an independent service and uses a decentralized data store. Services communicate using multiple channels, including REST, GraphQL, gRPC, RabbitMQ, and Kafka. [Figure 1.6](#) shows how the services communicate with each other.

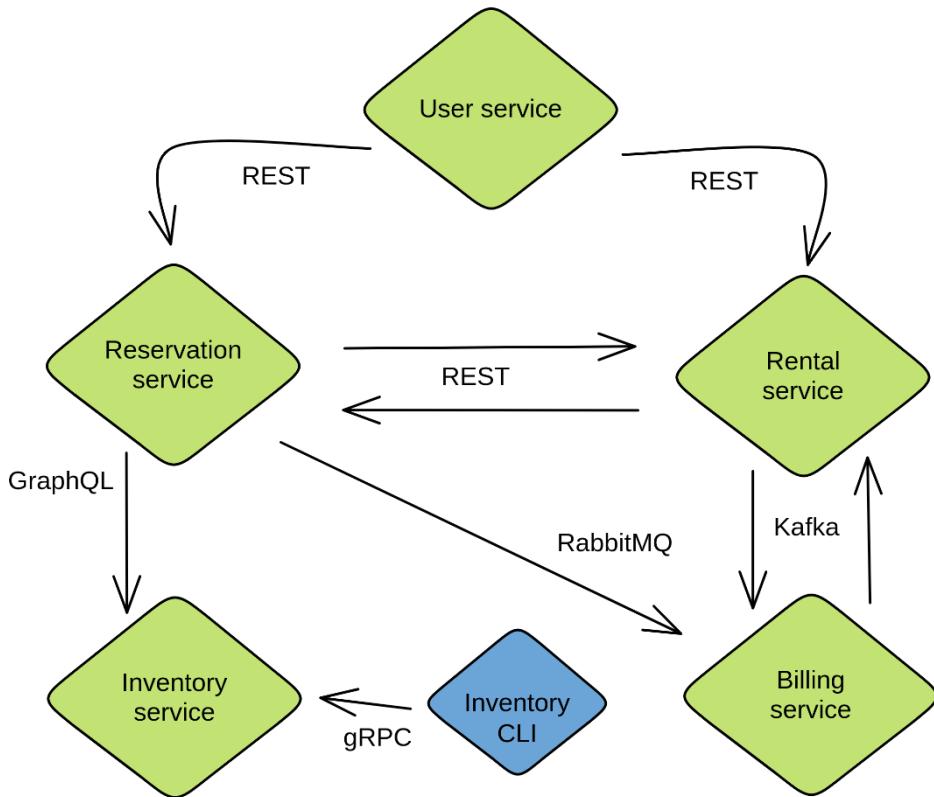


Figure 1.6 Car Rental architecture

Real-life applications usually limit themselves to just a couple of different communication technologies, depending on their needs. However, we designed the car rental application for teaching purposes and thus it does not have such limits. It covers as many communication technologies as possible. The same applies to database technologies. Each service utilizes different data store ranging from traditional relational to NoSQL (non-SQL) databases.

Since microservices systems rarely consist of only business services, our car rental also contains a few third-party services (not demonstrated in the diagram) that provide some functionality for the whole system. Apparent examples would be the Kafka and RabbitMQ brokers that will propagate the messages between our services. But our car rental system also includes other services, for instance, the metrics registry (Prometheus) or the traces collector (Jaeger) integrated with all business microservices to collect their metrics or call traces respectively.

We need to point out the Inventory CLI application at the bottom part of the architecture diagram. It represents a standalone command line Quarkus application used to group operations over cars in the Inventory service. All other services compose together as Quarkus microservices expected to run simultaneously to provide the overall car rental system functionality.

We also need to mention that the User service provides a simple user interface that allows users to log in and make reservations.

Acme Car Rental represents a small but coherent system. For real-world application, it might be too complex since it relies on many software stacks which are quite different. However, since our goal is to explain how Quarkus integrates with all these technologies, car rental serves as excellent learning material.

1.7.3 Implementation

For the implementation of the Quarkus services in Acme Car Rental, we develop in Java as our primary language. However, we need to also point out that Quarkus supports two alternative JVM languages—Kotlin and Scala. We use the Java Development Kit (JDK) version 21, which is the latest LTS (Long Term Support) release version available at the time of this writing.

TIP Quarkus 3.5.0 requires minimal JDK version 11.

The primary build tool we use in car rental is Maven. A similarly popular build tool for JVM projects is also Gradle. The required version of Maven is 3.8.1+. Nevertheless, Quarkus often comes with Maven integrated into the projects (Maven wrapper), which handles Maven versioning for you.

Since each chapter examines a particular technology, we want to prioritize the teaching of its software stack and its Quarkus integration in each respective chapter. Nonetheless, for the completeness of your Quarkus learning, we explain the use of Kotlin and Scala together with the Quarkus application building on top of Gradle in the Appendix Alternative Languages and Build Tooling.

1.8 Wrap up and next steps

This initial chapter introduced the Quarkus framework as a capable modern Java applications runtime. We summarized many exciting features that streamline the requirements of the current cloud-native application development, many of which are available only in Quarkus. We also designed the Acme Car Rental application that demonstrates the functionalities we discuss throughout this book.

In the rest of the book, we focus on individual development enhancements and the diverse portfolio of technologies that Quarkus supports. In the next chapter, we start by creating our first Quarkus application.

1.9 Summary

- Quarkus is a full-stack Java framework for developing modern cloud-native enterprise applications (monolithic, microservices, or serverless).
- Quarkus has been built from the ground up with containers and Kubernetes in mind.
- Since Quarkus supports many well-known standards and popular libraries, the learning curves are often short as users can utilize the well-known APIs.
- Quarkus provides excellent support for building native binaries with GraalVM.
- The new development workflow called Dev mode introduced in Quarkus improves user productivity and can be a great learning tool.

2

Your first Quarkus application

This chapter covers

- Creating Quarkus applications
- Analyzing the content of Quarkus applications
- Demonstrating the packaging and running of Quarkus
- Explaining Quarkus extensions

Imagine that you are asked to evaluate Quarkus for your current company. Your boss asks you to get acquainted with the tools required to run and package Quarkus applications, different strategies for the ease of the learning curve for your colleagues, and of course, a small performance measurement that can prove that Quarkus is the right choice. You should evaluate the extensibility and usability of the Quarkus with the demonstration of simple application deployment on your computer and be able to state the reasons why you would choose Quarkus as your application runtime.

Any first contact with every new technology represents an interesting experience — either positive or negative. After the previous chapter, you should have a good idea of what Quarkus is and what it aims to achieve. Now is the time to get our hands dirty.

In this chapter, we will learn how easy and fast it is to get started with Quarkus development. Following several easy steps we will generate, package, and run a working Quarkus application. Exploring what the Quarkus development tools generate for us out-of-the-box will also allow us to take this experience one step further—we will package the application not only into an executable JAR format which can be run with `java -jar` but also into a native executable built with GraalVM. As a small bonus, Quarkus also generates Dockerfiles, allowing us to build Docker images for all available packaging formats. This is just what Quarkus generates out-of-the-box, if you add more extensions it can generate even more useful files (e.g., Kubernetes resources). So, let's dive right in and create your first Quarkus application.

2.1 Generating Quarkus applications

Quarkus is a very developer-focused framework. It tries to provide developers with a number of available options to generate a starting application depending on user preference. In this section, we explore different choices available for generating a new Quarkus application. You can easily apply these options to start building your business application as demonstrated with the Acme Car Rental services.

We can create Quarkus applications in three distinctive ways:

- Quarkus Maven plugin
- Quarkus CLI (Command Line Interface)
- The website's graphical starter interface

You may ask why there are three different options available for the same task. Quarkus chose this because the target developer audience differs broadly in inclinations to various technologies. For instance, some programmers don't like to click-configure anything, so using a graphical interface is probably impractical. Conversely, Quarkus CLI requires additional steps to get the CLI installed. The Maven plugin is a good alternative because Maven is also broadly used as the build tool for Quarkus projects. However, there is an evenly big community of engineers that prefer to use Gradle as the build tool of their choice. And in that case, they probably don't have Maven installed.

In the following sections, we will establish how Quarkus integrates with these tools to create new Quarkus applications that generate Java projects based on the Apache Maven. If you prefer the Gradle build tool, we mention the necessary flags that tell Quarkus to create Gradle-based projects after each command.

TIP Quarkus applications can also be generated through other means (e.g., integrated development environments (IDEs)) that provide wrappers around the tooling mentioned in this chapter.

2.1.1 Generating Quarkus applications with Maven

To generate the Quarkus application with Apache Maven, we utilize the Quarkus Maven plugin (`io.quarkus:quarkus-maven-plugin`) and its `create` goal. The following `mvn` command invocation will create a new Quarkus project generated in the `quarkus-in-action` directory.

```
$ mvn io.quarkus.platform:quarkus-maven-plugin:3.5.0:create \
-DplatformVersion=3.5.0 \
-DprojectGroupId=org.acme \
-DprojectArtifactId=quarkus-in-action \
-Dextensions="resteasy-reactive"
```

The `quarkus-maven-plugin` invocation on the first line requires the full Quarkus Maven plugin *GAV* (`groupId`, `artifactId`, and `version`) because we are creating a new Maven project without any initial setup. By default, Quarkus uses the latest available Quarkus platform version. So we need to explicitly set older version used in this book declared by the `-DplatformVersion` system property. Feel free to continue with the latest version by removing this property from the `mvn` command. Next, we set the generated project Maven `groupId`, `artifactId`. Lastly, we specify the optional extensions list to be included in the generated project. This command generates a Maven project in the `quarkus-in-action` directory.

The `resteasy-reactive` extension represents a default REST layer implementation. It implements the Jakarta RESTful Web Services (JAX-RS) API used in Quarkus. This extension is included even if you don't specify any extension parameters. The snippet includes this extension explicitly to provide an example of defining extensions. If you also specify the `-DnoCode` property, the plugin won't generate any code, so it also doesn't include `resteasy-reactive` extension.

TIP An important thing to note is although the term `reactive` is in the name of the extension, this does not mean that the services we create with it need to be reactive. Although one of the key promises of Quarkus is that it makes developing reactive applications much simpler, reactivity is an optional feature that we will cover in later chapters.

If you need to generate a project utilizing the Gradle build tool, you can add `-DbuildTool=gradle` system property.

If you don't specify the project's `groupId` or `artifactId` with the system properties, the `create` goal invocation prompts you for this information during its invocation. Don't worry about the extensions if you don't know the extensions yet. We will cover them in detail later in this chapter. And this is all you need to do to get the base project code generated with the Quarkus Maven plugin.

TIP You can use the mvn io.quarkus.platform:quarkus-maven-plugin:3.5.0:help -Ddetail=true -Dgoal=create command to check all available parameters of the create goal.

2.1.2 Generating Quarkus applications with CLI

Quarkus Command Line Interface (CLI) is an intuitive command line program called quarkus that allows users to utilize basic Quarkus operations like generating application code, managing extensions, or launching Quarkus.

The installation of quarkus CLI is optional, however, preferred as it is utilized throughout this book. The installation instructions are available in the Appendix B Installations or at <https://quarkus.io/guides/cli-tooling>.

To generate a Quarkus application with the same setup as with the Maven plugin above, but utilizing the Quarkus CLI quarkus you need to run the following command:

```
$ quarkus create app org.acme:quarkus-in-action -P 3.5.0
→ --extension resteasy-reactive
```

TIP The -P 3.5.0 (or its long version --platform-bom) flag is only required because otherwise the quarkus CLI uses the latest available Quarkus version.

Similarly, as with the Maven plugin, to generate a project utilizing Gradle build tool, you can add --gradle flag.

This command generates the same application as the Maven command above. The result is again available in the quarkus-in-action directory. One difference from the Maven plugin invocation is that when you run this CLI command without arguments (just quarkus create app), the generated application will not ask you to provide them. It will use preconfigured defaults (for instance, the code is generated in code-with-quarkus directory).

TIP To get more information about what quarkus CLI can do, you can run commands with -h or --help flag as for instance in quarkus create app -h or just quarkus -h.

2.1.3 Generating Quarkus applications from code.quarkus.io

Next, we analyze the graphical user interface (GUI) of the Quarkus application generator available at <https://code.quarkus.io/>. This website, detailed in [Figure 2.1](#), specifies all parameters of the generated Quarkus application through a sophisticated graphical form. This UI is very useful as many users prefer to click-configure new Quarkus applications. However, this Quarkus online generator also provides a REST API that users might utilize to generate a ZIP file with the new Quarkus application remotely. Note that to open the full UI, you need to click MORE OPTIONS button.

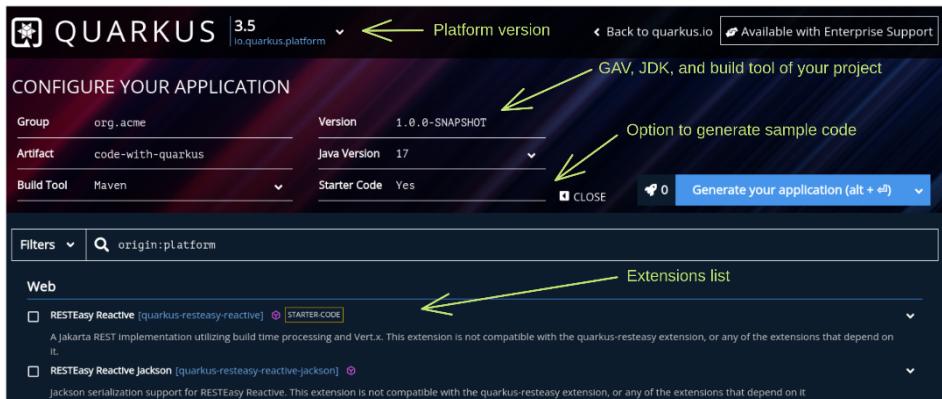


Figure 2.1 Quarkus Generator interface

This GUI provides the same project configuration capabilities of the generated Quarkus applications as the Maven plugin or the `quarkus` CLI. Starting first with the setting of the Quarkus platform version at the top of the page, you then need to set up the project GAV or you can use the default values. Next, you can choose the JDK version and decide whether to generate some starting code or an empty Quarkus project (this is implicitly enabled with the plugin and CLI). In the very long list at the end of the page, you can browse and choose individual Quarkus extensions that you want to include in your application by checking them in the extensions list.

Finally, after you click the `Generate your application` button, you are provided with an option to download a ZIP file containing your application. If you check just the `quarkus-resteasy-reactive` extension, you will generate the same application as we did with Maven and with the CLI. If you prefer Gradle, you can easily choose it from the `Build tool` dropdown menu.

The generator also provides a REST API exposed at <https://code.quarkus.io/api> root. It also provides an OpenAPI document together with the corresponding SwaggerUI that you can use to investigate available calls and options at <https://editor.swagger.io/?url=https://code.quarkus.io/q/openapi>. Utilizing this REST API, we can generate the same quarkus-in-action application with the following HTTP GET request:

```
curl "https://code.quarkus.io/api/download?  
-g=org.acme&a=quarkus-in-action&e=quarkus-resteasy-reactive"  
-o quarkus-in-action.zip
```

Unzipping the downloaded ZIP file from both the GUI and REST API download creates the same application in the `quarkus-in-action` directory which contains only the `resteasy-reactive` extension.

2.2 Contents of the generated application

No matter which way you prefer to generate the starting Quarkus application, if you add the `quarkus-resteasy-reactive` extension, you see the same results as we analyze in this section. Remember that we are focusing on the Maven project structure, so if you generated the Gradle project, the result is slightly different, but if you are a Gradle user, you know the differences from Maven.

When you learn about new technology, it is best to start with the general overview. If you list the files of the generated `quarkus-in-action` application, the directory structure appears as shown in the [Figure 2.2](#). Depending on your background, this might seem like a lot or maybe not that much. But rest assured that it is everything that you need to get working with Quarkus.

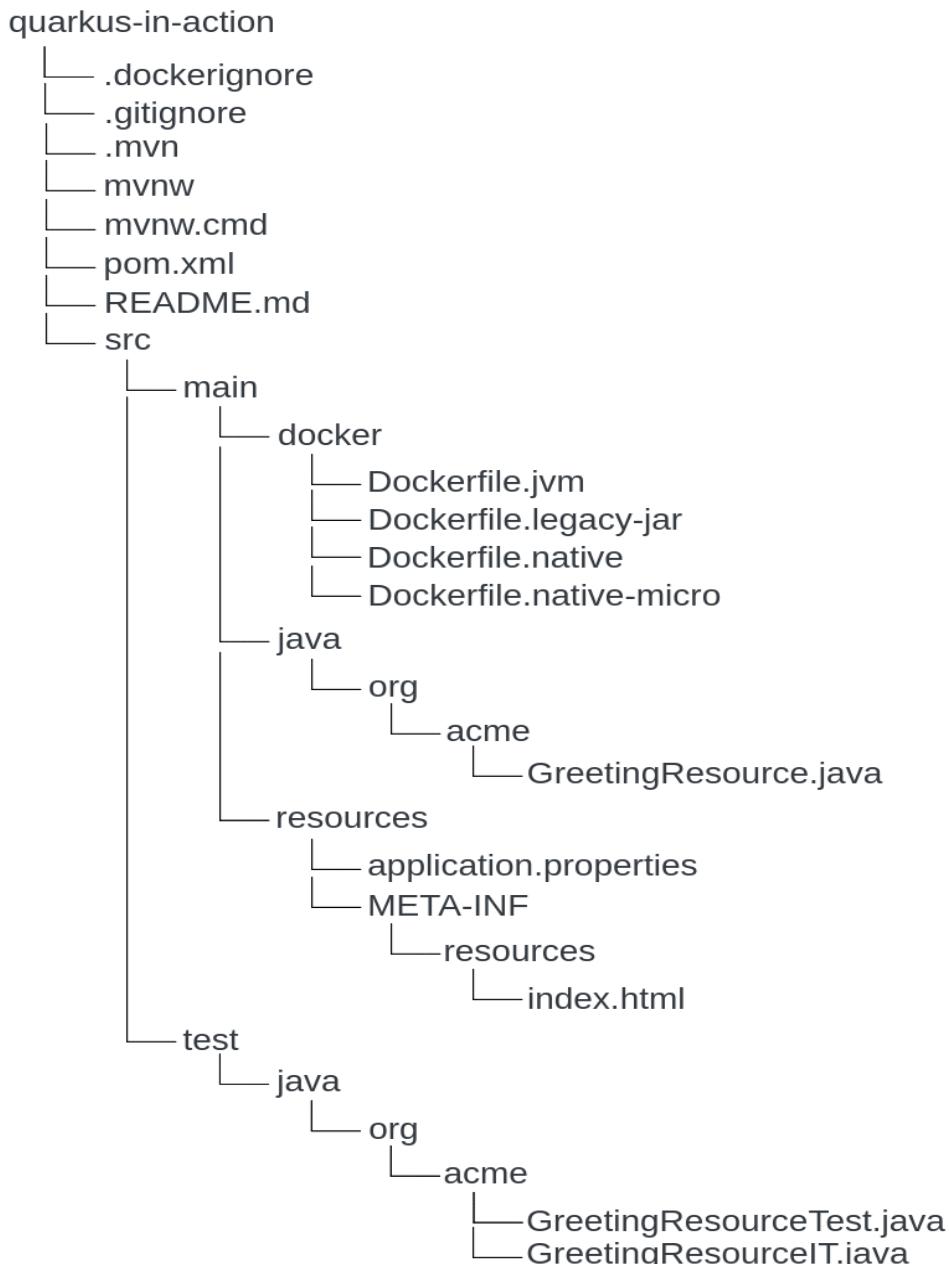


Figure 2.2 Quarkus application directory structure

The generated content follows the standard structure of the Java Maven project. The `pom.xml` file represents the Maven build configuration file. In the directory `src/main/java`, we develop the Java source code files (`.java`). In `src/main/resources`, we place other non source code related files as properties (configuration) or HTML pages. Quarkus additionally creates also the `src/main/docker` directory (not typically included in Maven projects) in which it provides generated Dockerfiles. The test source code is composed in the `src/test/java` directory.

The remaining set of files located in the root directory of the generated Quarkus project are either related to the Maven wrapper that is included within the generated application (e.g., `mvnw` or `mvnw.cmd`) or represent useful configuration files used in common environments that Quarkus applications usually utilize (for instance, `.gitignore` in Git or `.dockerignore` in Docker).

TIP The structure might differ if you didn't set the option to generate sample code when you were generating the Quarkus application.

Let's analyze individual parts in detail step-by-step. We do not explain all the files that you can see in the [Figure 2.2](#) as they are not required for our purposes.

2.2.1 Maven `pom.xml` file

The primary build descriptor of Maven projects is available in the `pom.xml` file. Maven defines the standard structure of this XML file, which is why we focus only on the Quarkus-specific parts.

DEPENDENCIES

The most important part of the generated `pom.xml` is the `<dependencies>` element listing mostly the Quarkus extensions, all represented as Maven artifacts. [Listing 2.1](#) details this part of the `pom.xml` file.

Listing 2.1 Quarkus <dependencies> element (including extensions)

```
<dependencies>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-reactive</artifactId> #1
  </dependency>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-arc</artifactId> #2
  </dependency>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.rest-assured</groupId> #3
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

#1 Explicitly added resteasy-reactive extension.

#2 Implicitly added extensions providing the dependency injection (arc) and testing framework (junit5) respectively.

#3 Not an extension (not in io.quarkus groupId).

If you examine the contents of this element a little closer, you can identify the quarkus-resteasy-reactive extension that we explicitly added to our application when it was generated. However, there are also two other Quarkus extensions included implicitly, as shown in [Listing 2.1](#). You can identify them by groupId `io.quarkus` and the artifactId starting with the `quarkus-` prefix. If you are familiar with Maven, you might notice that the versions of these Quarkus artifacts are not defined. This is because they are all defined in the Quarkus BOM (Bill Of Materials) specified in the `<dependencyManagement>` section of the `pom.xml` file. The BOM is always versioned per a specific Quarkus platform version and contains all platform extensions together with other useful artifacts (e.g., for testing) with the correct versions. Using the BOM also ensures that all utilized extensions are guaranteed to work together.

TIP Some Quarkus extensions, particularly the ones not available in the Quarkus core platform, do not need to be versioned with the same version as the Quarkus platform. The versions of these (Quarkiverse) extensions need to be specified explicitly.

MAVEN PLUGINS

In the `<build>` section of the generated `pom.xml`, we can also locate the definition of the `io.quarkus.platform:quarkus-maven-plugin` - the same plugin we use to generate the Quarkus application. [Listing 2.2](#) details its definition. The main purpose of this plugin is to package the application into the correct target artifacts and manage the Dev mode's lifecycle (we will detail the Dev mode in Chapter 3). Because it is included in the `pom.xml` build descriptor of our project, we can refer to it simply as `quarkus` without the need to specify also the full GAV as when we generated the application in section [2.1.1](#). The version of this plugin also aligns with the platform version of Quarkus specified for this project. The individual goals are specific to Quarkus lifecycle, and users don't need to modify them.

Listing 2.2 Quarkus Maven plugin declaration

```
<plugin>
  <groupId>${quarkus.platform.group-id}</groupId>
  <artifactId>quarkus-maven-plugin</artifactId>
  <version>${quarkus.platform.version}</version>
  <extensions>true</extensions>
  <executions>
    <execution>
      <goals>
        <goal>build</goal> #1
        <goal>generate-code</goal> #2
        <goal>generate-code-tests</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

#1 Generates the final application artifacts (e.g., the executable JAR).

#2 Source code generating before the source compilation (e.g., gRPC proto files).

The other three defined plugins in the `<build>` section—`maven-compiler-plugin`, `maven-surefire-plugin`, and `maven-failsafe-plugin` allow Quarkus to pass some additional configuration parameters to the Java compiler and the test execution, respectively. Feel free to check the official documentation at <https://maven.apache.org> if you want to learn more about these plugins.

PROFILES

The `<profiles>` section of the `pom.xml` contains the definition of the native Maven profile. [Listing 2.3](#) contains the declaration of this profile as it is defined in the `pom.xml`. This profile modifies Quarkus's build to also package your application as a native GraalVM executable file that we learn about in the following section [2.4](#). The important part is the property `quarkus.package.type` which is set to `native`. This tells the Quarkus Maven plugin to package the application into a native executable. The `skipITs` configuration is connected to the execution of the native tests run as integration tests (the native executable is started separately and the test calls the exposed API).

Listing 2.3 Quarkus native Maven profile

```
<profile>
  <id>native</id>
  <activation>
    <property>
      <name>native</name> #1
    </property>
  </activation>
  <properties>
    <skipITs>false</skipITs>
    <quarkus.package.type>native #2
    </quarkus.package.type>
  </properties>
</profile>
```

#1 Allows to run the profile also with `-Dnative` (in addition to `-Pnative`).

#2 The Quarkus package type configuration property.

2.2.2 Generated code and resources

If the Quarkus application generates with the option to enable sample code, you can locate the generated code in the `src/main/java/` directory. [Listing 2.4](#) contains the constructed `org.acme.GreetingResource` class which is created because we included `resteasy-reactive` extension.

Listing 2.4 GreetingResource class—generated sample JAX-RS endpoint

```

package org.acme;

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

@Path("/hello") #1
public class GreetingResource {

    @GET #2
    @Produces(MediaType.TEXT_PLAIN) #3
    public String hello() {
        return "Hello from RESTEasy Reactive"; #4
    }
}

```

#1 The HTTP path on which this resource is exposed.

#2 The HTTP method (GET) used to invoke this method.

#3 The media type for the Content-Type HTTP header.

#4 Returns Hello from RESTEasy Reactive text for GET /hello HTTP requests.

This class composes a sample Java API for RESTful Web Services (JAX-RS) server that exposes a single endpoint at `/hello` path. The server is defined by a few annotations specified in the JAX-RS specification (`jakarta.ws.rs.*`). The `GreetingResource` exposes a single HTTP GET `/hello` call by its API.

Moving on to the `src/main/resources` directory, we discover a very important file called `application.properties` that contains our Quarkus configuration (Chapter 3 details the configuration options of Quarkus applications). This folder also includes a simple HTML page in the `META-INF/resources` subdirectory, which provides a sample HTML that is displayed when you access Quarkus application at the root path in your browser.

The last subdirectory under the `src/main` directory, `docker`, separates four different Dockerfiles for building Docker images for various Quarkus packaging types. Here you can, for instance, discover and choose between the default Dockerfile for the JVM image in `Dockerfile.jvm` or the Dockerfile for the native executable in `Dockerfile.native`.

In the only test subdirectory `src/test/java`, which contains the Java test source files, two Java classes are generated in the same package `org.acme`. The `GreetingResourceTest` class provides a sample of the Quarkus test class. Its source code is available in the [Listing 2.5](#).

Listing 2.5 Quarkus test example with REST Assured

```

package org.acme;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
            .when().get("/hello")
            .then()
                .statusCode(200) #1
                .body(is("Hello from RESTEasy Reactive")); #2
    }
}

```

#1 Verifies that the GET /hello HTTP request is successful (200 OK).

#2 Verifies that the GET /hello HTTP request actually returns Hello from RESTEasy Reactive.

The test class is annotated with `@QuarkusTest` annotation that denotes it as a Quarkus test and by utilizing JUnit 5 and REST Assured APIs it calls the JAX-RS endpoint exposed by the `GreetingResource` class. It then asserts that the server responded successfully (200) and that the response's content is equal to the String `Hello from RESTEasy Reactive`.

The second test class, `GreetingResourceIT` extends the first test class `GreetingResourceTest` and it is annotated with the `@QuarkusIntegrationTest` annotation to mark it for execution with the built artifacts—either the executable JAR, native image, or container (if built with Quarkus).

Listing 2.6 Quarkus integration test

```
package org.acme;

import io.quarkus.test.junit.QuarkusIntegrationTest;

@QuarkusIntegrationTest
public class GreetingResourceIT extends GreetingResourceTest {
    // Execute the same tests but in packaged mode.
}
```

As demonstrated in [Figure 2.3](#), with integration tests, the test execution JVM runs separately from the Quarkus application artifact. The test only calls the application's exposed API. This is also the only way to execute tests with the native image or container. By extending the `GreetingResourceTest` we aim to execute the same tests for both traditional (Java unit tests, in our case, even in the same JVM) and integration test executions. The integration test isn't required to extend a unit test if the tests don't apply in both modes.

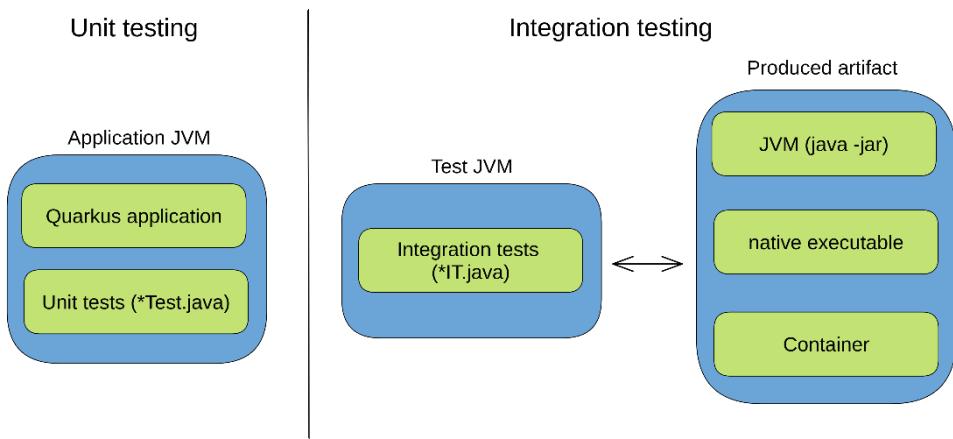


Figure 2.3 Unit versus integration testing with Quarkus

2.3 Running the Quarkus Application

We have learned how Quarkus functions and what the Quarkus application contains. Now it's time to run the Quarkus application we generated in the previous sections and investigate where Quarkus excels.

If you are not already in the generated `quarkus-in-action` directory, change your working directory to it. Before running any Maven (or Gradle) project, you need to package it. But does this need to be done with Quarkus too? The answer is both yes and no. One of the best Quarkus features, which makes the development with Quarkus such an enjoyable experience, is called the Development mode (also called Dev mode). In short, Dev mode is a continuous run of your Quarkus application that allows you to change your application code dynamically. With just a simple browser refresh (or any HTTP request), it recompiles and reruns the whole application to display the results of your changes in mere milliseconds. The point here is to demonstrate that Quarkus applications can run without manual building/packaging operations commonly required for Java applications.

To start the Quarkus application in Dev mode, you can run the command available in the following snippet, where we invoke Quarkus Maven plugin goal `dev`. We reference `quarkus-maven-plugin` only as `quarkus` this time. This is possible because our Maven project configuration, namely the `pom.xml` file, contains the Quarkus plugin definition.

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

If the Quarkus application builds on top of the Gradle tool, then you can run this command instead:

```
$ ./gradlew --console=plain quarkusDev
```

The same result can be achieved with the `quarkus` CLI. The invocation is simply run as:

```
$ quarkus dev
```

You might choose one of the previous tooling-specific commands depending on your preference. However, the `quarkus` CLI gives you the benefit of working in a unified way with both Maven and Gradle projects in the same way.

If you run the Dev mode for the first time, you will see the following question:

```
-----
--- Help improve Quarkus ---
-----
* Learn more: https://quarkus.io/usage/
* Do you agree to contribute anonymous build time data to the Quarkus
  community? (y/n and enter)
```

You can decide yourself whether you want to contribute the build time statistics from your local machine or not but we recommend you answer this question when you're asked because otherwise it times out after 10 seconds, and you will be asked again next time you start the Dev mode.

TIP When you answer this question, quarkus CLI creates a new file `io.quarkus.analytics.localconfig` in your `$HOME/.redhat` directory where you can change this setting if you change your mind.

When you execute the `quarkus-in-action` application in Dev mode, you can see the output similar to [Listing 2.7](#) in the console.

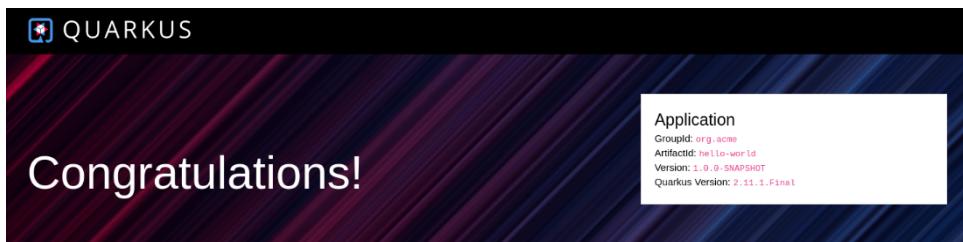
Listing 2.7 The output of the Quarkus application started in Dev mode

```
$ quarkus dev
...
_____
--\_\_\_\_/_/ |/_/|/_/|_/
- /_/_/_/_/_/ | / , _/ ,< /_/_/\ \
--\_\_\_\_/_/ |/_/|/_/|_|\_\_/_/
2023-07-11 09:46:31,977 INFO [io.quarkus] (Quarkus Main Thread)
quarkus-in-action 1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.2.0.Final)
started in 1.868s. Listening on: http://localhost:8080

2023-07-11 09:46:31,981 INFO [io.quarkus] (Quarkus Main Thread) Profile
dev activated. Live Coding activated.
2023-07-11 09:46:31,982 INFO [io.quarkus] (Quarkus Main Thread) Installed
features: [cdi, resteasy-reactive, smallrye-context-propagation, vertx]
```

You might notice that we have two additional extensions listed in the "Installed features" list at the end of the output - `smallrye-context-propagation`, responsible for correctly passing contexts between threads and `vertx`, the underlying reactive engine of Quarkus. These extensions are implicitly added to the `quarkus-in-action` application as dependencies of `resteasy-reactive`.

Our project is now up and running in the Quarkus Dev mode. You can now open <http://localhost:8080> in your browser to check the generated welcome page of your first Quarkus application. It will look like the one in [Figure 2.4](#). It contains some basic information about different resources in your application (we learned about them in the section [2.2](#)) and pointers to the documentation.



You just made a Quarkus application.

This page is served by Quarkus.

[VISIT THE DEV UI](#)

This page: <src/main/resources/META-INF/resources/index.html>
 App configuration: <src/main/resources/application.properties>
 Static assets: <src/main/resources/META-INF/resources>
 Code: <src/main/java>
 Generated starter code:
 RESTEasy Reactive Easily start your Reactive RESTful Web Services
 > @Path: /hello
 > [Related guide](#)

Figure 2.4 Quarkus welcome page

Additionally, you can also access the exposed endpoint at <http://localhost:8080/hello> in your browser or with any similar tool able to make HTTP GET request as cURL (`curl http://localhost:8080/hello`), HTTPie (`http :8080/hello`). Note that if you want to use these command line tools, they need to be installed separately.

Opening the URL in browser or running any of the above commands gets back 200 OK HTTP response with Hello from RESTEasy Reactive content from our JAX-RS method `GreetingResource#hello` as we demonstrate in the following snippet.

```
$ http :8080/hello
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
content-length: 28

Hello from RESTEasy Reactive
```

Quarkus Dev mode is very useful for the development cycles, but when you are ready to run your application in production, you need to package it to ship it to your platform. Quarkus packages with the build system that you defined your project with, either with Maven or Gradle. The package goal is executed with one of the following commands available in listing [Listing 2.8](#). The CLI again has the advantage of abstracting from the actual build tool in use.

Listing 2.8 Quarkus package commands in Maven, CLI, and Gradle

```
# Maven
./mvnw clean package

# CLI
quarkus build

# Gradle
./gradlew build
```

Quarkus package goals produce several output files in the `./target` (or `./build` for Gradle) directory which follow the standard Maven build artifacts. The output of the application's code and resources compiled from the `quarkus-in-action` project is included in the `quarkus-in-action-1.0.0-SNAPSHOT.jar` JAR. This JAR is standard Maven output. In this sense, it is not runnable.

The main Quarkus artifacts are located in the `quarkus-app` directory. This directory contains the executable `quarkus-run.jar` which you can run with `java -jar`. It also includes the `lib` directory into which Quarkus copies the application's dependencies. This is purposely done because `quarkus-run.jar` is not a fat JAR (also known as über JAR) that packages the application together with all its dependencies like you might be familiar with from other frameworks. Instead, the dependencies are externalized and referenced in the `quarkus-run.jar`'s `MANIFEST.MF` file. For instance, this is useful in the Docker image where you might want to put application dependencies, which usually do not change that often, into a separate layer beneath the application layer which in turn would contain your `quarkus-run.jar`. In fact, this is already done for you in the generated Dockerfiles (you can check, for instance, the `src/main/docker/Dockerfile.jvm` file). This means that if you are to move your executable JAR `quarkus-run.jar`, you are also required to move the `lib` folder with it to move its dependencies.

[Listing 2.9](#) demonstrates how you can run your packaged application as an executable JAR. The output looks similar to the Dev mode, but notice that the application now runs in the prod (production) mode.

Listing 2.9 Running Quarkus with java -jar

```
$ java -jar target/quarkus-app/quarkus-run.jar
_____
--/ _ \ / / / _ | / _ \ / / / / / _ /
-/ / / / / / / / , _/ , < / / / / \ \
--\_\_\_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/
2023-07-11 09:53:53,817 INFO [io.quarkus] (main) quarkus-in-action
1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.2.0.Final) started in 0.521s.
Listening on: http://0.0.0.0:8080
2023-07-11 09:53:53,840 INFO [io.quarkus] (main) Profile prod activated.
2023-07-11 09:53:53,840 INFO [io.quarkus] (main) Installed features:
[cdi, resteasy-reactive, smallrye-context-propagation, vertx]
```

And now you can access the application in the same way as we did in the Dev mode at <http://localhost:8080> or <http://localhost:8080/hello> in your browser or in the terminal.

2.4 Native compilation with GraalVM

Now that you understand that Quarkus is quite performant even with normal JDK runs (`java -jar`), it is time to take it one step further and investigate the native executions with GraalVM.

2.4.1 GraalVM

Graalvm.org defines GraalVM as “a high-performance JDK designed to accelerate the execution of applications written in Java and other JVM languages” (<https://www.graalvm.org>). It covers a lot of functionalities with its rich feature set. The main feature that is interesting for Quarkus is called the native-image. According to www.graalvm.org, native-image represents “a technology that allows compiling of Java code ahead-of-time to a binary – a native executable. A native executable includes only the code required at run time, that is the application classes, standard-library classes, the language runtime, and statically-linked native code from the JDK.”. The native executable follows the closed world assumption - it statically analyzes the execution paths of the application to provide a platform-specific, self-contained, small, and most importantly performant executable binary. Such binaries usually start in tens of milliseconds, making Java usable in very restrictive environments such as serverless architectures.

TIP The ahead-of-time compilation refers to the process of compiling Java bytecode into system-specific machine instructions.

There are three distributions of GraalVM available:

- Oracle GraalVM Community Edition (CE)
- Oracle GraalVM Enterprise Edition (EE)
- Mandrel (<https://github.com/graalvm/mandrel>)

GraalVM EE is an enterprise (paid) distribution of GraalVM CE with better performance than GraalVM CE. Mandrel is a downstream (forked) distribution of the Oracle GraalVM CE with the main goal of providing a way to build a native executable specifically designed to support Quarkus. It is thus the preferred GraalVM distribution to be used for creating Quarkus native executables. Mandrel focuses mainly on the `native-image` build tool. It doesn't provide full GraalVM toolset.

TIP Mandrel is currently supported only on Linux and Windows. If you need to build native executables on MacOS, you need to use GraalVM.

But before we move on, there is one more thing: if for any reason you are not able to install Mandrel or GraalVM, don't worry. Quarkus also provides for you a way of building your native executable in a Docker build that embeds your Maven build with integrated GraalVM inside a Docker container. But this can only build Linux based native binary that you can't directly run on different operating system (but it can run in container). This process is detailed in section [2.4.3](#). So if your operating system is Linux, you don't need to install GraalVM. But in every other case (Windows or Mac) you need to install GraalVM in order to produce native executable that are compatible with your platform.

If you need to install GraalVM locally, the instructions on how to install and configure GraalVM or Mandrel are available in the Appendix GraalVM. If you installed and configured Mandrel or GraalVM correctly, you can run the following commands available in the [Listing 2.10](#) to verify your configurations. The `GRAALVM_HOME` environment variable should be pointing to your installation of Mandrel/GraalVM. If you also use GraalVM directly, you should verify that you installed the `native-image` binary correctly.

Listing 2.10 Verifying Mandrel or GraalVM installation

```
$ echo $GRAALVM_HOME
/path/to/graalvm

# only for GraalVM CE/EE
$ $GRAALVM_HOME/bin/native-image --help

GraalVM native-image building tool
...
```

2.4.2 Packaging Quarkus as a native executable

To demonstrate the native builds, we utilize the same `quarkus-in-action` application we generated earlier in this chapter. You may remember that the generated `pom.xml` contains a Maven profile called `native` ([Listing 2.3](#)). To package your application into a native executable, you can run one of the commands available in [Listing 2.11](#) depending on the used build tool and preference.

Listing 2.11 Quarkus native compilation commands

```
# Utilizing Maven profile
$ ./mvnw package -Pnative

# Gradle needs to use the system property directly
$ ./gradlew build -Dquarkus.package.type=native

# Or quarkus CLI for general approach
$ quarkus build --native
```

The build takes longer (usually several minutes) as producing a native executable requires a lot of processing. It also requires a notable portion of memory. But that is it. [Listing 2.12](#) shows how you can run your Quarkus application now by directly running the generated binary. If you check now the generated output in the `target` directory, you will find a new artifact called `quarkus-in-action-1.0.0-SNAPSHOT-runner`. Of course, this might differ if you run this build on a different platform (`.exe`, for instance).

Listing 2.12 Running Quarkus native executable binary

```
$ ./target/quarkus-in-action-1.0.0-SNAPSHOT-runner

_____
--/ _ \V/ / / _ | / _ \V ///_// / / _/
-/ /_ / /_ / __| / , _/ ,< / /_ / \ \
--\_\_\_/_/_|/_/_/_/_|\_\_/_/_/
2023-07-11 09:58:00,705 INFO [io.quarkus] (main) quarkus-in-action
1.0.0-SNAPSHOT native (powered by Quarkus 3.2.0.Final) started in 0.035s.
Listening on: http://0.0.0.0:8080
2023-07-11 09:58:00,705 INFO [io.quarkus] (main) Profile prod activated.
2023-07-11 09:58:00,705 INFO [io.quarkus] (main) Installed features:
[cdi, resteasy-reactive, smallrye-context-propagation, vertx]
```

Did you notice how fast it started? Very impressive, right? Getting this kind of performance out of the box is undoubtedly not standard for most projects. At least not in Java.

2.4.3 Native executable with a container build

It is also possible to build a GraalVM native executable **without** GraalVM or Mandrel installed. Quarkus delegates the build and packaging of your application to a Docker container that contains the GraalVM that is appropriately configured.

TIP The installation of Docker (or Podman) is outside the scope of this book.

But why do we then need to configure Mandrel/GraalVM at all? Because building native executables this way can only produce a Linux-specific executable application. But even if you don't run on top of Linux, you can still utilize this binary in your Docker images. However, since most of the targeting platforms like production, continuous integration (CI) systems, containers, clouds, and similar usually run on top of some Linux distribution, this functionality is very appreciated in case installing GraalVM presents a problem for you. These environments often try to limit the amount of installed software. So Quarkus provides you with this option if you just want to produce an executable application that you can then easily package and run in a containerized environment (e.g., Kubernetes).

Quarkus executes the native build inside the Docker container in two cases — either on user request (in that case, it needs to be explicitly specified by a configuration property `-Dquarkus.native.container-build=true`) or as a fallback in case it is not able to find a valid GraalVM environment (e.g., `GRAALVM_HOME` is not set).

To explicitly request build in a container you can run the build with the system properties as defined in the [Listing 2.13](#).

Listing 2.13 An explicit request for Docker native build

```
# Maven
$ ./mvnw package -Pnative -Dquarkus.native.container-build=true

# Gradle
$ ./gradlew build -Dquarkus.package.type=native \
-Dquarkus.native.container-build=true

# CLI
$ quarkus build --native -Dquarkus.native.container-build=true
```

Quarkus, by default, detects which container runtime (Docker or Podman) you have, but you are also allowed to override this manually with the configuration property `-Dquarkus.native.container-runtime=docker|podman`. If you do not want to repeat these properties every time you build your native executable, you can also specify them in the `application.properties` (more information about Quarkus configuration is available in the following chapter). In the output of the build, you then see the following output (the messages can mention `podman` in case that is your container environment):

```
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildContainerRunner]
Using docker to run the native image builder
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildContainerRunner]
Checking status of builder image
'quay.io/quarkus/ubi-quarkus-mandrel-builder-image:jdk-17'
```

In the second (implicit) case, when you do not have a GraalVM environment, you can see the following message in addition to the same build information as above.

```
[WARNING] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Cannot
find the `native-image` in the GRAALVM_HOME, JAVA_HOME and System PATH.
Install it using `gu install native-image` Attempting to fall back to
container build.
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildContainerRunner]
Using docker to run the native image builder
```

The produced Linux binary `quarkus-in-action-1.0.0-SNAPSHOT-runner` is available in the target directory. So essentially, if your platform is Linux, you do not need to install GraalVM or Mandrel to produce a native compatible with your system. In every other case (Windows and Mac), you must install one of them to produce native compatible with your platform. Later on, we look into how you can integrate a creation of the Docker container that can utilize this Linux binary within your Maven (or Gradle) build. Then in one step, it creates a runnable container with your application that you can easily run on any of the platforms mentioned above.

2.5 Unequaled Performance

Probably all of us have been asked to optimize applications for performance at least once in our careers. The metrics, such as startup time, typically don't fall into the most optimized category. However, in the move to a cloud environment managed by Kubernetes where you don't control when precisely Kubernetes decides to restart your application or in the architectures with rapid scale-downs, such as serverless, the startup time is essential. You probably don't want to waste time starting your application for several minutes during the rush hour to keep your customers waiting until you can process their requests.

As you probably noticed, Quarkus starts really fast and that's not the last performance benefit Quarkus provides. So how does Quarkus achieve such a great performance? Quarkus utilizes a concept of build time processing. This means that Quarkus aims to move as much processing as possible to build time (when your application is being compiled) rather than to do them during runtime, as is typically the case with Java runtimes. [Figure 2.5](#) provides a visual representation of this processing.

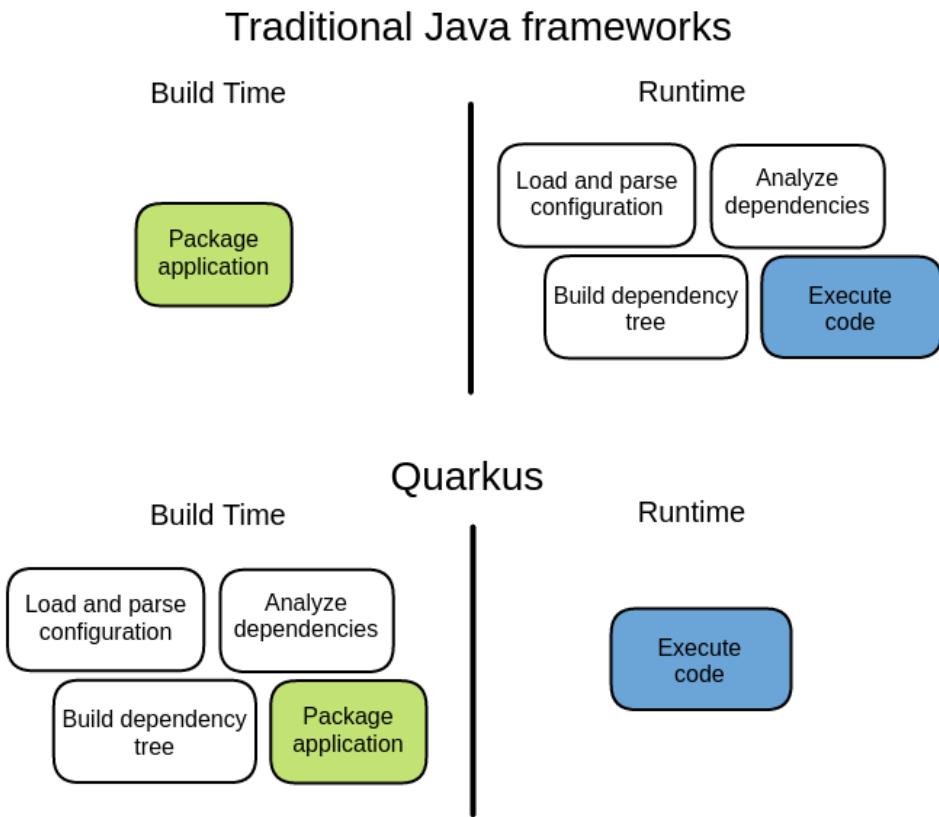


Figure 2.5 Quarkus build time vs runtime processing

In traditional Java frameworks, packaging your application is the first and only thing done at build time. These frameworks package all the classes and configuration needed for your application to run into the produced artifacts. This also includes everything that the framework and any libraries that your application uses need. However, a lot of the packaged classes will only be used once for the runtime initialization, and then they will just be left there, unused, for the whole application run.

Build time processing or ahead-of-time compilation is the idea that a lot of the tasks that are usually done when your application is first started (and that is why it typically takes a long time for Java applications to start) can be executed during the application compilation, the results can be recorded, and then utilized when the application is started.

In Quarkus, pre-processing, including class loading, annotation scanning, configuration processing, and more, are all executed during the application build. Classes only used for the application initialization are never loaded into the runtime (production!) JVM resulting in very low memory usage and unquestionably fast startup times unequaled in the Java world.

Surely, Quarkus cannot perform all pre-processing tasks to record them at build time. For instance, hardcoding the HTTP port on which the application runs during compilation might be problematic (since we might want to run the application on different ports in different environments). For this reason, Quarkus provides a mechanism that gives the developer an option to move as much processing as needed to build time while still integrating it with potential required runtime processing. The extensions framework directly accommodates this mechanism, as we learn in section [2.7](#).

TIP As a consequence of the build time initialization architecture, built-in Quarkus configuration properties are split into two groups: those that are applied at build time and those that are applied at runtime. Build time properties generally define things like enabling various optional features. Runtime properties are used for things that are unknown during the build, or would be impractical to fix during the build, like the already mentioned HTTP port for listening (because if Quarkus took that property into consideration already at build time, then you wouldn't be able to change the HTTP port when actually starting your application). If you're unsure whether a particular configuration property is fixed at build time or overridable at runtime, you can check the Quarkus documentation that lists all available properties - <https://quarkus.io/guides/all-config>.

If you are evaluating Quarkus, you might also be asked to demonstrate some real performance benefits. Let's take a look at some numbers detailing namely in the startup times and memory utilization (representing the main cloud requirements for your application), that you can present to your management.

Let's start by looking at the artifacts we created in the previous sections. GraalVM is often described as the main driver for these performance enhancements. Still, in Quarkus, both JAR and native formats certainly demonstrate the value that Quarkus processing brings to the table. Starting the `quarkus-in-action` application in the JVM mode ([Listing 2.14](#)), we can analyze the startup time logged when the application is starting.

Listing 2.14 Running Quarkus in JVM mode as runnable JAR

```
$ java -jar target/quarkus-app/quarkus-run.jar
-- ____ -- ____ -- ____ -- ____ -- ____ --
-/ __ \ / / / _ | / _ \ / / / / / / _/
-/ /_ / / / / _ | / , _/ , < / / / \ \
--\_\_\_\_\_\_\_/_/ |/_/|/_/|_| \_\_\_/_/
2023-11-02 17:19:01,067 INFO [io.quarkus] (main) quarkus-in-action
1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.5.0) started in 0.395s.
Listening on: http://0.0.0.0:8080
2023-11-02 17:19:01,071 INFO [io.quarkus] (main) Profile prod activated.
2023-11-02 17:19:01,071 INFO [io.quarkus] (main) Installed features: [cdi,
resteasy-reactive, smallrye-context-propagation, vertx]
```

Wow! Starting any Java application in just 0.395 seconds is incredibly fast! Of course, this number can differ depending on your environment, but the starting time for typical Quarkus REST applications rarely exceeds 2 seconds.

Let's also look at memory utilization as another critical performance driver for cloud-native applications because the less memory service consumes, the more services we can run in the same amount of memory (for which we pay in the cloud). [Listing 2.15](#) illustrates the memory usage of our Quarkus application started in JVM mode using the `ps` command. We use Resident Set Size (RSS) for our measurements which outputs the amount of RAM used by the individual programs.

Listing 2.15 Quarkus JVM memory utilization

```
$ ps -eo command,rss | grep quarkus
java -jar target/quarkus-ap 121332
```

Again, a JVM application utilizing only 121 MB of RAM is a great result, especially since we just started it. So Quarkus in the JVM mode is already very performant, and there are a lot of use cases where this mode is suitable for production deployment. But let's take this one step further and follow the exact measurements for the generated GraalVM native executable. [Listing 2.16](#) indicates both the startup time and the RSS usage of the generated Quarkus native binary.

Listing 2.16 Quarkus performance measurements of the native executable

```
$ ./target/quarkus-in-action-1.0.0-SNAPSHOT-runner
_____
-/ \ / / _ | / _ \ // / / / / /
-/ / / / / / | / , _/ , < / / / \ \
--\_\_\_\_/_/ |/_/|/_/|_| \_\_/_/
2023-11-02 17:16:56,514 INFO [io.quarkus] (main) quarkus-in-action
1.0.0-SNAPSHOT native (powered by Quarkus 3.5.0) started in 0.009s.
Listening on: http://0.0.0.0:8080
2023-11-02 17:16:56,514 INFO [io.quarkus] (main) Profile prod activated.
2023-11-02 17:16:56,514 INFO [io.quarkus] (main) Installed features: [cdi,
resteasy-reactive, smallrye-context-propagation, vertx]

# in a new terminal
$ ps -eo command,rss | grep quarkus
./target/quarkus-in-action- 40064
```

And now this is really impressive! Just 9 milliseconds to start and only 40 MB of RAM used? These values make Java comparable with generally faster (scripting) languages like Node.js or Go. Adding 9 ms to the request handling is essentially negligible. This is a significant milestone for Java developers since it allows teams to migrate to, for instance, the serverless architecture without the need to learn a different programming language.

The continuous growth of cloud migration and, in addition, serverless environments make the low memory footprint and fast startup very valuable metrics when choosing the application framework. Quarkus, with its build time processing, presents an excellent choice in this regard. So it is not only about how easy and enjoyable the work with Quarkus is for you during the development. You can simply demonstrate the practical production value as well.

2.5.1 When to use JVM and when to compile to native

With Quarkus, the compilation into the native executable image for users represents only a simple switch of a build time parameter. So it makes sense to consider compiling a native executable once the development cycle is done to get such kind of performance benefits. So does it always make sense to compile native binary? Well, not really.

Native compilations with GraalVM in Quarkus, one of the first frameworks that embraced native images, are often misunderstood as the primary performance benefit and, thus the required last step before the application deploys to production. However, this is not the case. As we have already learned, Quarkus utilizes a concept of build time processing which is the main reason for achieving better performance. Native images take it to another level but does it mean we should use them everywhere?

The answer is no. Using a native image or a runnable JAR should always depend on the target use case. Why? Because of the way how JVM (JIT - Just-in-time compiler) optimizes your code when the application runs. Depending on the execution paths of your application, it can eventually outperform the native image's great performance. Remember that native compilations work statically, ahead-of-time before any code is actually run. In this sense, they cannot know which paths will be utilized in which way and thus cannot put the best optimizations in place.

TIP GraalVM Enterprise Edition also provides a Profile-Guided Optimizations that allow providing profiling data taken from the running application to the native image compilation, which helps with this limitation.

If we imagine how the application's execution in time ([Figure 2.6](#)), we can analyze that the JVM when all optimizations take place, or in other words, when the application reaches the peak performance, it can outperform the native image performance which is already high from the beginning. However, it is static for the whole run of the application.

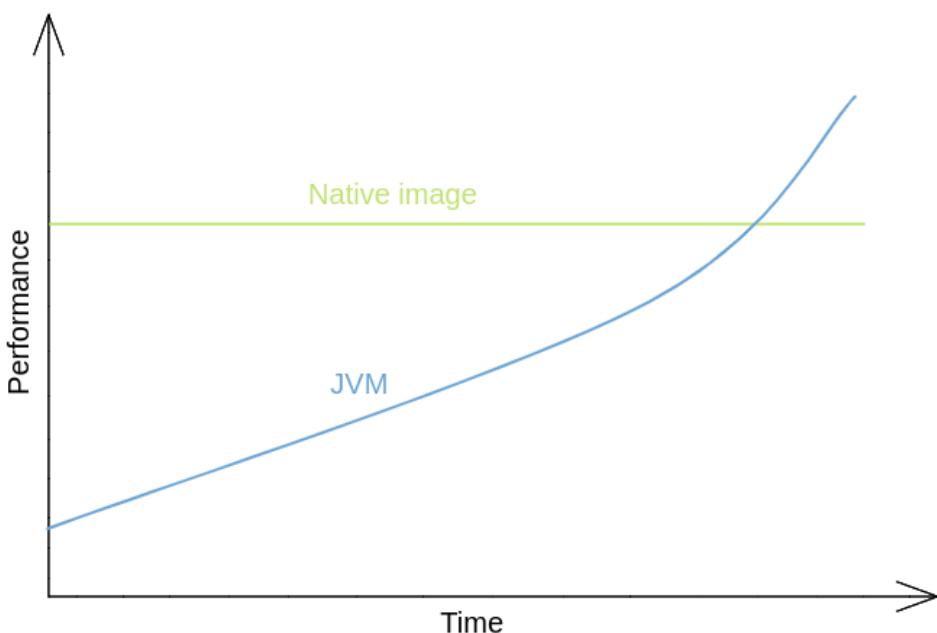


Figure 2.6 JVM vs native image performance over time comparison

Typically, in an environment where the startup time is not critical and the application is expected to be running for some time (classic microservices), the JAR can outperform the native executable. However, if the startup time is critical (serverless environments), then the native executable is the better choice.

2.6 Building container images with Quarkus

With the continuous push on the workloads to move to the cloud, building Docker (<https://www.docker.com/>) or Podman (<https://podman.io/>) container images is now an integral part of our everyday work for almost all of us. As we have learned in section [2.2](#), the generated application contains the included Dockerfiles that we can use to create Docker images. However, Quarkus was built with Kubernetes (or cloud in general) in mind. This means that it is not mandatory to use these files directly, and there is possibly a better way to create (and also deploy) your images which is represented in its own extension discussed in the later chapters. Nevertheless, you might still run into a use case requiring a Dockerfile. So let's take a look into what Quarkus created for us out of the box.

In the `src/main/docker` directory, you can see four Dockerfiles:

- `Dockerfile.jvm` — For the Quarkus ran in the JVM mode as a fast JAR (default).
- `Dockerfile.legacy-jar` — For the Quarkus ran in the JVM mode as the legacy JAR format. Quarkus used the legacy JAR format before the team came up with the idea of fast JAR, which is now the default. You can find more information about the available JAR formats at <https://developers.redhat.com/blog/2021/04/08/build-even-faster-quarkus-applications-with-fast-jar>.
- `Dockerfile.native` — For the Quarkus ran as a native executable.
- `Dockerfile.native-micro` — Similarly packages the native executable but utilizing a custom micro base image that is tuned for Quarkus native executables which results in a smaller size of the resulting image. More information about this micro image is available at <https://quarkus.io/guides/quarkus-runtime-base-image>.

Quarkus describes how you can create an image from each Dockerfile directly in the respective Dockerfile's comments. Each of these files contains instructions on how to package your Quarkus application and how to utilize Docker commands to build the image and run the container. Let's take a closer look at the last Dockerfile from the list `Dockerfile.native-micro` in [Listing 2.17](#).

Listing 2.17 Dockerfile.native-micro Dockerfile

```
#####
# This Dockerfile is used in order to build a container that runs the
# Quarkus application in native (no JVM) mode.
```

```

# It uses a micro base image, tuned for Quarkus native executables.
# It reduces the size of the resulting container image.
# Check https://quarkus.io/guides/quarkus-runtime-base-image for further
# information about this image.
#
# Before building the container image run:
#
# ./mvnw package -Pnative #1
#
# Then, build the image with:
#
# docker build -f src/main/docker/Dockerfile.native-micro -t
# quarkus/quarkus-in-action . #2
#
# Then run the container using:
#
# docker run -i --rm -p 8080:8080
# quarkus/quarkus-in-action #3
#
####
FROM quay.io/quarkus/quarkus-micro-image:1.0 #4
WORKDIR /work/
RUN chown 1001 /work \
    && chmod "g+rwx" /work \
    && chown 1001:root /work
COPY --chown=1001:root target/*-runner /work/application

EXPOSE 8080
USER 1001

CMD ["/./application", "-Dquarkus.http.host=0.0.0.0"]

```

#1 Instructions on how to package your Quarkus application.

#2 Instructions on how to build a Docker (Podman) image using this Dockerfile.

#3 Instructions on how to run the created image.

#4 The actual Dockerfile content.

The file starts with a short description of the produced image. Next, it demonstrates the individual instructions needed to package your Quarkus application correctly and to build a Docker image with the packaging output utilizing this Dockerfile. It also illustrates how you can start a container utilizing the created Docker image to start a container. At the end of the file, there is the actual content of the Dockerfile.

The following snippet details the produced container image. The interesting part is the image size which is only 74 MB.

```
$ docker image ls | grep quarkus-in-action
localhost/quarkus/quarkus-in-action latest 96591a254073
About a minute ago 74 MB
```

For comparison, the `Dockerfile.native` is around 150 MB. So the micro image saves 50 % of the image size. The JAR Docker image built with `Dockerfile.jvm` is about 450 MB. This might seem as much, however, because of the way the Quarkus is packaged in the JVM mode, splitting application JAR into a separate artifact (`quarkus-run.jar`) and the dependencies into an independent directory (`lib`, see section [2.3](#)), the image is built with dependencies in the layer before the layer that contains the runnable JAR. In this way, every time you rebuild this image, and if you do not change the dependencies, which would result in also recreating the `lib` layer, your build will be very small containing only your application which needs to be recompiled because of the changes that you are making. This strategy can also be utilized in a remote repository (cloud, Kubernetes) where you can push the base image layers which do not change that often and push only the smaller application layers through the network on changes.

TIP If you use Podman, we recommend you'll adjust your installation according to the information provided in <https://quarkus.io/guides/podman> to allow Podman to correctly handle all Quarkus scenarios.

2.7 Extensions

Extensions are an integral part of Quarkus architecture. They represent the mechanism that allows users to pick and choose only the functionality that is absolutely necessary for their particular Quarkus applications. This means that each Quarkus instance packages purely the indispensable dependencies required for its correct behavior. In this way, Quarkus packages less (only required) code and resources, meaning less processing and in turn, meaning faster startup times and better memory utilization that your application brings into production.

2.7.1 What is an extension?

From a developer's point of view, Quarkus extensions are simply modules that provide additional functionality by integrating a library or a framework with the Quarkus core — an integral Quarkus code base that provides wiring code, including all the extensions need. A couple of prominent examples of extensions are those for Hibernate (database management) and RESTEasy (REST server/client). By providing such a modular system, Quarkus allows you to choose whatever functionality you need in your individual applications without delivering everything in a single monolithic package, thus achieving a better runtime footprint.

If we break down the `pom.xml` of the `quarkus-in-action` application we generated, we see the `resteasy-reactive` extension (which implements the JAX-RS specification for implementing REST Services). It is located in the dependencies section, as shown in the [Listing 2.18](#). In Quarkus projects, the extensions are managed as Maven dependencies. The `io.quarkus:quarkus-resteasy-reactive` Maven dependency represents the `resteasy-reactive` extension.

Listing 2.18 Quarkus RESTEasy Reactive extension Maven dependency

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-reactive</artifactId>
</dependency>
```

From the architectural point of view, the integration of extensions into Quarkus is demonstrated in the [Figure 2.7](#).

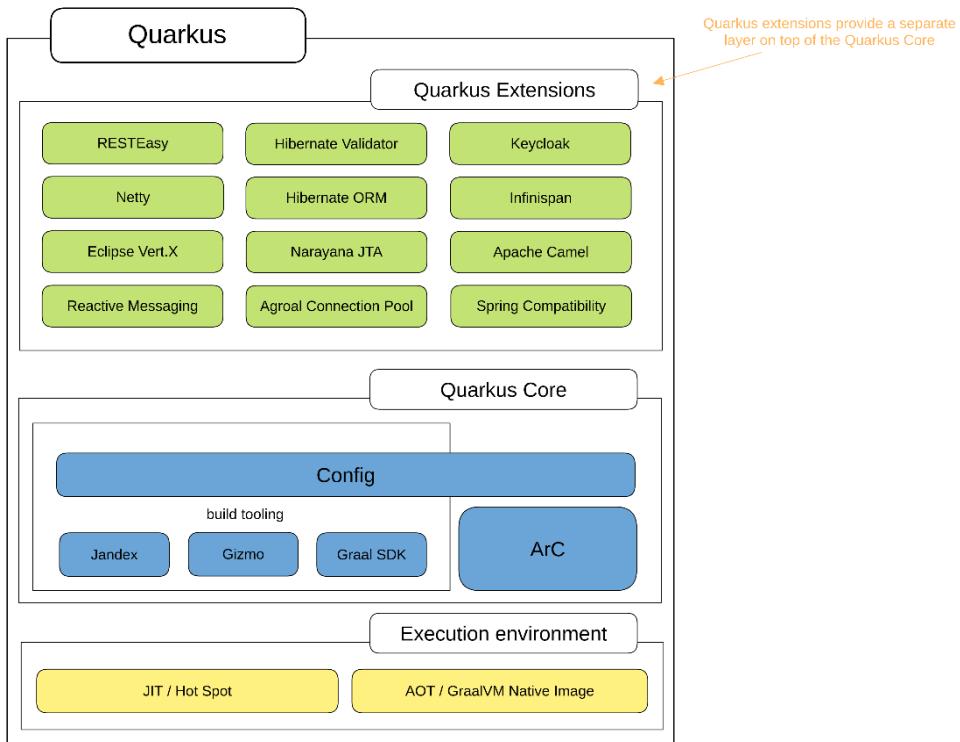


Figure 2.7 Quarkus component view

The extensions are the top layer users interact with the most. You can integrate only the needed functionality, which differs per application. Some extensions are further divided into more extensions when some parts of their functionality vary. For instance, if you need to use JSON with RESTEasy, you can choose from two different JSON serialization libraries (Jackson and JSON-B), meaning Quarkus provides two additional extensions that provide correct wiring for RESTEasy and the respective JSON library.

The ArC extension (also available in `pom.xml` in the `<dependencies>` section) is listed beneath other extensions because it is unique. ArC provides an implementation of the dependency injection framework. If you are familiar with Java EE, ArC implements the Contexts and Dependency Injection (CDI) specification. This extension provides an integral functionality that almost all other extensions and the Quarkus core rely on. However, some applications (e.g., the command line applications) might not require such functionality, so you can remove it if needed.

Some extensions are also added to Quarkus applications implicitly, typically as dependencies of other already added extensions. You probably noted that when you run the `quarkus-in-action` application (for instance, [Listing 2.9](#)) there are four extensions in the list - `cdi` (ArC), explicitly added `resteasy-reactive`, and two implicit extensions: `smallrye-context-propagation` and `vertx`.

2.7.2 Native compilations and build time processing

Another important benefit of the extensions approach in Quarkus is their ability to shield users from the complexity of generating native executable applications with GraalVM-based compilations. Such compilation must adhere to several rules that are mandated by the form of its execution. For example, the compiler needs to know all application's reflectively accessed program elements ahead-of-time, meaning often specified manually in the packaging command (however, it does try to deduct as much as it can from static analysis). This is where Quarkus delegates the responsibility of knowing such details to the individual extensions. Since the extension provides integration/implementation of a particular library or framework, it knows what kind of code the user writes. This way, it can help with denoting this kind of information to the GraalVM compiler automatically, without any user intervention. To compile into a native executable, you can simply add a command line flag, as demonstrated in the section [2.4](#).

As long as your application uses only Quarkus extensions, Quarkus guarantees almost seamless native compilation (of course, edge cases might happen). In a Java runtime, this is a very useful feature. Why? For instance, many popular Java enterprise libraries need to access components of your application reflectively. This is problematic for GraalVM native compilations because the reflectively accessed classes need to be explicitly listed during the native build. The integration code provided by the extension can do that for you. Using extensions is thus always the preferred way of adding additional functionality to your Quarkus application.

Something that your application also benefits from, but you also can't directly see when using extensions is the ability to specify which parts of library/framework integration execute during the build time and which need to be relayed to the runtime. It can also define which parts of the code compile during the build to record the bytecode that can be included in the packaged application for execution during runtime. Since the extension developer knows the details of the integrated code, it is often straightforward to define these respective areas. It is also possible to directly scan the user application in which the extension is added to get the information about available user classes and resources so the extension can make dynamic decisions depending on the code that you typed in your Quarkus application. This mechanism allows the integration of the libraries to fully take advantage of the build time principle of Quarkus.

Surely, you can still add any Java dependency to your Quarkus application. It will generally work in the JVM mode as in any other Java runtime. However, you might notice problems when compiling into a native executable with GraalVM or Mandrel. Many users or maintainers of popular libraries thus choose to implement the Quarkus integration with their respective libraries in custom Quarkus extensions that they can easily expose to all Quarkus users. If you are interested in writing your custom extensions, we dedicate an entire Chapter 11 to this subject.

2.7.3 Working with Quarkus extensions

The number of available Quarkus extensions is enormous. Nevertheless, the Quarkus extensions management tooling can compose for you a coherent experience of finding, adding, and removing Quarkus extensions. For better clarity, the examples utilized in this chapter will focus on the commands available in the `quarkus` CLI tool. However, all presented commands are also available in the Quarkus Maven plugin goals and Gradle tasks.

TIP To get all available goals of the Quarkus Maven plugin, you can invoke `./mvnw quarkus:help` and to get available Gradle tasks, you can use `./gradlew tasks`.

To list all extensions already installed in the Quarkus application, you can run the following command demonstrated in the [Listing 2.19](#) which executed in the `quarkus-in-action` application directory correctly outputs the `quarkus-resteasy-reactive` extension we installed at the project creation.

Listing 2.19 Listing installed extensions in the Quarkus application

```
$ quarkus extension list
Looking for the newly published extensions in registry.quarkus.io
Listing extensions (default action, see --help).
Current Quarkus extensions installed:

* ArtifactId                                Extension Name
* quarkus-resteasy-reactive                  RESTEasy Reactive

To get more information, append `--full` to your command line.
```

As an application grows over time and it needs to provide additional business value, it will leverage more and more of the built-in capabilities of Quarkus. If you want to list all available installable extensions (which is a very long list since we currently have only one extension installed) you might run the command detailed in the [Listing 2.20](#).

Listing 2.20 The installable extensions list

```
$ quarkus extension --installable
Listing extensions (default action, see --help).
Current Quarkus extensions installable:

* ArtifactId                                Extension Name
* blaze-persistence-integration-quarkus    Blaze-Persistence
* camel-quarkus-activemq                     Camel ActiveMQ
...

```

As browsing and searching in this long list might be difficult, the capabilities provided by extensions are grouped into categories to make it easier to select the desired ones. You can list the categories by invoking `extension categories` command, as demonstrated in the [Listing 2.21](#).

Listing 2.21 Listing Quarkus extensions categories

```
$ quarkus extension categories
Available Quarkus extension categories:

alt-languages
alternative-languages
business-automation
cloud
compatibility
core
data
integration
messaging
miscellaneous
observability
reactive
security
serialization
web
grpc
```

To get more information, append `--full` to your command line.

To list extensions in given category, use:

```
`quarkus extension list --installable --category "categoryId"``
```

Note the `--full` flag available for all these commands. With this flag, the output contains more information about available categories (or extensions) to help you if you are unsure what category or extension you are looking for.

Say that you need to interact with Kafka from your Quarkus application. Based on the output above, you might already guess to look into `messaging` category, but if not, you can still invoke the `quarkus extension categories --full` command to find that the `messaging` category directly mentions Kafka in its description. To find what extensions would suit this use case, you can invoke the following command to get installable extensions from the `messaging` category as shown in [Listing 2.22](#).

Listing 2.22 Listing all extensions in the messaging category

```
$ quarkus extension list --installable --category "messaging"
Current Quarkus extensions installable:

* ArtifactId                                     Extension Name
  quarkus-artemis-jms                           Artemis JMS
* quarkus-google-cloud-pubsub                   Google Cloud Pubsub
* quarkus-kafka-client                          Apache Kafka Client
* quarkus-kafka-streams                         Apache Kafka Streams
* quarkus-qpidd-jms                            AMQP 1.0 JMS client
  - Apache Qpid JMS
  quarkus-rabbitmq-client                       RabbitMQ Client
  quarkus-reactive-messaging-http               Reactive HTTP and
    WebSocket Connector
* quarkus-smallrye-reactive-messaging           SmallRye Reactive
  Messaging
* quarkus-smallrye-reactive-messaging-amqp      SmallRye Reactive
  Messaging - AMQP Connector
* quarkus-smallrye-reactive-messaging-kafka     SmallRye Reactive
  Messaging - Kafka Connector
* quarkus-smallrye-reactive-messaging-mqtt      SmallRye Reactive
  Messaging - MQTT Connector
* quarkus-smallrye-reactive-messaging-rabbitmq   SmallRye Reactive
  Messaging - RabbitMQ Connector
```

To get more information, append `--full` to your command line.

Add an extension to your project by adding the dependency to your pom.xml or use `quarkus extension add "artifactId"`

TIP We purposely use the long forms of the parameters for the quarkus CLI. However, most commands have shorter aliases, so users don't need to type full names. The command quarkus extension list --installable --category "messaging" can thus be shortened to quarkus ext list -ic "messaging". The CLI also comes with autocompletion that you can integrate with your Bash or ZSH shell. You can find all available options with quarkus extension -h.

From this output, we can summarize that the quarkus-smallrye-reactive-messaging-kafka extension is appropriate for the Kafka integration. Additionally, the output also makes it clear what command we need to invoke to actually add this extension to our project:

```
$ quarkus extension add "quarkus-smallrye-reactive-messaging-kafka"
...
[SUCCESS] ✓ Extension
↳ io.quarkus:quarkus-smallrye-reactive-messaging-kafka has been installed
```

This means that `pom.xml` has been modified and a new Maven dependency has been added to the dependencies section. The new `smallrye-reactive-messaging-kafka` extension is detailed in the [Listing 2.23](#).

Listing 2.23 smallrye-reactive-messaging-kafka extension in pom.xml dependency

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-reactive-messaging-kafka</artifactId>
</dependency>
```

Removing Quarkus extensions is done similarly. The `pom.xml` is again modified by removing the Maven dependency mentioned above. Analogous to adding an extension, you can locate the success message in the build's output. The command and the message are available in the [Listing 2.24](#).

Listing 2.24 Removing Quarkus extension example

```
$ quarkus extension remove "quarkus-smallrye-reactive-messaging-kafka"
...
[SUCCESS] ✓ Extension
↳ io.quarkus:quarkus-smallrye-reactive-messaging-kafka has been
↳ uninstalled
```

Using the `quarkus` CLI (or Maven plugin, Gradle tasks) represents a better utilization of Quarkus extensions. Namely, in browsing and searching for available extensions to achieve a particular task. However, Quarkus only modifies the `pom.xml` dependencies section in the background. So you are free to directly modify `pom.xml` if you already know what you need to do.

2.7.4 Quarkus guides

So far, we have learned how to manage the Quarkus extensions in your projects. However, knowing where to find the official documentation about what a Quarkus extension offers is often beneficial. For most extensions, Quarkus documentation provides a guide that offers the most helpful information about the extension and how to use it. It also usually contains a sample code for getting started with the extension. It also lists all available configuration properties applicable for the respective extension.

The Quarkus guides are hosted at <https://quarkus.io/guides/>. The website provides an intuitive UI that allows users to search for available guides and thus extensions that they might want to use in their Quarkus applications. Furthermore, the `quarkus` CLI conveniently provides a link to an official guide for the extension (if such a guide exists) when used together with `--full` flag. You can see this in action in the last column of the [Listing 2.25](#). Because the output of this command is too long, we display only the Kafka extension we used previously with columns split per line.

Listing 2.25 Quarkus messaging category list with the --full flag

```
$ quarkus extension list --installable --category "messaging" --full
Current Quarkus extensions installable:

* ArtifactId quarkus-smallrye-reactive-messaging-kafka

  Extension  SmallRye Reactive Messaging - Kafka Connector

  Version    3.2.0.Final

  Guide      https://quarkus.io/guides/kafka-reactive-getting-started

  ...


```

The structure of the Quarkus guide generally consists of a set of step-by-step instructions that build a small Quarkus application utilizing the described extension and showing its functionalities. It also provides a final solution in the form of a final built application available in the Quarkus quickstarts repository. At the end of each guide, there is a list of feasible configuration properties that can configure mentioned extension. These guides provide concise documentation for respective extensions and are the best place to retrieve information about different functionalities that your Quarkus applications can utilize.

2.7.5 Quarkiverse

Since initially introduced in 2019, Quarkus has grown at an unprecedented rate. With its frequent releases, robust support, and innovation, the community around Quarkus expands rapidly still today, moving Quarkus even further. Naturally, when more and more developers try Quarkus, they want to integrate their libraries with Quarkus to provide a custom extension that would make the use of their library easier. However, the Quarkus repository started to grow more than expected, and such a considerable number of extensions in the core Quarkus repository became unmaintainable.

For this reason, the Quarkus community created the Quarkiverse (Quarkus + Universe). Quarkiverse is a GitHub organization maintained by the Quarkus team that provides hosting of community extensions. The extensions in Quarkiverse are fully integrated into all the tooling we learned about in this chapter. Quarkus users can get all the benefits of the community extensions in the same way as with the core Quarkus extensions. The only difference is in the GAV definition of the Quarkiverse extensions, which typically includes different groupIds and versions.

When a developer wants to create an extension in Quarkiverse, the Quarkus team creates a new repository in the Quarkiverse organization with full administration rights given to the developer. The Continuous Integration (CI) and build/publish setup is already provided in the created repository. Developers publish their extension code as they see fit. Quarkiverse infrastructure takes care of validating it with the core Quarkus platform and possibly publishing it for the end users to consume.

Quarkiverse extensions are fully integrated extensions that can be used in Quarkus applications. The only thing to remember is that since they are versioned separately, users should check if the updates are available manually. You can find all available Quarkiverse extensions at <https://github.com/quarkiverse>.

2.8 Wrap up and next steps

In this chapter, we created our first Quarkus applications. We analyzed the project structure and described the application packaging into both runnable JAR and native executable formats. We then explained running Quarkus in these various formats and building Docker or Podman images from them. Lastly, we learned what Quarkus extensions are and how to utilize them for Quarkus development. All these concepts are the essential building blocks on which we later start developing the Acme Car Rental microservices. Hopefully, you now have a running Quarkus application, and you can't wait to learn more about what you can do with it.

This chapter is just the tip of the iceberg. Quarkus's extensions provide a vast ecosystem of different possibilities, which we will be continuously diving into throughout the rest of this book. However, in the next chapter, we will start by learning what functionalities make Quarkus development such an enjoyable experience.

2.9 Summary

- There are three distinctive ways how you can generate Quarkus applications in various ways—Maven plugin, command line interface, or at web starter code.quarkus.io.
- Quarkus applications follow a standard Maven (or Gradle) content structure. Quarkus also contains many useful additional files you can utilize in development and production.
- Compilations produce either runnable JAR of user applications but can also utilize GraalVM or Mandrel to create native executable applications.

- Quarkus extensions represent optional modules containing integrations with different libraries and frameworks that can be dynamically added and removed from the Quarkus application. They also provide integrations that allow straightforward native (GraalVM) compilations without user interventions.

3

Enhancing developer productivity with Quarkus

This chapter covers

- Speeding up application development with Quarkus' Dev mode
- Configuration mechanisms of Quarkus applications
- Experimenting with Dev UI to get insights into a running application
- Automatically running development instances of remote services using Dev Services
- Adding continuous testing into your application development workflow

As you recall from previous chapters, Quarkus offers several tools that make application development much faster, smoother, and the developer's life easier. In fact, developer productivity is one of the central ideas of Quarkus design. It is possible to develop in Java in a way where you simply make a change to your source file, hit the Refresh button in your browser, and all the changes you did are immediately visible! Gone are the days when you had to manually stop the application, run a Maven build, and start it again. You can enjoy the quick redeploy that is more common to dynamic languages like JavaScript but still use a statically typed compiled language - Java or Kotlin.

To give you a sneak peek before diving into it, Dev Services is another feature that you will surely enjoy. If you've worked on real-world projects, you've probably used databases a lot. In that case, you're probably familiar with the tediousness of managing databases for development - spinning them up, tearing them down, and controlling the schema/data. This is where Dev Services come into play. By using Dev Services, Quarkus can spin up disposable database instances for the application in a matter of seconds, fill them with some basic data, and automatically provide the wiring between the application and the database. By the way, databases are not the only feature offered by Dev Services. Quarkus can provide instances of many other types of remote services, like messaging brokers, for example. And you can write support for your own Dev Services, too (however, it would require writing a custom extension, which we will touch on in chapter 12).

TIP The majority of Dev Services implementations use the Testcontainers upstream library, but it's not the only supported way.

We also introduce the concept of continuous testing, where your tests execute automatically in the background on each code change while you can continue working with the application. This is another massive improvement over the classic workflows where you have to run tests separately and wait for them to finish, and you immediately notice how much time you can save.

This chapter focuses on the features Quarkus offers to make your life much easier as a developer. In each section, we'll explore one of them, along with hands-on examples based on further experimenting with the `quarkus-in-action` application from the previous chapter. Because these examples modify the source code of the `quarkus-in-action` application, you can find the updated sources in the directory named `chapter-03`.

3.1 Development mode

Quarkus' development mode, often referred to as Dev mode, radically changes how we develop Java applications. Traditionally, the development loop workflow has looked something like [Figure 3.1](#). Generally, to test your changes, you need to manually hit recompilation, run tests (at least once in a while), and restart the application. Of course, the exact workflow varies depending on what kind of application you're writing. If you're using a traditional Jakarta EE application server, the "Restart the application" part becomes "Redeploy the application".

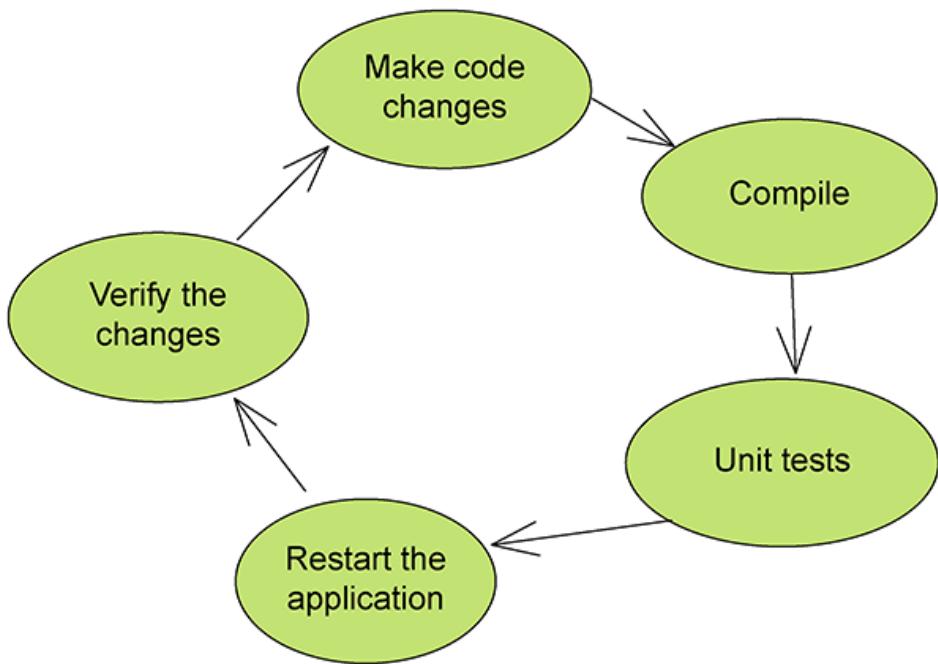


Figure 3.1 Traditional development workflow with Java

With Quarkus development mode, all this gets simpler, as shown in [Figure 3.2](#). When you make changes to your application and then hit the `Refresh` button in your browser, you immediately see the changes because the application gets recompiled and reloaded in the background within milliseconds.

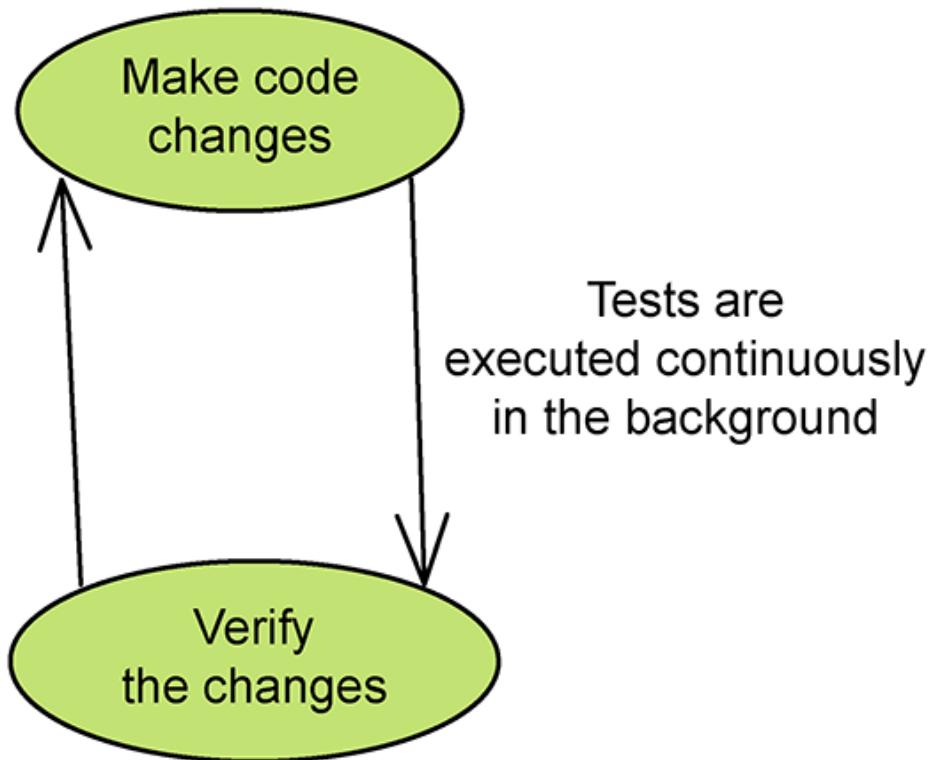


Figure 3.2 Simplified development workflow with Quarkus

This reload loop happens without the need to start a new Java Virtual Machine, the original one is reused to host the new version of your application.

And even better, your unit tests are executed automatically in the background, without you having to do anything or without interfering with your application, which allows you to work with your application while the tests are still running (if they take a long time to complete). We will talk more about Continuous testing in the section [3.5](#).

3.1.1 Trying live coding with the Quarkus project

Let's try some live coding and play around with the `quarkus-in-action` project that we created in the previous chapter. Open a terminal, go to the directory containing the project, and start it in Dev mode:

```
$ quarkus dev
```

TIP If you prefer using Maven directly instead of the CLI, use `mvn quarkus:dev` (or `./mvnw quarkus:dev` if using the Maven wrapper).

We already know that opening <http://localhost:8080/hello> in your browser shows you a greeting. Now, try changing that greeting a little. In your IDE of choice, open the `GreetingResource` class that implements our application's REST resource available in the source file `src/main/java/org/acme/GreetingResource.java` and change the string returned from the `greeting` method to something else, for example:

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

Now save the file (maybe that's not even needed because your IDE does it automatically), go back to your browser, and hit refresh (F5). After refreshing, you should see the updated greeting right away. Quarkus did the work to recompile and redeploy your application in the background transparently, and it probably took only a few hundred milliseconds after it received the HTTP request from your browser. You can verify that a reload occurred by looking into the console logs in your terminal. In particular, it relates to these two lines (omitting the timestamps):

```
INFO [io.quarkus] (Quarkus Main Thread) quarkus-in-action stopped in 0.014s
(...)
INFO [io.qua.dep.dev.RuntimeUpdatesProcessor] (vert.x-worker-thread-0)
  Live reload total time: 0.350s
```

These lines tell you that the application was stopped and then reloaded, and how long the reload took. By the way, it's not always necessary to send a request to trigger a reload - you can also trigger it at any time by pressing the `s` key while the terminal window with your running application is in focus.

TIP If you run a full build `mvn package` that includes tests, you probably have a failing test now because the included `GreetingResourceTest` asserts that the greeting is `Hello` from RESTEasy Reactive. You can either update the test or skip it with `-DskipTests` when running a Maven build. We will experiment with tests in section 3.5. We haven't yet enabled the continuous testing feature, so tests only run as part of the full build with `mvn package`.

You might also be asking what happens if you introduce a syntax error in your code. This would result in a failed build in the traditional Java development world, depending on your build tool. With Quarkus Dev mode, it's different. Try deliberately injecting a syntax error, for example, remove the semicolon after the `return` statement that we changed previously. Then, hit refresh again.

Instead of the `GreetingResource` showing you a text greeting, you should now see an error page (and the HTTP response code is 500). A shortened version of the included exception looks something like this:

Listing 3.1 Compilation failure in Dev mode

```
java.lang.RuntimeException: Compilation Failed:  
/quarkus-in-action/src/main/java/org/acme/GreetingResource.java:14:  
error: ';' expected  
    return "Hello Quarkus"  
           ^  
(stack trace omitted...)
```

A very similar snippet appears in the console log too. So instead of not being able to build and run your application at all, Quarkus is still running, and it can handle requests, and it also returns information about the error instead of serving the application itself. If you now fix your error by adding the missing semicolon and refresh again, the application gets back up.

This concludes the hands-on example of the Development mode. If you want to exit the application running in Dev mode, bring focus to the terminal window where it is running, and then either press `q`, which is the built-in command for quitting, or send the process a signal to terminate itself by pressing `Ctrl+C`.

3.1.2 How does it work?

Live coding and automatic reloading work not only for changes in your code. Changes to resource files, such as the main configuration file, trigger a reload too. When you add a new dependency, it even works for changes in your build descriptor, `pom.xml` or `build.gradle`.

Reload does not happen immediately when a change is detected. Instead, the application waits for an HTTP request to arrive (or for the user to trigger a reload manually). When Quarkus detects a change when an HTTP request comes, it triggers the reload. The request is held for a while to be then handled by the 'new' version of your application. Of course, this means that your application takes a bit longer to respond than usual, but unless the change is something that slows down the reload substantially (like adding a new dependency that is not present in the local Maven repository and has to be downloaded), it should finish within a few hundred milliseconds which is hardly noticeable.

All of this is possible thanks to a unique class loader architecture that allows reusing a single Java Virtual Machine for running multiple versions of the application in parallel in an isolated manner. The architecture is an advanced topic that we won't dive into here, but if you're interested in details about how it works, review the official documentation at <https://quarkus.io/guides/class-loading-reference>.

It's important to mention that the application loses any state it had before the reload. All application classes get loaded again in a new class loader, and the garbage collection processes the previously created (now released) objects. The so-called instrumentation-based reload can partially mitigate this. It can be enabled using the `quarkus.live-reload.instrumentation` property set to `true` (inside the `application.properties` file, which we describe in the next section) or by pressing `i` inside the terminal window or the Dev UI. When enabled, some specific minor changes can be applied to the source code without a full reload, only by replacing the relevant bytecode dynamically, without dropping the affected objects along with their state. However, due to various technical reasons, this is only possible for changes that only update bodies of methods. Changes such as adding new classes, methods, fields, or configurations still trigger a full reload and lose the application's state. After a reload, the console log tells you whether a full or instrumentation-based reload occurred. Instrumentation-based reload is disabled by default because it can sometimes lead to confusing behavior and should be used carefully. Experiment with it if you want. You will notice that an instrumentation-based reload is even quicker than the full one.

3.2 Application configuration

One challenge that developers of enterprise Java applications face is configuring their applications. Every framework you use might have its own configuration style, be it an XML or YAML file, system properties, or specific configuration models that Jakarta EE application servers have (`persistence.xml`, `web.xml`, ...). Quarkus put much effort into making application configuration as easy as possible. In fact, in most cases, all you need is a simple file that contains a set of properties (unless you need to, for example, dynamically obtain configuration values from a remote service). In that file, you can configure all aspects of your application, regardless of which extensions you are using, because each Quarkus extension that you add to your project contributes a set of properties that you can use for configuring the application behavior related to that extension. That single file is usually named `application.properties`, and by default, it resides in the `src/main/resources` directory (for Maven projects as well as Gradle projects).

TIP Quarkus also supports an equivalent YAML-based configuration style with the `quarkus-config-yaml` extension. In this case, config property names are mapped to YAML keys, and everything is expected to be in an `application.yaml` file instead. In this book, we focus on using `application.properties`, but we can also map all configuration properties into YAML if needed. Another way to configure Quarkus applications is using environment variables - each property can be overridden by an environment variable with a name that is derived from the property name. We will touch on that later too.

The `application.properties` file is used not only for configuring the behavior related to Quarkus extensions - but any of your application-specific configurations can also reuse it. The application's code can access each property listed in this file. To make this possible, Quarkus tightly integrates with the MicroProfile Config API (see <https://micropatterns.io/specifications/micropatterns-config/>).

TIP Built-in configuration properties (those that control the behavior of Quarkus itself and define data sources, thread pools, etc.) have names starting with the `quarkus.` prefix. You must name your application-specific configuration properties so that they start with a different word or unexpected behavior may result. Quarkus logs a warning if it encounters a defined property in the `quarkus` namespace that it doesn't recognize as a built-in property (such log entry can also help you notice that you've made a typo).

3.2.1 Experimenting with application configuration

Let's now externalize some application-specific configuration into properties in the `application.properties` file. We will do this to improve the `quarkus-in-action` application so that the configuration defines the returned greeting rather than a hard-coded string.

With the application still running in Dev mode, open the `application.properties` file in the `src/main/resources/` directory. It should be empty for now. Add the following line:

```
greeting=Hello configuration
```

Then, change the `GreetingResource` to look into the configuration instead of using a hard-coded greeting. Inject the value of the `greeting` property as shown in [Listing 3.2](#).

Listing 3.2 Injecting a configured greeting message

```
import org.eclipse.microprofile.config.inject.ConfigProperty;
...
public class GreetingResource {
    @ConfigProperty(name = "greeting")
    String greeting;
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return greeting;
    }
...
}
```

Refresh the <http://localhost:8080/hello> page in your browser and verify that the greeting is Hello configuration.

3.2.2 Configuration profiles

In many cases, you need the configuration values to be different depending on the lifecycle stage your application is running. For example, you usually need to connect to a different database instance when developing (or testing) the application than the one used during production. For that reason, Quarkus introduces the concept of configuration profiles.

By default, there are three configuration profiles:

- `prod` — active when the application is running in production mode
- `dev` — active when application runs in Dev mode
- `test` — active during tests

TIP It is also possible to define additional custom user profiles, which can be activated with the `quarkus.profile` configuration property.

Each of these profiles can use a different set of configuration values. To declare a property value for a particular profile, the property name in your `application.properties` file needs to be prepended with a `%` character followed by the configuration profile's name and then a dot, after which the regular property name and value. For example, a `quarkus.http.port` property defines the port on which Quarkus listens for HTTP traffic. By default, it's 8080. If you want to change this HTTP port specifically when running in Dev mode, add this to your configuration file:

Listing 3.3 Example of a property declaration only applied in Dev mode

```
%dev.quarkus.http.port=7777
```

When determining the set of values that will be active, Quarkus gives preference to the lines that declare a value specific to a particular profile if this profile is active. If there is no declaration specifically for this profile, Quarkus looks for a general declaration (without a profile prefix). If there is no such declaration, the property resorts to its default value if the application requires it.

3.2.3 Overriding application.properties

Some deployment environments (e.g., Kubernetes) require you to tweak the configuration values dynamically, which is not possible to do with the `application.properties` since this file is included in the compiled artifact. Out of the box, Quarkus provides two separate ways of overriding configuration - JVM system properties and environment variables.

The configuration values are read in the following priority: system properties, environment variables, and only then `application.properties` (potential additional configuration sources have their own custom priority). We can easily experiment with the `quarkus-in-action` application to demonstrate that. Stop the application (or Dev mode) if it is running and then run the application as demonstrated in [Listing 3.4](#).

Listing 3.4 Config override with system property

```
$ GREETING="Environment variable value" quarkus dev \
-Dgreeting="System property value"
```

This command sets the `GREETING` environment variable to Environment variable value and system property `-Dgreeting` to System property value when the application runs in Dev mode. Remember that in the `application.properties` we still define the greeting config value as `Hello` configuration. Notice also that we need to capitalize the environment variable name. This is the intended way of defining environment variables. So to define a config property as an environment value, the name of the property needs to be capitalized, and the dot delimiters (.) need to be replaced by underscores (_). For instance, config property `my.greeting` would be defined as `MY_GREETING`.

Invoking the <http://localhost:8080/hello> endpoint now correctly outputs "System property value" since the system property override took the precedence. If you now restart the application like shown in [Listing 3.5](#), that means without the system property definition, and repeat the call to the /hello endpoint, you will see that, as we learned this time, the environment variable takes precedence, and the returned value is "Environment variable value".

Listing 3.5 Config override with environment variable

```
$ GREETING="Environment variable value" quarkus dev
```

Of course, if you don't define either of them, Quarkus picks the `application.properties` configuration value, as we have already seen in [Listing 3.2](#).

`application.properties` is thus the main configuration file that includes most of the configuration of the Quarkus applications. However, in cases where some values need to be changed dynamically depending on different factors, you can override configurations by specifying either the system property or the environment variable when running the Quarkus application.

TIP The example that we showed for overriding a property was for an application-specific property (meaning that it's not a built-in property from a Quarkus extension, and thus its name doesn't start with `quarkus.`). If you apply this to built-in Quarkus properties, keep in mind that such overriding doesn't work for properties that are fixed at build time, these were briefly explained in [Chapter 2](#).

3.3 Dev UI

Dev UI is a browser-based tool that allows you to gain insights into your application running in Dev mode and even interact with it (change its state). It further facilitates application development by visualizing some framework-level abstractions that your application is using. In some cases, you might find that the insights gained by using the Dev UI can even substitute using a debugger. To gain an idea of what you can accomplish with the Dev UI, here is a list of notable examples:

- List all configuration keys and values (this can be invaluable, especially if you have multiple sources of configuration that supply different values for the same keys).
- List all CDI beans in the application, and for each of them, list their associated interceptors and priorities.
- List CDI beans detected during the build as unused, therefore not included in the resulting application.
- List JPA entities along with their mapping to database tables.

- Wipe data from a development database to be able to start from scratch.
- If you are using the Scheduler extension to schedule periodic tasks, you can manually trigger a task's execution outside the schedule.
- Re-run unit tests with a single click.
- View reports from completed test runs.

The architecture allows each Quarkus extension separately to plug its own tools into the Dev UI, so the complete list of options depends on which extensions the Quarkus application includes.

3.3.1 Experimenting with Dev UI

Run the `quarkus-in-action` application again in Dev mode. To open the Dev UI, you have two options. One of them is to open <http://localhost:8080/q/dev-ui> in your web browser manually. The easier way is to press the `q` key while the focus is on the terminal window where your application is running. This should open your browser automatically.

The landing page is named Extensions (see the menu on the left side of the page). In the Extensions page, you see panels, and each panel represents one Quarkus extension active in your project. Some extensions have Dev UI features, these are then available via buttons inside the extension's panel. Some extensions don't offer any Dev UI features, so their panel only shows a short description of the extension. For extensions that have user guides, the top-right corner provides a link to that user guide (the link looks like a book icon). At the bottom of the page, should see the server log.

The `quarkus-in-action` application does not use a lot of extensions. In fact, you probably see only a few panels, as shown in [Figure 3.3](#): The most prominent ones are RESTEasy Reactive and ArC, where ArC is the extension that handles contexts and dependency injection (CDI). You might potentially see more panels if you've added some other extensions to your project.

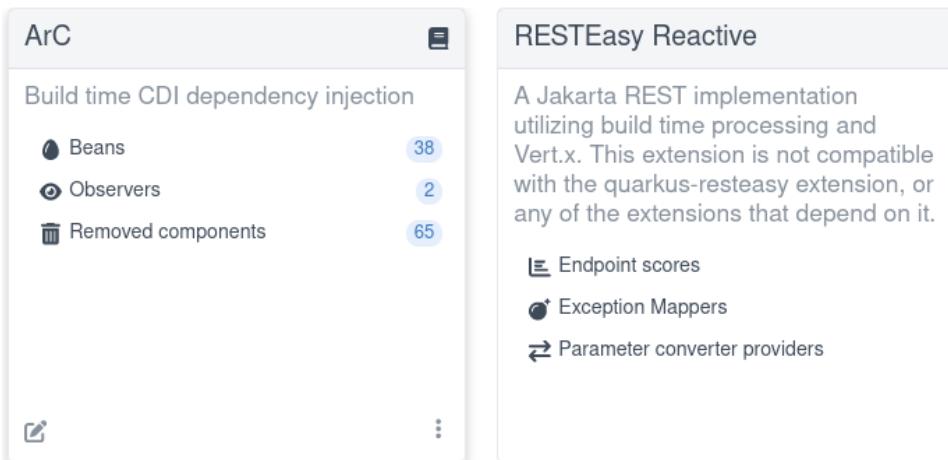


Figure 3.3 Dev UI panels

CHANGING THE CONFIGURATION

But before we play with the ArC extension, let's briefly explore the Config Editor. Click the Configuration panel in the menu on the left. You should then see a very long table containing all the configuration properties your application can understand. Either because they are built-in `quarkus.*` properties relevant to an extension you're using or because you have defined them as an application-specific property. If you scroll down, you will also see pure JVM system properties and environment variables.

All configuration properties understood by Quarkus (by the extensions that you're using) are listed here, even those for which you didn't specify any value. For those, you see their default value. All properties are writable. The `application.properties` file reflects any changes you make here to make them persistent. If you change the value of a property not listed in your `application.properties`, it gets added to the file.

TIP Even if you override something in the bottom part (which contains system properties and environment variables), a new configuration property of the same name will also appear in the `application.properties` file. Note that it doesn't mean that your application will see this new value of such system property or environment variable when calling `System.getProperties()` and `System.getenv()`. [Contents](#) of `application.properties` are not automatically translated to system properties. You need to use the MicroProfile Config API to read these values from your application.

To test out a change to a Quarkus built-in property, let's change the `quarkus.log.console.darken` property that controls the color of log messages in the terminal. The table should list it relatively high. If you can't find it, start typing its name into the search bar (filter) at the top of the page. The default value of this property is 0, so change it to 1, and then either click the corresponding 'save' icon on the right or hit `Enter`. After doing this, several things happen:

- Your application gets automatically reloaded because the configuration has changed.
- A new line containing `quarkus.log.console.darken=1` appears in `your application.properties`
- The configuration update also has the effect that new log messages (since the reload) in the terminal are darker.

TIP If you're running Quarkus inside an IDE, the color change might not be visible, depending on the terminal's capabilities. In that case, you might want to run the application in a regular shell terminal. Similarly, it might not be visible in the log view that is shown in the browser, you might have to look at the actual terminal window.

To revert this change, change the value back to 0 and hit `Enter` (or the save icon) again. Note that the new entry in `application.properties` stays there even if you change the value back to the default, so you might need to remove the line manually if you don't want it there.

LISTING APPLICATION'S CDI BEANS

Next, let's look at the control panel of an extension that offers Dev UI features, and that is ArC. It allows you to introspect things related to the CDI container. Go back to the 'Extensions' page of the Dev UI (using the left-side menu) and find the panel titled ArC.

If you click the Beans link, it takes you to a table that contains the list of all CDI beans in your application. In our case, most of them are built-in beans created by Quarkus, but there is one application bean, and that is our REST endpoint, `org.acme.GreetingResource` (every REST endpoint is a CDI bean even if you don't add any CDI annotation to it). The table shows the application beans first, so it is the first entry. It looks like in [Figure 3.7](#).

- You can see that its CDI scope is `@Singleton`, the default one.
- Notice that the class name is a clickable link. If you click it, Quarkus will do its best to detect what IDE you use, open it if it's not running, and open the source code of that class! Only application beans have clickable names.

3.4 Dev Services

When developing enterprise applications, dealing with remote resources like databases and message brokers is one of the most annoying aspects that slow you down. To be able to run your application and verify your changes, you need to have a development instance of them running somewhere, and most of the time, you have to manage it yourself. That's where Quarkus comes in - Quarkus can handle that for you by leveraging the so-called Dev Services. This means that Quarkus will automatically run and manage an instance of such resource for you. Generally, if you add an extension that supports Dev Services and you don't provide configuration for the remote service, Quarkus attempts to run an instance of that service and supply the wiring between it and your application.

TIP Dev Services design allows them to only work in Dev mode and during tests. You can't use them in production. To run your application in production mode, you have to provide the actual connection configuration of a remote service instance.

The following extensions listed in [Table 3.1](#) provide support for Dev Services. This list serves as an example reference. It is not exhaustive and will probably contain more extensions when you read this book.

Table 3.1 List of extensions that support automatic management of remote services via Dev Services.

Service	Extension that supports it	Description
AMQP	quarkus-smallrye-reactive-messaging-amqp	AMQP message broker
Apicurio	quarkus-apicurio-registry-avro	API registry
Databases	All SQL database drivers, including reactive ones, excluding Oracle	JDBC drivers and reactive database clients
Infinispan	quarkus-infinispan-client	Distributed key-value store
Kafka	quarkus-kafka-client	Kafka message broker
Keycloak	quarkus-oidc	Keycloak server as an OpenID Connect provider
Kogito	kogito-quarkus or kogito-quarkus-processes	Data Index for Kogito business automation
MongoDB	quarkus-mongodb-client	Document-based database
Neo4j	quarkus-neo4j	Graph database
RabbitMQ	quarkus-smallrye-reactive-messaging-rabbitmq	RabbitMQ broker to be used with Reactive Messaging
Redis	quarkus-redis-client	In-memory data structure store
Vault	quarkus-vault	HashiCorp Vault for storing secrets

For documentation about how to use Dev Services with each supported extension, and a list of all related configuration properties, refer to the official documentation of Dev Services at <https://quarkus.io/guides/dev-services>.

Dev Services generally require Docker (or Podman) to be available because they use containers to run the underlying services. There are some exceptions, for example, the H2 and Derby databases - these are run directly inside the JVM of your application.

3.4.1 Securing the Quarkus application using OpenID Connect and Dev Services

Let's learn how to use Dev Services to easily add an OpenID Connect (OIDC) authentication layer to your application. Similarly, we could add a connection to a database, but since the application is so simple that it doesn't need a database, showcasing some basic security features is more fitting.

The authentication workflow works like this:

- Quarkus Dev Services automatically spins up an instance of Keycloak, an OIDC provider. This requires a working Podman or Docker runtime, because the instance runs as a container.

- Then, we will use the Dev UI to obtain a security token from the embedded Keycloak instance.
- Finally, we will use that token to send an authenticated request to the application and verify that we can log in to it and print the name of the currently logged-in user.

Start by adding the `quarkus-oidc` extension to the application. For example, using the CLI:

```
$ quarkus extension add oidc
```

It adds the extension and triggers a reload of our application. The reloading will take longer than usual because Quarkus has to use your container runtime to spin up an instance of Keycloak. This message appears in the log when the reload finishes:

```
INFO [io.qua.oid.dep.dev.key.KeycloakDevServicesProcessor] (build-11) Dev
Services for Keycloak started.
```

You can also verify that a container for Keycloak is running. For example, if using Docker as shown in [Listing 3.6](#).

Listing 3.6 Verifying that Keycloak is running in a Docker container

```
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
36c3ee54c938 quay.io/keycloak/keycloak-x:16.1.0 start --http-enab...
26 minutes ago Up 26 minutes ago 0.0.0.0:43345->8080/tcp frosty_spence
```

Another way to show information about the Keycloak container is to press the 'c' key in the terminal where your Dev mode application is running. This shows information about all running Dev Services containers, as shown in [Listing 3.7](#).

Listing 3.7 Listing running Dev Services containers

```

keycloak
  Container:      ca64196216a9/confident_banach
→ quay.io/keycloak/keycloak:21.0.2
  Network:        podman (ca64196216a9) - null:37787->8080/tcp
  Exec command:   podman exec -it ca64196216a9 /bin/bash
  Injected config: - client.quarkus_oidc.auth-server-url=
→ http://localhost:37787/realm/quarkus
  - keycloak.realms=quarkus
  - keycloak.url=http://localhost:37787
  - oidc.users=alice=alice,bob=bob
  - quarkus_oidc.application-type=service
  - quarkus_oidc.auth-server-url=
→ http://localhost:37787/realm/quarkus
  - quarkus_oidc.client-id=quarkus-app
  - quarkus_oidc.credentials.secret=secret

```

TIP From now on, every time you run the application in Dev or test mode, it spins up an instance of Keycloak. Because the project's tests don't touch Keycloak, this unnecessarily slows down test execution. If you want to avoid that, add the `%test.quarkus_oidc.enabled=false` configuration line to your `application.properties`. This configuration completely disables the OIDC extension when running tests, and tests won't start a Keycloak instance anymore.

Now, to verify that we can log in to the application, we need an endpoint that prints the currently logged-in user's name. Open the `GreetingResource` class and add the following code as demonstrated in [Listing 3.8](#).

Listing 3.8 Source of the GreetingResource#whoAmI method

```
// import jakarta.ws.rs.core.Context;
// import jakarta.ws.rs.core.SecurityContext;
// import java.security.Principal;

@GET
@Path("/whoami")
@Produces(MediaType.TEXT_PLAIN)
public String whoAmI(@Context SecurityContext securityContext) {
    Principal userPrincipal = securityContext.getUserPrincipal();
    if (userPrincipal != null) {
        return userPrincipal.getName();
    } else {
        return "anonymous";
    }
}
```

From now on, when you invoke the `/hello/whoami` endpoint, the response contains the username of the authentication user or `anonymous` when invoked without authentication. Try opening `localhost:8080/hello/whoami` in your browser. It shows `anonymous` - you didn't specify the token, but you can still open the page because we didn't configure authentication to be mandatory. Beware that if you send an HTTP request with an invalid access token (rather than not specifying any token at all), the server will return a 401 Unauthorized response instead of the message `anonymous`.

To obtain an authentication token, go to the Dev UI (press `d` with a focus on the Quarkus terminal window), you will see a new panel present: OpenID Connect, as shown in [Figure 3.4](#).

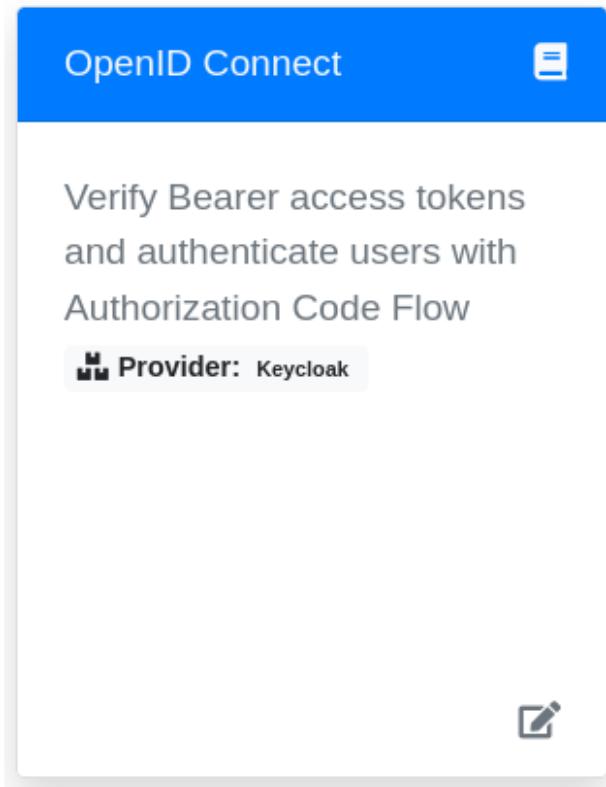


Figure 3.4 OpenID Connect panel in the Dev UI

Click the Provider: Keycloak button, and then the big green button saying Log into Single Page Application. It takes you to Keycloak's login page visualized in [Figure 3.5](#).

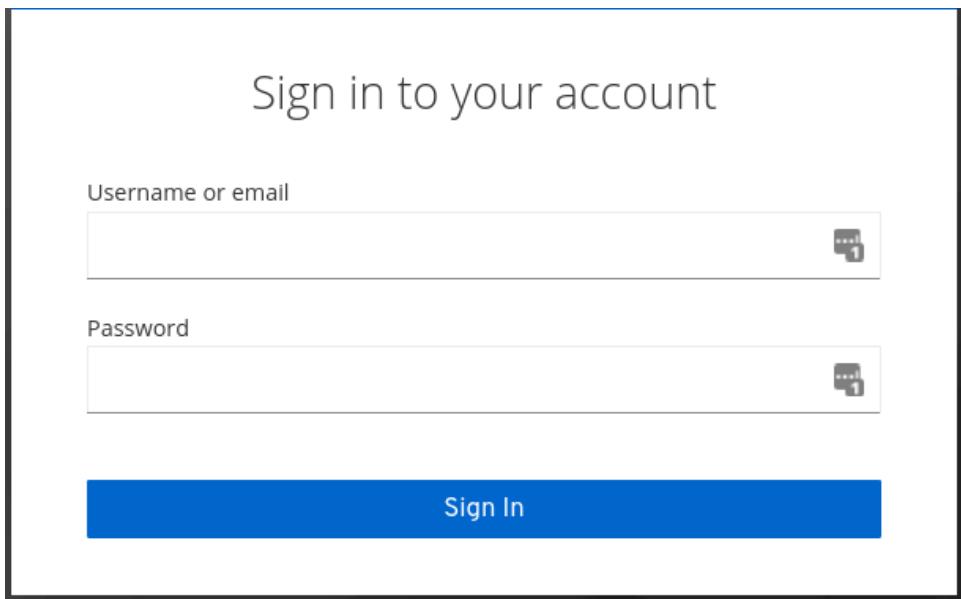


Figure 3.5 Login page of Keycloak

Use the username `alice`, and the password is also `alice`. This username-password pair automatically exists in Keycloak instances started through Dev Services, so don't worry - this combination won't work on a production instance! After logging in, Keycloak takes you back to the Dev UI, now as an authenticated user. To view your token, click the `View Access Token` link. The right column (`Decoded`) shows the raw token as a JSON document, while the left column (`Encoded`) shows its Base64-encoded version - a long alphanumeric string. This is what you need to attach to your HTTP request when calling the JAX-RS endpoint. Click on it and copy it to your clipboard, or save it into a text file.

TIP By default, the access token is only valid for 5 minutes, so you need to do the next step within that timeframe. If 5 minutes is not enough, you can extend it in the administration console of Keycloak (there is a link to it next to the `View Access Token` button, and the credentials are `admin:admin`) by changing the `Access Token Lifespan` value in the `Tokens` tab. You must log in again if the token expires to obtain a new token.

Now try invoking the endpoint while supplying the token. In the following snippet, replace `$AUTH_TOKEN` with the Base64-encoded version of Alice's token. If you don't use curl, you can also use tooling provided by your web browser that allows you to add custom HTTP headers to requests.

```
curl -H "Authorization: Bearer $AUTH_TOKEN" localhost:8080/hello/whoami
```

The response says `alice` since the token corresponds to this user.

3.5 Continuous testing

We can all agree that automated tests are essential to any real-world software project. Every application platform should provide tools to enable writing tests for projects using that platform, and Quarkus is no different. It defines a unique testing framework that tightly integrates with JUnit. This test framework allows you to quickly spin up an instance of your application (of course, with the possibility to change some parts of it) and perform tests by either working with individual parts of your application (by injecting them into the test, for example). For instance, it can take it to a higher level and communicate with the application as a black box by sending HTTP requests. Nevertheless, features of the testing framework are not what we focus on in this section - we will cover something more exciting - the capability to run tests continuously. Chapter 5 contains a deeper dive into the testing framework and its capabilities.

Now, what do we mean by continuous testing? We already know the Dev mode and live reloading of your application. To take it one step further, Quarkus doesn't only quickly reload your application upon every change, but it can also re-run your tests along with it! Every reload generally triggers a test run if continuous testing is enabled. That's another tremendous improvement to the developer's workflow because the tests can run parallel with any other tasks you do while developing without spending time starting their execution or waiting for them to finish. Even though tests execute within the same JVM where your development instance lives, you don't have to worry about affecting the application's state. This is because tests are run in a separate class loader and so have their isolated application instance on which they run. If your tests take a long time to complete, you can safely interact with your application while tests are still running!

3.5.1 Testing the Quarkus in Action project

Once again, run the Quarkus in Action application in Dev mode (`quarkus dev`). Look into its `src/test/java` directory. You will notice that there are two test classes, `GreetingResourceTest` and `GreetingResourceIT`. The latter is specifically for testing in native mode and is not relevant to this example, so we're going to ignore it for now and only look at `GreetingResourceTest`. The source code of the only testing method looks like the [Listing 3.9](#).

Listing 3.9 Source code of a simple test

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

The test uses the `RestAssured` library, which is a toolkit for running tests against REST endpoints. The code is relatively easy to read. It verifies that when you send an HTTP GET request to the `/hello` endpoint of your application, then the result is a 200 response (meaning success) and that the response body contains the string `Hello from RESTEasy Reactive`. Let's try to run our test now. Bring focus to the terminal with your running Dev mode and hit the `r` button, which starts continuous testing.

TIP To have continuous testing enabled automatically at application start, you can set the `quarkus.test.continuous-testing` property to `enabled`.

If you followed the previous exercises and are continuing from there, it is very likely that the test fails because we have changed that response from `Hello from RESTEasy Reactive` to `Hello Quarkus` in the Dev mode section, and then to `Hello configuration` in the configuration section.

If the test failed, you see a red failure report (including a stack trace) in the log, and this is in the status line at the bottom:

Listing 3.10 Status line in terminal shows a failure in continuous testing.

```
1 test failed (0 passing, 0 skipped), 1 test was run in 2537ms. Tests
completed at 09:30:38.
```

If the test passed, you see a green status line:

Listing 3.11 Status line in terminal after shows successful run of continuous testing.

```
All 1 test is passing (0 skipped), 1 test was run in 2987ms. Tests
completed at 09:33:55.
```

TIP This time, the test took a relatively long time to finish (almost 3 seconds in our case). This only occurs the first time you run tests after starting Dev mode. Subsequent runs will be faster, which you can verify immediately by hitting the `r` button to re-run tests. In our case, the time drops to about 300 milliseconds.

If your test failed, fix it by changing the response back to `Hello` from RESTEasy Reactive (either directly in the `GreetingResource` class or, if you changed it to read the configuration by updating the `greeting` configuration property. If your test passed, try the opposite, and deliberately change the response to make the test fail.

After performing your change in the source code (or `application.properties`), save the source file. Notice that the tests get re-run immediately! When continuous testing is enabled, the behavior of Dev mode changes so that a live reload triggers by merely detecting a source file change (otherwise, it is when an HTTP request or some other specific event is received). Can you already see how much of a time saver this is for developers?

TIP You don't have to worry about breaking the running application or the testing pipeline by saving a source file that contains a syntax error. If you do, then simply, instead of running your tests and showing you the results, Quarkus will offer you an error message in the terminal to help you fix that.

CONTROLLING TEST OUTPUT

You can use the `o` key in the terminal window to control the test output. If the test output is disabled (it is by default), you only see the failure stack traces after a test run. If you hit `o` to enable test output, your terminal will start showing logs produced by the testing instance of your application, including logs produced by the test itself. For example, it will look like the [Listing 3.12](#). You can tell that these logs come from the testing instance by the name of the thread executing them. It is the `Test runner` thread. Notice that the testing instance binds to a different HTTP port (8081 as opposed to 8080) and uses the `test config` profile.

Listing 3.12 Logs produced by running a test when test output is enabled

```
2023-07-11 11:47:49,319 INFO [io.quarkus] (main) quarkus-in-action
1.0.0-SNAPSHOT on JVM (powered by Quarkus 3.2.0.Final) started in
1.325s. Listening on: http://localhost:8081
2023-07-11 11:47:49,320 INFO [io.quarkus] (main) Profile test activated.
2023-07-11 11:47:49,321 INFO [io.quarkus] (main) Installed features:
[cdi, resteasy-reactive, smallrye-context-propagation, vertx]
```

USING THE DEV UI FOR TESTING AND VIEWING TEST REPORTS

So far, we have used the terminal window to control your application's testing. You might prefer a graphical interface, though, especially for reviewing test reports because a bunch of stack traces lying about in the terminal log is probably not something you want to see when diagnosing failures resulting from running your test suite. This is where the Dev UI comes in again. Everything test related that the terminal controls can do, the Dev UI can do too. Open the Dev UI (press 'd' in the terminal) and choose 'Continuous Testing' in the left-side menu. If you haven't enabled continuous testing yet, you will need to click the 'Start' button to enable it and run the whole set of tests. Shortly after that, you should see the results, as shown in [Figure 3.6](#) if the tests are passing, or [Figure 3.7](#) in case of failures.

The screenshot shows the 'Continuous Testing' section of the Quarkus Dev UI. At the top, there are three buttons: 'Stop' (blue), 'Run all' (green), and 'Only run failing tests' (grey). The 'Run all' button is highlighted. Below the buttons is a table with three columns: 'Test Class', 'Name', and 'Time'. The first row shows 'Test Class' as 'org.acme.GreetingResourceTest', 'Name' as 'testHelloEndpoint()', and 'Time' as '731ms'. The entire table has a light grey background.

Figure 3.6 Controls in the bottom right corner in the Dev UI

This screenshot is similar to Figure 3.6 but shows a failed test. The 'Run all' button is now highlighted in red. The table shows the same structure: 'Test Class' is 'org.acme.GreetingResourceTest', 'Name' is 'testHelloEndpoint()', and 'Time' is '30ms'. The '30ms' value is highlighted in red, indicating a failure.

Figure 3.7 Controls in the bottom right corner in the Dev UI

To re-run tests, click the 'Run all' button. To view details of failing tests, click the 'Run failed' button. Dev UI will switch from showing the regular server log at the bottom of the page to showing the test-specific logs. You can see that the two sets of logs are completely independent, these are two different Quarkus instances that don't affect each other.

TIP As a Java developer, you might be used to HTML or XML test reports generated by Maven or Gradle plug-ins. As our tests are JUnit-based and continuous testing embedded in the Quarkus process is just a special way to run them, of course, they also get executed the 'normal' way during a Maven build (by the Surefire plug-in during the `test` Maven goal) and produce traditional reports in the `target/surefire-reports` directory. The reports that you can find in the Dev UI try to mimic them, but might not include all the information that you would get after running the `mvn test`.

3.6 Wrap up and next steps

We learned about the features of Quarkus that improve the life of application developers and save a tremendous amount of time. You can enjoy the quick development turnaround time known from dynamic languages but still, use a static language like Java and Kotlin. The concept that a single properties file can store all necessary configurations for the application is another piece of improvement. You don't have to deal with several bloated XML configuration files like in the old times of working with Java EE. Most of the tedious work in managing database instances for development is now gone, with Quarkus managing them automatically (we will explore databases in chapter 5). The Dev UI provides invaluable insights into a running application for troubleshooting purposes, and it can save you from using a debugger. Another time saver is getting your tests executed automatically in the background after each source code change without preventing you from further experimenting with the application while the tests are still running.

These features together make up the productivity boosts and development experience improvements that Quarkus provides. Now that we've learned about the general benefits of using Quarkus, this book's Part 2 will focus on particular frameworks and libraries that Quarkus supports. We will start by examining the parts related to remote communication (various protocols for communicating with other applications and services).

3.7 Summary

- The development mode of Quarkus significantly improves the development process and saves a lot of time when writing applications.
- Live reloading happens in the background automatically after changing the source code and interacting with the application. It is also very fast.
- If the developer introduces an error, be it a syntax error or invalid usage of a library, Dev mode will still continue running, but will respond to HTTP requests with details about the error instead of serving the actual application content.
- In most cases, one properties file is enough to contain all configurations related to the application (both the configuration of Quarkus built-ins and application-specific properties). Usually, the file is `src/main/resources/application.properties`.
- Dev UI is a browser-based tool that provides insights into applications running in Dev mode, facilitating troubleshooting during development. Different Quarkus extensions contribute different features available in the Dev UI if that extension is active.
- Dev UI also allows manipulating the application, for example by changing the configuration values, triggering scheduled events out of schedule, wiping databases, etc.

- Dev Services is a tool that further simplifies development by automatically managing instances of remote resources such as databases or message brokers and providing the wiring between the application and the resource. It works by spawning containers with these resources, automatically shutting them down when Dev mode is stopped.
- Quarkus is tightly integrated with JUnit 5 to allow a smooth experience for writing and executing tests.
- Continuous testing is a way to quickly run your application's tests after each source code change in a fully automated fashion. The developer can continue experimenting with the application without interruption and only check the test results as they appear in the console or the Dev UI.

4

Handling communications

This chapter covers

- Learning how to expose and consume APIs using the REST paradigm
- Exploring GraphQL as an alternative to the REST paradigm
- Evaluating gRPC and Protocol Buffer as another way to expose and consume APIs

Microservices applications require a lot of network communication between the various parts. Recall that we plan to develop a system comprising the User, Reservation, Rental, Inventory, and Billing services. All of these services need to exchange information between them. This communication can be synchronous, where an application that requires some data from another system makes a network call and waits for the result, or asynchronous, where the requesting service receives a notification of the result when it completes. In this chapter, we focus on synchronous communication since it still represents the primarily utilized method of network communication in the microservices architecture. Specifically, we will learn how Quarkus makes it very easy to develop client and server applications that produce and consume data over the following protocols:

- Representational State Transfer (REST)
- GraphQL
- gRPC

REST is a ubiquitous paradigm that probably doesn't require a long introduction. Quarkus makes it easy to design your own REST API with a simple annotation-based model. We also learn how to write the client side - it's just as easy since you can use the same annotations on the client.

GraphQL may be new to some developers. It's an alternative to REST that gives the client more control over what data exactly it wants to receive. Unlike REST, it also allows the client the flexibility to request multiple types of data in a single HTTP request, making it a good fit for constrained environments where limiting the number of requests and the amount of transferred data is important. Just like with REST, Quarkus uses annotations to define your GraphQL API as well as to consume it. It also offers a UI which we can use to manually test our GraphQL APIs.

gRPC is a high-performance, open-source universal RPC framework that provides support for multiple languages. As you might have guessed, Quarkus also provides a simple annotation-based programming model for creating gRPC services and clients, and UI support to manually experiment with them.

The high-level view of the Acme Car Rental system looks as presented in [Figure 4.1](#). This diagram shows all the services comprising the system and the remote protocols utilized for communication between them. In this chapter, we develop several of these services to demonstrate, compare and explain the applicability of the analyzed network protocols.

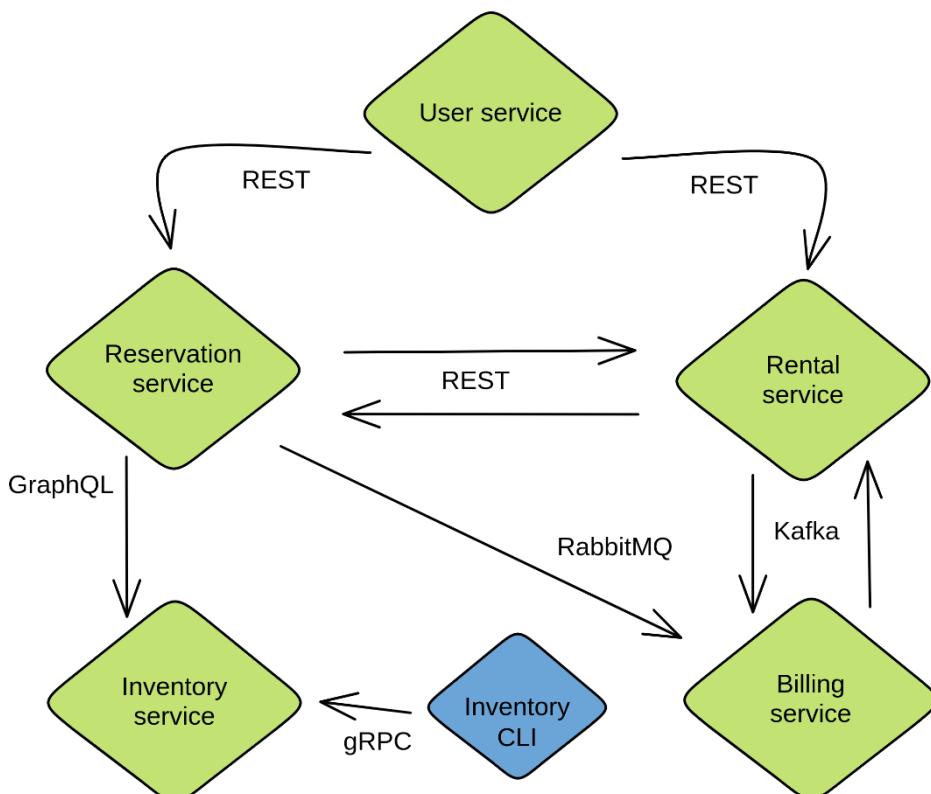


Figure 4.1 The Acme Car Rental system architecture

This is a lot of microservices to run on a single machine simultaneously, even for development. This is why we must distinguish different ports on which the individual services run. For simplicity, we will use the predefined port mapping for our local development environments, detailed in [Table 4.1](#). It also helps to categorize different services when we talk about them. For instance, if we call an application running on port 8081, you can deduct we call the Reservation service.

Table 4.1 The port mapping of the car rental services for local development

Service	Port
User	localhost:8080
Reservation	localhost:8081
Rental	localhost:8082
Inventory	localhost:8083
Billing	localhost:8084

The code of all services is available in the public code repository <https://github.com/xstefank/quarkus-in-action>. Individual directories split the book's content per chapter. Each directory is the final version of all developed services in that chapter. Furthermore, all commits are composed gradually per each section of this book to provide even greater detail on all individually developed parts throughout this book.

4.1 Developing REST-based Quarkus applications

REST (Representational state transfer) is the most common architectural style to communicate between services, mainly because it relies on the universal and well-understood HTTP protocol, and because it's relatively easy to use. Especially in the Java ecosystem, REST services are top-rated. Quarkus makes developing such services straightforward via an annotation-based model utilizing a project called RESTEasy Reactive, which contains both an implementation of the Java API for RESTful Web Services (JAX-RS) specification for exposing REST services and implementation of the MicroProfile REST Client specification for consuming REST services. The `quarkus-resteasy-reactive` extension (and its derivatives) achieves the former, while `quarkus-rest-client-reactive` (and similarly its derivatives) accomplishes the latter. As we will see later on, Quarkus also offers a bunch of tools for easily invoking REST endpoints manually.

TIP As we've learned in previous chapters, the term `reactive` in the name of both server and client extensions doesn't mean the services we create with them need to be reactive.

The following [Table 4.2](#) gives a brief overview of the most important JAX-RS annotations:

Table 4.2 The list of the most important JAX-RS annotations and their descriptions

Annotation	Description
@Path	Identifies the relative URI path that a resource class or method will serve requests for.
@GET	Indicates that the annotated method handles HTTP GET requests.
@POST	Indicates that the annotated method handles HTTP POST requests.
@Consumes	Defines the media types that a resource class or method accepts.
@Produces	Defines the media types that a resource class or method produces.

We will put these annotations to good use shortly.

4.2 Car Rental Reservation service

Let's start coding the first service. We now develop a simplified version of the Reservation service. This service exposes a REST endpoint for working with car reservations, which is what we need to implement first.

We begin by creating a new Quarkus application by executing the following command. The `--no-code` flag specifies that we don't want any example code generated.

```
$ quarkus create app org.acme:reservation-service -P 3.2.0.Final \
--extension quarkus-resteasy-reactive-jackson,rest-client-reactive-jackson, \
quarkus-smallrye-openapi --no-code
```

We choose the `quarkus-resteasy-reactive-jackson` extension because we want to use the Jackson library for handling JSON serialization and deserialization of requests and responses of the REST services that the Reservation service exposes. The `quarkus-resteasy-reactive` extension, which provides the core REST functionality, is a dependency of `quarkus-resteasy-reactive-jackson` (which adds just the Jackson support to it), so we don't have to add `quarkus-resteasy-reactive` explicitly.

We use the `rest-client-reactive-jackson` extension because we want to use Jackson also for the handling of JSON serialization and deserialization of the downstream REST requests that the Reservation service creates. Again, the core REST client functionality is brought in transitively as the `rest-client-reactive` extension.

The `quarkus-smallrye-openapi` extension adds the ability to generate an OpenAPI document, which documents our exposed REST API. It also creates a simple UI that we can use to test the REST endpoints that we develop.

To avoid port conflicts when running multiple services together, open the `application.properties` configuration file in the newly generated `reservation-service` directory and add the property `quarkus.http.port=8081`.

For this part of our journey, the Reservation service needs to do two things:

- For user-provided start and end dates, return a list of cars that are available for rent.
- Make a reservation for a specific car for a set of dates.

Before creating any code, make sure to launch Quarkus Dev mode, which can be done, as you might remember, by running the Quarkus CLI:

```
$ quarkus dev
```

4.2.1 Checking car availability

First, we need to define a model for cars, which in our case, is very simple. This car model contains information about the license plate number of the car, its manufacturer, and the model name. Moreover, it will also contain an `id` field to identify each car uniquely. To create this model, add a new `org.acme.reservation.inventory.Car` class into the `src/main/java` directory containing the following code available in [Listing 4.1](#):

Listing 4.1 The source code of the Car class which represents a car available for rental

```
package org.acme.reservation.inventory;

public class Car {

    public Long id;
    public String licensePlateNumber;
    public String manufacturer;
    public String model;

    public Car(Long id, String licensePlateNumber,
              String manufacturer, String model) {
        this.id = id;
        this.licensePlateNumber = licensePlateNumber;
        this.manufacturer = manufacturer;
        this.model = model;
    }
}
```

In this example, we used public fields instead of regular properties (a private field with a getter and setter) because it's shorter. If you prefer private fields along with getters and setters, you may use them instead, it will work the same way.

TIP If you use the Lombok library to make your code shorter through its `@Data` classes, that will work too.

As you might recall from our architecture diagram, the list of available cars will be provided by the Inventory service, so we need to model that communication. We do so by creating an interface abstracting these calls named `InventoryClient` in the package `org.acme.reservation.inventory` that contains one method: `allCars` used to obtain all cars. The code for this interface looks as shown in the [Listing 4.2](#):

Listing 4.2 The source code of the `InventoryClient` interface which abstracts the communication with the Inventory service

```
package org.acme.reservation.inventory;

import java.util.List;

public interface InventoryClient {

    List<Car> allCars();
}
```

As we have not written the Inventory service yet (we will do this when we discuss GraphQL), we will, for the time being, use a simple static list as a stub for the calls of the `Inventory` service. Create a new `InMemoryInventoryClient` class that implements the `InventoryClient` interface in the same package with code as shown in [Listing 4.3](#):

Listing 4.3 The source code of the InMemoryInventoryClient class which is the simplest possible implementation of InventoryClient

```
package org.acme.reservation.inventory;

import java.util.List;
import jakarta.inject.Singleton;

@Singleton #1
public class InMemoryInventoryClient implements InventoryClient {

    private static final List<Car> ALL_CARS = List.of(
        new Car(1L, "ABC-123", "Toyota", "Corolla"),
        new Car(2L, "ABC-987", "Honda", "Jazz"),
        new Car(3L, "XYZ-123", "Renault", "Clio"),
        new Car(4L, "XYZ-987", "Ford", "Focus")
    );

    @Override
    public List<Car> allCars() {
        return ALL_CARS;
    }
}
```

#1 This annotation is part of the Context and Dependency Injection (CDI) specification. It allows us to use this class as the implementation of InventoryClient wherever the application needs such implementation.

It goes without saying that we need a way to model reservations, so we keep it simple by adding only the necessary fields - the id of the car relevant to the reservation, the start and end dates, and a unique id for the reservation. Create a new class `org.acme.reservation.Reservation` with code as shown in [Listing 4.4](#). Additionally, we also include the `isReserved` method that checks whether the reservation overlaps with any of the days in the duration passed as arguments.

Listing 4.4 The source code of the Reservation class, which models reservations

```
package org.acme.reservation.reservation;

import java.time.LocalDate;

public class Reservation {

    public Long id;
    public String userId;
    public Long carId;
    public LocalDate startDay;
    public LocalDate endDay;

    /**
     * Check if the given duration overlaps with this reservation
     * @return true if the dates overlap with the reservation, false
     * otherwise
     */
    public boolean isReserved(LocalDate startDay, LocalDate endDay) {
        return (!(this.endDay.isBefore(startDay) ||
            this.startDay.isAfter(endDay)));
    }
}
```

Having this model in place, we now turn our attention to persisting these reservations, as a reservation system that does not persist reservations makes little sense in practice! Following good Object-Oriented design principles, we model the interactions with the persistence layer via an interface named `ReservationsRepository`. We only need this layer to do two things: Save a reservation and retrieve all reservations. The new interface `org.acme.reservation.reservation.ReservationsRepository` can be written as shown in [Listing 4.5](#).

Listing 4.5 The source code of the ReservationsRepository interface which abstracts how the reservations are stored to and retrieved from the data store

```
package org.acme.reservation.reservation;

import java.util.List;

public interface ReservationsRepository {

    List<Reservation> findAll();

    Reservation save(Reservation reservation);
}
```

As we are not using a full-blown database just yet, we are using a simple list as a stub created in the `InMemoryReservationsRepository` class with code that looks as demonstrated in [Listing 4.6](#).

Listing 4.6 The source code of the InMemoryReservationsRepository class which is the simplest possible implementation of ReservationsRepository

```
package org.acme.reservation.reservation;

import java.util.Collections;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicLong;
import jakarta.inject.Singleton;

@Singleton
public class InMemoryReservationsRepository
    implements ReservationsRepository {

    private final AtomicLong ids = new AtomicLong(0);
    private final List<Reservation> store =
        new CopyOnWriteArrayList<>(); #1

    @Override
    public List<Reservation> findAll() {
        return Collections.unmodifiableList(store);
    }

    @Override
    public Reservation save(Reservation reservation) {
        reservation.id = ids.incrementAndGet(); #2
        store.add(reservation);
        return reservation;
    }
}
```

#1 This (thread-safe) list is essentially our data store.

#2 Assign a unique id to the reservation we are about to save.

We can now turn our attention to creating a REST endpoint handling the GET HTTP requests under the `/reservation/availability` path, which returns all available cars for the given start and end date (supplied as query parameters). Let's create a new class representing our REST resource in `org.acme.reservation.rest` package called `ReservationResource` with the code as shown in [Listing 4.7](#). If there's a generated `GreetingResource` class, you may remove it (but it shouldn't be there if you used the `-no-code` flag when creating the project).

The implementation of the `availability` method isn't complicated. It retrieves all the available cars from the inventory, checks which are already reserved for the dates in question and returns the remaining ones.

Listing 4.7 The source code of the `ReservationResource` class which handles HTTP GET requests for the `/reservation/availability` path

```
package org.acme.reservation.rest;

import java.time.LocalDate;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
import org.acme.reservation.inventory.Car;
import org.acme.reservation.inventory.InventoryClient;
import org.acme.reservation.reservation.Reservation;
import org.acme.reservation.reservation.ReservationsRepository;
import org.jboss.resteasy.reactive.RestQuery;

@Path("reservation") #1
@Produces(MediaType.APPLICATION_JSON) #2
public class ReservationResource {

    private final ReservationsRepository reservationsRepository;
    private final InventoryClient inventoryClient;

    public ReservationResource(ReservationsRepository reservations,
                               InventoryClient inventoryClient) { #3
        this.reservationsRepository = reservations;
        this.inventoryClient = inventoryClient;
    }

    @GET
    @Path("availability") #4
    public Collection<Car> availability(@RestQuery LocalDate startDate,
                                         @RestQuery LocalDate endDate) { #5
        // obtain all cars from inventory
        List<Car> availableCars = inventoryClient.allCars();
    }
}
```

```

// create a map from id to car
Map<Long, Car> carsById = new HashMap<>();
for (Car car : availableCars) {
    carsById.put(car.id, car);
}

// get all current reservations
List<Reservation> reservations = reservationsRepository.findAll();
// for each reservation, remove the car from the map
for (Reservation reservation : reservations) {
    if (reservation.isReserved(startDate, endDate)) {
        carsById.remove(reservation.carId);
    }
}
return carsById.values();
}
}

```

#1 By annotating the class with `@Path("reservation")` we ensure that Quarkus uses this class to handle all HTTP requests under the `/reservation` path.

#2 By annotating the class with `@Produces(MediaType.APPLICATION_JSON)` we ensure that all methods that handle REST calls return a JSON response. This makes Quarkus serialize the result of the method to JSON when returning the HTTP response.

#3 Quarkus creates the instance of `ReservationResource` for us by calling this constructor providing the implementations of both the `ReservationsRepository` and the `InventoryClient` interfaces that we provide (as they are `@Singleton CDI beans`).

#4 By annotating the method with `@GET` and `@Path("availability")` we establish that the framework invokes this method when an HTTP GET call is made to the `/reservation/availability` path (the `reservation` part comes from the value of the `@Path` annotation used on the class).

#5 The two parameters are annotated with `@RestQuery`, meaning that they represent the values of the HTTP query parameters named `startDate` and `endDate` respectively (matching the names of the variables). The values need to be in the format of `YYYY-MM-DD`.

4.2.2 Making a reservation

Now that we have seen how to handle the case where, given a start and end date, we need to return a list of available cars for rent, the next step is to make a reservation of a specific car for a set of dates. This POST HTTP endpoint (because it creates a new reservation) accepts JSON input representing the reservation. The new `ReservationResource#make` method handles this endpoint, as shown in [Listing 4.8](#).

Listing 4.8 Partial source of the ReservationResource class which will handle HTTP POST requests under /reservation

```
// import jakarta.ws.rs.Consumes;

@Consumes(MediaType.APPLICATION_JSON) #1
@POST #2
public Reservation make(Reservation reservation) { #3
    return reservationsRepository.save(reservation); #4
}
```

#1 The method accepts a JSON input, so we use the proper @Consumes annotation for this method.

#2 This method accepts HTTP POST requests, so we use the @POST annotation.

#3 By simply declaring Reservation as a method parameter (without any annotation), we establish that Quarkus deserializes the HTTP request body into an instance of this class.

#4 All the method has to do is save the reservation and return it, thus ensuring that Quarkus serializes it as the HTTP response body.

4.2.3 Experimenting with the exposed REST API using the Swagger UI

Having written all this code, we need a way to exercise it and see how it behaves in practice. There are multiple ways to do this, but we introduce a graphical method using Quarkus' built-in Swagger UI (provided by the `smallrye-openapi` extension) for this section. This UI is available at <http://localhost:8081/q/swagger-ui> URL. When accessed and the application still runs in Dev mode, it looks like presented in [Figure 4.2](#). The UI shows the two REST endpoints we developed and provides an easy way to test them.

TIP By default, only the Dev mode includes the Swagger UI. If you would like to have it also in production, you can add the `quarkus.swagger-ui.always-include=true` to your configuration.

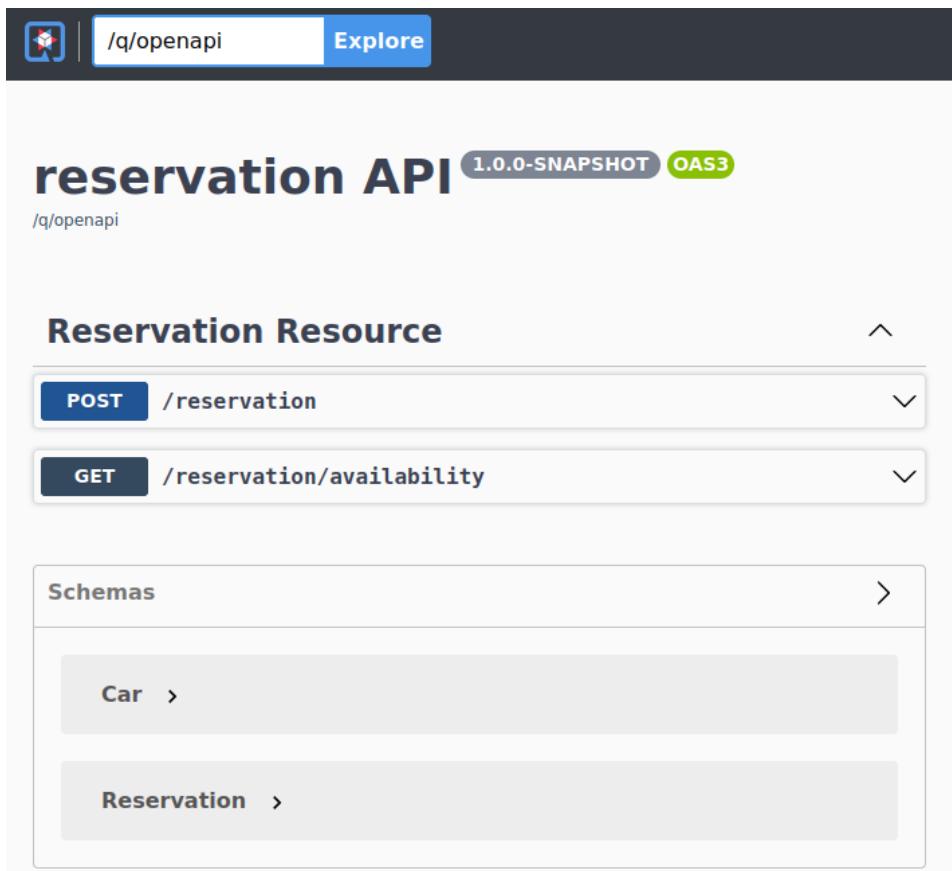


Figure 4.2 Quarkus provides a UI for executing REST operations by leveraging Swagger UI

The Swagger UI is a graphical representation of the OpenAPI document available at <http://localhost:8081/q/openapi>. This document generates as defined by the MicroProfile OpenAPI specification. It is also possible to customize how it generates. However, that is beyond our scope. If you are interested in how the OpenAPI extension works, you can check the official documentation at <https://quarkus.io/guides/openapi-swaggerui>. Listing 4.9 provides a part of the generated OpenAPI document. This part shows some high-level metadata of the service (title and version), as well as the definition of the method for making a reservation (`ReservationResource#make`) - you can see that it accepts as well as produces a JSON format with the corresponding schema to a `#/components/schemas/Reservation` object (which is a representation of the `Reservation` class from our code). You can check the response from <http://localhost:8081/q/openapi> if you want to see the whole document.

Listing 4.9 The OpenAPI document of the Reservation service

```
---
openapi: 3.0.3
info:
  title: reservation-service API
  version: 1.0.0-SNAPSHOT
paths:
  /reservation:
    post:
      tags:
        - Reservation Resource
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Reservation'
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Reservation'
```

After trying out the `GET /reservation/availability` endpoint (either in the Swagger or with a different tool like cURL) using any start and end dates, we see that the application responded with an HTTP 200 response code and a response body containing all the cars (since there are no reservations yet in the system). [Listing 4.10](#) shows an example of such a response.

TIP If you want to try out the REST API using cURL, you will find it helpful that the Swagger UI, whenever you execute an operation in its UI, generates and shows an example cURL command that you can copy and paste into your terminal.

Listing 4.10 Example JSON response when no reservations exist in the system

```
[
  {
    "id": 1,
    "licensePlateNumber": "ABC-123",
    "manufacturer": "Toyota",
    "model": "Corolla"
  },
  {
    "id": 2,
    "licensePlateNumber": "ABC-987",
    "manufacturer": "Honda",
    "model": "Jazz"
  },
  {
    "id": 3,
    "licensePlateNumber": "XYZ-123",
    "manufacturer": "Renault",
    "model": "Clio"
  },
  {
    "id": 4,
    "licensePlateNumber": "XYZ-987",
    "manufacturer": "Ford",
    "model": "Focus"
  }
]
```

To make a reservation for the first car with a starting date of 2024-01-01 and end date of 2024-01-05, we use the `POST /reservation` endpoint with the following JSON shown in [Listing 4.11](#):

Listing 4.11 The JSON request body to make a reservation for the car with id 1

```
{
  "carId": 1,
  "startDay": "2024-01-01",
  "endDay": "2024-01-05"
}
```

The response we get back is also a JSON representing a persisted reservation with the reservation id set in this case to 1 as demonstrated in [Listing 4.12](#).

Listing 4.12 The JSON response acknowledging the reservation

```
{
  "id": 1,
  "carId": 1,
  "startDay": "2024-01-01",
  "endDay": "2024-01-05"
}
```

Let's test the Reservation service to ensure we can't reserve the same car for any days in the 2024-01-01 - 2024-01-05 range. We do that by invoking the GET /reservations/availability endpoint with 2024-01-02 as the start date and 2024-01-04 as the end date (as we previously made a reservation for this same range). The result is available in [Listing 4.13](#).

Listing 4.13 The JSON response when the first car has been reserved for the dates we are using

```
[
  {
    "id": 2,
    "licensePlateNumber": "ABC-987",
    "manufacturer": "Honda",
    "model": "Jazz"
  },
  {
    "id": 3,
    "licensePlateNumber": "XYZ-123",
    "manufacturer": "Renault",
    "model": "Clio"
  },
  {
    "id": 4,
    "licensePlateNumber": "XYZ-987",
    "manufacturer": "Ford",
    "model": "Focus"
  }
]
```

We see that car with id 1 is missing from this list, as expected.

4.3 Using the REST Client

In addition to exposing REST endpoints, Quarkus also makes it very easy to consume REST endpoints. To showcase this capability, we create the Rental service that the Reservation service calls when making a reservation with the starting day equal to today, meaning that the rental starts with the confirmed reservation.

For the time being, the Rental service is a REST endpoint with a simple actual functionality of rental tracking. In the parent directory, we can create it using this command:

```
$ quarkus create app org.acme:rental-service -P 3.2.0.Final --extension \
resteeasy-reactive-jackson --no-code
```

TIP Notice that we don't need to provide the full name of the extensions. Quarkus deducts the correct extension if possible.

To avoid potential port conflicts when running multiple services together, open the `src/main/resources/application.properties` file and add the configuration property `quarkus.http.port=8082`.

Once created, we can develop an `org.acme.rental.Rental` class that represents a simple model of a rental with the code available in [Listing 4.14](#):

Listing 4.14 The source code of the Rental class which models rentals

```
package org.acme.rental;

import java.time.LocalDate;

public class Rental {

    private final Long id;
    private final String userId;
    private final Long reservationId;
    private final LocalDate startDate;

    public Rental(Long id, String userId, Long reservationId,
                  LocalDate startDate) {
        this.id = id;
        this.userId = userId;
        this.reservationId = reservationId;
        this.startDate = startDate;
    }

    public Long getId() {
        return id;
    }

    public String getUserId() {
        return userId;
    }

    public Long getReservationId() {
        return reservationId;
    }

    public LocalDate getStartDate() {
        return startDate;
    }
}
```

The `org.acme.rental.RentalResource` is the REST endpoint that initiates a rental request. It can be composed as demonstrated in [Listing 4.15](#).

Listing 4.15 The source code of the RentalResource class which handles HTTP requests under /rental/start

```
package org.acme.rental;

import io.quarkus.logging.Log;

import java.time.LocalDate;
import java.util.concurrent.atomic.AtomicLong;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/rental")
public class RentalResource {

    private final AtomicLong id = new AtomicLong(0);

    @Path("/start/{userId}/{reservationId}") #1
    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Rental start(String userId,
                        Long reservationId) { #2
        Log.infof("Starting rental for %s with reservation %s", #3
                  userId, reservationId);
        return new Rental(id.incrementAndGet(), userId, reservationId,
                          LocalDate.now());
    }
}
```

#1 The {} syntax is used to capture the value of the subpaths.

#2 Note that userId and reservationId match the names in the subpaths of the @Path annotation.

Quarkus can therefore capture the values of these subpaths.

#3 We're using a Quarkus-specific class for logging. This is the recommended approach. You could also instantiate an org.jboss.logging.Logger instance instead, but the use of static methods is simpler.

Taking a step back, we see how easy it has been to create the various pieces of the microservice architecture we intend to implement. We have so far bootstrapped a working version of the Reservation and Rental services with a couple of commands and a small amount of clean, simple Java code. The only missing part is implementing the communication layer between reservation and rental to close the loop between them.

The first order of business to accomplish this task is to define the model for rentals on the reservation side of the communication. As this model is the same as the model used on the rental side, we copy the `Rental` class into the `org.acme.reservation.rental` package of the Reservation service. Next comes the definition of REST-style communication, which we implement in the Reservation service by creating an interface named `org.acme.reservation.rental.RentalClient`, as demonstrated in [Listing 4.16](#). This interface only has one method that takes start and end dates and returns a rental. The real value Quarkus provides here is that by requiring a few very simple annotations from us, it can provide an implementation of the interface, so we don't have to write any tedious HTTP-related code at all.

Listing 4.16 The source code of the `RentalClient` class which consumes the `/rental/start` REST endpoint

```
package org.acme.reservation.rental;

import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;

import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.reactive.RestPath;

@RegisterRestClient(baseUri = "http://localhost:8082") #1
@Path("/rental")
public interface RentalClient {

    @POST #2
    @Path("/start/{userId}/{reservationId}")
    Rental start(@RestPath String userId, #3
                  @RestPath Long reservationId);
}
```

#1 The `@RegisterRestClient` annotation informs Quarkus that this interface consumes a REST endpoint. Quarkus will use the metadata of this interface to provide an implementation of it automatically. The value of `baseUri` is set to the base HTTP path where our `Rental` service listens for requests. The configuration would typically provide this value (or it can even be received dynamically from a service discovery mechanism), but we make it simple for the time being.

#2 As this method consumes the `start` REST endpoint of the `Rental` Service, which is an HTTP POST endpoint, we annotate the interface method with `@POST`.

#3 To make Quarkus populate the HTTP paths that the `start` REST endpoint of the `Rental` service expects, we annotate the two parameters with `@RestPath`.

Finally, we update the `ReservationResource` to use our new client. The updated code looks as demonstrated in [Listing 4.17](#).

Listing 4.17 Updated source code of the ReservationResource class which handles HTTP POST requests under /reservation

```

import org.acme.reservation.rental.Rental;
import org.acme.reservation.rental.RentalClient;
import org.eclipse.microprofile.rest.client.inject.RestClient;

@Path("reservation")
@Produces(MediaType.APPLICATION_JSON)
public class ReservationResource {

    private final ReservationsRepository reservationsRepository;
    private final InventoryClient inventoryClient;
    private final RentalClient rentalClient;

    public ReservationResource(ReservationsRepository reservations,
                               InventoryClient inventoryClient,
                               @RestClient RentalClient rentalClient) { #1
        this.reservationsRepository = reservations;
        this.inventoryClient = inventoryClient;
        this.rentalClient = rentalClient;
    }

    @Consumes(MediaType.APPLICATION_JSON)
    @POST
    public Reservation make(Reservation reservation) {
        Reservation result = reservationsRepository.save(reservation);
        // this is just a dummy value for the time being
        String userId = "x"; #2
        if (reservation.startDay.equals(LocalDate.now())) {
            rentalClient.start(userId, result.id); #3
        }
        return result;
    }

    ...
}

```

#1 To make Quarkus use our RentalClient in the ReservationResource, we need to annotate the constructor (injected) parameter with @RestClient.

#2 In later parts of the book, where we talk about security, we will use a proper userId

#3 Starting the rental is simply a matter of calling the start method of the RentalClient interface which makes the remote HTTP call.

If you now try to invoke Reservation’s `POST :8081/reservation` with a start date of the current date, it will make the call to the Rental service, and you can verify with the log message in the Rental service that it receives the request as expected. Make sure that the Rental service is started (e.g., `quarkus dev`).

In the next section, we focus on creating the Inventory service which we then use to properly implement `org.acme.reservation.inventory.InventoryClient` of the Reservation Service, thus replacing the static list implementation we added in [Listing 4.3](#).

4.4 Developing Quarkus applications with GraphQL

GraphQL is an emerging query language that gives clients the power to ask for exactly what they need without over-fetching (receiving data that you don’t need) or under-fetching (having to execute many queries sequentially to get all the data you need), which helps reduce the number of requests required actually to perform what is needed, and saves network traffic. This makes GraphQL a good fit for constrained environments, like mobile applications.

Depending on how you put it to use, GraphQL can be viewed as an alternative to both SQL (which is more focused on data querying) and REST (which is more focused on APIs and operations). Thanks to statically defined schemas for endpoints, it enables powerful developer tools and easy schema evolution without having to introduce breaking changes.

The backing data storage can be a persistent database, any kind of in-memory data structure, or a combination of both - GraphQL merely defines the querying language. It doesn’t assume anything about the source of the data. You can also use GraphQL to create an API that connects multiple completely disparate data sources and exposes a unified view of the data obtained from these sources. In this section, we will show a relatively basic example of our data model derived from a set of Java classes, and we will only store the data inside in-memory data structures. Later, in Chapter 7, we will enhance the service further by using a persistent database for storing the exposed data.

The GraphQL schema describes the GraphQL data model. The main components of a schema are types and fields. A GraphQL type, when using GraphQL with a data model defined in Java, most often corresponds to a Java class containing several fields, which conform to the fields of the type. When a client asks for data, it specifies which fields need to be fetched.

A client interacts with a GraphQL endpoint by executing so-called GraphQL operations. You can evaluate these as rough equivalents of SQL queries or REST methods. GraphQL operations also have a clearly defined contract. They accept parameters and return a particular type. There are three types of operations that clients can execute when communicating with a GraphQL endpoint:

- `Query` - This is an equivalent of a `SELECT` with SQL or a `GET` operation with RESTful endpoints. It allows clients to ask for data and is a read-only operation, so it should not change any data.

- **Mutation** - This is an equivalent of `UPDATE` or `INSERT` in SQL terms and `PUT` or `POST` in REST terms. It's an operation that can also change existing data or add new data. Compared to the SQL equivalents, it can also return some data back to the client along with the update.
- **Subscription** - This special query does not return just one response. Instead, the result is a (potentially infinite) stream of responses in which new items can appear over time. You can use this for notifications about events on the data, for example, new orders that arrive in a shop or newly registered users. This is usually implemented using WebSockets.

We won't dive deep into all the features that GraphQL offers. For more information and tutorials you may visit <https://graphql.org/>.

In Quarkus, the support for GraphQL follows the MicroProfile GraphQL specification. At the time of writing this book, only the server-side components of GraphQL were part of this specification, but work was underway to integrate the client-side APIs. However, Quarkus supports both sides as two separate extensions. They are named `quarkus-smallrye-graphql` and `quarkus-smallrye-graphql-client`. Let's learn how to use them, starting with the server-side extension.

4.5 Car Rental Inventory service

In this section, we create the Inventory service for our car rental project. This service will manage the inventory of cars that are available for rental. It will expose a GraphQL-based API with relevant operations that allow retrieving information about the car inventory and managing it. Later, we will update the Reservation service to use a GraphQL client to obtain the list of cars from the Inventory service.

4.5.1 Exposing the Inventory service over GraphQL

For the GraphQL API that we're about to create, we use a code-first approach, meaning we describe the API with Java code, and the SmallRye GraphQL extension then takes care of translating this model into a GraphQL schema.

To create the skeleton for the project, use the following CLI command in the parent directory:

```
$ quarkus create app org.acme:inventory-service -P 2.14.1.Final \
--extension smallrye-graphql --no-code
```

To avoid port conflicts when running multiple services together, open the configuration file `application.properties` and add the property `quarkus.http.port=8083`.

TIP You can always refer to the [Table 4.1](#) from the beginning of this chapter that lists which services run on which ports.

Run the project in Dev mode. Now, create the `org.acme.inventory.model` package for domain model classes, `org.acme.inventory.service` for the GraphQL API, and `org.acme.inventory.database` for the database stub. Let's now add the `Car` class code in the `model` package. Remember, the Reservation service already has a `Car` class, and the plan is that it will be able to exchange information about cars with Inventory service. To make the classes compatible, let's name all fields the same so that their corresponding GraphQL fields will be equal. The code of the `Car` class is available in [Listing 4.18](#).

Listing 4.18 The source code of the `Car` class which represents a car available for rental

```
package org.acme.inventory.model;

public class Car {

    public Long id;
    public String licensePlateNumber;
    public String manufacturer;
    public String model;

}
```

We need the `CarInventory` class in the same `database` package to serve as our database stub (we will only use in-memory data structures for now). [Listing 4.19](#) demonstrates the code of the `CarInventory` class.

Listing 4.19 The source code of the `CarInventory` class which serves as a database stub for the car inventory

```
package org.acme.inventory.database;

import org.acme.inventory.model.Car;

import jakarta.annotation.PostConstruct;
import jakarta.enterprise.context.ApplicationScoped;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicLong;

@ApplicationScoped #1
```

```

public class CarInventory {

    private List<Car> cars;

    public static final AtomicLong ids = new AtomicLong(0);

    @PostConstruct
    void initialize() { #2
        cars = new CopyOnWriteArrayList<>();
        initData();
    }

    public List<Car> getCars() {
        return cars;
    }

    private void initData() { #3
        Car mazda = new Car();
        mazda.id = ids.incrementAndGet();
        mazda.manufacturer = "Mazda";
        mazda.model = "6";
        mazda.licensePlateNumber = "ABC123";
        cars.add(mazda);

        Car ford = new Car();
        ford.id = ids.incrementAndGet();
        ford.manufacturer = "Ford";
        ford.model = "Mustang";
        ford.licensePlateNumber = "XYZ987";
        cars.add(ford);
    }

}

```

#1 `@ApplicationScoped` defines this class as a CDI bean, similarly as `@Singleton`. Making this bean `@ApplicationScoped` ensures that Quarkus creates only one instance of it, so all beans injecting this inventory will share the same car database.

#2 Framework calls the method annotated with `@PostConstruct` when it instantiates the bean. In our case, we use it to initialize the database stub.

#3 The `initData` method, called within the `initialize` method, stores some cars in the car inventory, so these will always be present after the start of the application. If you want to have more than just two cars, feel free to add more per your imagination. IDs are automatically generated from an `AtomicLong`.

Next, we need to add the `GraphQLInventoryService` class in the `service` package that exposes our model as a GraphQL service. To do so, it utilizes annotations from the MicroProfile GraphQL specification. The code of this class is available in [Listing 4.20](#).

Listing 4.20 The source code of the `GraphQLInventoryService` class which exposes the fleet over GraphQL

```
package org.acme.inventory.service;

import org.acme.inventory.database.CarInventory;
import org.acme.inventory.model.Car;
import org.eclipse.microprofile.graphql.GraphQLApi;
import org.eclipse.microprofile.graphql.Mutation;
import org.eclipse.microprofile.graphql.Query;

import jakarta.inject.Inject;
import java.util.List;
import java.util.Optional;

@GraphQLApi
public class GraphQLInventoryService {

    @Inject
    CarInventory inventory;

    @Query
    public List<Car> cars() { #1
        return inventory.getCars();
    }

    @Mutation
    public Car register(Car car) { #2
        car.id = CarInventory.ids.incrementAndGet();
        inventory.getCars().add(car);
        return car;
    }

    @Mutation
    public boolean remove(String licensePlateNumber) { #3
        List<Car> cars = inventory.getCars();
        Optional<Car> toBeRemoved = cars.stream()
            .filter(car -> car.licensePlateNumber
                .equals(licensePlateNumber))
            .findAny();
    }
}
```

```

        if(toBeRemoved.isPresent()) {
            return cars.remove(toBeRemoved.get());
        } else {
            return false;
        }
    }

}

```

#1 The cars query retrieves the current list of cars belonging to the fleet.

#2 The register mutation serves for registering new cars into the fleet.

#3 The remove mutation serves for removing cars from the fleet. It returns true if a car was removed after invoking it, false otherwise.

4.5.2 Invoking GraphQL operations using the UI

Now, let's open the GraphQL UI (also known as GraphiQL) and play around by executing various GraphQL operations manually. While the service still runs in Dev mode, open <http://localhost:8083/q/graphql-ui/> in your browser. The UI looks like the one presented in [Figure 4.3](#).

There are three main parts to the page:

- The top-left window serves for writing your GraphQL queries.
- The bottom-left window provides values for variables if you use variables in your queries and also for supplying HTTP headers that the query should include.
- The gray window on the right displays the result of operations that you execute.

And there's also the blue Play button that executes the requested operation.

The screenshot shows the GraphiQL UI interface. At the top, there's a navigation bar with icons for file operations (New, Open, Save, etc.) and tabs for "GraphQL UI", "Prettify", "Merge", "Copy", and "History". Below the navigation is a code editor containing a GraphQL query:

```

1 • query {
2 •   cars {
3 •     licensePlateNumber
4 •     model
5 •     id
6 •   }
7 }

```

Below the code editor is a toolbar with "QUERY VARIABLES" and "REQUEST HEADERS" buttons. Under "QUERY VARIABLES", there is a single entry "1".

Figure 4.3 GraphiQL is an HTML and JavaScript based UI for manually executing GraphQL operations

Now, let's try some queries. Remember that we have a query named `cars` used to retrieve a list of cars currently registered in the fleet. The `Car` type contains fields named `licensePlateNumber`, `manufacturer`, and `model`. All in all, this means that the query that we need to obtain all cars along with all their fields is the one provided in [Listing 4.21](#).

Listing 4.21 This GraphQL query obtains all fields about all cars in the fleet

```

query {
  cars {
    id
    licensePlateNumber
    model
    manufacturer
  }
}

```

When transferring this query to the UI, note that the UI supports autocompletion based on the type information obtained from the GraphQL schema. It also creates the ending curly brackets when you type opening curly brackets. So, try typing just:

```
query {
}
```

And then place your cursor on the empty line in the middle, either press `Ctrl+Space` to get all possible suggestions right away or start typing the word `cars` (just the `c` will suffice). You will see something like this in [Figure 4.4](#) because the UI knows about all the queries offered by the GraphQL endpoint. Use the arrow keys and `Enter` to pick `cars` from the suggestions.

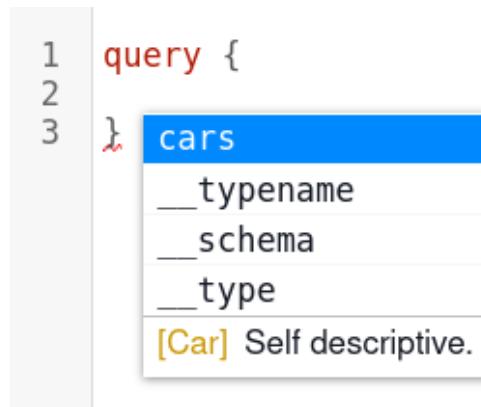


Figure 4.4 After pressing `Ctrl+Space` in the GraphQL UI, you get autocomplete suggestions

TIP The also offered `__typename`, `__schema` and `__type` suggestions represent built-in fields that allow reading metadata about the types in the schema. For example, if you request the `__typename` field as a subselection of a `Car`, the returned value will be `Car`.

Similarly, by autocompleting the query name, you get suggestions for fields of the `Car` type after pressing `Ctrl+Space` with the cursor in the middle of this unfinished query (or start typing the name of one of the fields):

```
query {
  cars {
    # selected fields from the `Car` type go here
  }
}
```

When the query completes and looks like in [Listing 4.21](#). Press the blue `Play` button to execute it. The result is in JSON format. It looks as presented in [Listing 4.22](#).

Listing 4.22 The result of the cars query contains information about all cars in the fleet

```
{  
  "data": {  
    "cars": [  
      {  
        "id": 1,  
        "licensePlateNumber": "ABC123",  
        "model": "6",  
        "manufacturer": "Mazda"  
      },  
      {  
        "id": 2,  
        "licensePlateNumber": "XYZ987",  
        "model": "Mustang",  
        "manufacturer": "Ford"  
      }  
    ]  
  }  
}
```

Try removing one of the subselection fields from the `cars` query and execute the query again. The result does not include that field.

Now that we've tried a query let's quickly also try a mutation. Recall that a GraphQL mutation is an operation that, unlike a query, can change data, meaning it is not read-only. The mutation that adds a new car into the fleet can look as demonstrated in [Listing 4.23](#). Again, `Ctrl+Space` can autocomplete most of this.

Listing 4.23 The register mutation serves for adding new cars into the fleet

```
mutation { #1
  register(car: { #2
    licensePlateNumber: "0123"
    model: "406"
    manufacturer: "Peugeot"
  }) { #3
    licensePlateNumber
    model
    manufacturer
    id
  }
}
```

#1 We use the mutation word to say this is a mutation, not a query.

#2 The register mutation has one parameter of type CarInput (we will explore that in a bit), so let's supply the fields to build an object of that type.

#3 The mutation not only creates an object but also returns the same object, as we described it in the source code. When the return type is a complex object, it is mandatory to specify which fields we want to fetch from that object. In our example, we fetch all of them, including the automatically generated ID.

After executing, the result looks like [Listing 4.24](#). You can see that the mutation echoes back the `Car` while storing it. The result contains the same data that we sent into the operation because we selected all fields from the returned type.

Listing 4.24 The output of the register mutation echoes back the car that we've added into the fleet

```
{
  "data": {
    "register": {
      "licensePlateNumber": "0123",
      "model": "406",
      "manufacturer": "Peugeot",
      "id": 3
    }
  }
}
```

You may now verify that the database contains the added car by executing the `cars` query again.

4.5.3 Reviewing the GraphQL schema

Recall that GraphQL schema is the contract that the service exposes, and all clients have to adhere to it to be able to communicate with that service. Let's briefly look at the GraphQL schema that the Quarkus GraphQL extension generated from the domain model. Access <http://localhost:8083/graphql/schema.graphql> either in your browser or by a terminal command of your choice. [Listing 4.25](#) shows the schema of the Inventory GraphQL service.

Listing 4.25 The GraphQL schema of the Inventory service contains definitions of all relevant GraphQL types

```

type Car { #1
    id: BigInteger
    licensePlateNumber: String
    manufacturer: String
    model: String
}

"Mutation root"
type Mutation { #2
    register(car: CarInput): Car
    remove(licensePlateNumber: String): Boolean!
}

"Query root"
type Query { #3
    cars: [Car]
}

input CarInput { #4
    id: BigInteger
    licensePlateNumber: String
    manufacturer: String
    model: String
}

```

#1 This defines a Car as an output type. The input counterpart is in bullet 4. The GraphQL language makes a strict distinction between output and input types, even when, like in our case, the types are identical (generated from the same Java class). We used the Car class as a parameter of the register mutation, which means the GraphQL extension created an input type, but we also used it as the return type of that mutation. Hence the extension also created the output type.

#2 The type Mutation clause shows all mutations in our schema and their parameters and return types.

#3 The type Query clause shows all queries in our schema and their parameters and return types. When a type is in square brackets, it means a collection of objects of that type. So the [Car] type corresponds to the List<Car> as the return type of the InventoryService#cars method.

#4 This is the input counterpart of the Car type, as described in bullet 1.

4.5.4 Consuming the Inventory service using a GraphQL client

We will now update the Reservation service to use a GraphQL client to obtain the real list of cars in the fleet from the Inventory service instead of using a hard-coded list of cars. So, the Reservation acts as a GraphQL client, whereas the Inventory acts as a GraphQL server. After this exercise, you can find the resulting state of the Reservation service in this book's GitHub repository `quarkus-in-action` inside the `chapter-04/reservation-service` directory.

We now need to run the Reservation and Inventory services at once so they can talk to each other. We will only change the Reservation code now, so you may run Reservation in Dev mode, and Inventory in production mode. But you can run both in Dev mode in parallel if you prefer! This is possible because we specified different default ports for both of them. So the Reservation runs on port 8081 while Inventory runs on port 8083 (see [Table 4.1](#) for reference).

Go to the directory of the Reservation service, and let's get started. First, add the GraphQL client extension by executing the following:

```
$ quarkus extension add smallrye-graphql-client
```

We will use a so-called *typesafe* client, meaning the code will use our domain classes directly. Quarkus also supports *dynamic* GraphQL clients. We will discuss the difference between these two types later. For a typesafe client, we need to add an interface that describes the client-side view of the GraphQL contract it uses to communicate with the server side. Create the `GraphQLInventoryClient` interface inside the `org.acme.reservation.inventory` package with the source code presented in [Listing 4.26](#).

Listing 4.26 The GraphQLInventoryClient interface describes the contract between the reservation and inventory

```
@GraphQLClientApi(configKey = "inventory") #1
public interface GraphQLInventoryClient extends InventoryClient {
    @Query("cars") #2
    List<Car> allCars();
}
```

#1 The `GraphQLClientApi` annotation declares this interface to be a client-side view of a GraphQL API.

The `configKey` parameter assigns a name to this particular GraphQL client. It serves to map the configuration values to this specific client. Any configuration that we will add related to this client will reference it by its name. This allows the definition of multiple GraphQL client APIs with different configuration values.

#2 The `InventoryClient` contains a method `allCars`, but the GraphQL query exposed by the `Inventory` service is named `cars` ([Listing 4.25](#)). By applying the `Query` annotation containing the correct name of the query, we instruct the library to execute the correct query when it invokes the `allCars` method.

The next step is to provide the configuration for our GraphQL client. The only required configuration value is the URL where the client should connect to the server. Add this line to `application.properties` of the `Reservation` service:

```
quarkus.smallrye-graphql-client.inventory.url=http://localhost:8083/graphql
```

Notice the word `inventory` inside the property's key. That refers to the `configKey` value in the `GraphQLClientApi` annotation that we added in the previous step.

Now update the `org.acme.reservation.reservation.ReservationResource` class to actually use the GraphQL-based `inventory` client instead of the `inventory` stub `InMemoryInventoryClient`. Change its constructor, as shown in [Listing 4.27](#).

By changing the injected type into `GraphQLInventoryClient` and specifying the name of the client that we want to inject in the `@GraphQLClient` annotation, we make the `ReservationResource` use a GraphQL client instead of the in-memory client stub. This also means that we can safely remove the stub we used before in class `InMemoryInventoryClient`.

Listing 4.27 The constructor of ReservationResource now injects an instance of GraphQLInventoryClient

```
public ReservationResource(ReservationsRepository reservations,
    @GraphQLClient("inventory") GraphQLInventoryClient inventoryClient,
    @RestClient RentalClient rentalClient) {
    this.reservationsRepository = reservations;
    this.inventoryClient = inventoryClient;
    this.rentalClient = rentalClient;
}
```

There is one more minor change needed before we can test it out. Because we use the `Car` class as a return value from the service, Quarkus needs to be able to deserialize it from JSON documents and construct its instances. It is, therefore it's necessary to provide a public no-argument constructor in the `Car` class. It can be empty. Add the following constructor the `org.acme.reservation.inventory.Car` class:

```
public Car() {
}
```

Now we can call the `availability` method and verify that it's using a GraphQL client instead of the `InMemoryInventoryClient`. For instance, with the HTTPie, it can be done as shown in [Listing 4.28](#). In the result, you can see the cars come from the Inventory service, as opposed to those hardcoded in `InMemoryInventoryClient`.

Listing 4.28 Calling the reservation availability which in turn collects the available cars from the Inventory service

```
$ http :8081/reservation/availability
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
content-length: 154

[
  {
    "id": 1,
    "licensePlateNumber": "ABC123",
    "manufacturer": "Mazda",
    "model": "6"
  },
  {
    "id": 2,
    "licensePlateNumber": "XYZ987",
    "manufacturer": "Ford",
    "model": "Mustang"
  }
]
```

HOW THE TYPESAFE CLIENT WORKS

In the Reservation service, we used a typesafe GraphQL client. The typesafe client works in a way that internally transforms Java classes into GraphQL types, as we saw with the `Car` class. On the server side, we called a query called `cars` that returns a list of `cars` (`[Car]` denotes the list in GraphQL terms). The typesafe client looks at all the fields that the `Car` class (in the client-side application) contains and builds an internal representation of the `Car` type from the server side. This representation does not have to be complete. It doesn't have to contain all fields of the target type. If it had to contain all fields, it would defeat one of the purposes of GraphQL, which is being able to choose which fields to fetch.

When the client API invokes the GraphQL operation, the library converts this invocation into a GraphQL request, in our case the request that obtains the list of cars. The constructed query request looks like this:

```

query {
  cars {
    id
    licensePlateNumber
    manufacturer
    model
  }
}

```

If we left out any of the fields of the `Car` class in the reservation application, then that field would not be part of the selection in the query. When the query is ready, the client sends it to the server over the HTTP protocol. The server responds with the JSON representation of a list of cars currently registered in the fleet. The typesafe client library converts this JSON array into a `List<Car>` returned from the `GraphQLInventoryClient#allCars` method.

DYNAMIC GRAPHQL CLIENT

While a typesafe GraphQL client works directly with model classes, there's also the so-called *dynamic* client as an alternative to it. Instead of automatically transforming classes to corresponding queries, the dynamic client offers a Domain Specific Language (DSL) that allows the developer to construct GraphQL documents using chained calls of Java methods while making the Java code looks as close as it can be to writing a raw GraphQL document directly. We won't go through a guided example for this one, but to give you a better idea of what the difference is, let's look at a few code snippets that implement an equivalent of the client that we built in the Reservation service. For example, to build the original `cars` query, we could write code as presented in [Listing 4.29](#).

Listing 4.29 An example `cars` query with a dynamic GraphQL client using the provided DSL

```

// import static io.smallrye.graphql.client.core.Document.document;
// import static io.smallrye.graphql.client.core.Field.field;
// import static io.smallrye.graphql.client.core.Operation.operation;

Document cars = document(
  operation("cars",
    field("id"),
    field("licensePlateNumber"),
    field("manufacturer"),
    field("model")
  )
);

```

The resulting `Document` object can then be passed to the client instance and sent over the wire to the target GraphQL service. The response is represented as an instance of the `Response` interface.

Listing 4.30 An example of request execution with dynamic GraphQL client

```
DynamicGraphQLClient client = ...; // obtain a client instance
Response response = client.executeSync(cars);
JSONArray carsAsJson = response.getData().asJSONArray();
// or, automatically deserialize a List of cars:
List<Car> cars = response.getList(Car.class, "cars");
```

The `Response` object contains all information about the response received from the server, which means the actual data and maybe a list of errors, if any occurred during the execution. The data can be retrieved in its raw form as a JSON document. Optionally, it's also possible to automatically deserialize from JSON and receive instances of domain objects.

Typesafe and dynamic clients offer similar features, but their use differs. Typesafe client is easier and more intuitive to use because you use domain classes directly but with a more complicated GraphQL schema, it might be tricky to properly construct Java counterparts of the GraphQL types in the schema. Also, if you have more operations that work with the same type, but they require fetching a different set of fields from that type, then you will need different Java classes that represent that type, each with exactly the fields that you require. With a dynamic client, this is a little easier because you don't include some fields in the documents that describe the operations.

4.6 Developing Quarkus services with gRPC

In previous sections, we learned about REST and GraphQL technologies. This section focuses on the gRPC (gRPC Remote Procedure Calls, <https://grpc.io/>), an open-source framework for remote process communication that is an excellent match for microservice architectures. Previous experience with gRPC is helpful but not required. We provide a brief overview of the technology for everyone to connect at the same level to explain the gRPC integration in Quarkus.

4.6.1 Understanding when to use gRPC

Have you ever implemented a connection to service over HTTP and felt that the limitations of the protocol constrict you? Has latency ever become a real problem for you and your team? Does the JSON format seems overkill for your needs, and are you looking for something more compact and efficient?

If you find these questions relatable, you may need to check out gRPC. In many real-world cases, many projects must keep the latency to the minimum. gRPC facilitates this by using a binary protocol (which is inherently faster than human-readable text-based protocols) and supporting multiplexing (sending multiple requests concurrently over one connection before receiving responses for them), which is faster than the typical HTTP/1 architecture, where requests and responses are serialized one after another.

[Figure 4.5](#) shows the sending of requests and responses over a single TCP connection with and without multiplexing. The first row displays HTTP/1.1, which does not support multiplexing. The second row shows HTTP/2, which does support multiplexing and is used under the hood by gRPC. Notice how multiple requests take less time to fulfill when using multiplexing.



Figure 4.5 Visualizing the value of multiplexing in HTTP2

4.6.2 Getting familiar with the protocol buffers

Protocol buffers specification, also known as protobuf, is a serialization format for typed structured data. It is compact, efficient, and language-independent. Protobuf can use binary and text formats, but the former is way more popular, and thus the book exclusively focuses on it. Compared to JSON, protobuf messages are smaller and easier to serialize/deserialize. On the other hand, binary messages are not readable.

Another critical characteristic of protobuf messages is that they have a schema definition. Users create a special file format called *proto* that describes the schema definition. Different tools then compile the proto file into language-specific code for representing, serializing, and deserializing the protobuf messages.

Let's use the `Car` example in the Inventory service and analyze how we can represent it in proto (see [Listing 4.31](#)). The file starts by setting the syntax to `proto3`, corresponding to the latest protocol buffer version format. The presented options are language-specific and define things like the Java package and how the framework should organize the generated code. Note that besides the Java package, we also define the protobuf package: `package inventory;`, which acts as a namespace for the message(s) defined below. Lastly, the proto file defines the structure of the `car` message. It specifies each field's type, name, and order (not to be confused with the default value).

The compilation process converts the fields from camel case (using the capital case to separate words) to snake case (using underscore to separate words) to align the code with the Protocol Buffer naming conventions. For example, `licensePlateNumber` is converted to `license_plate_number`.

Listing 4.31 Describing the Car model using Protocol Buffers

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.acme.inventory.model";
option java_outer_classname = "InventoryProtos";

package inventory;

message Car {
    string license_plate_number = 1;
    string manufacturer = 2;
    string model = 3;
    int64 id = 4;
}
```

There are multiple ways to compile a proto file, including the `protoc` compiler, build tool plugins, or the `quarkus-grpc` extension. Regardless of the compilation technique, the generated code includes a representation of the `Car` message in Java class accompanied by a builder implementation and methods for serializing / deserializing. [Listing 4.32](#) provides an example use of this generated code. The example shows how we can create an instance of `Car` programmatically and write it to an `OutputStream`. Note that code uses the builder pattern implemented as part of the generated code and then the `writeTo` method of the `Car` instance that writes it into an `OutputStream`.

Listing 4.32 An example using the generated protobuf code

```
public void writeCarData(OutputStream os, String licensePlateNumber,
    String manufacturer, String model) throws IOException {
    Car.newBuilder()
        .setLicensePlateNumber(licensePlateNumber)
        .setManufacturer(manufacturer)
        .setModel(model)
        .setId(id)
        .build()
        .writeTo(os);
}
```

4.7 Adding gRPC support to your project

As you might have already noticed from the architectural diagram ([Figure 4.1](#)), for the demonstration of the gRPC, we add the support for gRPC into the Inventory service, to show you the differences to the GraphQL approach. So if you want to follow the implementation, please execute the examples provided in this section in the `inventory-service` directory.

Message definitions are not the only thing we can define in proto files. We can also define remote service methods. In this case, the generated code includes client and server stubs. The server stubs are the base classes that we can utilize to implement the remote method. The client stubs are classes we can use from the client side to call the remote method. Quarkus users can use the `quarkus-grpc` extension for generating Java code from their proto files whether they are defining remote methods or not. Let's add this extension to the Inventory service. We can do this by either the CLI as demonstrated in [Listing 4.33](#) or manually. The resulting `pom.xml` file contains the dependency listed in [Listing 4.34](#).

Listing 4.33 Adding the gRPC extension using the CLI

```
$ quarkus extension add grpc
```

The command adds the dependency listed in [Listing 4.34](#) into your `pom.xml`.

Listing 4.34 Adding or verifying the presence of gRPC extension

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-grpc</artifactId>
</dependency>
```

The next step is to add the proto file `inventory.proto` under `src/main/proto` directory. Let's use the Protocol Buffers definition from [Listing 4.31](#) as a base. We modify it to a service that adds and removes cars from the inventory. The `InventoryService` in [Listing 4.35](#) defines two methods. One that sends an `InsertCarRequest` and one that sends a `RemoveCarRequest`. Both methods return a `CarResponse` message representing either the added or the removed car. Note, that `Car` is renamed to `InsertCarRequest` and `CarResponse` for styling purposes. This is also convenient as it helps to prevent naming collisions with the code that we introduced in previous sections (e.g., [Listing 4.18](#)).

Listing 4.35 Defining an Inventory service contract in gRPC (`inventory.proto` file)

```

syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.acme.inventory.model";
option java_outer_classname = "InventoryProtos";

package inventory;

message InsertCarRequest {
    string licensePlateNumber = 1;
    string manufacturer = 2;
    string model = 3;
}

message RemoveCarRequest {
    string licensePlateNumber = 1;
}

message CarResponse {
    string licensePlateNumber = 1;
    string manufacturer = 2;
    string model = 3;
    int64 id = 4;
}

service InventoryService {
    rpc add(InsertCarRequest) returns (CarResponse) {}
    rpc remove(RemoveCarRequest) returns (CarResponse) {}
}

```

TIP Note that we only included the `id` field in the response object, not the request object (`InsertCarRequest`). IDs are automatically generated, so let's not allow the client to specify it.

The compilation process (`mvn clean package`) triggers the `quarkus-grpc` extension. The extension detects the proto file under `src/main/proto` directory and generates Java code under `target/generated-sources/grpc`.

The generated code includes Java classes for the protobuf messages `InsertCarRequest`, `RemoveCarRequest`, and `CarResponse`. It also includes a gRPC client and server stubs. The complete list of generated files can be seen in the output of the tree command available in [Listing 4.36](#).

Listing 4.36 The tree structure of the files generated from the the proto file definition

```
target/generated-sources/grpc
└── org
    └── acme
        └── inventory
            └── model
                ├── CarResponse.java
                ├── CarResponseOrBuilder.java
                ├── InsertCarRequest.java
                ├── InsertCarRequestOrBuilder.java
                ├── InventoryProtos.java
                ├── InventoryServiceBean.java
                ├── InventoryServiceClient.java
                ├── InventoryServiceGrpc.java
                ├── InventoryService.java
                ├── MutinyInventoryServiceGrpc.java
                ├── RemoveCarRequest.java
                └── RemoveCarRequestOrBuilder.java
```

If we compiled the proto file using the protoc compiler or any tool other than the quarkus-grpc extension, the generated code would be the same; with a minor exception. The `MutinyInventoryServiceGrpc` interface is unique to quarkus-grpc, and it represents the `InventoryService` using the SmallRye Mutiny library (<https://smallrye.io/smallrye-mutiny/>). Mutiny is a reactive programming library that is extensively used in Quarkus and is in-depth covered in Chapter 8 of this book. In this case, Mutiny acts as an alternative to the StreamObserver API. The StreamObserver API is what the protoc compiler uses by default. This chapter focuses exclusively on Mutiny, as it provides a more straightforward API. Additionally, it is the recommended API approach for the use with gRPC in Quarkus. Don't worry if you don't understand all the details yet. Chapter 8 focuses on reactive programming and will explain Mutiny concepts in more detail.

4.8 Implementing a gRPC service using Quarkus

Among the generated code lies the `InventoryService` interface. Implementing this interface is the only thing we need to expose the Inventory service via gRPC. Let's take a closer look at this file. It contains one Java method per RPC method in the proto file. In our case, it contains both methods for adding and removing cars to and from the inventory, as shown in [Listing 4.37](#). Both methods return a `Uni` of `CarResponse`. `Uni` (this type comes from Mutiny) represents a single asynchronous action. It's a stream that emits a single item or a failure.

Listing 4.37 The gRPC Inventory service interface

```
package org.acme.inventory.model;

import io.quarkus.grpc.MutinyService;

@io.quarkus.grpc.common.Generated(
    value = "by Mutiny Grcp generator",
    comments = "Source: inventory.proto") #1
public interface InventoryService extends MutinyService {
    #2
    io.smallrye.mutiny.Uni<org.acme.inventory.model.CarResponse>
        add(org.acme.inventory.model.InsertCarRequest request);
    #3
    io.smallrye.mutiny.Uni<org.acme.inventory.model.CarResponse>
        remove(org.acme.inventory.model.RemoveCarRequest request);
}
```

#1 The `@Generated` annotation states the proto source file.

#2 The add car RPC method.

#3 The remove car RPC method.

The [Listing 4.38](#) demonstrates the implementation of the gRPC `InventoryService` provided in a new class `org.acme.inventory.grpc.GrpcInventoryService` that we need to create in the Inventory service.

Listing 4.38 Implementing the gRPC Inventory service

```
package org.acme.inventory.grpc;

...

@GrpcService #1
public class GrpcInventoryService
    implements InventoryService { #2

    @Inject
    CarInventory inventory; #3

    @Override
    public Uni<CarResponse> add(
        InsertCarRequest request) { #4
        Car car = new Car();
        car.licensePlateNumber = request.getLicensePlateNumber();
        car.manufacturer = request.getManufacturer();
        car.model = request.getModel();
        car.id = CarInventory.ids.incrementAndGet();
        Log.info("Persisting " + car);
        inventory.getCars().add(car); #5

        return Uni.createFrom().item(CarResponse.newBuilder()
            .setLicensePlateNumber(car.licensePlateNumber)
            .setManufacturer(car.manufacturer)
            .setModel(car.model)
            .setId(car.id)
            .build()); #6
    }

    @Override
    public Uni<CarResponse> remove(
        RemoveCarRequest request) { #7
        Optional<Car> optionalCar = inventory.getCars().stream()
            .filter(car -> request.getLicensePlateNumber()
                .equals(car.licensePlateNumber))
            .findFirst();
    }
}
```

```

        if (optionalCar.isPresent()) {
            Car removedCar = optionalCar.get();
            inventory.getCars().remove(removedCar);
            return Uni.createFrom().item(CarResponse.newBuilder()
                .setLicensePlateNumber(removedCar.licensePlateNumber)
                .setManufacturer(removedCar.manufacturer)
                .setModel(removedCar.model)
                .setId(removedCar.id)
                .build());
        }
        return Uni.createFrom().nullItem();
    }
}

```

#1 The `@GrpcService` annotation tells Quarkus to manage this class as gRPC service.

#2 The class implements the `InventoryService` to provide gRPC operations.

#3 Injection of the `CarInventory` (an example of field injection as opposed to the constructor injection we used before).

#4 Override the `add` method to implement the add car request handling.

#5 Insert car into the inventory.

#6 Send response wrapped in Mutiny's `Uni`.

#7 Override the `remove` method to implement the remove car request handling.

The `quarkus-grpc` extension takes care of managing the actual gRPC server, and thus no other code is needed. The gRPC is ready to be used. The best way to try out the implementation is using the Dev mode as demonstrated in [Listing 4.39](#).

Listing 4.39 Starting the Dev mode with gRPC extension

```

$ quarkus dev
...
INFO [io.qua.grp.run.GrpcServerRecorder] (vert.x-eventloop-thread-0)
Started gRPC server on 0.0.0.0:9000 [TLS enabled: false]
...

```

The gRPC server starts using port 9000. It can be accessed using the command line tools like `grpcurl` (<https://github.com/fullstorydev/grpcurl>) or the Quarkus Dev UI available at <http://localhost:8083/q/dev>. Since the Dev UI doesn't require us to install anything, we will focus on it for now. On the main page of the Dev UI, there is now a visible gRPC panel, as shown in [Figure 4.6](#).

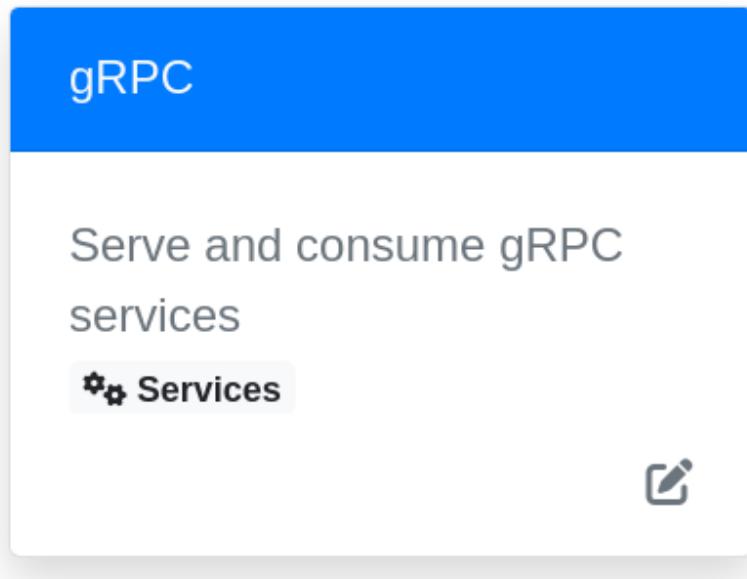


Figure 4.6 The gRPC entry point in the Dev UI of the Inventory service

The `Services` button takes you to the gRPC service table, which looks like the one shown in [Figure 4.7](#). The list contains a health endpoint `grpc.health.v1.Health` and our defined `inventory.InventoryService`. Each service has a `Test` link on the right side of the table that points to the respective service test page.

#	Name and Status	Implementation Class	Methods	
1.	✓ <code>inventory.InventoryService</code>	<code>org.acme.inventory.InventoryServiceImpl</code>	<ul style="list-style-type: none"> • <code>UNARY</code> remove • <code>UNARY</code> add 	✖ Test
2.	✓ <code>grpc.health.v1.Health</code>	<code>io.quarkus.grpc.runtime.health.GrpcHealthEndpoint</code>	<ul style="list-style-type: none"> • <code>UNARY</code> Check • <code>SERVER_STREAMING</code> Watch 	✖ Test

Figure 4.7 Listing the available gRPC services

Following the `Test` button of the `InventoryService`, the test page ([Figure 4.8](#)) contains text fields for both the `add` and `remove` requests, the `Send` buttons, and text areas for displaying the output. The input text areas are pre-populated with a blank request message. For instance, fill in the values for the `add` request and press the `Send` button. The car is now registered in the system.

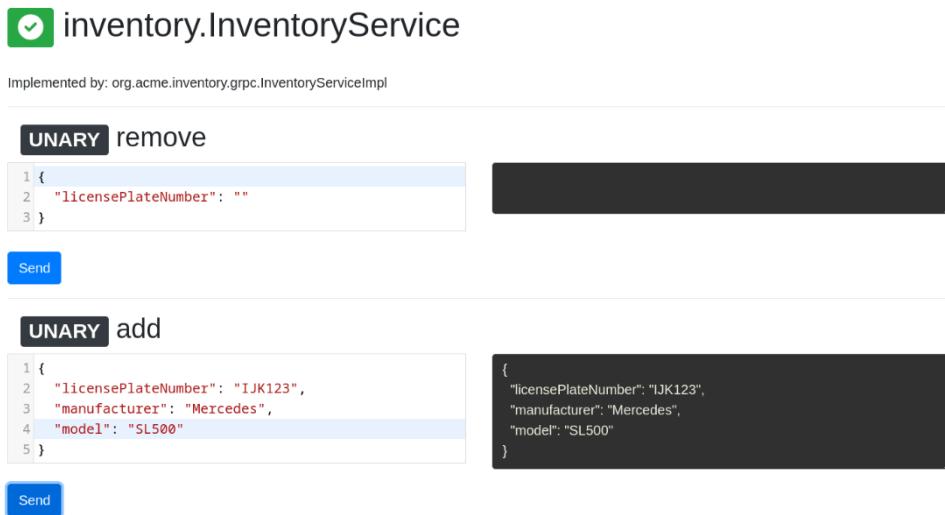


Figure 4.8 Adding a new car using the gRPC console

The in-memory list in the Inventory service now contains the newly persisted car. You can verify it using the GraphQL query returning all cars that is provided in [Listing 4.21](#).

The Dev UI makes it easy to try out the service and does not require implementing a client or using external tools like `grpcurl`. More importantly, all changes in the actual implementation take immediate effect, and you can test them without rebuilding and restarting the project. This includes changes to the actual proto file. We advise letting the Dev mode run through the next section to experience the added value of Dev mode firsthand.

4.8.1 Working with gRPC and streams

We have just learned the steps of defining protobuf messages and gRPC services, adding gRPC support in the Quarkus project, and implementing a gRPC using a `Uni` to send back a response. To build on top of that, we can extend the support of the server to send back multiple responses. In this case, we need to prefix the message type in the proto file with the `stream` keyword.

In the same spirit, the client may stream multiple requests too. To do that we need to use the `stream` keyword with the input messages of the RPC methods.

So, we have four distinct classes of gRPC services based on their streaming capabilities:

- Unary: The client sends a single request, and the server responds with a single response.
- Server streaming: Server returning multiple responses per call.
- Client streaming: The client sends multiple requests per call.
- Bidirectional streaming: The client and server send multiple requests and responses, respectively.

When we work with the `stream` of gRPC messages, we need to use Mutiny's `Multi` as the input or output type wrapper. For instance, the `InsertCarRequest` message in our case. `Multi` is similar to `Uni` but is capable of emitting multiple (potentially unbounded) items. Additionally, it emits a completion event to signal the end of the stream.

Let's convert the unary example of adding a new car we created in the Inventory service to bidirectional streaming so that we can add multiple cars per single connection and directly see how our Quarkus application processes them in the responses. The first step is to update the proto file (`inventory.proto`) and add the `stream` keywords to the `add` method, as shown in [Listing 4.40](#).

Listing 4.40 Adding the stream keywords to the add method for bidirectional streaming

```
rpc add(stream InsertCarRequest) returns (stream CarResponse) {}
```

The Dev mode should still be running, but if it's not, start it again. If it's already running, you can only refresh the Dev UI and see that the `add` method of the `InventoryService` now shows as bidirectional streaming. [Figure 4.9](#) shows how the method is now labeled as `BIDI_STREAMING`.

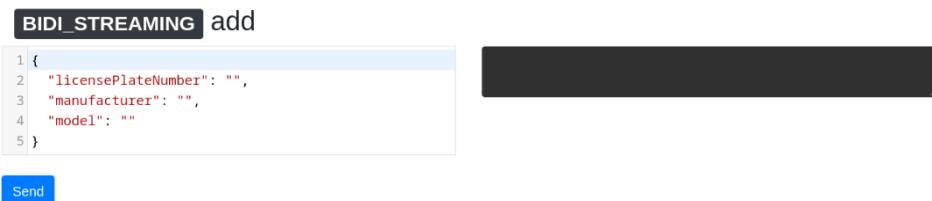


Figure 4.9 The BIDI_STREAMING version of the add RPC method

The `Send` button will not work just yet, as the `GrpcInventoryService` service implementation needs to be aligned. If pressed, the response text area displays an error message.

It should be fairly easy to align the service implementation. All we need to do is to change the implementation of the `add` method, as shown in [Listing 4.41](#). The `add` method now accepts a `Multi` of `InsertCarRequest` and produces a `Multi` of `CarResponse`. The implementation performs the same operations as the unary operation before on each `InventoryCarRequest`. First, it maps the received `InsertCarRequest` to the `Car` entity. Then it invokes the (in-memory) `persist` operation on each mapped car. Lastly, it maps the car to the `CarResponse` streamed back to the client. You may compare it to the unary version in [Listing 4.38](#).

Listing 4.41 Implementing the gRPC Inventory service with bidirectional streaming

```

@Override
public Multi<CarResponse>
add(Multi<InsertCarRequest> requests) { #1
    return requests
        .map(request -> { #2
            Car car = new Car();
            car.licensePlateNumber = request.getLicensePlateNumber();
            car.manufacturer = request.getManufacturer();
            car.model = request.getModel();
            car.id = CarInventory.ids.incrementAndGet();
            return car;
        }).onItem().invoke(car -> { #3
            Log.info("Persisting " + car);
            inventory.getCars().add(car);
        }).map(car -> CarResponse.newBuilder() #4
            .setLicensePlateNumber(car.licensePlateNumber)
            .setManufacturer(car.manufacturer)
            .setModel(car.model)
            .setId(car.id)
            .build());
}

```

#1 The add method accepts and produces types wrapped in Multi.

#2 Map each InsertCarRequest to Car instance.

#3 On each instance, invoke the persist operation.

#4 Map the Car to the CarResponse returned to the client.

With the implementation in place, our service Dev UI is back in its working state. Note that after adding the first car, a `Disconnect` button appears next to the `Send` button, allowing the client to stop sending requests (by disconnecting the connection). Send as many requests as you feel like, and then hit the `Disconnect` button. The car pops up in the response window with each sent car.

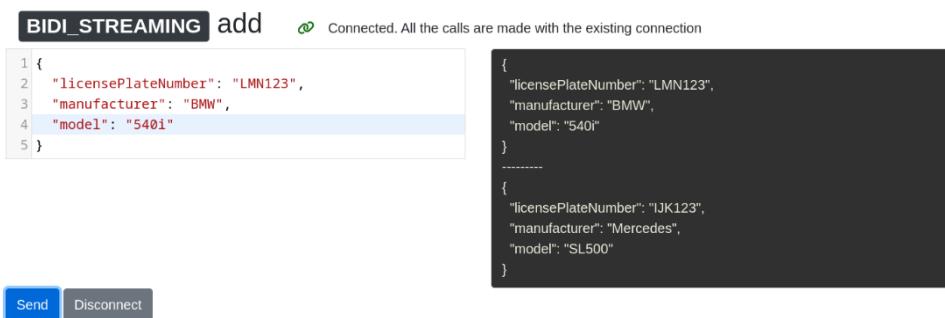


Figure 4.10 Adding more cars per single connection

4.8.2 Using a gRPC client with Quarkus

Previous sections teach how to set up gRPC in Quarkus to generate code from proto files and implement services using gRPC. The last step is learning how to use Quarkus to consume gRPC services. This section focuses on writing gRPC clients with Quarkus.

In particular, this section consumes the Inventory service already exposed via gRPC through a simple CLI application. For example, we can utilize such an application for the out-of-bounds administration tasks of the Inventory service. All that is required is just a Quarkus project with the gRPC extension and the proto file that describes the contract of the exposed Inventory service. Let's start with a unary version of the service that is simpler to consume. We are going to need the unary version of the Inventory service proto file defined in [Listing 4.35](#).

Let's create a new `inventory-cli` application with the `grpc` extension. In the parent directory, you can execute the following command.

```
$ quarkus create app org.acme:inventory-cli -P 2.14.1.Final \
--extension grpc --no-code
```

We can copy the proto file directly from the previous example [Listing 4.35](#) to the `src/main/proto/inventory.proto` file. Before we proceed with the client code, we need to define the connection details. The host and port that expose the Inventory service can be specified in the `application.properties` with the addition of these two properties respectively:

- `quarkus.grpc.clients.{client-name}.host` - target server host.
- `quarkus.grpc.clients.{client-name}.port` - target server port.

The `client-name` is the logical name of the client that we use as an identifier in case multiple clients are present. The `application.properties` excerpt in [Listing 4.42](#) contains the connection details of the Inventory service that you need to add to the `inventory-cli` configuration file. Note that the `client-name` in the configuration is `inventory`.

Listing 4.42 Configuring the gRPC client in the inventory-cli

```
quarkus.grpc.clients.inventory.host=localhost  
quarkus.grpc.clients.inventory.port=9000
```

To consume the service, all we need to do is add the `@GrpcClient` annotation on a property that has one of the generated stubs as type. The [Listing 4.43](#) presents a new class `org.acme.inventory.client.InventoryCommand` that uses a `@GrpcClient` with the logical name `inventory`. The role of the name is to correlate between code and configuration. In this case, the client connects to `localhost:9000`, the host/port combination configured for the name `inventory` in the configuration file `application.properties`.

Listing 4.43 Consuming the gRPC service with InventoryCommand

```

package org.acme.inventory.client;

import io.quarkus.grpc.GrpcClient;

import org.acme.inventory.model.InsertCarRequest;
import org.acme.inventory.model.InventoryService;
import org.acme.inventory.model.RemoveCarRequest;

public class InventoryCommand {

    @GrpcClient("inventory") #1
    InventoryService inventory; #2

    public void add(String licensePlateNumber, String manufacturer,
                    String model) {
        inventory.add(InsertCarRequest.newBuilder()
            .setLicensePlateNumber(licensePlateNumber)
            .setManufacturer(manufacturer)
            .setModel(model)
            .build()); #3
    }

    public void remove(String licensePlateNumber) {
        inventory.remove(RemoveCarRequest.newBuilder()
            .setLicensePlateNumber(licensePlateNumber)
            .build()); #4
    }
}

```

#1 Specify which client to inject with correlating to the configuration.

#2 Select the generated service stub to use, which is the `InventoryService` in our case.

#3 Invoke the remote service to add a car.

#4 Invoke the remote service to remove a car.

The code in [Listing 4.43](#) is perfectly valid, but it does need an entry point to trigger the invocation of the `add` and `remove` methods. We need a way to batch-import cars in the inventory from CSV (comma-separated value) files, so let's expose the application as a command line tool. To do so, add the `@QuarkusMain` annotation to the class and implement the `run` method of the `QuarkusApplication` interface. This is all that we need to do as shown in [Listing 4.44](#).

TIP We don't dive into the command line applications in this book. But if you are interested in writing CLIs with Quarkus, you can find more information at <https://quarkus.io/guides/command-mode-reference>.

You probably remember that the result of the `InventoryService` operation is `Uni`, which represents an asynchronous computation. So we need to block our thread if we want the operation to complete before we return from CLI invocation with `await().indefinitely()` which effectively makes it again a blocking call.

Listing 4.44 Consuming the gRPC service from the command line

```
package org.acme.inventory.client;

import io.grpc.GrpcClient;
import io.quarkus.runtime.QuarkusApplication;
import io.quarkus.runtime.annotations.QuarkusMain;
import org.acme.inventory.model.InsertCarRequest;
import org.acme.inventory.model.InventoryService;
import org.acme.inventory.model.RemoveCarRequest;

@QuarkusMain #1
public class InventoryCommand
    implements QuarkusApplication { #2

    private static final String USAGE =
        "Usage: inventory <add>|<remove> " +
        "<license plate number> <manufacturer> <model>";

    @GrpcClient("inventory")
    InventoryService inventory;

    @Override
    public int run(String... args) { #3
        String action =
            args.length > 0 ? args[0] : null; #4
        if ("add".equals(action) && args.length >= 4) {
            add(args[1], args[2], args[3]);
            return 0;
        } else if ("remove".equals(action) && args.length >= 2) {
            remove(args[1]);
            return 0;
        }
    }
}
```

```

}

System.err.println(USAGE);
return 1;
}

public void add(String licensePlateNumber, String manufacturer,
                 String model) {
    inventory.add(InsertCarRequest.newBuilder()
        .setLicensePlateNumber(licensePlateNumber)
        .setManufacturer(manufacturer)
        .setModel(model)
        .build())
    .onItem().invoke(carResponse ->
        System.out.println("Inserted new car " + carResponse))
    .await().indefinitely(); #5
}

public void remove(String licensePlateNumber) {
    inventory.remove(RemoveCarRequest.newBuilder()
        .setLicensePlateNumber(licensePlateNumber)
        .build())
    .onItem().invoke(carResponse ->
        System.out.println("Removed car " + carResponse))
    .await().indefinitely(); #6
}
}

```

#1 Marks this class as the main class.

#2 Tell Quarkus that this is a Quarkus application.

#3 Override the main entry point for the Quarkus application.

#4 Determine the action (add or remove).

#5 Invoke the add method and wait for the result.

#6 Invoke the remove method and wait for the result.

Once built (`mvn clean package`), the tool can be used from the command line, as demonstrated in the following snippet. Don't forget to start the Inventory service if it's not already running, so the CLI has a server to connect.

```
$ java -jar target/quarkus-app/quarkus-run.jar add KNIGHT Pontiac TransAM
```

When executed, the CLI client connects to the Inventory service acting as the gRPC server and inserts a new car. The CLI log prints the following message confirming that it inserted the car:

```
Inserted new car licensePlateNumber: "KNIGHT"
manufacturer: "Pontiac"
model: "TransAM"
```

We could also easily compile the `inventory-cli` application into a native image that we can utilize in various environments even without JVM.

4.9 Wrap up and next steps

In this chapter, we experimented with several communication protocols we can utilize in Quarkus applications. The first introduced protocol was REST. As we learned, Quarkus employs the JAX-RS specification through the new RESTEasy reactive project. Next, we looked into the raising alternative called GraphQL that solves the over-fetching and under-fetching problems of REST. For GraphQL, Quarkus utilizes the MicroProfile GraphQL specification through the SmallRye GraphQL extensions. Lastly, we analyzed the gRPC support in Quarkus with the gRPC extension that generated stubs for our implementation utilizing asynchronous types from the SmallRye Mutiny project.

We also created the first services of the Car Rental system: Reservation, Rental, and Inventory utilizing the mentioned protocols for communications. Additionally, we also developed an administration CLI application, `inventory-cli` that acts as a command line utility managing the car inventory in the Inventory service to which it connects through gRPC.

The subsequent chapters build on top of this base by providing various concepts that represents a requirement for communications handling in modern application deployments like security, testing, contract negotiations, fault tolerance, etc.

4.10 Summary

- Quarkus makes it trivial to implement and also consume REST-based services.
- Quarkus uses the RESTEasy project that implements the JAX-RS specification to define REST resources.
- The `smallrye-openapi` extension generates a Swagger API that describes REST resources present in an application and shows them as a HTML page that also allows testing them out directly in the browser.
- Consuming REST resources is possible in a typesafe way through an annotated interface.
- GraphQL is a great way to be more specific about the information a client is requesting from the server, limiting the size of data that travels through the network.
- GraphQL in Quarkus utilizes the MicroProfile GraphQL specification with additions from the SmallRye GraphQL project.
- Quarkus offers two separate GraphQL extensions - for the server and the client side.

- Server-side GraphQL support in Quarkus is provided using the code-first approach, where the developer provides the Java API, and Quarkus generates a schema out of it.
- gRPC is a protocol designed for dealing with the scaling issue of traditional HTTP-based communications.
- Quarkus simplifies the code generation process of gRPC-related services and manages the low-level gRPC code so users can focus on the business implementations.
- gRPC uses a serialization format named Protocol Buffers. Quarkus generates relevant classes from protobuf files provided by the developer.
- Apart from long-running server applications that listen for requests, it is possible to write command line applications or executable scripts with Quarkus. This is achieved by creating an entrypoint class that implements `QuarkusApplication` that is annotated with the `@QuarkusMain` annotation.
- gRPC and GraphQL are both focused on minimizing the network usage, so they are a good fit for cloud deployments where billing is based on network traffic.

5

Testing Quarkus applications

This chapter covers

- Discovering how Quarkus testing integrates with JUnit
- Developing tests that can execute with a traditional JVM as well as a native binary
- Using testing profiles to run tests with different configurations
- Creating testing mocks of remote services

In chapter 3, we already discussed some aspects of Quarkus application testing, namely the ability to test them continuously using the Dev mode. We also experimented with the Dev Services, which is the ability to automatically run instances of services needed for testing, such as databases or message brokers.

In this chapter, we dive a bit deeper into the capabilities that Quarkus provides for the testing of your applications. We explain the integration of Quarkus with JUnit 5, learn how to execute tests in native mode easily, and how to use testing profiles to execute tests with different configurations in one go. We also look at the facilities for creating mocks of CDI beans that are not suitable to be used directly during test execution. All these features are neatly integrated in Quarkus. We're sure you'll get accustomed to them quickly, saving a lot of time, effort and nerves compared to traditional testing approaches with different platforms.

For many software developers, writing and executing tests is just a necessary evil (perhaps just to satisfy some company policies). If this is your case, we hope that Quarkus' continuous testing has already started changing your mind. Let's see if we can go even further! For all the practical parts of this chapter, we will be enhancing the Reservation service developed in Chapter 4. The starting point from which we start adding things is in the `chapter-04/reservation-service` directory.

5.1 Writing tests

To write tests for your applications, Quarkus comes with its lightweight testing framework based on the JUnit 5 library (<https://junit.org/junit5/>). This framework takes care of spinning up actual application instances under test and tearing them down after tests finish. Quarkus utilizes this same framework for testing both in JVM mode and in native mode. Most tests work in both modes without changes. Among the facilities that the testing framework offers to test developers are:

- Injecting instances of the application's CDI beans directly into a test.
- Injecting URLs of the application's endpoints so you don't have to build the URL manually to invoke it.
- Integration with Mockito for creating mocks.
- Ability to override beans from the application with another implementation.
- Testing profiles so that different tests can run against a separate application instance with a different configuration.
- Starting and shutting down various services that the application needs (this is in addition to the Dev Services discussed in chapter 3 and can be used for custom services that Dev Services do not support). This is possible through providing an implementation of the `QuarkusTestResourceLifecycleManager` interface.

TIP Mocking and the ability to inject CDI beans from the application into the test are unavailable when testing in native mode. This is because while in JVM mode, the testing code runs in the same JVM as the application, native mode tests run as a JVM process communicating with a running pre-built binary, so the processes are separated and have to communicate over remote protocols, such as HTTP. Application beans can still use CDI injection, this limitation only applies to injecting into the test case itself.

5.1.1 Writing a simple test for the Reservation repository

In this practical part, we develop `ReservationRepositoryTest`, a test inside the codebase of the Reservation service. This test intends to verify that the `InMemoryReservationsRepository` can properly save a reservation and that it assigns an ID to it. For this test, we use a rather white-box approach that is directly injecting an instance of the `InMemoryReservationsRepository`, rather than communicating with it through a REST endpoint (`ReservationResource`).

One important note before we get to actually implementing the test. The Reservation service is set to bind to HTTP port 8081 (we added `quarkus.http.port=8081` into `application.properties`), but that's also the default port used by the testing instance during tests. When using Dev mode with continuous testing, these two instances run together so that they will clash. To mitigate this, set `quarkus.http.test-port=8181` in the `application.properties` configuration file of the Reservation service. It's also possible to set the port to 0. In that case, Quarkus automatically chooses any free port where the instance can be bound.

Let's create the test. Create a file named `ReservationRepositoryTest.java` inside the `src/test/java/org/acme/reservation` directory, the contents are in [Listing 5.1](#).

Listing 5.1 White-box testing of ReservationsRepository

```

package org.acme.reservation;

import io.quarkus.test.junit.QuarkusTest;
import org.acme.reservation.reservation.Reservation;
import org.acme.reservation.reservation.ReservationsRepository;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

@QuarkusTest      #1
public class ReservationRepositoryTest {

    @Inject
    ReservationsRepository repository; #2

    @Test
    public void testCreateReservation() {
        Reservation reservation = new Reservation(); #3
        reservation.startDay = LocalDate.now().plus(5, ChronoUnit.DAYS);
        reservation.endDay = LocalDate.now().plus(12, ChronoUnit.DAYS);
        reservation.carId = 384L;
        repository.save(reservation); #4

        Assertions.assertNotNull(reservation.id); #5
        Assertions.assertTrue(repository.findAll().contains(reservation));
    }
}

```

#1 Ensure that the Quarkus application gets automatically started and makes all the testing facilities available (for example, CDI injection).

#2 Inject an instance of ReservationsRepository (which will resolve to the InMemoryReservationsRepository singleton).

#3 Create an instance of Reservation.

#4 Save the reservation in the repository.

#5 Assert that the repository has assigned the reservation a non-null Id.

In this test, we create an instance of `Reservation`, set the reservation to start five days from now, end twelve days from now, and supply any car Id (the repository doesn't verify that such a car exists - we don't actually connect to the `Inventory` service here). Then we save the reservation in the repository and assert that the repository has assigned the reservation a non-null Id, and that the repository now contains it.

To execute the test, you can either use the Dev mode and run it via continuous testing (you can enable it by pressing `r` in the terminal), or use the traditional Maven-based way, where you add a test and then execute it using `mvn test`. Both approaches should yield the same result.

TIP If your IDE supports running tests directly from the UI, this should generally work too. At minimum, IntelliJ IDEA works fine.

5.2 Native testing

Because Quarkus offers a first-class integration with GraalVM and compiling into native binaries, this support naturally extends to testing as well. It is possible to run the same tests in JVM mode as well as native mode. In many cases, tests will continue working without changes when executed in native mode. There are a few caveats, though. As explained in the introduction to this chapter, mocking and CDI injection (into the test itself) are not supported in native mode tests.

Because native mode tests require compiling a full binary of the application, which takes a lot of time, they also aren't supported for Continuous testing. To execute them, one has to execute a separate Maven or Gradle build.

For tests meant to run in native mode, we use the `@QuarkusIntegrationTest` annotation. Tests annotated with `@QuarkusIntegrationTest`, as opposed to `@QuarkusTest`, are executed against a pre-built result of the Quarkus build, whether it is a JAR or a native binary. This means that an integration test can also be run in JVM mode, but in that case, the same limitations as for native mode apply (no injection or mocks) because the test runs in a separate JVM from the application. For this reason, it's not very practical to use `@QuarkusIntegrationTest` with JVM mode, and thus integration tests are skipped by default in JVM mode if you generated your project using standard Quarkus tools like the CLI or Maven plugin, they get enabled only when the native profile is active. This is controlled by the `skipITs` property that is added into the project model with a default value of `true`. In most cases, the recommended pattern is to use `@QuarkusTest` for tests meant to be used in JVM mode and `@QuarkusIntegrationTest` for native mode, where the integration test case can inherit from a JVM test case, and if some particular test methods are not compatible with native mode, they can be marked for skipping in native mode by `@DisabledOnIntegrationTest(forArtifactTypes = DisabledOnIntegrationTest.ArtifactType.NATIVE_BINARY)` annotation.

The diagram in [Figure 5.1](#) explains the difference between unit tests, integration tests, native mode, and JVM mode.

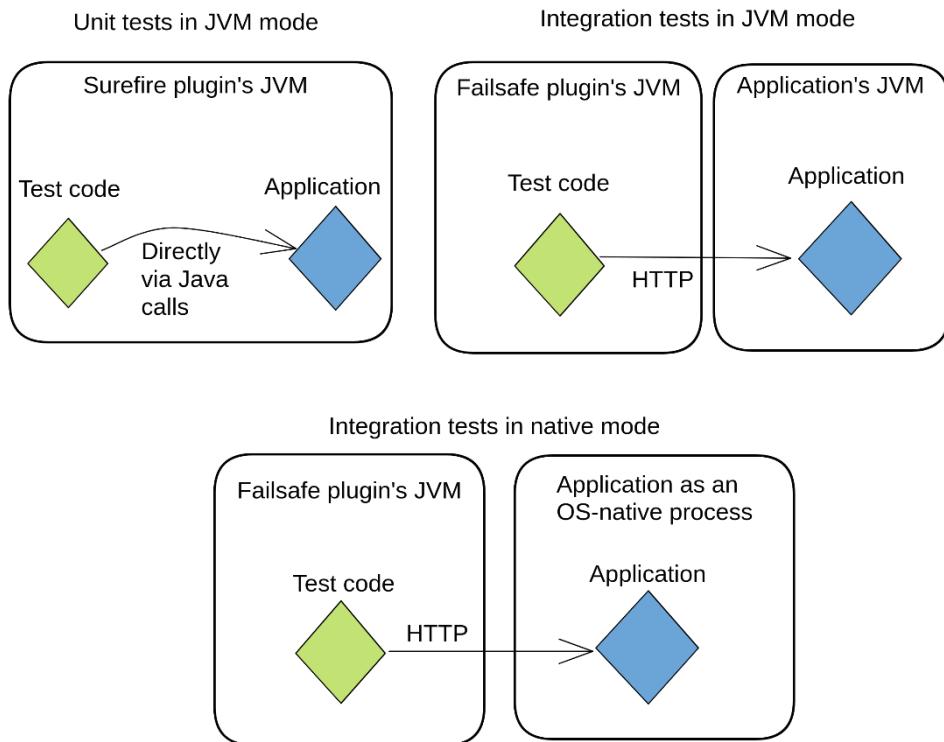


Figure 5.1 The difference between unit tests, integration tests, and integration tests in native mode

5.2.1 Writing a test for the Reservation resource

In this next part, we develop a test that can run both in JVM and native mode. We stick to the described pattern where we have a JVM test class annotated with `@QuarkusTest`, and the test class for the native mode that extends that class and it is itself annotated with `@QuarkusIntegrationTest`. The class for the native test is empty and only inherits tests from the JVM test case. If the JVM test can run without changes in native mode, then nothing else is necessary.

In the previous section, we wrote a low-level test for the `ReservationRepository`, injecting it as a CDI bean. That's why that test can't work in native mode. To develop a test that can work in native mode, we need to go a level higher - we still interact with the `ReservationRepository`, but this time, it is through a REST endpoint, `ReservationResource`. The communication between the test code and the application under test runs over HTTP protocol. Functionally, the test is very similar, but it looks quite different due to the use of REST instead of CDI.

TIP As you may remember, methods of the `ReservationResource` endpoint use a REST client and GraphQL client to connect to different services (`Rental` and `Inventory`). Our test avoids the need for that. The `make` method for creating a reservation calls the `Rental` service only if the starting day of the reservation is today, so we will avoid creating reservations starting today. The `availability` method needs a connection to the `Inventory` service, but this test doesn't use that method. It will be the subject of the next section, where we will substitute the calls to the `Inventory` service with a mock.

When developing Quarkus tests that invoke REST (or any HTTP-based) endpoints, it's recommended to use the `RestAssured` library, so we will do that. It's an API that significantly simplifies calling HTTP endpoints and verifying the expected results. The testing framework that comes with Quarkus also contains some built-in integration with `RestAssured` to make it easier to use - it automatically manages the target URLs, so you don't have to specify the URL of the server under test. The new test we're about to create, similar to the `ReservationRepositoryTest` creates a reservation, submits it, and then verifies that everything went correctly — an Id was assigned to the reservation, etc.

Because we chose not to include any generated code when we created the `Reservation` service (the `--no-code` option), we didn't include the `RestAssured` library that would be present if any tests were generated. So we need to add this dependency into the `pom.xml` in the `reservation-service` directory manually, as demonstrated in the following snippet, before we can utilize this library in the `ReservationRepositoryTest`:

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <scope>test</scope>
</dependency>
```

The test source of the `ReservationResourceTest` is as follows in [Listing 5.2](#).

Listing 5.2 The source code of `ReservationResourceTest`

```
package org.acme.reservation;

import io.quarkus.test.common.http.TestHTTPEndpoint;
import io.quarkus.test.common.http.TestHTTPResource;
import io.quarkus.test.junit.QuarkusTest;
import io.restassured.RestAssured;
import io.restassured.http.ContentType;
import org.acme.reservation.reservation.Reservation;
```

```

import org.acme.reservation.rest.ReservationResource;
import org.junit.jupiter.api.Test;

import java.net.URL;
import java.time.LocalDate;

import static org.hamcrest.Matchers.notNullValue;

@QuarkusTest
public class ReservationResourceTest {

    @TestHTTPEndpoint(ReservationResource.class)                      #1
    @TestHTTPResource
    URL reservationResource;

    @Test
    public void testReservationIds() {
        Reservation reservation = new Reservation();                      #2
        reservation.carId = 12345L;
        reservation.startDay = LocalDate
            .parse("2025-03-20");
        reservation.endDay = LocalDate
            .parse("2025-03-29");
        RestAssured                                         #3
            .given()
            .contentType(MediaType.APPLICATION_JSON)
            .body(reservation)
            .when()
            .post(reservationResource)
            .then()
            .statusCode(200)
            .body("id", notNullValue());                                #4
    }
}

```

#1 Inject the URL where a particular REST endpoint (in this case, `ReservationResource`) is available during the test run.

#2 Similar to the `ReservationRepositoryTest`, build an instance of a reservation.

#3 This RestAssured request will be converted to an HTTP POST request against the `/reservation` endpoint, passing our `Reservation` instance as the body in the JSON format.

#4 Because the response of the `/reservation` endpoint is a JSON representation of the reservation that we passed earlier, but with an Id already assigned, verify that the Id really did get assigned.

Note that the URL injection via `@TestHTTPEndpoint` is not an injection in the CDI sense, so it also works in native mode. This injected URL can then be passed to the `RestAssured` invocations when building requests.

You might now similarly execute this test as in the previous section either by running complete test execution (`mvn test`) or by running Continuous testing in Dev mode.

We have a test that works in JVM mode, so let's create the native mode counterpart. As explained earlier, we can achieve this by simply writing a class that extends the original test and is annotated with `@QuarkusIntegrationTest`. Such test class is shown in [Listing 5.3](#).

Listing 5.3 ReservationResourceIT, the test case that runs in native mode

```
package org.acme.reservation;

import io.quarkus.test.junit.QuarkusIntegrationTest;

@QuarkusIntegrationTest
public class ReservationResourceIT extends ReservationResourceTest {
}
```

Now, if you execute `mvn test`, only the JVM tests will run. To have native mode tests execute too, you have to execute `mvn verify -Pnative`, which activates the `native` profile. This instructs Quarkus to build a native binary as the build result and also to execute the `verify` goal, where the `failsafe` plugin picks up and executes integration tests, including our newly created `ReservationResourceIT`. The execution takes longer than usual because of the native binary compilation.

5.3 Mocking

Mocking is a technique used when you need to run a high-level test program that includes components that are not suitable for running in the test environment. This can be due to reasons like budgeting (for example, if a component sends SMS messages, which costs money, and you don't want to spend money on this while testing your product), or because the product needs to access something that is not available in the testing environment, or simply for performance purposes, where you use a mock of remote service to speed things up. The mocked service shouldn't constitute the main target that the test is supposed to verify because then your test would actually verify the functionality of a mock instead of the production code.

In this section, we learn about two mocking approaches that Quarkus' testing framework offers — mocking with CDI beans and the Mockito framework. We also look at practical examples for both methods.

5.3.1 Mocking by replacing implementations

One approach to mocking is to override a CDI bean with another implementation. In CDI terms, this can be achieved by annotating the mock (the implementation that should be used during tests) by `@Alternative` along with `@Priority(1)` annotations. If such bean shares some bean types with the original bean, this ensures that injection points requesting these bean types receive an instance of the mock during tests. Quarkus further simplifies this approach by offering a built-in `@io.quarkus.test.Mock` annotation. This is a CDI stereotype that applies the `@Alternative`, `@Priority(1)` and `@Dependent` annotations when it's placed on a class to provide all required code in a single annotation that defines a mocked CDI bean.

For example, you could mock the `InventoryClient` developed in chapter 4 with a bean shown as code in [Listing 5.4](#) and as a diagram in [Figure 5.2](#). To remind you what this client does, it is used to retrieve information about all cars from the `Inventory` service. The shown mock implementation returns a list containing a single hard-coded car. Note that this mock can be used to substitute any implementation of the `InventoryClient` interface, including the GraphQL-based implementation present in the Reservation service.

Listing 5.4 Mock implementation of InventoryClient

```
@Mock
public class MockInventoryClient implements InventoryClient {

    @Override
    public List<Car> allCars() {
        Car peugeot = new Car(1L, "ABC 123", "Peugeot", "406");
        return List.of(peugeot);
    }
}
```

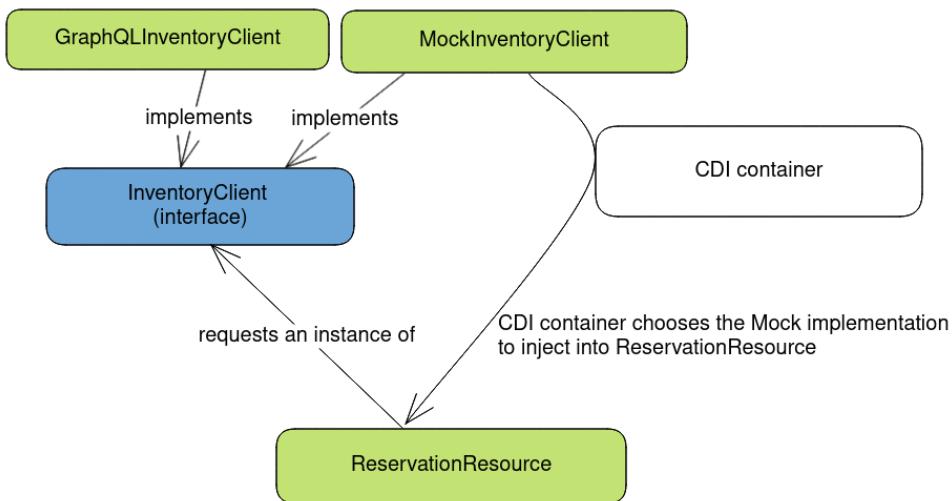


Figure 5.2 Mocking by replacing the implementation with a mock bean

It's enough to place this class somewhere in the sources for the tests (`src/test/java`) and it will be picked up automatically.

TIP Mock beans replace regular CDI beans and are resolved through regular CDI mechanisms. Therefore, for example, if you have multiple mocks that satisfy the requirements of a single injection point, the test will fail to start because of this ambiguity.

5.3.2 Mocking with Mockito

Another approach to mocking, as opposed to replacing bean implementations with different classes, is to use a mocking framework such as Mockito. Mockito offers a toolkit for building mock objects dynamically and defining what behavior they should exhibit when their methods are invoked.

TIP Mocking with Mockito, as opposed to using CDI alternatives, is not limited to using only CDI beans. Any object can be replaced with a mock. When used with CDI, the bean has to be of a normal scope, which means `@Singleton` and `@Dependent` beans can't be mocked out (because Mockito needs to use proxies for such beans, and these two scopes are non-proxyable). All other built-in CDI scopes will work.

Quarkus offers its integration with Mockito in the `io.quarkus:quarkus-junit5-mockito` Maven artifact. If you use Mockito in your tests, make sure you import this artifact (with a `test` scope) like this:

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

MOCKING THE INVENTORY CLIENT USING MOCKITO

Let's now add another test for the `Reservation` service. This test is the most complex and high-level of the three we're developing in this chapter. To reiterate, we have these two tests now:

- `ReservationRepositoryTest#testCreateReservation()`: Saves a reservation into the reservation repository by directly injecting the reservation repository using CDI. This is a white-box test.
- `ReservationResourceTest#testReservationIds()`: Saves a reservation into the reservation repository by calling the `ReservationResource` REST endpoint. This is more of a black-box test. It also supports running in native mode.

The new test, `testMakingAReservationAndCheckAvailability`, that we are going to create in the `ReservationResourceTest` class, will do the following:

- Call the `availability` method of the `ReservationResource` to get a list of all available cars for the requested date.
- Choose one of the cars (the first one) and make a reservation for the requested date.
- Call the `availability` method again with the same dates to verify that the car is not returned as available anymore.

We need mocking in this test because calling the `availability` method requires the `ReservationResource` to call the `Inventory` service. This is normally done with a GraphQL client (the interface `GraphQLInventoryClient`). But in our test, we don't assume that the `Inventory` service is available to be called. By replacing the GraphQL client with a mock, we ensure that a running `Inventory` service instance is not required for running the test.

TIP Of course, it is possible to design an integration test suite where multiple services are running simultaneously and can call each other. This is generally complicated because it involves controlling various applications' lifecycles, resourcing requirements, etc. In this case, we decided to simplify and replace calls to the other service with a mock.

[Listing 5.5](#) shows the relevant code. All this code should be added to the already existing `ReservationResourceTest`.

Listing 5.5 The `ReservationResource#testMakingAReservationAndCheckAvailability()` test

```
// If you're struggling with the static imports, here they are:  
//import static org.hamcrest.Matchers.hasSize;  
//import static org.hamcrest.Matchers.is;  
//import static org.hamcrest.Matchers.notNullValue;  
  
@TestHTTPEndpoint(ReservationResource.class)  
@TestHTTPResource("availability")  
URL availability; #1  
  
// uses mocks  
@DisabledOnIntegrationTest(forArtifactTypes =  
    DisabledOnIntegrationTest.ArtifactType.NATIVE_BINARY) #2  
@Test  
public void testMakingAReservationAndCheckAvailability() {  
    GraphQLInventoryClient mock =  
        Mockito.mock(GraphQLInventoryClient.class); #3  
    Car peugeot = new Car(1L, "ABC 123", "Peugeot", "406");  
    Mockito.when(mock.allCars())  
        .thenReturn(Collections.singletonList(peugeot));  
    QuarkusMock.installMockForType(mock,  
        GraphQLInventoryClient.class); #4  
  
    String startDate = "2022-01-01";  
    String endDate = "2022-01-10";  
    // Get the list of available cars for our requested timeslot  
    Car[] cars = RestAssured #5  
        .given()  
            .queryParam("startDate", startDate)  
            .queryParam("endDate", endDate)  
        .when().get(availability)  
        .then().statusCode(200)  
        .extract().as(Car[].class);
```

```

// Choose one of the cars
Car car = cars[0];
// Prepare a Reservation object
Reservation reservation = new Reservation(); #6
reservation.carId = car.id;
reservation.startDay = LocalDate.parse(startDate);
reservation.endDay = LocalDate.parse(endDate);
// Submit the reservation
RestAssured #7
    .given()
        .contentType(MediaType.APPLICATION_JSON)
        .body(reservation)
    .when().post(reservationResource)
    .then().statusCode(200)
        .body("carId", is(car.id.intValue()));
// Verify that this car doesn't show as available anymore
RestAssured #8
    .given()
        .queryParam("startDate", startDate)
        .queryParam("endDate", endDate)
    .when().get(availability)
    .then().statusCode(200)
        .body("findAll { car -> car.id == " + car.id + "}", hasSize(0));
}

```

- #1 Inject the URL of the /availability path under the root of the Reservation resource's path.
- #2 Because we are using mocks in this test, mark it as skipped when running in native mode.
- #3 Create a mock object that implements the GraphQLInventoryClient interface. When its allCars() method is called, return a hard-coded list of cars.
- #4 Override the GraphQLInventoryClient CDI bean type with the mock.
- #5 Call the availability method to retrieve the list of all available cars. This should return the single Peugeot we hardcoded into the mock inventory client.
- #6 Create a reservation object for the returned car.
- #7 Submit the reservation request and verify that the created reservation contains the correct carId field.
- #8 Call the availability method again and verify that the car with our Id doesn't exist anymore. Use a matcher to select from the list of returned cars and then assert that the selection returned an empty list (hasSize(0)).

5.4 Testing profiles

With the way we wrote tests until now, all tests execute against a single instance of the Quarkus application. This might not be optimal in all cases for two reasons. First, it might be hard to write the tests properly so that they are isolated and don't affect one another because a full cleanup of the state changes introduced by the test might not be practical or possible. The second reason is that it's impossible to test different configurations - all tests share the same application configuration. For these reasons, Quarkus offers testing profiles. These can be used to group test cases into groups (profiles) where test cases belonging to the same profile are run together on the same Quarkus instance. That instance is then torn down to be replaced by another instance for another test profile. While this increases the time needed to run the whole test suite, it gives the test developer a lot of flexibility.

A test profile can specify its own:

- Base Quarkus configuration profile from which configuration values are taken.
- The set of configuration properties overrides on top of the base configuration profile.
- Enabled alternatives (bean with an `@Alternative` annotation that should override the original beans).
- `QuarkusTestResourceLifecycleManager` implementations (custom resources that should be started before the test and shut down when they are no longer needed).
- A set of tags (see below).
- Command line parameters - only applicable when the test is a script with a `main` method.

A test profile may declare zero, one, or multiple tags. This allows the filtering of tests that should be run in each testing execution. Tags are simple strings, and when `quarkus.test.profile.tags` is defined when starting a test run (this property can list multiple tags separated by commas), only those tests that have at least one tag listed by the property will be executed. For example, if test 1 has tags `a` and `b`, test 2 has tags `b` and `c`, then if `quarkus.test.profile.tags` is `b`, then both tests will run. If the value is `c, d`, then only test 2 will run, and test 1 will be skipped because it doesn't declare any of the tags `c` or `d`.

A custom test profile is defined as a user-defined implementation of the `io.quarkus.test.junit.QuarkusTestProfile` interface. This interface contains several methods that define behavior related to the items listed in the above list. All of them have default implementations, so you can override just the ones you need. A minimal example that only provides some tags and overrides some configuration properties is shown in [Listing 5.6](#).

Listing 5.6 A test profile is an implementation of the QuarkusTestProfile interface

```
import io.quarkus.test.junit.QuarkusTestProfile;

import java.util.Map;
import java.util.Set;

public class RunWithStaging implements QuarkusTestProfile {

    // use the staging instance of a remote service
    @Override
    public Map<String, String> getConfigOverrides() {
        return Map.of("path.to.service",
                      "http://staging.service.com");
    }

    @Override
    public Set<String> tags() {
        return Set.of("staging");
    }
}
```

To mark a test to use a specific profile, you can use the `@TestProfile` annotation, as shown in [Listing 5.7](#).

Listing 5.7 The `@TestProfile` annotation is used to declare which test profile a test case uses

```
@QuarkusTest
@TestProfile(RunWithStaging.class)
public class StagingTest {

    @Test
    public void test() {
        // do something
    }
}
```

Given this declaration, the `StagingTest` will have the `path.to.service` property set when it runs. Furthermore, if the `quarkus.test.profile.tags` property is set, it has to contain the tag `staging` to enable the test, otherwise this test will be skipped.

5.5 Wrap up and next steps

In this chapter, we explored the facilities that Quarkus offers related to testing. It's not only the ground-breaking continuous testing that we introduced in chapter 3. It's also a set of neatly integrated tools on top of JUnit 5 and GraalVM. We learned how to write tests that can be run in native mode out-of-the-box, use CDI injection in tests, create mocks with Mockito, and split tests into isolated groups running with different setup and configuration values.

Now that we have covered the testing aspect, in the next chapter, we will focus on another crucial and cross-cutting part of application development—security.

5.6 Summary

- Tests for Quarkus applications run in JVM as well as native mode.
- Native mode isn't supported for tests that use mocks or CDI injection into the test itself because the application under test runs in a different OS process than the testing logic.
- Quarkus is tightly integrated with Mockito to allow easy mocking of application classes.
- Mocking is achieved either by replacing a CDI bean with an alternative implementation or by describing the mock's behavior using Mockito DSL.
- Testing profiles group tests into groups that run with a separate application instance and a potentially different configuration.
- Dev Services, the feature that automatically manages instances of databases, messaging brokers, etc., is supported during tests as well as when developing in Dev mode.

6

Exposing and securing web applications

This chapter covers

- Developing a basic secured web application with HTML
- Creating a more advanced HTMX-based UI
- Propagating the security context for calls between a web application and a REST service
- Exploring other alternatives for frontend development

In this chapter, we focus on two concepts: creating an HTML-based frontend for your application, and securing it to require authentication. We have already touched security with OIDC and Keycloak a little in chapter 3. We will now apply this concept in the car rental project.

Security, just like testing, is another aspect of software development that is often viewed as a boring necessary evil, and thus is often neglected in the early stages of a project. Properly securing an application is generally not an easy task. We will see how Quarkus addresses this and makes security as simple as possible, while still providing a high level of flexibility. Changing most security aspects of a Quarkus application is often just a matter of changing a few configuration properties, without having to update the code.

For the frontend development, Quarkus allows you to choose from many approaches and frameworks. There are pure Java solutions that try to minimize the need to write HTML and JavaScript (like Vaadin and its server-side rendering), but you can also use any framework that runs purely on the client side - based on HTML and JavaScript files that you bundle into the application. In this chapter, we present just one of the many approaches that you can take. We will use *Qute* (pronounced as "cute"), a server-side templating engine, and *HTMX*, a client-side toolkit for building responsive web applications that greatly simplifies asynchronous communication with the backend services. While Qute is provided by Quarkus extension and is an integral first-class member of the Quarkus ecosystem, HTMX is a third-party JavaScript-based library completely independent of Quarkus. Nevertheless, we will see that these two frameworks work together very nicely.

TIP While security and frontend development are generally separate topics, we decided to combine them together in this chapter for rather practical purposes, because parts of our frontend will require security to be already set up.

The outcome of this chapter will be a new service called `Users`, that exposes a simple HTML frontend allowing logged-in users to view their car reservations, view available cars for given dates and create new reservations.

[Figure 6.1](#) depicts the architecture what we will create in this chapter.

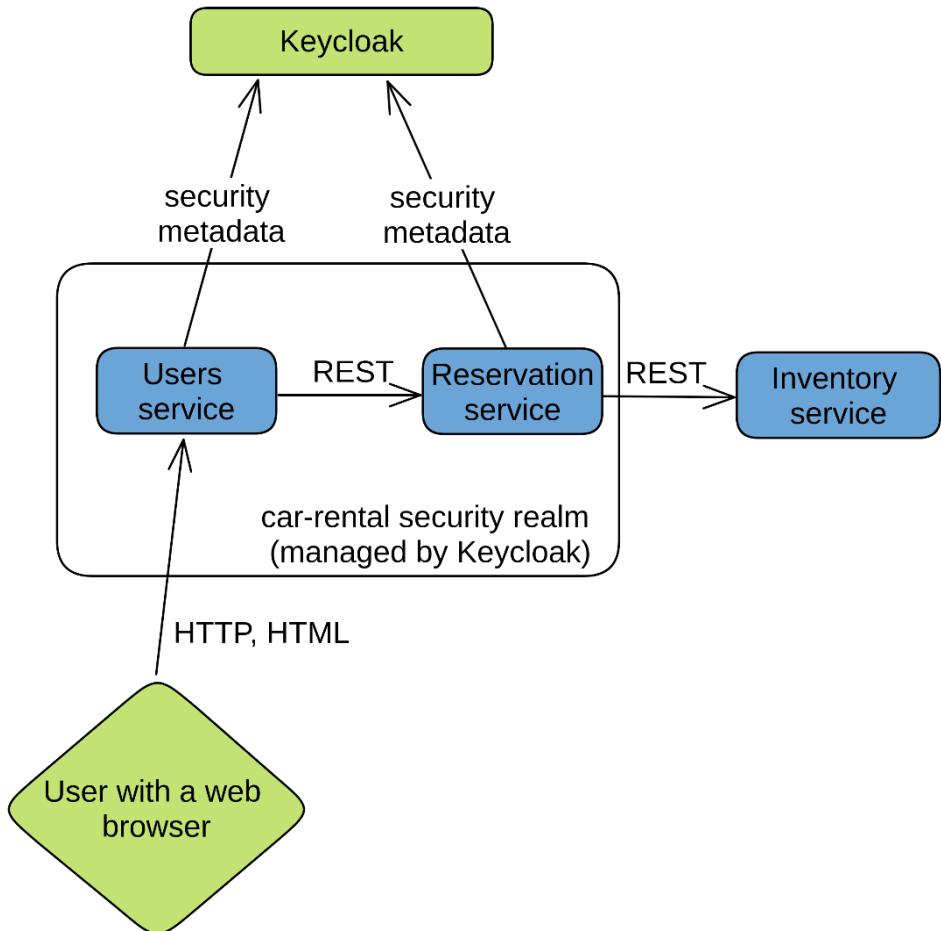


Figure 6.1 Diagram showing the architecture part used in chapter 6

In the practical parts of this chapter, we will use three services:

- The `Users` service, which we develop here from scratch.
- The `Reservation` service, which we already developed in chapters 4 and 5, but we will add security features to it now. The `Users` service, which makes REST calls to `Reservation`, propagates the information about the currently logged-in user (this is sometimes called "security context") along with these calls so that the `Reservation` service knows who's logged in.
- The `Inventory` service, also developed in chapter 4. We won't make any changes to it now, but it has to be running because it is required by the `Reservation` service to be able to work with cars and reservations.

The frameworks that we will use for the frontend parts are:

- Qute — a server-side templating engine designed mostly (but not only) for rendering HTML pages. It comes as a built-in Quarkus extension.
- HTMX — a client-side toolkit for building responsive web applications. It greatly simplifies asynchronous communication with the backend server, sometimes referred to as AJAX. This is achieved using special HTML elements, and in most cases, it removes or greatly reduces the need to write JavaScript code. It is a third-party open-source library not directly associated with Quarkus, but as you will see, it plays very nicely together with Qute.

Obviously, Qute and HTMX are not the only options you have for creating UI applications with Quarkus. If you're interested in seeing the alternatives, see Appendix A.

The solution is located in the `chapter-06` directory of the GitHub repository - here, you can find the new `users-service` project along with the updated `reservation-service`. The `Inventory` service doesn't receive any updates in this chapter, thus you may use the version from chapter 4, which you can find in the `chapter04/inventory-service` directory.

Before we actually create the frontend for working with reservations, we first learn how to secure web applications by creating a very simple HTML page that shows the username of the logged-in user.

6.1 Creating a secured web application

Let's dive right in by creating a new project for the `Users` service. If you're using the Quarkus CLI, then this is the command that you will need:

```
$ quarkus create app org.acme:users-service -P 3.2.0.Final --extension
- qute,resteasy-reactive-qute,oidc,rest-client-reactive-jackson,
- quarkus-oidc-token-propagation-reactive --no-code
```

We're using these extensions:

- `qute` is the templating engine that we will use to generate the HTML content.
- `resteasy-reactive-qute` allows exposing templates processed by Qute via a REST endpoint - we will create a REST endpoint that serves HTML resources and provides methods for asynchronously updating their contents in the next section.
- `oidc` for the security mechanisms. Just as in chapter 3, Keycloak will be used as the OIDC provider.
- `rest-client-reactive-jackson`, because the `Users` service communicates with the REST api exposed by the `Reservation` service, therefore we need the REST client extension.

- `quarkus-oidc-token-propagation-reactive`, because we need the `@AccessToken` annotation for propagating the OIDC ID token from the `Users` service to the `Reservation` service.

TIP When you run the application in Dev mode, a Dev Services Keycloak instance will automatically start because we've added the `oidc` extension. We will use that for our security concerns later. For now, note that it will slow down the start of Dev mode a bit because the Keycloak container takes a few seconds to start. Live reload only reloads the application itself though, so don't worry, it won't keep restarting the Keycloak container over and over when you apply changes in the application's code.

6.1.1 Creating a simple HTML page

We start by creating a very simple HTML page that shows the username of the currently logged-in user. Because the application doesn't have configured security just yet, the displayed username will be empty - we aren't required to log in, so the application is called anonymously. As the next step, we will configure the application to require authentication, so only then will the page actually show a username.

We're using the `Qute` engine to generate the web pages. `Qute` is a simple but powerful templating engine and comes as a Quarkus extension. It is most commonly used for generating HTML pages, but it's not limited to that. You may use it to generate any kind of document where you require the mixing of a template together with injectable parameters. Of course, an example is the best way to show how it works, so let's create a file `src/main/resources/templates/whoami.html` now in the `users-service` project as shown in [Listing 6.1](#):

Listing 6.1 The Qute template used for the WhoAmI HTML webpage

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Who am I</title>
</head>
<body>
<p>Hello {name ?: 'anonymous'}!</p>          #1
<a href="/logout">Log out</a>                  #2
</body>
</html>
```

#1 `{name}` is a placeholder that should be replaced by the value of the `name` parameter.

#2 This will create a hyperlink that allows a logged-in user to log out. We will use this later.

The only Qute-specific part of this template is the `{name}` placeholder. It is replaced by the value of the `name` parameter. This parameter contains the username of the currently logged-in user (we will set this up next). We also specify the name's default value, `anonymous`, if the template doesn't receive any value for this parameter.

To expose an HTML page generated from this template, we use a REST endpoint that produces the HTML content type and uses the `whoami.html` template to generate the output. This is achieved using the `org.acme.users.WhoAmIResource` class, as shown in [Listing 6.2](#).

Listing 6.2 The REST endpoint that exposes the WhoAmI template as an HTML page

```

package org.acme.users;

import io.quarkus.quot.Template;
import io.quarkus.quot.TemplateInstance;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;

@Path("/whoami")
public class WhoAmIResource {

    @Inject
    Template whoami;                      #1

    @Inject
    SecurityContext securityContext;        #2

    @GET
    @Produces(MediaType.TEXT_HTML)
    public TemplateInstance get() {
        String userId = securityContext.getUserPrincipal() != null
            ? securityContext.getUserPrincipal().getName() : null;
        return whoami.data("name", userId);
    }
}

```

#1 Inject an object representing the `whoami.html` template. This is automatically matched to the `whoami.html` file.

#2 Inject the current security identity to be able to see the metadata about the currently logged-in user.

The `get` method is a REST resource method that takes the `whoami` template, adds the necessary parameters to it (in this case, just the `name` parameter), and returns it - this is then passed to the `resteasy-reactive-quot` extension that takes the returned `TemplateInstance`, processes it and returns the resulting HTML document as the result of the REST method.

Notice that if there's no active user session, we pass `null` as the value of the `name` parameter, so the template will be rendered using the default value, which is `anonymous`. Of course, it is also an option to properly handle the default behavior here instead of delegating that to the template.

Now, supposing you have the `users-service` project running in Dev mode, you should be able to open <http://localhost:8080/whoami> in your browser. Because we don't have security set up yet, you won't be required to log in, and the resulting web page should say:

```
Hello anonymous!
```

TIP We haven't set up security yet, but if you're using the finished project from the book's git repository rather than writing the project from scratch, then security is already set up. In that case, you will be required to log in. Use `alice` as both the username and the password.

The logout link is not usable for now, but we will get it working as we actually enable security for the application.

6.1.2 Adding security to the application

Now that we have a simple HTML page, it's time to secure the application so we can see an actual username on the WhoAmI page instead of just `anonymous`. As already mentioned, we will use Keycloak as the OIDC provider that handles all metadata about registered users. Quarkus will delegate to Keycloak for all authentication purposes. To make things simple, we use a Dev Services Keycloak instance. Remember that Dev Services is the feature of Quarkus that makes development much easier by managing instances of remote services like databases, messaging brokers, and (like in this case) Keycloak as an OIDC provider.

Normally, setting up security for applications is very complicated - when you manage your own Keycloak instance, you have to configure the security realm and manage the list of users. But, as we're focusing on development aspects in this book, we aim to make it as simple as possible by setting up a Keycloak instance very easily, thanks to Dev Services. This can't be used in production mode, so you will have to manage more things manually there. One important aspect to note is that we will secure not only the `Users` service but also the `Reservation` service. This might sound complicated because you need to manage how both applications use the same Keycloak instance, right? Not really. With Dev Services, the managed Keycloak instance can be easily shared between multiple applications running simultaneously. With the right Dev Services configuration, when you run two or more applications in Dev mode that need a Keycloak instance, Quarkus can automatically create a single Keycloak instance (a single container) shared by all applications. Quarkus will normally boot up a Keycloak container when you start the first application in Dev mode. When you run the next application, it detects that there already is a shared Keycloak instance, and the application will wire itself up to that one instead of starting a new instance.

TIP With shared Dev Services Keycloak, the Keycloak instance is managed by the first application that started it, so if you stop that one, the Keycloak container will also be stopped, and the other applications using it might stop working properly.

Let's get to it. We already have the `quarkus-oidc` extension present in the `Users` service, so Keycloak should already get started and wired up (in Dev mode) properly, along with some sensible default settings for the security realm, so we don't probably need to configure Keycloak itself (but of course, it is possible to customize the Dev Services Keycloak instance via Quarkus configuration properties, if that is necessary). We now configure only the application itself. Add these lines into `application.properties` of the `Users` service:

Listing 6.3 Security-related configuration properties of the Users service

```
quarkus.http.auth.permission.all-resources.paths=/* #1
quarkus.http.auth.permission.all-resources.policy=
-authenticated #2
quarkus.oidc.application-type=web_app #3
quarkus.keycloak.devservices.shared=true #4
quarkus.oidc.logout.path=/logout #5
```

#1 Denote a collection of resource paths (in this case, /* means everything) that follow a common authentication policy.

#2 Set the authentication policy for the resource collection.

#3 Set the application type.

#4 Share the Dev Services Keycloak instance between applications.

#5 Create a /logout endpoint.

Security configuration involves creating collections of resources that share a common authentication policy. In this case, we have only one collection, and we named it `all-resources`. The `paths` attribute is set to `/*` and the `policy` is set to `authenticated`, which means that access to all resources exposed by the application will require authentication. Anonymous access is forbidden.

We choose `web_app` as the application type to say that the preferred authentication method is the so-called Authorization Code Flow, which means that Quarkus redirects any unauthenticated request to the Keycloak URL that allow the user to authenticate. The other possible type is `service`, which we would generally use for applications that are a set of HTTP (REST) resources rather than a web application, and in this case, the preferred authentication mode would be using the `Authorization` HTTP header.

We also made the Dev Services keycloak instance to be shared between multiple applications running in Dev mode on the same machine. When you start Dev mode and Quarkus detects that such an instance is already running, the application will use that instance instead of spinning up a new Keycloak.

The `quarkus.oidc.logout.path` setting enables an endpoint on the `/logout` path that invalidates the authenticated session, logging the user out when invoked by the application. If you remember the `/logout` hyperlink from the `whoami.html` template, this is exactly where it points.

Because we've changed the configuration of the Dev Services Keycloak, just for this one time, we recommend restarting Dev mode completely rather than just letting it perform a live reload. That should make sure all changes are correctly applied. Now, if you refresh the <http://localhost:8080/whoami> webpage, you are redirected to a screen handled by Keycloak, which asks you to enter your credentials, as shown in [Figure 6.2](#).

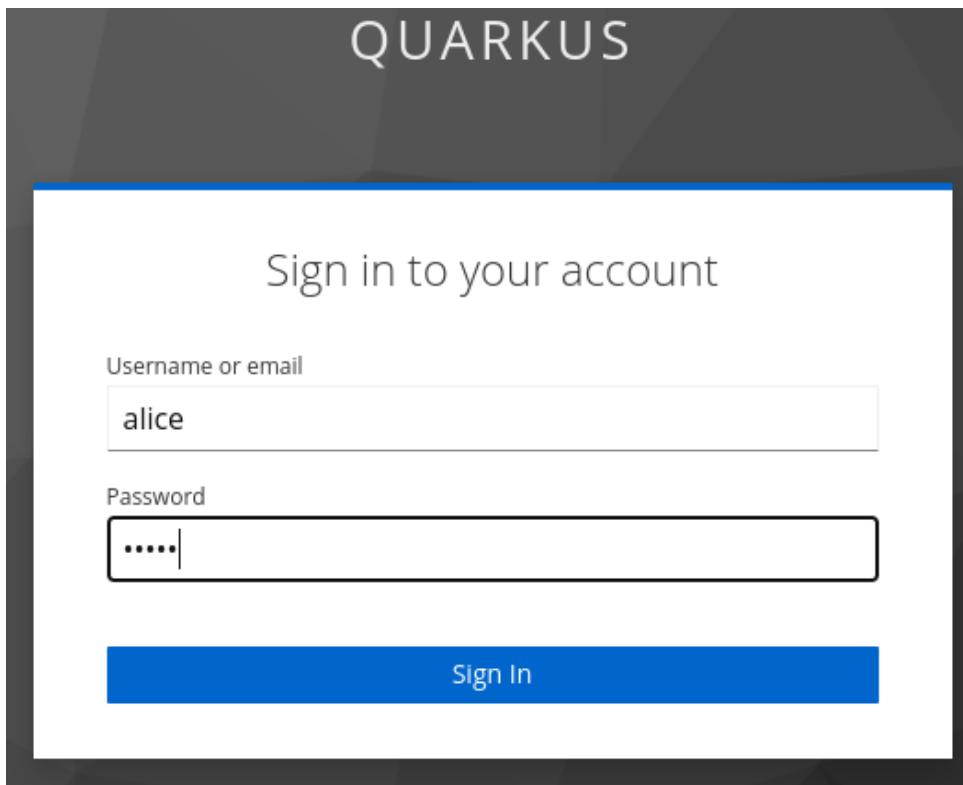


Figure 6.2 Keycloak login screen

A Dev Services Keycloak with default configuration contains a user named `alice` with the password `alice`. Enter this into the login form and click the `Sign In` button. If you log in successfully, Keycloak redirects you back to <http://localhost:8080/whoami>, but as a logged-in user. The page will say:

Hello alice!

TIP The default users (`alice` and `bob`) are present by default in a Keycloak instance that was spawned via Dev Services. This set of automatically added users is configurable through the configuration property called `quarkus.keycloak.devservices.users`. In production mode, where you would have to manage the instance manually, Keycloak doesn't contain any such users by default. Instead, it offers highly configurable features like user registration forms, activation of accounts through email verification, password resets, blocking existing accounts, etc. The specification of all features that Keycloak provides is out of the scope of this book.

The logout link below also works now. You will get logged out if you click it, receiving a screen confirming that. We could specify a different URL where you get redirected after logout, but since all resources of this application require authentication, we don't really have any other reasonable page where to automatically redirect the user because they all wouldn't work after logging out. To log in, manually navigate to <http://localhost:8080/whoami> in your browser again. This time, try logging in as another user with the included credentials out of the box. As you might have guessed, the username is `bob`, and the password is `bob`. The WhoAmI page should then say:

```
Hello bob!
```

TIP If it happens that you restart the Keycloak instance (for example, by stopping and restarting the whole Quarkus Dev mode - NOT just a live reload of the application) while you're logged in to the application in your browser, it might happen that after refreshing the page, the browser will again pass the original cookie that identifies the authenticated session, but with a new Keycloak instance. This session is not valid anymore, thus you will get an error (a blank page and a warning in the application's log). We suggest using the logout button before restarting Dev mode to avoid this. If you don't, you might have to manually tell the browser to forget the `q_session` cookie or wait until the cookie expires. In Dev mode, session cookies are valid for 10 minutes by default.

6.2 Creating a UI for managing car reservations

The main goal of this chapter, as we already mentioned, is to create a very simple secured UI in the `Users` service that makes use of the `Reservation` service and allows authenticated users to manage their reservations. It allows viewing the list of reservations for the logged-in user, cars that are available for renting given a start and end date, and creating a reservation by clicking a single button. The resulting application will look like in [Figure 6.3](#), plus a header showing the name of the current user.

List of reservations

ID	Car ID	Start day	End day
1	1	2023-02-04	2023-02-10

Available cars to rent

Start date: 02/04/2023

End date: 02/10/2023

Car ID	Plate number	Manufacturer	Model	Reservation
2	XYZ987	Ford	Mustang	<input type="button" value="Reserve"/>

Figure 6.3 Reservation management page

Under the `List of reservations` header, there's a table that lists information about all reservations that the current user has made.

Under `Available cars to rent`, you can select a desired start and end date, and after clicking the `Update list` button, the table below will be updated to show all cars that are available to be reserved on the selected dates. Each row in the table has a `Reserve` button that creates a reservation for this car and for these start/end dates.

The application behaves as a single-page application. Clicking buttons does not lead to reloading the whole page. Instead, it dynamically updates parts of the DOM (Domain Object Model) based on responses to asynchronous HTTP requests. But don't worry, even though we're using asynchronous requests to the backend, we won't need to write a single line of JavaScript code.

6.2.1 Updates to the Reservation service needed by the Users service

Let's first enable security for the `Reservation` service. We make it use a shared Dev Services Keycloak instance together with the `Users` service when developing both applications on the same machine in Dev mode. Find the directory with the `Reservation` service project (you can find it in the `chapter-05/reservation-service` directory; we recommend copying it into your `chapter-06` work). We need to add the `quarkus-oidc` extension, so either add it into the `pom.xml` manually or use the easier way and execute this CLI command (in the root directory of the `reservation-service` project):

```
quarkus ext add oidc
```

Add these two lines into the project's `application.properties`:

```
quarkus.oidc.application-type=service
quarkus.keycloak.devservices.shared=true
```

`quarkus.oidc.application-type=service` means that the application is a set of RESTful HTTP resources, so the preferred authentication method is the use of the `Authorization` HTTP header, rather than a browser cookie.

`quarkus.keycloak.devservices.shared` means we share a Keycloak instance together with the `Users` service running in Dev mode on the same machine.

For simplicity's sake, in this case, we didn't make the authentication mandatory (we did that in the `Users` service), so there are no `quarkus.http.auth.permission.*` properties. We develop the `Reservation` service in a way that allows anonymous access. If there is no `Authorization` header on incoming requests, they will be allowed to go through, but the security context will be empty. Thanks to this simplification, you can still call the service's REST endpoints without obtaining and passing an authentication token.

Now that we have set up security for the `Reservation` service, we can finish some of the things we originally replaced with dummy values inside the `ReservationResource` endpoint. Open the class `org.acme.reservation.rest.ReservationResource`.

Somewhere at the beginning of the class, inject an instance of `SecurityContext` so that we can access information about the logged-in user:

```
@Inject
javax.ws.rs.core.SecurityContext context;
```

TIP Notice that in CDI, we can mix the field injection that we use for `SecurityContext` together with the constructor injections from chapter 4.

Change the `make` method (it serves to create new reservations) so that it retrieves the current user's name and stores it in the object representing the reservation:

Listing 6.4 Updating the `make` method for creating reservations to store the current user's name

```
@Consumes(MediaType.APPLICATION_JSON)
@POST
public Reservation make(Reservation reservation) {
    reservation.userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() :
        "anonymous"; #1
    Reservation result = reservationsRepository.save(reservation);
    if (reservation.startDay.equals(LocalDate.now())) {
        rentalClient.start(reservation.userId, result.id);
    }
    return result;
}
```

#1 Here, we retrieve the current user's name. We set the name to anonymous if there's no logged-in user.

The UI in the `Users` service will also be able to list all reservations belonging to the logged-in user, so we need a way to ask the `Reservations` service for this list. Add this as a new method to the `ReservationResource`:

Listing 6.5 New method for listing all reservations belonging to the current user

```
@GET
@Path("all")
public Collection<Reservation> allReservations() {
    String userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : null;
    return reservationsRepository.findAll()
        .stream()
        .filter(reservation -> userId == null ||
            userId.equals(reservation.userId))
        .collect(Collectors.toList());
}
```

Remember that the `userId` is `null` when no user is logged in. When we filter through all reservations to pick the ones belonging to the current user, in this case, we include all reservations in the result (that is because of the `userId == null` condition). If there's a user logged in, we only include reservations where the `userId` matches the current user.

Now that the `Reservation` service is secured, it has everything that our frontend application needs. Let's next create the actual frontend that manages reservations for users.

6.2.2 Preparing backend parts in the Users service to be used by the UI

The UI in the `Users` service will need to call the REST endpoints of the `Reservation` service and also propagate the security credentials of the logged-in user into these calls. The two domain model classes that we need to use are `Car` and `Reservation`, so let's create simplified copies of them in the `Users` service. In the `Users` service, create the class `org.acme.users.model.Car`:

Listing 6.6 The Car class needed by the Users service

```
package org.acme.users.model;

public class Car {
    public Long id;
    public String licensePlateNumber;
    public String manufacturer;
    public String model;
}
```

And similarly, `org.acme.users.model.Reservation`:

Listing 6.7 The Reservation class needed by the Users service

```
package org.acme.users.model;

import java.time.LocalDate;

public class Reservation {
    public Long id;
    public String userId;
    public Long carId;
    public LocalDate startDay;
    public LocalDate endDay;
}
```

Again, if you prefer private fields with getters and setters, feel free to use them instead of public fields. It's a matter of taste.

Next, create the interface for the REST client that calls the `Reservation` service. We saw a REST client interface before in chapter 4, so this should be familiar. The `@AccessToken` annotation is the only new concept in this interface. The REST client is represented by the `org.acme.users.ReservationsClient` class:

Listing 6.8 The code of the ReservationsClient class

```
package org.acme.users;

import io.quarkus.oidc.token.propagation.AccessToken;
import org.acme.users.model.Car;
import org.acme.users.model.Reservation;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.reactive.RestQuery;

import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import java.time.LocalDate;
import java.util.Collection;

@RegisterRestClient(baseUri = "http://localhost:8081")
@AccessToken #1
@Path("reservation")
public interface ReservationsClient {

    @GET
    @Path("all")
```

```

Collection<Reservation> allReservations(); #2

@POST
Reservation make(Reservation reservation); #3

@GET
@Path("availability")
Collection<Car> availability( #4
    @RestQuery LocalDate startDate,
    @RestQuery LocalDate endDate);
}

```

#1 Ensure that the current user's ID token propagates to all calls made by this REST client. It works by adding the token into the Authorization HTTP header.

#2 The resource method for retrieving all reservations belonging to the current user.

#3 The resource method for creating a new reservation.

#4 The resource method for obtaining a list of available cars for the given start and end dates.

If you need to refresh your memory on the actual API and implementation of the REST endpoints that we're calling here, go back to the `Reservation` service and check out the `ReservationResource` class.

6.2.3 Creating the UI using Qute, HTMX and a REST backend

ABOUT HTMX

HTMX is a client-side toolkit for building responsive web applications. Its main goal is to simplify asynchronous communication with the backend server and provide dynamic updates of the pages' contents. Doing this usually requires writing JavaScript code. But HTMX achieves this declaratively with unique HTML attributes. In simpler cases, you won't need to write your own JavaScript at all.

To show a simple example:

Listing 6.9 A simple example of HTMX usage

```

<script src="https://unpkg.com/htmx.org@1.8.4"></script>
<button hx-get="/click" hx-swap="innerHTML" hx-target="#greeting">
    Greet me
</button>
<div id="greeting">
</div>

```

With this piece of pure HTML, you get a `button` and a `div`. When the user clicks the button, HTMX sends an HTTP GET request to the backend server's `/click` endpoint. Then the contents of the `div` element (with id `greeting`, as denoted by the `hx-target` attribute) are replaced by the response's body. No JavaScript code is required.

HTMX is a third-party library, so it isn't associated with Quarkus. Nevertheless, some parts of the Qute engine's design were influenced by HTMX to make them work nicely together. More information about HTMX can be found at <https://htmx.org/>.

CREATING TEMPLATES USING QUTE+HTMX AND SERVING THEM

The reservation management page that we're about to create consists of three parts:

- A header containing a title, the current user's name, and a logout button.
- A table with all reservations that belong to the logged-in user.
- A table listing all cars that are available to rent for the selected dates and two fields to allow selecting the start and date.

Let's create the Qute templates necessary for this page. We use the plural form because we have separate templates for specific parts of the page. Because the most common and basic usage of HTMX, as we just showed in the previous section, is to dynamically replace contents of HTML elements with different HTML content received from the backend server, we create a separate Qute template for each case where we do such upgrades. Namely, we need a template for the following:

- The main page itself (`index.html`).
- Table listing the reservations (`listofreservations.html`).
- Table listing the cars that are available for selected dates (`availablecars.html`).

For each of these templates, we create a REST method that renders a page from that template.

Let's start by implementing the index page. In the `Users` project, create the new file `index.html` in the `src/main/resources/templates/ReservationsResource` directory. For now, it only contains the header with the username, and a logout button, as shown in [Listing 6.10](#).

Listing 6.10 Initial (almost blank) version of the reservation management page

```

{@java.lang.String name}                                     #1
{@java.time.LocalDate startDate}
{@java.time.LocalDate endDate}

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Reservations</title>
    <link rel="stylesheet"
        href="https://cdn.simplecss.org/simple.min.css">    #2
    <script src="https://unpkg.com/htmrx.org@1.7.0">          #3
    </script>
</head>
<body>

<header>
    <h1>Reservations</h1>
    <p>For logged-in user: {name}</p>                      #4
    <a href="/logout">Log out</a>
</header>

</body>
</html>
```

#1 Declare expected types of the template's parameters, see below for explanation.

#2 Import the SimpleCSS stylesheet to make the HTML output a bit nicer.

#3 Import the HTMX library.

#4 The header includes the `name` parameter (name of the logged-in user) and a logout link.

The first three lines are optional. They constitute a declaration of the parameters this page requires to render along with their types. If you provide these declarations, the Quie templating engine does its best to ensure the type safety of the template. It also throws an error at build time in the case of incorrect use of any parameter (e.g., if you refer to an attribute that doesn't exist in the class of the parameter). Our page takes three parameters - `username` and the initial start and end dates (these can be overridden later by the user form). They serve for filtering available cars and for creating a reservation.

Now we need a REST endpoint that serves this template. In the `Users` service, create the `ReservationsResource` class in the `org.acme.users` package, as shown in [Listing 6.11](#).

Listing 6.11 Initial version of the REST resource serving the index.html page

```
package org.acme.users;

import io.quarkus.qute.CheckedTemplate;
import io.quarkus.qute.TemplateInstance;
import io.smallrye.common.annotation.Blocking;
import org.eclipse.microprofile.rest.client.inject.RestClient;
import org.jboss.resteasy.reactive.RestQuery;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;
import java.time.LocalDate;

@Path("/")
@Blocking
public class ReservationsResource {

    @CheckedTemplate
    public static class Templates {
        public static native TemplateInstance index(
            LocalDate startDate,
            LocalDate endDate,
            String name);
    }

    @Inject
    SecurityContext securityContext;

    @RestClient
    ReservationsClient client;

    @GET
    @Produces(MediaType.TEXT_HTML) #2
    public TemplateInstance index(@RestQuery LocalDate startDate,
                                 @RestQuery LocalDate endDate) {
```

```

        if (startDate == null) {
            startDate = LocalDate.now().plusDays(1L); #3
        }
        if (endDate == null) {
            endDate = LocalDate.now().plusDays(7);
        }
        return Templates.index(startDate, endDate,
            securityContext.getUserPrincipal().getName());
    }
}

```

#1 Declare the template with its parameters as a checked template. See below for an explanation.

#2 The resource method that instantiates the template and returns it.

#3 The startDate and endDate parameters are considered optional, so if no values are provided, we assume that we're looking for available cars for the timeslot starting tomorrow and ending seven days from today.

In this example, we decided to use a slightly different approach to work with templates - checked templates. When you declare a template inside a static class annotated with `@CheckedTemplate` like this, Qute performs type-safety checks on the template usage during build time. As you can see, instead of passing parameters to the template using the `data` method (which we did in the WhoAmI example), you can declare the template's parameters in a type-safe way as parameters of the method that returns an instance of the template. A method that wants to serve this template can now build an instance of it by calling `Templates.index(parameters...)`. The `index` in the method name corresponds to the name of the template file (`index.html`).

With this code in place and dev mode running, you may now navigate to <http://localhost:8080/> to verify that we're on the right track. The page redirects you to the Keycloak instance, which asks you to provide credentials. Remember that that's either `alice` or `bob`, with the password being the same as the username. The page you see after successful login looks like [Figure 6.4](#). There is just the header for now. The rest of the page is blank.

Reservations

For logged in user: alice

[Log out](#)

Figure 6.4 Header of the reservation management page

The next step is to list available cars for particular dates, along with the button to create a reservation for the selected car and dates immediately. For that, we need a new template. In the `src/main/resources/templates/ReservationsResource` directory create a new file `availablecars.html` as shown in [Listing 6.12](#).

Listing 6.12 Template for the table of available cars

```
{@org.acme.users.model.Car[] cars} #1
{@java.time.LocalDate startDate}
{@java.time.LocalDate endDate}
<div id="carlist">
<table>
  <thead>
    <tr>
      <th>Car ID</th>
      <th>Plate number</th>
      <th>Manufacturer</th>
      <th>Model</th>
      <th>Reservation</th>
    </tr>
  </thead>
  <tbody>
    {@for car in cars} #2
      <tr>
        <td>{car.id}</td> #3
        <td>{car.licensePlateNumber}</td>
        <td>{car.manufacturer}</td>
        <td>{car.model}</td>
        <td>
          <form hx-target="#reservations"
                hx-post="/reserve" > #4
            <input type="hidden" name="startDate" value="{startDate}" />
            <input type="hidden" name="endDate" value="{endDate}" />
            <input type="hidden" name="carId" value="{car.id}" />
            <input type="submit" value="Reserve"/> #5
          </form>
        </td>
      </tr>
    {@/for}
  </tbody>
</div>
```

#1 To receive the type safety checks, declare that the `cars` parameter is a collection of `Car` instances.

#2 A for-loop over the collection of cars. For each car, we generate an HTML form and a button that creates a reservation.

- #3 Add relevant attributes of cars as cells into the table.
- #4 Define the action taken upon submitting the form - a HTTP POST request to /reserve.
- #5 Add all parameters required by the /reserve endpoint into the form.

The `hx-post` attribute on a `<form>` element means that when the user clicks the submit button, the page sends an HTTP POST request to the specified endpoint (`/reserve`, we will create the actual handling method later). It also adds all values of the form to the request as parameters. In this case, the parameters are all data necessary for creating a new reservation. The `hx-target="#reservations"` attribute specifies that the contents of the response to such request should replace the HTML element with ID `reservations`. Again, we will create this element later. This immediately updates the list of existing reservations after we make a new one.

Now, to show this table of available cars, add a reference to it into `index.html`, along with the simple form that allows you to select the start and end dates for reservations. The code is in [Listing 6.13](#). Place it somewhere inside the `<body>` element, after the end of the `</header>` element.

Listing 6.13 Including the list of available cars in the index page

```

<h2>Available cars to rent</h2>
<form hx-get="/available" hx-target="#availability"> #1
  <p>Start date:<input id="startDateInput" type="date"
    name="startDate" value="{startDate}" /></p>
  <p>End date:<input id="endDateInput" type="date"
    name="endDate" value="{endDate}" /></p>
  <input type="submit" value="Update list"/>
</form>
<div id="availability" hx-get="/available" #2
  hx-trigger="load, update-available-cars-list from:body" #3
  hx-include="[id='startDateInput'],[id='endDateInput']"> #4
<!-- To be replaced by the result of calling /available -->
</div>

```

#1 Create a form where users can choose a start and end date to filter through available cars.

#2 The div that the `availablecars.html` template populates, it renders after sending a GET request to the `/available` endpoint.

#3 Set the events which trigger updating the contents of this div.

#4 Include the selected `startDate` and `endDate` parameters in the GET requests.

The first half of this snippet is the form that allows users to select the start and end dates for reservations. The initial value of these two inputs will be the `startDate` and `endDate` parameters passed to the template, but the users can change these values. The `hx-get="/available"` attribute means that when the form is submitted, a GET request is sent to the `/available` endpoint, along with all values from this form as parameters. `hx-target="#availability"` means that when a response is received, the response's body will replace the contents of the element with ID `availability`, not this `<form>` element itself. When the user selects the start date and end dates and then clicks the `Update list` button, the page requests a list of cars available for that dates and updates the available cars' table with the received data.

The second part is the table of available cars, populated by the `availablecars.html` template. The `availability` div needs to update its contents in response to three different events. The first is loading the page - we want the table to appear automatically when the page loads in the browser - this triggers the built-in event named `load`, which is why we include `hx-trigger="load"`. The second event is when the user makes a new reservation - we use a custom event named `update-available-cars-list` for that, and we add it to the `hx-trigger` list. We will explain how to trigger this event later. We also have to add a `from:body` modifier to make it work properly (HTMX requirement). The third event that triggers a reload of this div is when the user submits the form with start and end dates. We already took care of this by the `hx-target="#availability"` attribute on the form.

When we ask the server for a list of available cars, we need to include the `startDate` and `endDate` parameters in the GET request. The form picks the values from the inputs above with IDs `startDateInput` and `endDateInput`. This `hx-include` attribute ensures that the current value of these inputs is included in the request when the user refreshes the table. Careful, we don't want to use the `{startDate}` and `{endDate}` (Qute) parameters of the `index.html` template. These are only the initial values that Qute uses when first rendering the page. If we did, we would ignore potential updates performed by the user in the form above.

Next, we need the template with a table of all reservations. This needs to be in a file named `listofreservations.html` that similarly as other templates resides inside the `src/main/resources/templates/ReservationsResource/` directory and is shown in [Listing 6.14](#).

Listing 6.14 Template for showing the list of reservations

```

{@org.acme.users.model.Reservation[] reservations} #1
<div id="listofreservations">
<table>
  <thead>
    <tr> #2
      <th>ID</th>
      <th>Car ID</th>
      <th>Start day</th>
      <th>End day</th>
    </tr>
  </thead>
  {#for i in reservations} #3
    <tr>
      <td>{i.id}</td>
      <td>{i.carId}</td>
      <td>{i.startDay}</td>
      <td>{i.endDay}</td>
    </tr>
  {/for}
</table>
</div>

```

#1 Again, for type safety, we specify that the `reservations` parameter is a collection of `Reservation` objects.

#2 Header of the table.

#3 Iterate over the collection of `reservations`, and add a row to the table for each item.

To include this table in the index page, add the code from [Listing 6.15](#) to `index.html`. Put it after the end of the `</header>` element containing the logout link.

Listing 6.15 Including the template with a list of reservations in the index page

```

<h2>List of reservations</h2>
<div id="reservations"
  hx-get="/get" #1
  hx-trigger="load"> #2
<!-- To be replaced by the result of calling /get -->
</div>

```

#1 The list of reservations is obtained by sending a GET request to the `/get` endpoint.

#2 Render the list when the page loads.

We trigger an update of the `reservations` div as a response to the `load` event to make sure it renders on page load. The other event triggering a refresh of this table is when the user makes a new reservation. You might remember that back in `availablecars.html`, we generated a form for each available car, and the `hx-target` of that form had the value `#reservations`, meaning that when the form is submitted (the user makes a reservation request), it will trigger an update of the reservation list.

Now, we need to implement the REST methods for serving the remaining templates and handling the creation of new reservations:

- `getReservations`, exposed on the `/get` endpoint, that returns an instance of the `listofreservations.html` with the list of all reservations of the current user.
- `getAvailableCars`, exposed on the `/available` endpoint, returning an instance of the `availablecars.html` template. It takes `startDate` and `endDate` parameters to narrow down the results.
- `create`, exposed on the `/reserve` endpoint, creates a new reservation and returns the updated list of reservations (the `listofreservations.html` template), already including the new reservation. It takes `startDate`, `endDate` and `carId` parameters.

But before we implement these methods, we also need to add factory methods to be able to instantiate the remaining two templates. Remember that we added the `listofreservations.html` template that takes a collection of reservations and the `availablecars.html` template that takes a collection of cars, start date, and end date. Update the `Templates` class (nested inside `ReservationsResource`) as shown in [Listing 6.16](#).

Listing 6.16 Adding a way to build template instances inside a REST class with the type safety

```
@CheckedTemplate
public static class Templates {
    public static native TemplateInstance index(
        LocalDate startDate,
        LocalDate endDate,
        String name);

    public static native TemplateInstance listofreservations(
        Collection<Reservation> reservations);

    public static native TemplateInstance availablecars(
        Collection<Car> cars,
        LocalDate startDate,
        LocalDate endDate);
}
```

Now add the mentioned three REST methods into the `ReservationsResource` class, as per [Listing 6.17](#).

Listing 6.17 Remaining necessary REST methods to be able to serve all templates

```

@GET
@Produces(MediaType.TEXT_HTML)
@Path("/get")
public TemplateInstance getReservations() {
    Collection<Reservation> reservationCollection
        = client.allReservations();
    return Templates.listofreservations(reservationCollection);
}

@GET
@Produces(MediaType.TEXT_HTML)
@Path("/available")
public TemplateInstance getAvailableCars(
    @RestQuery LocalDate startDate,
    @RestQuery LocalDate endDate) {
    Collection<Car> availableCars
        = client.availability(startDate, endDate);
    return Templates.availablecars(
        availableCars, startDate, endDate);
}

@POST
@Produces(MediaType.TEXT_HTML)
@Path("/reserve")
public RestResponse<TemplateInstance> create(
    @RestForm LocalDate startDate,
    @RestForm LocalDate endDate,
    @RestForm Long carId) {
    Reservation reservation = new Reservation();
    reservation.startDay = startDate;
    reservation.endDay = endDate;
    reservation.carId = carId;
    client.make(reservation);
    return RestResponse.ResponseBuilder
        .ok(getReservations())
        .header("HX-Trigger-After-Swap",      #1
            "update-available-cars-list")
        .build();
}

```

#1 Trigger a custom event - see explanation below

The only new concept here is that we want to trigger a custom event named `update-available-cars-list` when the user creates a new reservation. The call to `/reserve` itself only updates the reservations table (remember the `hx-target="#reservations"` attribute inside `availablecars.html`). Still, we also need to update available cars because, after a successful reservation, that car is no longer available. We trigger this event by adding an `HX-Trigger-After-Swap` HTTP header to the response. This means that after the response is received and the reservations table is updated, we also trigger an update of the list of available cars. This links to the `update-available-cars-list` value inside a `hx-trigger` in `index.html`. Of course, there are better solutions than this because typically, you would want to update both tables simultaneously, whereas here, the table of available cars updates asynchronously after the update of the reservations table. It is also an option to have just one template together for both tables and have this REST method return it. Also note that to be able to add headers to the response, we had to wrap the return type in a `RestResponse` and use the `ResponseBuilder` to build the response that contains the necessary `TemplateInstance` as well as the header.

6.2.4 Trying the application

And that's it. Now let's try it out. Make sure that you have all three necessary services running:

- `Users` service on `localhost:8080`. It has to run in Dev mode for now because we didn't set up a Keycloak instance except for the one provided to us by Dev Services.
- `Reservation` service (the secured version) on `localhost:8081`. It also needs to run in Dev mode because it shares a Keycloak instance with `Users`.
- `Inventory` service on `localhost:8083`. It can run in either Dev mode or production mode.

TIP If you are going to create reservations that start on the current date, you will also need the `Rental` service on `localhost:8082` because in this case, the `Reservation` service contacts the `Rental` service to start a rental. If you refrain from using the current date as a start date, then the `Rental` service doesn't need to be running.

Now navigate to <http://localhost:8080> and experiment with the application. Use the form in the Available cars to rent section to set start and end dates, then click the Update list button to refresh the list of available cars on those dates. Once you click the Reserve button, a reservation for those dates appears, and the relevant car disappears from the list of available cars. This happens via asynchronous HTTP requests, not by reloading the whole page.

We didn't create a way to cancel existing reservations. This is left to the reader as an exercise. You would probably want to achieve this by adding a new column into the first table and a button similar to the `Reserve` button, but it would call a new REST method that deletes a reservation. Though, support for deleting a reservation also needs to be implemented in the `Reservation` service!

TIP If you need to undo all your changes and start anew from the original state, trigger a live reload of the `Reservation` service (press `s` in its terminal window, for example). It will delete all reservations because they are stored in-memory inside the application. After reloading the `Reservation` service and hitting `F5` in your browser, you can start experimenting from scratch.

If you feel that you need more cars for experimenting, feel free to add more cars in the `Inventory` service (in the `CarInventory#initialData` method that fills the inventory with initial data), or add them dynamically through the GraphQL or gRPC API (with this approach, they will be lost on restart!). If you don't remember, for GraphQL, you can do this by navigating to <http://localhost:8083/q/graphql-ui> and executing a mutation similar to the one in [Listing 6.18](#):

Listing 6.18 Adding a car via GraphQL

```
mutation {
  register(car: {
    licensePlateNumber: "123LAMBO"
    model: "Huracan"
    manufacturer: "Lamborghini"
    id: 5
  }) {
    id
  }
}
```

6.3 Running in production mode

So far, we have developed the `Users` and `Reservation` services in a way that they run only in Dev mode because they rely on Quarkus spinning up an instance of Keycloak and providing all the necessary Keycloak configuration (the `Inventory` service can already run in production mode in the current state because it has no such requirement). In this section, let's see what it takes to be able to run them in Quarkus' production mode. This is the checklist of what is missing:

- Manually run an instance of Keycloak.
- Provide suitable configuration for the `car-rental` security realm in Keycloak.

- Also, run a PostgreSQL database because Keycloak requires it.
- Enhance the configuration of the two applications to provide the necessary information for connecting to Keycloak. They share the same instance, just like we did in Dev mode.

6.3.1 Running Keycloak and PostgreSQL as containers

We use the Docker Compose (or you may be using Podman Compose, it works the same) project to easily spin up an instance of Keycloak along with an instance of PostgreSQL that it requires. We have prepared an entire `docker-compose.yml` file for this. It's located in the book's repository in the directory `chapter-06/production`, along with the file `car-rental.json`, a definition of the Keycloak security realm that the Reservation and Users services will use. A realm created by importing this file is very similar to the realm that Quarkus creates when running Keycloak via Dev Services. The set of users is the same (`alice` and `bob`, where passwords are the same as the usernames).

The `docker-compose.yml` file already contains the logic that makes Keycloak import the realm on startup, as is visible in the following excerpt from the file available in [Listing 6.19](#).

Listing 6.19 Automatically importing a security realm in Keycloak on startup

```
volumes: #1
  - "./car-rental.json:/opt/keycloak/data/import/car-rental.json:Z"
command:
  - start-dev
  - --import-realm #2
```

#1 Mount the JSON file describing the realm into the container's `/opt/keycloak/data/import` directory. This is where Keycloak looks for realms to import.

#2 The `--import-realm` tells Keycloak to enable importing of realms in the `import` directory.

To start everything up, go to the `chapter06/production` directory and issue the following command (or with Podman Compose, use `podman-compose`):

```
$ docker-compose up
```

As specified in the `docker-compose.yml` file, Keycloak starts and listens on port 7777, and PostgreSQL is available on port 5300.

6.3.2 Wiring the services to use Keycloak

Now we need to configure the Reservation and Users services to be able to connect to our Keycloak instance because when running in production mode, we can't have Quarkus do this for us automatically.

Add these lines into `application.properties` in the `reservation-service` directory:

Listing 6.20 Wiring the Reservation service to manually managed Keycloak

```
%prod.quarkus.oidc.auth-server-url=http://localhost:7777/realmss/car-rental
%prod.quarkus.oidc.client-id=reservation-service
%prod.quarkus.oidc.token-state-manager.split-tokens=true
```

TIP The `quarkus.oidc.token-state-manager.split-tokens` is most likely necessary in prod mode, because the security realm is configured in a way that the total size of session cookies might exceed 4 kilobytes (a session cookie contains three encrypted tokens - ID, access and refresh). Some browsers might decide to ignore such cookies, and thus authentication would not work. With this property enabled, Quarkus will split the authentication cookie into three separate cookies, one for each of the mentioned tokens. In dev mode, the generated security realm is configured to use smaller tokens.

And similarly in the `users-service` directory:

Listing 6.21 Wiring the Users service to manually managed Keycloak

```
%prod.quarkus.oidc.auth-server-url=http://localhost:7777/realmss/car-rental
%prod.quarkus.oidc.client-id=users-service
%prod.quarkus.oidc.token-state-manager.split-tokens=true
```

All these properties are prepended with `%prod` prefix, meaning they are only considered when running in production mode, not Dev mode. Dev mode with an automatically managed Keycloak instance is still usable, unaffected by these changes. But now, you can run these two services as in production by running these commands in each of their respective root directories:

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

Everything should stay very similar to when we were using Dev mode, except you won't be able to do live reloads. In a real-world production environment, this might probably be even more complicated. For example, Keycloak and PostgreSQL can be managed as deployments in a Kubernetes cluster. This section aimed to show the easiest way to get it running using regular containers.

6.4 Wrap up and next steps

This chapter combined two critical aspects of application development - security and UI. We decided to incorporate them into a single chapter because they intertwine a lot, especially for smaller applications like the one we are developing in our examples. It's easier to write one with the other.

We used Keycloak as the lynchpin of our security solution. Keycloak is very well integrated with Quarkus and offers many security-related features, including user registration, sending verification emails, defining password policies, blocking users, single sign-on for multiple services, and much more. Quarkus uses these features to communicate with the Keycloak instance and delegate user authentication to it. We showed how to easily propagate an ID token between two Quarkus services over a REST client. The receiving instance re-verifies the token's validity by connecting to Keycloak under the hood.

For the UI part, we used a combination of Qute and HTMX. Qute is a core extension of Quarkus, and on the basic level, it's a templating engine that renders HTML fragments on the server side and sends them to the client. HTMX, on the other hand, is a purely client-side library. It's a separate project completely independent of Quarkus, but the Qute extension plays with it very nicely. HTMX greatly simplifies the development of interactive AJAX-based applications that dynamically update their content based on communication with the backend, except that with HTMX, in most cases you won't need to write any JavaScript code.

In the next chapter, we will return to the backend. We will look at how Quarkus works with databases and how your applications can make use of them.

6.5 Summary

- Quarkus' OIDC extension comes with first-class support for Keycloak as the tool for managing almost everything related to securing an application.
- With Dev Services, a Keycloak instance is initialized with minimal configuration needed on the user's part (most configuration uses sensible defaults, including creating some users).
- A Dev Services Keycloak instance can be shared by multiple Quarkus applications running in Dev mode on the same machine.

- Propagating the ID token between services with a REST client is as easy as adding the `quarkus-oidc-token-propagation-reactive` extension and then adding an `@AccessToken` annotation on the REST client interface.
- In a REST endpoint, you may inject a `SecurityContext` object to inspect the logged-in user.
- Qute is a server-side templating engine for rendering (not only) HTML pages.
- Qute offers (optional) build-time type safety checks on the usage of all Java objects passed into templates.
- HTMX's most basic usage is sending asynchronous HTTP requests to the backend server and then dynamically swapping contents of HTML elements using bodies of received responses. In most cases, you can achieve this without needing to write any JavaScript code.
- The best way to use HTMX with Quarkus is by writing REST endpoints that handle the asynchronous requests dispatched by HTMX and return pieces of HTML code that should replace the relevant parts of the HTML page that the client is viewing.
- The ecosystem of UI frameworks usable with Quarkus is growing. The most notable other frameworks are Renarde and Quinoa (more information about these in appendix A).

7

Database access

This chapter covers

- Learning how to connect Quarkus application to the database
- Introducing Hibernate ORM with Panache
- Explaining Quarkus approaches to the database access
- Integrating reactive database access to Quarkus applications

Databases. We believe all of us can agree that we must do something with the database at least a few times in our careers. From simple column adjustments to sophisticated migrations, these might become challenging problems to solve. This is why there are a lot of libraries helping us with database access, object-relational mapping (ORM), and migrations every time we need them.

In the previous chapter, we delegated the user management and credentials storing to the Keycloak server that manages its own persistent store. While this is suitable for user handling, it surely isn't ideal for our custom business data that the Car rental system manages. In this chapter, we concentrate on learning how Quarkus manages database access for your applications.

This chapter introduces Panache, an API that Quarkus adds on top of the well-known Hibernate ORM framework. Panache implements the active record pattern, and provides even further simplification over the object-relational mapping provided by Hibernate. However, we don't stop there. As we explain, Panache is also usable with the repository pattern (a repository is a Java object that represents the database table and has methods for interacting with it).

Another very useful feature is using Panache as a REST mapper. This means that you can have REST endpoints with methods for CRUD (create, read, update, delete) generated automatically for your entities. We also differentiate databases and connections that your Quarkus applications can utilize, explaining both the traditional SQL databases like PostgreSQL and the NoSQL stores like MongoDB. Switching between SQL and NoSQL is extremely easy thanks to the abstractions that Panache provides. We then also show how to access all these databases reactively.

The `chapter-07` directory contains the final versions of services developed in this chapter. We are starting with the most current state of all services (Chapter 6 for `reservation-service`, Chapter 4 for `rental-service` and `inventory-service`). We hope you are already interested in how Quarkus handles databases, so let's dive right in.

7.1 Panache

Hibernate (<https://hibernate.org/>) is a very well-known open-source object-relational mapping library that you hopefully have already at least heard about. It continues to provide the de facto standard for Java persistence that is also officially standardized via the Jakarta Persistence API (JPA) specification. There are already a lot of materials, including books provided by Manning, that teach you how to do database access with Hibernate ORM. Which is why we have different goals here. In this chapter, we focus on the new API called Panache.

Panache, as an idea, came from the very same Hibernate team that wanted to introduce a simplified model for database access. Panache took a number of different approaches to choose from. In this section, we focus on the active record pattern. We discuss other approaches in the following sections.

7.1.1 Active record pattern

The active record pattern aims to encapsulate the operations required for the record manipulation in the database into the record (corresponding to an object in Java) itself! This approach is straightforward. Instead of injecting `EntityManager` as you used to do with traditional Hibernate/JPA practice, you now have the option to include all database access operations in the entity itself. How does this work? Let's look at a short example of how Panache works in [Listing 7.1](#). The individual operations are either represented as instance methods (e.g., `persist`, `delete`) if you call an operation working on a specific entity, or as static methods of the entity class (`Car` in the example), which serve to perform more general operations on the table that the entity represents (e.g., `listAll`, `count`).

Listing 7.1 Panache example of active record manipulation

```
// Assume we have an entity named Car

// persisting into the database called on the object instance
car.persist();

// deleting an entity
car.delete();

// listing all Cars from the database called as a static method
Car.listAll();

// find Car by id
Car.findById(carId);

// list all blue Cars
Car.list("color", Color.Blue);

// count all cars
Car.count();

// delete Car
Car.deleteById(carId);
```

Panache is based on HQL (Hibernate Query Language), which is an extension of JPQL (Java Persistence Query Language) and allows you to put parts of these languages into the fluent API model to build database queries. This allows us to write very readable and maintainable code without the need to write any boilerplate. We use many of the available operations in this chapter, but not all. But because this model provides a fluent API, discovering different options is as simple as pressing the autocomplete button in any modern IDE.

As you might remember, we only used a database stub in all the Car rental services until now. We cannot justify using hardcoded static lists representing production data in any modern application. In this chapter, we introduce different database integrations in the respective services of our application. We start by adding a traditional SQL database, PostgreSQL, to the Reservation service.

7.1.2 Getting started with Panache

To be able to use the Panache with the active record pattern approach in our application, we need to do three things:

- Add the relevant configuration for the connection information.
- Create our entities.

- Make the entities extend the `PanacheEntity` class.

PANACHE EXTENSIONS AND APPLICATION CONFIGURATION

As with other functionality in Quarkus, if we want to use Panache in our Quarkus application, we start by adding a Quarkus extension(s) for it. For Panache, we need two extensions: `quarkus-hibernate-orm-panache` for Panache itself, and a database driver that is specific to the database that our application uses. In our case, it is the `quarkus-jdbc-postgresql` for the PostgreSQL database.

TIP The JDBC database driver is required for Hibernate Panache to communicate correctly with the underlying database. As we will learn in the following sections, depending on the database you use, you may choose a different driver (e.g., `quarkus-jdbc-mysql`).

If you are not already in the `reservation-service` directory (remember that we start with the current version from Chapter 6), change your current directory to it. Now you can execute the command available in [Listing 7.2](#) to add required extensions. Of course, you can keep the Dev mode running throughout this chapter. You might notice that the `quarkus-jdbc-postgresql` extension starts a new Dev services instance of the PostgreSQL container for us (e.g., with `docker ps`).

Listing 7.2 Adding extensions required for Panache

```
$ quarkus ext add quarkus-hibernate-orm-panache quarkus-jdbc-postgresql
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✅ Extension io.quarkus:quarkus-hibernate-orm-panache has
→ been installed
[SUCCESS] ✅ Extension io.quarkus:quarkus-jdbc-postgresql has
→ been installed
```

Or you can add the Maven dependencies directly, as shown in [Listing 7.3](#).

Listing 7.3 Adding extensions required for Panache to pom.xml

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

Next, as with any other external service, if we want to connect to a database, we must provide the connection configuration specifying its location and how to connect to it (i.e., the credentials). We first configure the data source as the database to which we are connecting, and then we can optionally provide additional configuration specific to Hibernate. The required configuration for the Reservation service needs to be placed in the `application.properties` as shown in [Listing 7.4](#). Note that we are defining the database URL only in the production mode to allow Dev services to start a PostgreSQL instance for us in Dev mode.

Listing 7.4 Adding the Panache configuration to the Reservation service's application.properties

```
quarkus.datasource.db-kind = postgresql #1
quarkus.datasource.username = user
quarkus.datasource.password = pass
%prod.quarkus.datasource.jdbc.url =
→ jdbc:postgresql://localhost:5432/reservation #2

# drop and create the database at startup #3
quarkus.hibernate-orm.database.generation = drop-and-create

#1 Database specific connection information.
#2 Intentionally define connection URL only in prod mode to allow Dev services.
#3 Hibernate specific properties.
```

This is all that we need to do from the configuration perspective. Next, let's look into how we can define our entities.

DEFINING ENTITIES

An entity in the Panache with the active pattern approach is simply a POJO (plain old Java object) annotated with the `@Entity` annotation and extending the `PanacheEntity` class.

To define the `Reservation` entity in the `Reservation` service, let's move the `Reservation` class into the new `org.acme.reservation.entity` package and modify it like shown in the [Listing 7.5](#) to update it to a valid Panache entity.

And that is all we need. If you are familiar with JPA, you can see this is a way too concise definition of a database entity. All the usually duplicated code in most of the entities, including the `id` field, the generation strategy, or the getters and setters can be omitted because Panache dynamically provides it for us.

TIP Optionally, instead of extending the `PanacheEntity`, you can also extend the `PanacheEntityBase` if you want to define your custom strategy to generate entity IDs (in the same way as generally with Hibernate).

Notice also that we can keep our fields public. Actually, Panache encourages this approach. So there is no need to generate a bunch of getters and setters as the JPA specification requires you to do now. But again, you can do so if you wish.

If you need to override the getter or setter, you are free to do so. And you can also still keep the field public. Panache will rewrite all direct accesses with the getter/setter dynamically for you.

Listing 7.5 The Reservation entity in the Reservation service

```
package org.acme.reservation.entity;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

import javax.persistence.Entity;
import java.time.LocalDate;

@Entity #1
public class Reservation extends PanacheEntity { #2

    public String userId;
    public Long carId;
    public LocalDate startDay;
    public LocalDate endDay;

    /**
     * Check if the given duration overlaps with this reservation
     * @return true if the dates overlap with the reservation, false
     * otherwise
     */
    public boolean isReserved(LocalDate startDay, LocalDate endDay) {
        return (!(this.endDay.isBefore(startDay) ||
            this.startDay.isAfter(endDay)));
    }

    @Override
    public String toString() {
        return "Reservation{" +
            "userId='" + userId + '\'' +
            ", carId=" + carId +
            ", startDay=" + startDay +
            ", endDay=" + endDay +
            ", id=" + id +
            '}';
    }
}
```

```

    }
}
```

#1 Add the `@Entity` annotation.

#2 Extend the `PanacheEntity` class that provides Panache functionality for the Reservation entity.

Optionally, you can also extend Panache entities with custom business methods that utilize the inherited static methods from the `PanacheEntity` class demonstrated in the example [Listing 7.1](#) at the beginning of this section. For instance, imagine you would need a method that returns all reservations for a particular car. This can be easily done as demonstrated in the [Listing 7.6](#). This method is just an example, but we will use this functionality later in different services.

Listing 7.6 Example business method in the Reservation entity

```

public static List<Reservation> findByCar(Long carId) {
    return list("carId", carId);
}
```

For more information about the available query options, you may read the Panache guide at <https://quarkus.io/guides/hibernate-orm-panache>.

7.1.3 Using Panache in the Reservation service

Now we can safely delete the generated stub repository that we created in Chapter 4 that is in the `InMemoryReservationsRepository` class. Remove this class. The `ReservationResource` needs to be adjusted as shown in the following listings. Let's start with the fields and constructor in [Listing 7.7](#). Because we are moving to the active record pattern, we can remove the `ReservationRepository` field and injection from the constructor.

Listing 7.7 Modification required in the ReservationResource after Reservation entity transformation to the active record pattern

```

@Path("reservation")
@Produces(MediaType.APPLICATION_JSON)
public class ReservationResource {

    private final InventoryClient inventoryClient;
    private final RentalClient rentalClient;

    @Inject
    javax.ws.rs.core.SecurityContext context;

    public ReservationResource(
            @GraphQLClient("inventory")
            GraphQLInventoryClient inventoryClient,
            @RestClient RentalClient rentalClient) {
        this.inventoryClient = inventoryClient;
        this.rentalClient = rentalClient;
    }

    ...
}

```

In [Listing 7.8](#), we can analyze how active record pattern handling differs from the original `ReservationRespository`. The `make` method shows an example of the instance method use with the `reservation.persist()` operation. We can also note the `@Transactional` annotation that the `persist` operation requires to save a record into a database. The `allReservations` and `availability` methods conversely demonstrate the use of static methods of the `Reservation` entity inherited from Panache.

Listing 7.8 The modified ReservationResource methods now working with active record pattern

```

@Consumes(MediaType.APPLICATION_JSON)
@POST
@Transactional #1
public Reservation make(Reservation reservation) {
    reservation.userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : "anonymous";
    reservation.persist(); #2
    Log.info("Successfully reserved reservation " + reservation);
    if (reservation.startDay.equals(LocalDate.now())) {
        Rental rental = rentalClient
            .start(reservation.userId, reservation.id); #3
        Log.info("Successfully started rental " + rental);
    }
}

```

```

    }

    return reservation;
}

@GET
@Path("all")
public Collection<Reservation> allReservations() {
    String userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : null;
    return Reservation.<Reservation>streamAll() #4
        .filter(reservation -> userId == null ||
            userId.equals(reservation.userId))
        .collect(Collectors.toList());
}

@GET
@Path("availability")
public Collection<Car> availability(@RestQuery LocalDate startDate,
                                      @RestQuery LocalDate endDate) {
    // obtain all cars from inventory
    List<Car> availableCars = inventoryClient.allCars();
    // create a map from id to car
    Map<Long, Car> carsById = new HashMap<>();
    for (Car car : availableCars) {
        carsById.put(car.id, car);
    }

    // get all current reservations
    List<Reservation> reservations =
        Reservation.listAll(); #5
    // for each reservation, remove the car from the map
    for (Reservation reservation : reservations) {
        if (reservation.isReserved(startDate, endDate)) {
            carsById.remove(reservation.carId);
        }
    }
    return carsById.values();
}

```

#1 The POST method must be executed in transaction. We explain transactions in a later section.

#2 Persisting the reservation into the database.

#3 The id field was generated when we persisted this reservation.

#4 An example use of static methods are inherited from the PanacheEntity. streamAll() needs to

get a hint of the entity type because Java type system can't yet deduct it directly.

#5 The `listAll()` method returns a list of all entities equivalent to `SELECT * FROM Reservation`.

If you are not already running then start the `Reservation` service in the dev mode:

```
$ quarkus dev
```

Since we intentionally didn't define the connection URL to our database in the `application.properties` for Dev mode, the Quarkus magic kicks in and boots a zero-config PostgreSQL database in a container for us. This container lives only during the lifecycle of the Dev mode. Once you stop the Dev mode, the container also stops. Additionally, our Quarkus application is already configured and connected to this PostgreSQL container! If you repeat the calls to make the reservation and get all reservations from Chapter 4, they are now propagated into the database.

Now, this is impressive. How little code is needed to connect an application to the database. This kind of experience is generally incomparable with any other framework.

If you make a few POST requests to the `/reservation` endpoint (easily in the Swagger UI provided at the <http://localhost:8081/q/swagger-ui/> or in the terminal, just remember to change the dates for the future) you can query the reservation from the database with a GET request to the same `/reservation/all` endpoint as before, but additionally you can also check (with any database connection tool of your liking) that they are persisted in the database. The example utilizing HTTPie and `psql` (PostgreSQL client) in terminal is available in [Listing 7.9](#). Note that the PostgreSQL container-mapped port generates randomly when the container starts, so you need to check which port your PostgreSQL container (`docker ps`) uses. The credentials for the database (set in the configuration) are `user:pass`.

TIP The `psql` command line utility is totally optional to verify performed operations. So if you don't have it installed, you don't need to install it just for these verifications.

Listing 7.9 Persisting entity and testing it is persisted in the database

```
$ http POST :8081/reservation <<< '{  
  "carId": 1,  
  "startDay": "3333-01-01",  
  "endDay": "3333-01-02"  
' #1  
  
...  
  
$ http :8081/reservation/all  
HTTP/1.1 200 OK
```

```

Content-Type: application/json
content-length: 69

[
    {
        "carId": 1,
        "endDay": "3333-01-02",
        "id": 1,
        "startDay": "3333-01-01",
        "userId": "anonymous"
    }
] #2
....
```

\$ psql -h localhost -p 37423 -U user \
-d quarkus -c 'SELECT * FROM Reservation' #3
Password for user user:

id	carid	endday	startday	userid
1	1	3333-01-02	3333-01-01	anonymous

#4
(1 row)

- #1 First, persist a new reservation with a **POST** call.
- #2 Check that the reservation is available in the application via a **GET** call.
- #3 Connect to the database directly with **psql** CLI.
- #4 The reservation persisted in the **Reservation** table.

If you are unsure about the name of the tables Hibernate generated for your entities, you can always check the scripts and the entities that the Hibernate extension knows about in the dev UI, as shown in [Figure 7.1](#).

The screenshot shows the Hibernate ORM Dev UI interface. At the top, there's a header bar with the title "Hibernate ORM". Below it, a sidebar on the left says "Define your persistent model with Hibernate ORM and JPA" and lists "Persistence units 1", "Entities 1", and "Named Queries 0". The main area is titled "Persistence Unit <default>". It contains a table with two columns: "# Class name" and "Table name". There is one row: "# 1. org.acme.reservation.entity.Reservation" and "Table name Reservation".

#	Class name	Table name
1.	org.acme.reservation.entity.Reservation	Reservation

Figure 7.1 Hibernate ORM in dev UI

Now we also can clean the remains of the `ReservationRepository` since the active record pattern replaced it. We can remove the whole `org.acme.reservation.reservation` package. However, this breaks the `ReservationRepositoryTest`. Let's update this test to use the `Reservation` record pattern, which will persist data into a real PostgreSQL database instance that starts with Dev Services for the test execution. Rename the `ReservationRepositoryTest` to `ReservationPersistenceTest` and update the code as demonstrated in [Listing 7.10](#).

Listing 7.10 Updated ReservationPersistenceTest test testing reservation persist operations

```
package org.acme.reservation;

import io.quarkus.test.junit.QuarkusTest;
import org.acme.reservation.entity.Reservation;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import javax.transaction.Transactional;
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

@QuarkusTest
public class ReservationPersistenceTest {

    @Test
    @Transactional #1
    public void testCreateReservation() {
        Reservation reservation = new Reservation();
        Assertions.assertEquals(null, reservation.getId());
        Assertions.assertEquals("2023-01-01", reservation.getDate());
        Assertions.assertEquals(1, reservation.getDuration());
        Assertions.assertEquals("2023-01-01T00:00:00", reservation.getFrom());
        Assertions.assertEquals("2023-01-01T23:59:59", reservation.getTo());
    }
}
```

```

reservation.startDay = LocalDate.now().plus(5, ChronoUnit.DAYS);
reservation.endDay = LocalDate.now().plus(12, ChronoUnit.DAYS);
reservation.carId = 384L;
reservation.persist(); #2

Assertions.assertNotNull(reservation.id);
Assertions.assertEquals(1, Reservation.count());
Reservation persistedReservation =
    Reservation.findById(reservation.id); #3
Assertions.assertNotNull(persistedReservation);
Assertions.assertEquals(reservation.carId,
    persistedReservation.carId);
}
}

```

#1 We need to start a transaction again since we persist in an actual database.

#2 Real database insert into Dev Services PostgreSQL database.

#3 Real select into the database.

7.2 Panache repository pattern

The active record pattern introduced in the previous section is not the right cup of tea for all developers. However, Panache is not constrained only to this model. A very popular alternative is the repository pattern. This pattern defines a custom object called the repository, which is responsible for all the database manipulation operations (remember that active record pattern encapsulated these operations in the record itself). Externalizing manipulation from the record entity classes indeed has benefits (e.g., flexibility or separation of concerns), but eventually, the developer chooses which pattern is more suitable for the application at hand.

We introduce the repository pattern in the Inventory service. Of course, we could again use PostgreSQL, but would you maybe prefer another database? So why we don't use a different one this time? Of course, for Panache, this is not a problem at all. Changing to another database is just a matter of changing the extension for the JDBC driver we include in our Quarkus application.

In the Inventory service, we use another very popular open-source relational database MySQL but this time with the repository pattern.

TIP Panache provides both models out of the box. So the choice of the database is orthogonal to the chosen API. We could also use a repository pattern with the same extensions in the previous section.

7.2.1 Utilizing repository pattern with MySQL

Similarly, as with the active record, we need to do three things in order to use Panache with repository pattern:

- Add required extensions and configuration.
- Create the entity classes.
- Define the repository class.

PANACHE EXTENSIONS AND CONFIGURATION FOR MYSQL

We also need two extensions—one for Panache and one for the MySQL JDBC driver. This is exactly the same as with the active record pattern in the previous section. We just utilize a different JDBC driver—this time for MySQL. Changing into the `inventory-service` directory, we can add the extensions either with the CLI or directly to the `pom.xml` as shown in [Listing 7.11](#).

Listing 7.11 Adding required Panache extensions to the Inventory service

```
$ quarkus ext add quarkus-hibernate-orm-panache quarkus-jdbc-mysql
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✅ Extension io.quarkus:quarkus-hibernate-orm-panache has been \
installed
[SUCCESS] ✅ Extension io.quarkus:quarkus-jdbc-mysql has been installed

# or add extensions to the pom.xml manually
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jdbc-mysql</artifactId>
</dependency>
```

The configuration is also very similar, but we are connecting to the MySQL database this time. To provide configuration for MySQL, we need to add the properties available in [Listing 7.12](#) to the `application.properties` in the `Inventory` service.

Listing 7.12 Properties required for MySQL connection

```
# configure your datasource
quarkus.datasource.db-kind = mysql #1
quarkus.datasource.username = user
quarkus.datasource.password = pass
%prod.quarkus.datasource.jdbc.url =
→ jdbc:mysql://localhost:3306/inventory #2

# drop and create the database at startup
quarkus.hibernate-orm.database.generation = drop-and-create
```

#1 Use the MySQL driver.

#2 Specify connection URL only in production mode to allow Dev Services.

The configuration is relatively similar, as we have learned with the active record pattern. If we developed entities in the same way as in the previous chapter, it would still work. But we want to utilize the repository pattern in the Inventory service, so the code we need to develop is slightly different.

DEVELOPING ENTITIES FOR USE WITH THE REPOSITORY

With the repository pattern, we don't need to do anything special about the created entities. Defining standard JPA entities work as we would expect. So let's use the JPA approach in the Inventory service. We can now modify the `org.acme.inventory.model.Car` class as shown in the [Listing 7.13](#) to make it a valid JPA entity:

Listing 7.13 The Car entity

```

package org.acme.inventory.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Car {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String licensePlateNumber;
    private String manufacturer;
    private String model;

    // getters, setters, equals, and hashCode omitted
}

```

TIP If you don't like to generate all getters and setters to keep them in your application you can still extend `PanacheEntityBase` or `PanacheEntity` and use public fields. During build time, Quarkus rewrites all accesses to the public fields to use the accessor methods.

CREATING THE REPOSITORY

The last step in using the repository pattern is the actual definition of our repository class that will manage persistent access for cars. This class must implement the `PanacheRepository` interface providing the record type as the generic parameter. It also needs to be defined as a CDI bean (e.g., with a bean-defining annotation). We could utilize the already created `CarInventory`, but since we are creating a repository, it is better to keep class names consistent. Create a new class `CarRepository` in the new package `org.acme.inventory.repository` as shown in the [Listing 7.14](#).

TIP If you need to define a custom ID type, you can also implement the `PanacheRepositoryBase` which presents you with the option to set your own ID type as a second type parameter.

Listing 7.14 The code of the CarRepository class

```
package org.acme.inventory.repository;

import io.quarkus.hibernate.orm.panache.PanacheRepository;
import org.acme.inventory.model.Car;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class CarRepository implements PanacheRepository<Car> {
}
```

Out of the box, this provides you with the same methods we had at our disposal with the static methods inherited from the active record pattern. Similarly, as with inheritance in the active record pattern, you can provide your custom methods if needed. An example of a custom business method in the `CarRepository` is demonstrated in the [Listing 7.15](#). Add this method to the `CarRepository`.

Listing 7.15 Custom business method in CarRepository

```
@ApplicationScoped
public class CarRepository implements PanacheRepository<Car> {

    public Optional<Car> findByLicensePlateNumberOptional(
        String licensePlateNumber) {
        return find("licensePlateNumber", licensePlateNumber)
            .firstResultOptional();
    }
}
```

Our application is now ready to utilize this repository to persist and retrieve rentals from the MySQL database.

7.2.2 Using the repository pattern in Inventory

Now that we have the repository in place, we can safely remove the generated stub `CarInventory`. However, there is one catch. We predefined two cars that populate the inventory with some initial data. It would be good to keep them. There are several ways of how we can do this—e.g., we could use the same `@PostConstruct` concept in the new `CarRepository` or observe the Quarkus startup event. But Panache provides a more elegant and portable solution. We can include a custom `import.sql` SQL script in the `src/main/resources` directory to prepopulate the database. This script automatically executes when Quarkus connects to the database. In our case, it populates the database with our initial data. Surely, this is useful only for testing and development. Hence, it is disabled in prod mode by default. We can add a new file in `src/main/resources/import.sql` with the content available in the [Listing 7.16](#). These SQL commands execute when Panache starts.

Listing 7.16 The example SQL import.sql file to prepopulate the database

```
INSERT INTO Car (licensePlateNumber, manufacturer, model)
  VALUES ('ABC123', 'Mazda', '6');
INSERT INTO Car (licensePlateNumber, manufacturer, model)
  VALUES ('XYZ987', 'Ford', 'Mustang');
```

Next, we can adjust the `GraphQLInventoryService` and the `GrpcInventoryService` (our exposed APIs) to persist and list the cars from the database. Notice that the methods that modify the database also need to be run in the transaction (`@Transactional`) for the change to be committed to the database.

The `GraphQLInventoryService` needs to replace the car access with the repository access. But since we now also have the custom business method that returns `Car` by the license number in the `CarRepository`, we can replace the manual filtering in the `remove` method with the invocation passed directly to the database. The code of the updated `GraphQLInventoryService` is available in [Listing 7.17](#).

Listing 7.17 The GraphQLInventoryService modified code

```
package org.acme.inventory.service;

import org.acme.inventory.model.Car;
import org.acme.inventory.repository.CarRepository;
import org.eclipse.microprofile.graphql.GraphQLApi;
import org.eclipse.microprofile.graphql.Mutation;
import org.eclipse.microprofile.graphql.Query;

import javax.inject.Inject;
import javax.transaction.Transactional;
import java.util.List;
```

```

import java.util.Optional;

@GraphQLApi
public class GraphQLInventoryService {

    @Inject
    CarRepository carRepository; #1

    @Query
    public List<Car> cars() {
        return carRepository.listAll();
    }

    @Transactional #2
    @Mutation
    public Car register(Car car) {
        carRepository.persist(car); #3
        return car;
    }

    @Mutation
    public boolean remove(String licensePlateNumber) {
        Optional<Car> toBeRemoved = carRepository
            .findByLicensePlateNumberOptional(
                licensePlateNumber); #4
        if (toBeRemoved.isPresent()) {
            carRepository.delete(toBeRemoved.get());
            return true;
        } else {
            return false;
        }
    }
}

```

#1 Inject the CarRepository as a CDI bean.

#2 We again need to use @Transactional if we want to persist or modify entities.

#3 Persisting is done by calling the persist method on the repository.

#4 Utilize the custom business method we additionally defined in the CarRepository.

The code of the `GrpcInventoryService` requires similar modifications. But because we are delegating the invocation of the `persist` method outside the CDI bean, and we still need to run it in a transaction, we need to wrap the `persist` invocation in a new transaction created with the programmatic API. as shown in [Listing 7.18](#). Each `persist` operation runs in its own transaction.

Listing 7.18 The modified code of the GrpcInventoryService

```

@GrpcService
public class GrpcInventoryService
    implements InventoryService {

    @Inject
    CarRepository carRepository; #1

    @Override
    @Blocking #2
    public Multi<CarResponse> add(Multi<InsertCarRequest> requests) {
        return requests
            .map(request -> {
                Car car = new Car();
                car.setLicensePlateNumber(request.getLicensePlateNumber());
                car.setManufacturer(request.getManufacturer());
                car.setModel(request.getModel());
                return car;
            }).onItem().invoke(car -> {
                Log.info("Persisting " + car);
                QuarkusTransaction.run( () ->
                    carRepository.persist(car) #3
                );
            }).map(car -> CarResponse.newBuilder()
                .setLicensePlateNumber(car.getLicensePlateNumber())
                .setManufacturer(car.getManufacturer())
                .setModel(car.getModel())
                .setId(car.getId())
                .build());
    }

    @Override
    @Blocking
    @Transactional #4
    public Uni<CarResponse> remove(RemoveCarRequest request) {
        Optional<Car> optionalCar = carRepository
            .findByLicensePlateNumberOptional(
                request.getLicensePlateNumber());

        if (optionalCar.isPresent()) {
            Car removedCar = optionalCar.get();
            carRepository.delete(removedCar);
            return Uni.createFrom().item(CarResponse.newBuilder()

```

```

        .setLicensePlateNumber(removedCar.getLicensePlateNumber())
        .setManufacturer(removedCar.getManufacturer())
        .setModel(removedCar.getModel())
        .setId(removedCar.getId())
        .build());
    }
    return Uni.createFrom().nullItem();
}
}

```

#1 We need to inject the CarRepository.

#2 Use the `@Blocking` annotation to tell Quarkus that this method now executes blocking operation (`persist`) on a database. Otherwise, Quarkus won't run it as gRPC defaults to fast event loop threads which cannot be blocked.

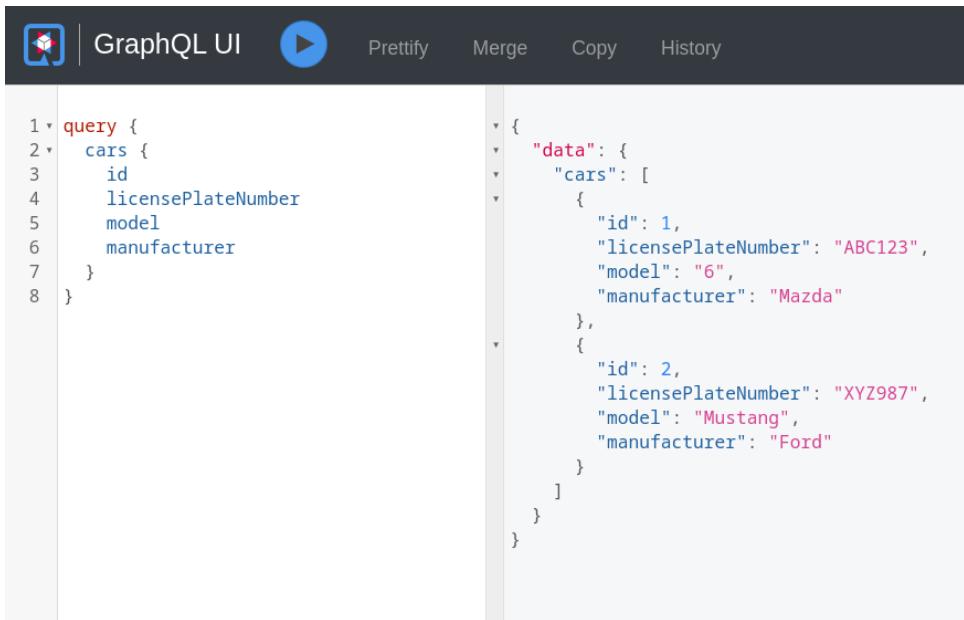
#3 Run `persist()` in a transaction which is committed immediately.

#4 Run the whole remove method in a single transaction committed only when the method ends.

We could have used a `@Transactional` annotation on the `add` method instead of using the `QuarkusTransaction.run()` method for each car separately. But with this approach, the whole operation would be run in a single transaction, and that transaction would be committed only when the whole stream of cars is processed (and the gRPC stream closed). So we would have a transaction that stays open for a long time, which could easily become a performance bottleneck, and we could also run into a transaction timeout if the stream stayed open for too long (the default transaction timeout is one minute). We will explain `QuarkusTransaction#run` method in more detail in section [7.5.2](#). Of course, if the requirement would be to process cars additions in the chunks defined by the stream, we would need to wrap the whole method in the transaction (and adjust transactions timeouts accordingly).

And now we can safely delete the `CarInventory` as its functionality is now fully replaced with the `CarRepository`.

If you now start the Inventory service in the Dev mode (`quarkus dev`), the MySQL container with your development database starts in the background, and your Quarkus application connects to it. If the Dev mode starts successfully, we can utilize the exposed GraphQL API as showed in the figure [Figure 7.2](#) in the GraphQL UI to query all cars and verify that the database was prepopulated with the two cars from `import.sql`. The GraphQL UI is available at <http://localhost:8083/q/graphql-ui>.



The screenshot shows the GraphQL UI interface. On the left, a code editor displays a GraphQL query:

```

1 * query {
2 *   cars {
3 *     id
4 *     licensePlateNumber
5 *     model
6 *     manufacturer
7 *   }
8 }

```

On the right, the results of the query are shown as JSON data:

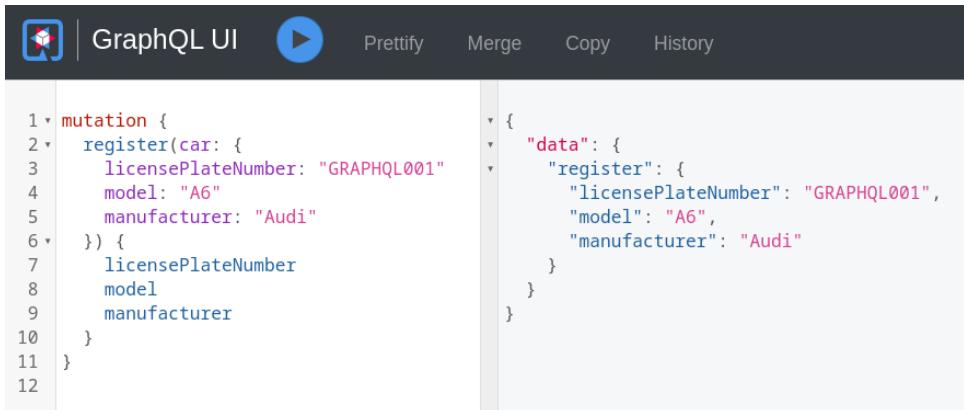
```

{
  "data": {
    "cars": [
      {
        "id": 1,
        "licensePlateNumber": "ABC123",
        "model": "6",
        "manufacturer": "Mazda"
      },
      {
        "id": 2,
        "licensePlateNumber": "XYZ987",
        "model": "Mustang",
        "manufacturer": "Ford"
      }
    ]
  }
}

```

Figure 7.2 Inventory GraphQL query example now pulling the data from the database

We can now register new cars either through the GraphQL ([Figure 7.3](#)) or through the gRPC ([Figure 7.4](#)) exposed APIs:



The screenshot shows the GraphQL UI interface. On the left, a code editor displays a GraphQL mutation:

```

1 * mutation {
2 *   register(car: {
3 *     licensePlateNumber: "GRAPHQL001"
4 *     model: "A6"
5 *     manufacturer: "Audi"
6 *   }) {
7 *     licensePlateNumber
8 *     model
9 *     manufacturer
10 }
11 }
12

```

On the right, the results of the mutation are shown as JSON data:

```

{
  "data": {
    "register": {
      "licensePlateNumber": "GRAPHQL001",
      "model": "A6",
      "manufacturer": "Audi"
    }
  }
}

```

Figure 7.3 Register car through the exposed GraphQL API

```

1 {
2   "licensePlateNumber": "GRPC001",
3   "manufacturer": "Hyundai",
4   "model": "i30"
5 }

```

Send

Figure 7.4 Register car through the exposed gRPC API

Finally, we can verify that both new registrations have been persisted in the database through the GraphQL UI again, as showed in the [Figure 7.5](#).

```

query {
  cars {
    id
    licensePlateNumber
    model
    manufacturer
  }
}

```

```

{
  "data": {
    "cars": [
      {
        "id": 1,
        "licensePlateNumber": "ABC123",
        "model": "6",
        "manufacturer": "Mazda"
      },
      {
        "id": 2,
        "licensePlateNumber": "XYZ987",
        "model": "Mustang",
        "manufacturer": "Ford"
      },
      {
        "id": 3,
        "licensePlateNumber": "GRAPHQL001",
        "model": "A6",
        "manufacturer": "Audi"
      },
      {
        "id": 4,
        "licensePlateNumber": "GRPC001",
        "model": "i30",
        "manufacturer": "Hyundai"
      }
    ]
  }
}

```

Figure 7.5 Verifications of the newly registered cars in the GraphQL UI

But to also verify that we have the results actually in the MySQL database, we can also run the following `mysql` CLI command available in [Listing 7.19](#) to list all records from the Car database:

TIP The `mysql` tool verification is again totally optional, and you don't need to install it if you don't already have it installed.

Listing 7.19 List all persisted Cars in the MySQL database

```
$ mysql -h localhost -P 46257 --protocol=tcp -u user -p quarkus \
<<< 'SELECT * FROM Car;'
Enter password:
+----+-----+-----+-----+
| id | licensePlateNumber | manufacturer | model |
+----+-----+-----+-----+
| 1  | ABC123  | Mazda   | 6      |
| 2  | XYZ987  | Ford    | Mustang |
| 3  | GRAPHQL001 | Audi   | A6     |
| 4  | GRPC001 | Hyundai | i30   |
+----+-----+-----+-----+
```

7.3 Traditional ORM/JPA data access

Both the active record and the repository pattern provide simplified data access management. However, if you prefer the traditional JPA/Hibernate APIs we have used in enterprise applications for years, you can still utilize the entire landscape of Hibernate ORM/JPA features in your Quarkus applications.

We don't use JPA in our Car rental services. However, we want to teach you how to set up your Quarkus application if you wish to do so.

To use Hibernate ORM in your application, you need to add the extensions for the `quarkus-hibernate-orm` and the JDBC driver for the database of your choice. You can develop your JPA entities in the exact same way as in JPA (an example is also provided in the section [7.2.1](#)). One difference from typical JPA applications is that you don't need to include `persistence.xml` in your application if you don't require more advanced configuration. You can configure everything in the `application.properties` with the same properties as shown in the previous sections. Of course, if you include the JDBC driver, the Dev services start the database for you in the Dev and test modes.

If you include all steps as described, you can inject the `EntityManager` from the JPA specification as a CDI bean, as shown in [Listing 7.20](#):

Listing 7.20 Utilizing standard JPA/ORM data access with EntityManager

```
@Path("/users")
public class PersonResource {

    @Inject
    EntityManager em; #1

    @POST
    @Transactional
    public void add(String name) {
        Person person = new Person();
        person.setName(name);

        em.persist(person); #2
    }
}
```

#1 Injecting the EntityManager.

#2 You can use the EntityManager as specified in the JPA.

7.4 REST Data

A large number of enterprise production applications expose REST APIs that provide CRUD (Create, Read, Update, Delete) operations over an entity in the database. To simplify this tedious and often repeated task, Quarkus provides a set of REST data extensions that expose CRUD operations over Panache entities through the generated JAX-RS resources.

At the time of writing this book, these extensions are still considered experimental, and their supported configurations don't cover all the combinations of data access methods that Quarkus supports in general. However, there are REST Data extensions for all varieties that we are discussing in this book (which also corresponds to the most popular user choices). Adnew combinations might be included in the future.

TIP Experimental extensions are not guaranteed to be stable, and Quarkus provides them with the intent to collect user feedback. However, REST Data extensions are not changing frequently in the last releases, so they might be already considered relatively stable.

The REST Data extensions include:

- `quarkus-hibernate-orm-rest-data-panache` — JDBC with Hibernate ORM.

- `quarkus-mongodb-rest-data-panache` — MongoDB (NoSQL) with Hibernate ORM.
- `quarkus-hibernate-reactive-rest-data-panache` — JDBC with Hibernate Reactive.

Since we haven't covered Hibernate Reactive and Panache with NoSQL databases (like MongoDB) yet, we focus on `quarkus-hibernate-orm-rest-data-panache` in this section. However, we learn about REST Data modifications for Hibernate Reactive and MongoDB in subsequent sections.

To demonstrate REST Data in action, we include it in the Reservation service, which already provides a subset of CRUD operations over the reservations.

To include REST Data that generate the JAX-RS resource over an entity from the relation database, we need to add three extensions (if not already included):

- The REST Data extension `quarkus-hibernate-orm-rest-data-panache`.
- The JDBC driver (e.g., `quarkus-jdbc-postgresql`).
- The RESTEasy JSON serialization extension (e.g., `quarkus-resteasy-reactive-jackson`).

You can easily add these extensions to the Reservation service with the command available in [Listing 7.21](#). Notice that `quarkus-jdbc-postgresql` and `quarkus-resteasy-reactive-jackson` are skipped because we already added them into Reservation.

Listing 7.21 Adding REST Data required extensions to the Reservation service

```
$ quarkus ext add quarkus-hibernate-orm-rest-data-panache \
  quarkus-jdbc-postgresql quarkus-resteasy-reactive-jackson
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✅ Extension io.quarkus:quarkus-hibernate-orm-rest-data-panache
→ has been installed
👍 Extension io.quarkus:quarkus-resteasy-reactive-jackson was already
→ installed
👍 Extension io.quarkus:quarkus-jdbc-postgresql was already installed
```

Next, we need to develop the Panache entities and/or repositories like in section [7.1](#). We define the `Reservation` entity as demonstrated in [Listing 7.5](#).

Lastly, we must define interfaces that denote which Panache entities or repositories should REST Data expose as JAX-RS resources. They are required to extend either the `PanacheEntityResource` interface or, in case of use of the repository pattern, the `PanacheRepositoryResource` interface. Quarkus generates the JAX-RS resources based on these marking files when the application is compiled. Since we used the active record pattern in Reservation service, we can define the `ReservationCrudResource` in the `org.acme.reservation.rest` package as shown in [Listing 7.22](#) (we need to define a different class name because we already have a `ReservationResource`).

Listing 7.22 The code of ReservationCrudResource

```
package org.acme.reservation.rest;

import io.quarkus.hibernate.orm.rest.data.panache.PanacheEntityResource;
import org.acme.reservation.entity.Reservation;

public interface ReservationCrudResource extends
    PanacheEntityResource<Reservation, Long> {
}
```

The individual operations definitions are in the `RestDataResource` parent interface with a detailed JavaDoc describing respective operations.

As these interfaces serve as marking files for the REST data extension to select entities or repositories that it should expose as JAX-RS resources, users shouldn't implement them. Also, the extension ignores any additional methods in these interfaces. However, you can provide overrides for the inherited methods in order to customize them. To do the customizations, you can annotate the REST Data interfaces with these two annotations:

- `@ResourceProperties` — e.g., to change the REST path of the resource or if the collection responses return paged results. This annotation should be applied on the REST data interface itself.
- `@MethodProperties` — e.g., to define that a particular method is not exposed. This annotation should be applied on an overriding method declaration in the REST data interface.

Since the default path of our resource generates from the class name (i.e., `ReservationCrudResource`) it is `/reservation-crud` which is probably not what users of our API would expect, even if it's descriptive. We also possibly don't want to expose these operations to everybody. It would be easier to move them under a different root path for more straightforward future enhancements. We can modify the `ReservationCrudResource` to add the `@ResourceProperties` annotation changing the path as shown in [Listing 7.23](#) to change the root path on which this generated REST Data endpoint is exposed to `/admin/reservation`:

Listing 7.23 Customization of the root path for ReservationCrudResource

```
package org.acme.reservation.rest;

import io.quarkus.hibernate.orm.rest.data.panache.PanacheEntityResource;
import io.quarkus.rest.data.panache.ResourceProperties;
import org.acme.reservation.entity.Reservation;

@ResourceProperties(path = "/admin/reservation")
public interface ReservationCrudResource extends
    PanacheEntityResource<Reservation, Long> {
}
```

If you start the application in Dev mode (if not already running) and go to <http://localhost:8081/q/swagger-ui/> in your browser, you see that the new JAX-RS resource was generated for us, and it exposes now all CRUD operations over the Reservation entity. The [Listing 7.6](#) presents an excerpt from the Swagger UI showing this new endpoint.



Figure 7.6 Generated REST Data resource in Swagger UI

We will not secure the /admin/reservation URL for simplicity. But feel free to utilize what you learned in Chapter 6 to secure it yourself as an exercise.

7.5 Transactions

Enterprise transactions are surely a topic worth a separate book. Especially in distributed environments like microservices, transactional processing is a complicated problem. We won't dive too deep into the details, but it's important to cover how you need to handle transactions in Quarkus, specifically when used for data access.

Quarkus comes with an integrated transaction manager. If you use any of the extensions that need transactions, they transitively include it for you. If you want to use transactions outside this scope, you might add the `quarkus-narayana-jta` extension to your Quarkus application (Narayana is the name of a project which is an open-source implementation of the transaction manager).

Quarkus provides two ways of handling transactions — declarative and manual.

7.5.1 Declarative transactions

Quarkus employs the JTA (Java/Jakarta Transactions API) to denote transactions. We already learned about the `@Transactional` annotation that we used to define transactions when we were persisting our Car rental entities. This annotation comes from JTA. If you put it on a CDI method, it states that the method runs inside a transaction. The transaction, by default, is started before the method begins, and it's committed or rolled back (depending on the method throwing specified exceptions) when it finishes. The `@Transactional` annotation contains a parameter of type `Transactional.TxType` which specifies how or if the transaction starts:

- `TxType.REQUIRED` — This is the default. If there is no incoming transaction, start a new one. Join the existing transaction otherwise.
- `TxType.REQUIRES_NEW` — Always execute the method in a new transaction. The framework suspends any existing transaction for the duration of the method execution.
- `TxType.MANDATORY` — Must be invoked with an already started transaction. Otherwise, it throws an exception.
- `TxType.SUPPORTS` — Method can run both with or without a transaction.
- `TxType.NOT_SUPPORTED` — Must be run without transaction. The framework suspends any incoming transaction for the duration of the method execution.
- `TxType.NEVER` — If called within an existing transaction, the framework throws an exception.

The type you choose depends on the use case of the business method. For instance, if the Reservation service would expose an endpoint computing some statistics over all reservations, we don't need to execute this operation in the transaction, so `SUPPORTS` or `NOT_SUPPORTED` would be preferred types. However, remember that any operation with transactions (starting or suspending) requires some processing that adds overhead to the method invocation. So it's better not to start transactions if you don't need them.

We can also use this annotation at the class level. It then propagates to all methods. If needed, you can also override the class-level annotation on individual methods. Quarkus additionally also provides the `@TransactionConfiguration` annotation that you can use to set transaction timeout.

7.5.2 Manual handling with QuarkusTransaction

The `UserTransaction` class, which comes from the JTA, can be easily injected into any CDI bean to utilize the manual management of transactions. However, Quarkus provides a more user-friendly custom API for manual transaction handling in the `QuarkusTransaction` class. `QuarkusTransaction` offers several ways of handling transactions.

The first option of utilization is very similar to the `UserTransaction` from JTA, where you can manually begin and commit/rollback transactions. But instead of JTA, you can do this with static methods of `QuarkusTransaction` so you don't need to keep a reference of the user transaction manually. The [Listing 7.24](#) provides an example of this API. The `begin` method also provides an override consuming the `BeginOptions` instance able to customize some transaction options like the timeout, for example.

Listing 7.24 QuarkusTransaction with manual begin and commit/rollback calls

```
QuarkusTransaction.begin();
// perform operation
result = ...
if (result != null) {
    QuarkusTransaction.commit();
} else {
    QuarkusTransaction.rollback();
}
```

Furthermore, `QuarkusTransaction` also provides a set of wrappers around `Runnable` or `Callable` that execute delegated code in the transaction. The [Listing 7.25](#) shows an example of this API. Additionally, Quarkus also allows users to provide an instance of the `RunOptions` class as a parameter to each transaction started in this way that allows changing the configuration of the transaction semantic (`REQUIRE_NEW`, `JOIN_EXISTING`, `DISALLOW_EXISTING`, `SUSPEND_EXISTING`), transaction timeout, or to provide an exception handler that can decide that, in some cases of exceptions, the transaction should still commit successfully.

Listing 7.25 Transaction wrappers provided by QuarkusTransaction

```
QuarkusTransaction.run(() -> { #1
    // perform operation
})

String result = QuarkusTransaction.call( #2
    QuarkusTransaction.runOptions() #3
        .semantic(RunOptions.Semantic.SUSPEND_EXISTING)
        .timeout(30)
        .exceptionHandler((throwable -> {
            if (throwable instanceof OKException) {
                return RunOptions.ExceptionResult.COMMIT;
            }
            return RunOptions.ExceptionResult.ROLLBACK;
        })),
    () -> {
    // perform operation
    return "result";
});
```

#1 Run a Runnable inside a transaction.

#2 Run a Callable inside a transaction.

#3 Optionally provide transaction configuration.

7.5.3 Transactions with Panache

In Quarkus, every operation that modifies the database (e.g., persist or update) must execute inside a transaction. Hibernate requires this to propagate the in-memory changes into the downstream database. Quarkus recommends doing transactions at the entry points as we did in our REST resource methods.

Quarkus also additionally provides two methods that force the changes to be sent to the database prematurely (by default, this is done only at the end of the transaction or before the query):

- `flush()` — Sends current changes to the database.
- `persistAndFlush(entity)` — Persist and flush in a single method call.

These methods are available both in the active record and in the repositories. They might be helpful if you need to execute record validation or if you work on big data sets. In every case, use them wisely, as these forms of communication with the database tend to cost a lot of application resources.

7.6 Panache with NoSQL

NoSQL (non-SQL but sometimes also referred to as Not-Only SQL) databases are a newer alternative to relational databases. These databases store the data in different formats than traditional relational databases. Many types of NoSQL data stores include key-value, wide column, graph, or document. Quarkus supports many distinct NoSQL databases (MongoDB, Cassandra, Elasticsearch, Neo4j, and Amazon DynamoDB, to name a few) to cover respective users' preferences.

One of the most popular NoSQL databases is MongoDB (<https://www.mongodb.com/>). MongoDB is the most widely used document-based NoSQL database. This is why we will introduce it in the Rental service as our data store.

Quarkus supports MongoDB with its provided API through the `mongodb-client` extension. However, it also provides an integration with the Panache framework that vastly simplifies the development of MongoDB documents. This means that we can write very similar code to what we did for the relational databases. However, Panache stores our entities as documents (represented as JSON) in a NoSQL database this time.

7.6.1 Utilizing MongoDB with Panache

If we want to integrate Panache with MongoDB into a Quarkus application, we need to do a few things:

- Add the `quarkus-mongodb-panache` extension and the relevant connection configuration.
- Define entity classes.
- Make entities extend `PanacheMongoEntity`.
- Optionally also define a repository class extending `PanacheMongoRepository`.

PANACHE MONGODB CONFIGURATION

The MongoDB comes integrated with Panache in a single extension, so we only need to add the `quarkus-mongodb-panache` extension. You can add it to the Rental service (`rental-service` directory), as demonstrated in [Listing 7.26](#):

Listing 7.26 Adding quarkus-mongodb-panache extension

```
$ quarkus ext add quarkus-mongodb-panache
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✓ Extension io.quarkus:quarkus-mongodb-panache has been
→ installed
```

Next, we must configure the connection information in the `application.properties`. The connection string is the only required property. However, if you don't use the `@MongoEntity` annotation on your entities, you also must define the name of the connected database. The [Listing 7.27](#) provides an example of MongoDB connection properties. As you might notice, we are again using the prod profile for the connection string, so we have Dev services for MongoDB started in the Dev mode (since in Dev mode, this property is undefined).

Listing 7.27 Properties for the MongoDB Panache connection information

```
%prod.quarkus.mongodb.connection-string = mongodb://mongo:27017
```

```
quarkus.mongodb.database = rental #1
```

#1 The name of the database is only required if you don't specify it in the entities.

DEVELOPING PANACHE MONGODB ENTITIES

The development is very similar to using Panache with relational databases. The only change is that entities now need to extend a specific MongoDB entity base in the class `PanacheMongoEntity`, or it can also extend the `PanacheMongoEntityBase` if you want to declare your own ID field.

Similarly, if you prefer to use the repository pattern, then you need to create a repository class extending the `PanacheMongoRepository` or if you want to change the type of the used IDs, then you can extend the `PanacheMongoRepositoryBase`.

In the Rental service, we use the active record pattern. We can move the created `org.acme.rental.Rental` class into `org.acme.rental.entity.Rental` to represent our entity and modify it as shown in [Listing 7.28](#):

Listing 7.28 The Rental Panache MongoDB entity in the Rental service

```
package org.acme.rental.entity;

import io.quarkus.mongodb.panache.PanacheMongoEntity;

import java.time.LocalDate;

public class Rental extends PanacheMongoEntity { #1

    public String userId; #2
    public Long reservationId;
    public LocalDate startDate;
    public LocalDate endDate;
    public boolean active;
}
```

#1 We need to extend PanacheMongoEntity.

#2 We can remove the original ID field since it is now provided automatically and only keep the userId referencing the user owning this rental.

If you need to change the name of the created collection, the name of the database, or the name of the client, you can add the optional `@MongoEntity` annotation that allows you to change this information. By default, the created collection is named `Rental`, but if we would like to call it `Rentals` (as some teams prefer naming collections in plural), we could do it with the `@MongoEntity` annotation as this:

```
@MongoEntity(collection = "Rentals")
public class Rental extends PanacheMongoEntity {
```

Now we are ready to persist cars into the MongoDB NoSQL databases.

Quarkus uses the MongoDB Java driver to serialize Panache entities into the MongoDB documents. This conversion uses the BSON (Binary JSON, <https://bsonspec.org/>) formats utilized in MongoDB. So if you need to customize any mapping information, you need to utilize annotations from the `org.bson.codecs.pojo.annotations` package. For instance, if you want to rename a particular field, it is done like this:

```
@BsonProperty("rental-active")
public boolean active;
```

The Rental service won't compile now. But we will fix it the following section.

7.6.2 Using the Panache MongoDB in Rental

We can now extend the Rental service functionality to persist data into MongoDB. Additionally, since we now track the rental state, we can add a few business methods into the `Rental` entity class that we can later utilize in the exposed REST API. Let's add two new methods into the `Rental` class as shown in [Listing 7.29](#).

Listing 7.29 Adding custom business methods to Rental entity

```
public static Optional<Rental> findByUserAndReservationIdsOptional(
    String userId, Long reservationId) {
    return find("userId = ?1 and reservationId = ?2",
        userId, reservationId)
        .firstResultOptional(); #1
}

public static List<Rental> listActive() {
    return list("active", true); #2
}
```

#1 Find Rental with passed `userId` and `reservationId` if it exists.

#2 List all active rentals.

Now we can update the JAX-RS resource in class `RentalResource` to look like presented in [Listing 7.30](#).

Listing 7.30 The code of the `RentalResource` with database integration

```
@Path("/rental")
public class RentalResource {

    @Path("/start/{userId}/{reservationId}")
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Rental start(String userId,
        Long reservationId) {
        Log.infof("Starting rental for %s with reservation %s",
            userId, reservationId);

        Rental rental = new Rental();
        rental.userId = userId;
        rental.reservationId = reservationId;
        rental.startDate = LocalDate.now();
        rental.active = true;
```

```

        rental.persist(); #1

        return rental;
    }

    @PUT
    @Path("/end/{userId}/{reservationId}")
    public Rental end(String userId, Long reservationId) {
        Log.infof("Ending rental for %s with reservation %s",
        userId, reservationId);
        Optional<Rental> optionalRental = Rental #2
            .findByUserAndReservationIdsOptional(userId, reservationId);

        if (optionalRental.isPresent()) {
            Rental rental = optionalRental.get();
            rental.endDate = LocalDate.now();
            rental.active = false;
            rental.update(); #3
            return rental;
        } else {
            throw new NotFoundException("Rental not found");
        }
    }

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

    @GET
    @Path("/active")
    public List<Rental> listActive() {
        return Rental.listActive(); #4
    }
}

```

#1 MongoDB supports transactions since version 4.0, but we are not required to use them. This is why we don't have `@Transactional` in this class.

#2 Use the custom business method to get the rental from MongoDB.

#3 If found, update the properties and update the record in the data store.

#4 Use the custom method to list only the active rentals.

If we now start the Rental service in the Dev mode (`quarkus dev`), we can verify that the persisted rentals are in the MongoDB started with Dev services as shown in [Listing 7.31](#).

Listing 7.31 Verification of the Rental service functionality

```
$ http POST :8082/rental/start/user1/1
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
content-length: 122

{
    "active": true,
    "endDate": null,
    "id": "640f540f00e031200c6230ec",
    "reservationId": 1,
    "startDate": "2023-03-13",
    "userId": "user1"
}

$ http PUT :8082/rental/end/user1/1
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
content-length: 131

{
    "active": false,
    "endDate": "2023-03-13",
    "id": "640f540f00e031200c6230ec",
    "reservationId": 1,
    "startDate": "2023-03-13",
    "userId": "user1"
}

$ http :8082/rental
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
content-length: 133

[
    {
        "active": false,
        "endDate": "2023-03-13",
        "id": "640f540f00e031200c6230ec",
```

```

    "reservationId": 1,
    "startDate": "2023-03-13",
    "userId": "user1"
}
]

$ http :8082/rental/active
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
content-length: 2

```

[] #1

#1 The result is empty since we don't have any active rentals.

And we can also verify that the document is stored in the MongoDB directly with, for instance, the `mongosh` command line client (you can find the mapped port with `docker ps`):

TIP This verification with `mongosh` is again optional. If you don't have it installed, you don't have to install it now.

```

$ mongosh --port 33209 rental --eval "db.Rental.find({})"
[
  {
    _id: ObjectId("640f540f00e031200c6230ec"),
    active: false,
    endDate: ISODate("2023-03-13T00:00:00.000Z"),
    reservationId: Long("1"),
    startDate: ISODate("2023-03-13T00:00:00.000Z"),
    userId: 'user1'
  }
]

```

7.6.3 Adjusting Reservation service to the Rental service updates

MongoDB uses a different type of ID field instead of a numeric value. You might have noticed it in the outputs in the previous section. Now that we extend the `Rental` entity extends the `PanacheMongoEntity`, the ID type has changed to `ObjectId`, which we can mainly interpret as a `String` value that is returned now from our endpoints. Of course, this is not an issue in the Rental service because the `Rental` had the ID field previously inherited from the `PanacheEntity`, which was a `Long` value. We didn't interpret it manually anywhere, and the Jackson parser can adjust to this change automatically.

However, the problem with the parsing of the `Rental` ID fields arises in the Reservation service that calls the Rental service if the reservation is being made on the current day. We must adjust the Rental JSON parsing in the Reservation service to accommodate this change. Namely, we need to change the type of the `id` field in the `org.acme.reservation.rental.Rental` class to type `String` and adjust the constructor and getter accordingly. The [Listing 7.32](#) presents this change.

Listing 7.32 Adjustments in the Reservation service required after the Rental upgrade to MongoDB

```
public class Rental {

    private final String id;

    ...

    public Rental(String id, String userId, Long reservationId,
                  LocalDate startDate) {
        ...
    }

    public String getId() {
        return id;
    }

    ...
}
```

And now we can safely call Rental service from the Reservation service again.

7.7 Reactive data access

Reactive programming is a programming paradigm in which users write non-blocking, responsive, asynchronous code based on message passing. We will dive deep into the reactive in Chapter 8, but here we want to look into how you can utilize non-blocking database drivers provided by some databases to integrate the database utilization into your reactive pipelines.

Quarkus uses the Hibernate Reactive project to integrate with different databases supporting reactive drivers. By their definition, “Hibernate Reactive is intended for use in a reactive programming environment like Vert.x or Quarkus, where interaction with the database should occur in a non-blocking fashion. Persistence operations are orchestrated via the construction of a reactive stream rather than via direct invocation of synchronous functions in procedural Java code” (<https://hibernate.org/reactive/> 2022-08-02). The stream in Hibernate Reactive is represented as either a `CompletionStage` coming from JDK, or as a `Uni` (single result) or as a `Multi` (multiple results), both of which come from the project called SmallRye Mutiny.

TIP We will cover SmallRye Mutiny in detail in Chapter 8. Here we only need to remember that both `Uni` and `Multi` represent an asynchronously computed single or multiple results, respectively.

Traditional Hibernate ORM and JPA were designed with blocking IO operations. If you invoke a request to the database, the active thread waits for the response — it’s blocked. However, when you write the application reactively, blocking is not allowed, at least not on event loop threads. If your operation needs to block, it must delegate the blocking operation execution to the so-called worker thread, passing the execution back to the event loop thread once the blocking operation finishes. This is what Quarkus with traditional Hibernate ORM does. However, with Hibernate Reactive and reactive JDBC drivers, purely reactive APIs are used, and, in most cases, no blocking operations need to be performed when accessing the database.

You may ask why this is important. And that is an excellent question. The main idea is that reactive code makes your applications more responsive, which is the primary metric that must be scalable in modern applications. But we will cover this in detail in the following chapter. For now, it is only important to learn how to use reactive types with persistence.

Quarkus provides direct integration of Hibernate Reactive, which allows users to utilize the API provided by the library. Quarkus also provides a Panache integration layer on top of this integration to simplify the code users need to write. In this section, we focus on Panache with Hibernate Reactive.

7.7.1 Using Hibernate Reactive with Panache

As we already learned about Panache, it dramatically simplifies the code for data access for simple data sets. With Hibernate Reactive Panache, we still utilize a very similar model but this time with the reactive programming paradigm and reactive SQL clients. From the development point of view, this boils down to using the Panache-provided methods (either in the entity with the active record or in the repository). Instead of standard types like `E` or `List<E>`, which we had in the blocking version, we now receive results of types `Uni<E>` or `Uni<List<E>>` where `E` is our entity type. This only says that the result `E` in the `Uni<E>` represents an asynchronous computation that completes in the future.

In this chapter, we change the Reservation service to use Panache with Hibernate Reactive on top of the non-blocking PostgreSQL driver.

PANACHE HIBERNATE REACTIVE EXTENSIONS AND CONFIGURATION

To start, we need to change the extensions that we originally included in the Reservation service. We have to replace the blocking versions of Panache (`quarkus-hibernate-orm-panache`) and PostgreSQL JDBC driver (`quarkus-jdbc-postgresql`) with their reactive counterparts — `quarkus-hibernate-reactive-panache` and `quarkus-reactive-pg-client` like demoed in [Listing 7.33](#) with these commands executed in the `reservation-service` directory. Additionally, since we also utilized REST data in the Reservation service, we also need to exchange the `quarkus-hibernate-orm-rest-data-panache` for the reactive version in the `quarkus-hibernate-reactive-rest-data-panache` extension.

Listing 7.33 Changing blocking ORM extensions for their respective reactive versions in the Reservation service

```
$ quarkus ext remove quarkus-hibernate-orm-panache quarkus-jdbc-postgresql
→ quarkus-hibernate-orm-rest-data-panache
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✓ Extension io.quarkus:quarkus-jdbc-postgresql has been
→ uninstalled
[SUCCESS] ✓ Extension io.quarkus:quarkus-hibernate-orm-rest-data-panache
→ has been uninstalled
[SUCCESS] ✓ Extension io.quarkus:quarkus-hibernate-orm-panache has been
→ uninstalled

$ quarkus ext add quarkus-hibernate-reactive-panache
→ quarkus-reactive-pg-client quarkus-hibernate-reactive-rest-data-panache
[SUCCESS] ✓ Extension io.quarkus:quarkus-hibernate-reactive-panache has
→ been installed
[SUCCESS] ✓ Extension io.quarkus:quarkus-reactive-pg-client has been
→ installed
[SUCCESS] ✓ Extension
→ io.quarkus:quarkus-hibernate-reactive-rest-data-panache has been
→ installed
```

The only required change in the application's configuration is the connection URL string. Other properties remain the same. The configuration in the `application.properties` of the `Reservation` service should be changed as shown in the [Listing 7.34](#):

Listing 7.34 The Reservation service configuration for Hibernate Reactive

```
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = user
quarkus.datasource.password = pass
%prod.quarkus.datasource.reactive.url =
→ vertx-reactive:postgresql://localhost:5432/reservation #1

# drop and create the database at startup
quarkus.hibernate-orm.database.generation = drop-and-create
```

#1 The only required change from the blocking Panache version.

DEFINING ENTITIES WITH HIBERNATE REACTIVE

The code of our entities is the same as with the blocking versions of the extensions. However, the packages of the classes that our classes extend or implement changed as now they provide the reactive variants. We need to do a few drop-in replacements in files `Reservation.java` and `ReservationCrudResource.java` as shown in the [Listing 7.35](#). But everything else can stay the same.

Listing 7.35 Replacements of extensions classes with reactive variants

```
// Reservation.java

-import io.quarkus.hibernate.orm.panache.PanacheEntity;
+import io.quarkus.hibernate.reactive.panache.PanacheEntity;

// ReservationCrudResource.java

-import io.quarkus.hibernate.orm.rest.data.panache.PanacheEntityResource;
+import io.quarkus.hibernate.reactive.rest.data.panache.
-PanacheEntityResource;
```

7.7.2 Reservation service updates for Hibernate Reactive usage

Since we now use reactive types in the Reservation service, we need to adjust our code that communicates with the database to be reactive too. Let's start with the `ReservationResource` class, which contains most of the required changes. Don't worry if you don't understand everything. We will apply our knowledge with a deeper analysis of reactive programming in Chapter 8. [Listing 7.36](#) demonstrates these changes.

Listing 7.36 Changes of JAX-RS methods of ReservationResource

```
// ReservationResource.java

...

@Consumes(MediaType.APPLICATION_JSON)
@POST
@WithTransaction #1
public Uni<Reservation> make(Reservation reservation) {
    reservation.userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : "anonymous";
    return reservation.<Reservation>persist().onItem()
        .call(persistedReservation -> {
            Log.info("Successfully reserved reservation "
                + persistedReservation);
```

```

        if (persistedReservation.startDay.equals(LocalDate.now()) &&
            persistedReservation.userId != null) {
            Rental rental = rentalClient
                .start(persistedReservation.userId,
                       persistedReservation.id);
            Log.info("Successfully started rental " + rental);
        }
        return Uni.createFrom().item(persistedReservation);
   }); #2
}

@GET
@Path("all")
public Uni<List<Reservation>> allReservations() { #3
    String userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : null;
    return Reservation.<Reservation>listAll()
        .onItem().transform(reservations -> reservations.stream()
            .filter(reservation -> userId == null ||
                    userId.equals(reservation.userId))
            .collect(Collectors.toList()));
}

@GET
@Path("availability")
public Uni<Collection<Car>> availability(...) { #4
    ...
    // get all current reservations
    return Reservation.<Reservation>listAll()
        .onItem().transform(reservations -> { #5
            // for each reservation, remove the car from the map
            for (Reservation reservation : reservations) {
                if (reservation.isReserved(startDate, endDate)) {
                    carsById.remove(reservation.carId);
                }
            }
            return carsById.values();
        });
}

```

#1 With Hibernate Reactive, we need to use a custom `@WithTransaction` annotation to mark the transaction. The method also needs to return a `Uni`.

#2 This is the correct way of handling reactive types, where we provide a declarative definition of what should happen when the `async` result of the `Uni` (computed from the `persist()` invocation) is received.

#3 The return value now makes the execution of this JAX-RS method reactive (the HTTP request suspends until the Uni produces a value; more details in Chapter 8). We also change the return value to List since Mutiny doesn't have an override for Collection collect.

#4 We now return a Uni<Collection<Car>> meaning asynchronously computed car collection.

#5 We pull the list of reservations from the database so we can directly transform it together with the filtering to the collection of available cars.

The ReservationPersistenceTest won't compile since it directly references the Reservation entity. We need a more sophisticated rewrite since we need to handle transactions in the reactive code. Quarkus provides a simple helper APIs that are contained in the quarkus-test-vertx extension. Add this extension to the pom.xml in test scope:

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

Now we can change the ReservationPersistenceTest as demonstrated in the [Listing 7.37](#).

Listing 7.37 Changes for reactive invocations in the ReservationPersistenceTest

```

@Test
@RunOnVertxContext
public void testCreateReservation(
    UniAssertor uniAssertor) {
    final UniAssertor assertor = new UniAssertorInterceptor(uniAssertor) {
        @Override
        protected <T> Supplier<Uni<T>> transformUni(
            Supplier<Uni<T>> uniSupplier) {
            return () -> Panache.withTransaction(uniSupplier);
        }
    };

    Reservation reservation = new Reservation();
    reservation.startDay = LocalDate.now().plusDays(5);
    reservation.endDay = LocalDate.now().plusDays(12);
    reservation.carId = 384L;

    assertor.<Reservation>assertThat(() -> reservation.persist(),
        r -> {
            Assertions.assertNotNull(r.id);
            assertor.putData("reservation.id", r.id);
        });

    assertor.assertEquals(() -> Reservation.count(), 1L);
    assertor.assertThat(() -> Reservation.<Reservation>findById(
        assertor.getData("reservation.id")),
        persistedReservation -> {
            Assertions.assertNotNull(persistedReservation);
            Assertions.assertEquals(reservation.carId,
                persistedReservation.carId);
        });
}
}

```

We have to utilize the `@RunOnVertxContext` annotation is needed for Quarkus to run the test method on the Vert.x (event loop) thread because the main thread wouldn't work with reactive code. This annotation also allows us to use the `UniAssertor` that provides an API for assertions on `Uni` results. `UniAssertor` provides similar API as JUnit `Assertions` so the use is pretty similar to normal JUnit usage. This is the recommended way of testing Hibernate Reactive.

Since the main processing is non-blocking, even the outgoing Rental service HTTP call can no longer utilize blocking APIs. If you invoke `POST /reservation` with the current day now, the request fails with the error message stating this (`BlockingNotAllowedException`). We need to update the return type of the `RentalClient#start` method, as showed in [Listing 7.38](#).

Listing 7.38 Return type change to non-blocking Uni in RentalClient

```
@RegisterRestClient(baseUri = "http://localhost:8082")
@Path("/rental")
public interface RentalClient {

    @POST
    @Path("/start/{userId}/{reservationId}")
    Uni<Rental> start(@RestPath String userId,
                      @RestPath Long reservationId);
}
```

This, of course, needs the adjustment in the `ReservationResource#make` method to consume now provided `Uni` return type. This is demonstrated in the [Listing 7.8](#).

Listing 7.39 The ReservationResource#make method update required to consume Uni return type from the RentalClient

```

@Consumes(MediaType.APPLICATION_JSON)
@POST
@ReactiveTransactional
public Uni<Reservation> make(Reservation reservation) {
    reservation.userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : "anonymous";
    return reservation.<Reservation>persist().onItem()
        .call(persistedReservation -> {
            Log.info("Successfully reserved reservation "
                + persistedReservation);
            if (persistedReservation.startDay.equals(LocalDate.now())) {
                return rentalClient.start(persistedReservation.userId,
                    persistedReservation.id)
                    .onItem().invoke(rental ->
                        Log.info("Successfully started rental " + rental))
                    .replaceWith(persistedReservation); #1
            }
            return Uni.createFrom().item(persistedReservation);
        });
}

```

#1 We chain the now reactive invocation of the remote HTTP request to the Rental service with the replacement of the returned value for the persisted reservation as the expected return value of the make method.

The Reservation service now works again as expected. But now, with Hibernate Reactive. Feel free to experiment with the available APIs as we did in [Listing 7.9](#). If you want a small reminder, you can find all available endpoints at <http://localhost:8081/q/swagger-ui/> when the Reservation service runs in Dev mode.

7.8 Wrap up and next steps

In this chapter, we learned how to store application data in a data store with Quarkus. We introduced the new Panache framework built on top of the well-known Hibernate ORM framework to provide simplified API for ORM development.

We differentiated two patterns that users can utilize with Panache—the active record pattern and the repository pattern. We demonstrated the use of both these patterns with relational databases (PostgreSQL and MySQL) and one NoSQL database (MongoDB). Additionally, we also introduced a new REST data framework that significantly simplifies the development of CRUD JAX-RS resources on top of Panache entities.

We also illustrated the use of Panache with Hibernate Reactive — a framework that can utilize the non-blocking reactive drivers provided by some databases. We also touched on the reactive programming model. Since Quarkus is built around reactive principles, we will learn how Quarkus applications can utilize reactive development in detail in the next chapter.

7.9 Summary

- Panache is a framework providing simplified ORM in Quarkus. It frees the developer from most of the boilerplate code that needs to be written with current data access libraries.
- Panache can be utilized with either the active record or the repository pattern, where both have the same level of expressiveness, and with Panache it is simple to switch between them.
- REST Data is a further simplification for developing JAX-RS CRUD services. It generates the CRUD resources for the specified Panache entities that can be adjusted for user requirements.
- Quarkus can utilize the reactive drivers provided by the database through the Hibernate Reactive library. The application code then uses the reactive, non-blocking, and asynchronous APIs, which makes the applications more scalable and responsive.

8

Reactive programming

This chapter covers

- Learning what reactive programming is
- Identifying why we want to write reactive code
- Explaining SmallRye Mutiny as the reactive library used in Quarkus
- Analyzing how the reactive paradigm integrates into the Quarkus architecture

8.1 Being Reactive

The reactive programming paradigm is an alternative way of writing software programs. In contrast to the imperative model, which executes the program as a sequence of ordered steps, reactive programming focuses on asynchronous executions that notify the caller about the result only when the result computes or an error occurs. In this way, the caller thread is free to do any other work in the meantime and return to the request processing only when there is progress. This means that reactive systems can be more responsive and scalable as the same resources (both software and hardware) can do more work in the same time window.

This benefit stems mainly from the fact that threads are not blocked while waiting for an I/O operation (a remote service or database call) to finish. Instead of waiting and doing nothing useful, they can be used to process other things. Therefore, your application needs fewer threads to handle the same number of work.

Quarkus is built on top of a reactive engine from the ground up, meaning writing reactive programs is straightforward. However, if you prefer the imperative model, you can still write applications the same way you are used to with imperative code, but you will lose some of the scalability benefits.

Reactive programming is generally considered to be more complex than imperative. But if you are willing to write reactive code, we explain the transition from the imperative model and the benefits that reactive programming brings to your applications in this chapter. Additionally, Quarkus also allows you to mix both paradigms if you want. Quarkus actually enables you to do this even in the same class, which can significantly help with gradual transition to a reactive programming model.

8.1.1 Handling of I/O operations

The first question you might ask is why reactive programming might be useful? I/O operations are a great example to demonstrate the distinction between imperative and reactive execution models. Modern applications require I/O, whether to access the database/filesystem, call another service, or process messages from a broker. These operations generally take a long time to complete. In the typical sequential imperative model, these calls represent blocking operations. Each incoming request is assigned to a worker thread from a predefined thread pool that handles it from start to finish, as shown in [Figure 8.1](#).

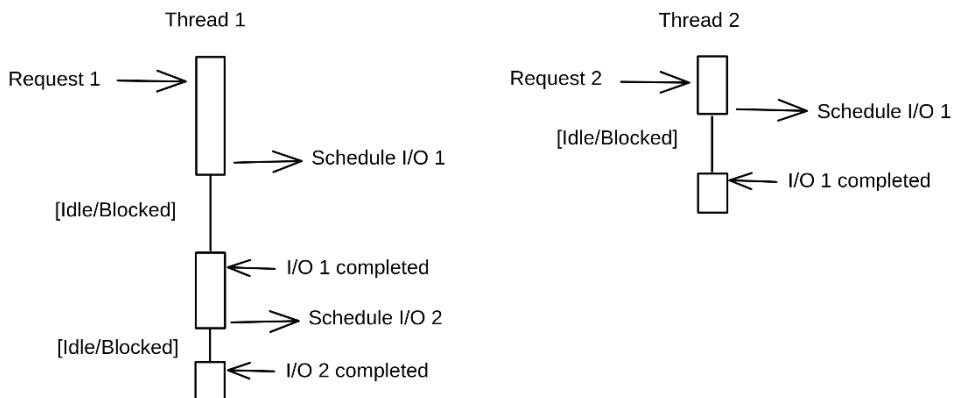


Figure 8.1 Blocking I/O request handling

If the request requires I/O operation, the whole thread is blocked. No other work can be done until the I/O operation completes even if the I/O operation itself delegates to a separate resource (e.g., network device, file system, database). The thread is just idly waiting for the operation to finish. With the move of application workloads to cloud, having a bigger number of threads to handle concurrent tasks also means higher memory and CPU cycles consumption which directly translates to cloud bills.

The reactive paradigm allows Quarkus to streamline this processing into a non-blocking I/O based on an asynchronous event-driven network application framework called Netty (<https://netty.io/>). By using only a few eventloop (or I/O) threads that handle concurrent I/O operations, there is no longer a need to delegate each request to a new thread. Instead, I/O interleaves in the same eventloop thread as shown in [Figure 8.2](#).

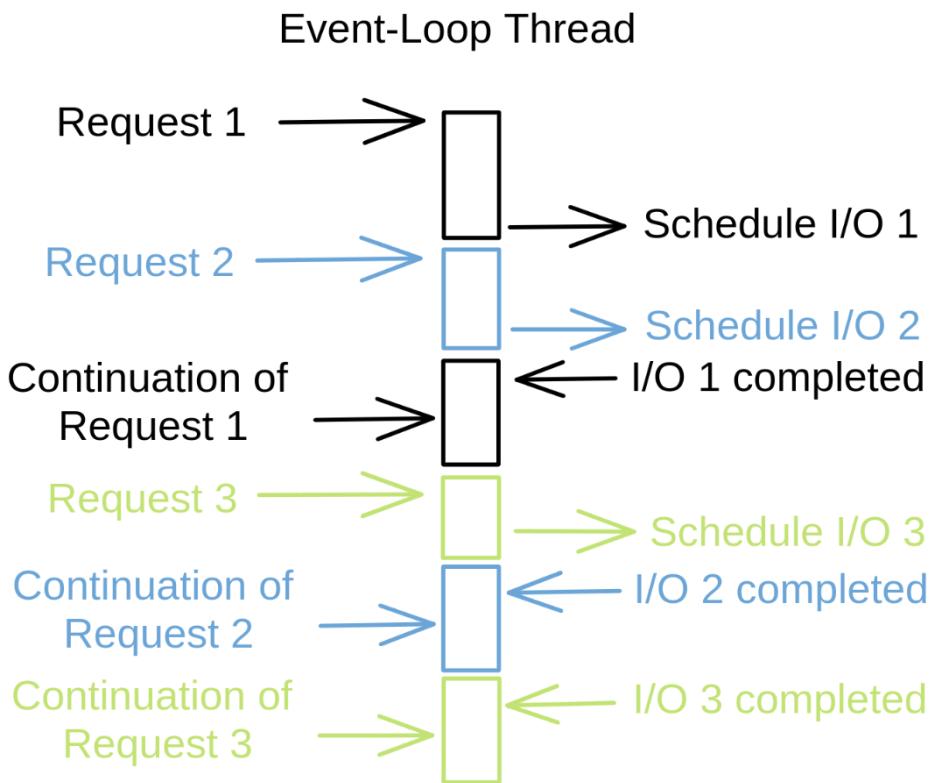


Figure 8.2 Non-blocking I/O request handling

Consequently, users must refrain from blocking these eventloop threads for this to function correctly. If you block the eventloop thread for the configured time (2 seconds by default), Quarkus will stop the execution of the operation.

Because each scheduled I/O operation doesn't block the eventloop thread, it might be utilized for other work while the operation progresses. It effectively executes operations concurrently in the same thread. If any work needs to block, it is outsourced to a thread from the worker thread pool.

Quarkus automatically deduces whether the method executes on the eventloop or the worker thread depending on the signature of the method. If it's reactive (returning reactive, nonblocking types), it chooses eventloop thread with the expectation that the method doesn't block it. Conversely, other return types mean that the method is expected to block, so it executes on the worker thread.

Quarkus also provides users with the `@Blocking` and `@Nonblocking` annotations which, if placed on the method, gives Quarkus a hint that the operation needs to execute on the worker (blocking) thread or on the eventloop (nonblocking) thread, respectively. For example, if you need to perform a blocking computation in your reactive pipeline, you can simply put the `@Blocking` annotation on the method that encapsulates this computation.

8.1.2 Writing reactive code

Most developers write their code in the imperative paradigm. When the program starts, the runtime invokes a top-level entrypoint method (e.g., the `main` method Java) that begins executing declared statements step by step. Conversely, the reactive programming model expresses executions in the form of *continuations*. A continuation defines the code executed when the operation completes. The continuation handles either the successful case, then we have a result available, or it can complete exceptionally, and we can handle the received exception. [Figure 8.3](#) shows the visualization of the difference in executions.

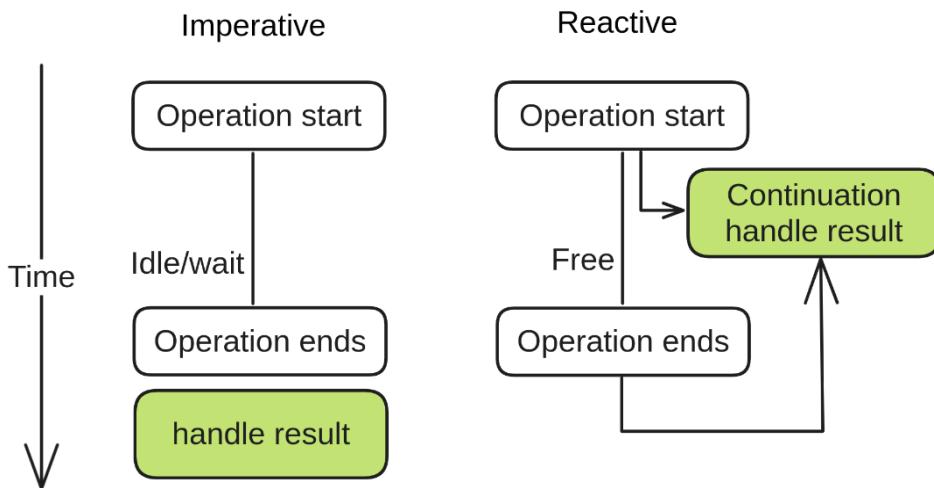


Figure 8.3 Comparison of the imperative and reactive code handling

Continuations typically define the same result handling code as in the imperative model. The difference is in when and how it is defined. While with step-by-step instructions, it is clear that operations handling the result start only when the operation ends, in this case, the processing blocks until the operation ends. Otherwise, we couldn't execute the next operation in the sequence which requires the result as argument. With reactive programming, we pass the continuation to the operation itself. When the operation ends, continuation code executes with the result. In other words, our code reacts to the event generated by the end of the operation. The operation invocation is fully asynchronous, and we are free to do any other work in the meantime.

This might sound a lot like an event-driven architecture. And it actually is. However, at a bigger scope, since modern applications need to handle an enormous number of events. These events frequently come from an external source—for instance, user clicks, message retrievals, database updates, or new service version rollout. However, they also include internal concepts like asynchronous operations, I/O (input/output) requests, or the definitions and executions of reactive pipelines (linking of continuations). Reactive programming thus requires a different way of thinking about the execution flow in the application. In turn, it provides a better scaling and more resilient model that performs better under altering loads.

8.2 Mutiny

Quarkus uses SmallRye Mutiny as its reactive API exposed to the users. Mutiny is a reactive library that provides a set of APIs for non-blocking, event-driven, and asynchronous applications. It implements the Reactive Streams specification, so it comes with integrated backpressure (a mechanism that allows consumers to signal producers to slow down the production of items). With Mutiny, user code reacts to events emitted when the result is computed or an error occurs.

TIP Reactive programming in Quarkus is also usable with Kotlin coroutines, which is outside the scope of this book.

8.2.1 Reactive streams

The Reactive Streams specification (<https://www.reactive-streams.org/>) provides an API for the non-blocking asynchronous stream executions with integrated backpressure. Since JDK 9, it is also integrated directly into the JDK as the `java.util.concurrent.Flow` class.

This specification consists of four interfaces:

- Publisher responsible for the publishing of items.
- Subscriber responsible for the consumption of items.
- Subscription represents the link between exactly one Publisher and Subscriber.

- Processor acts as both Publisher and Subscriber.

The demonstration of the message exchange between the Publisher and Subscriber is provided in [Figure 8.4](#).

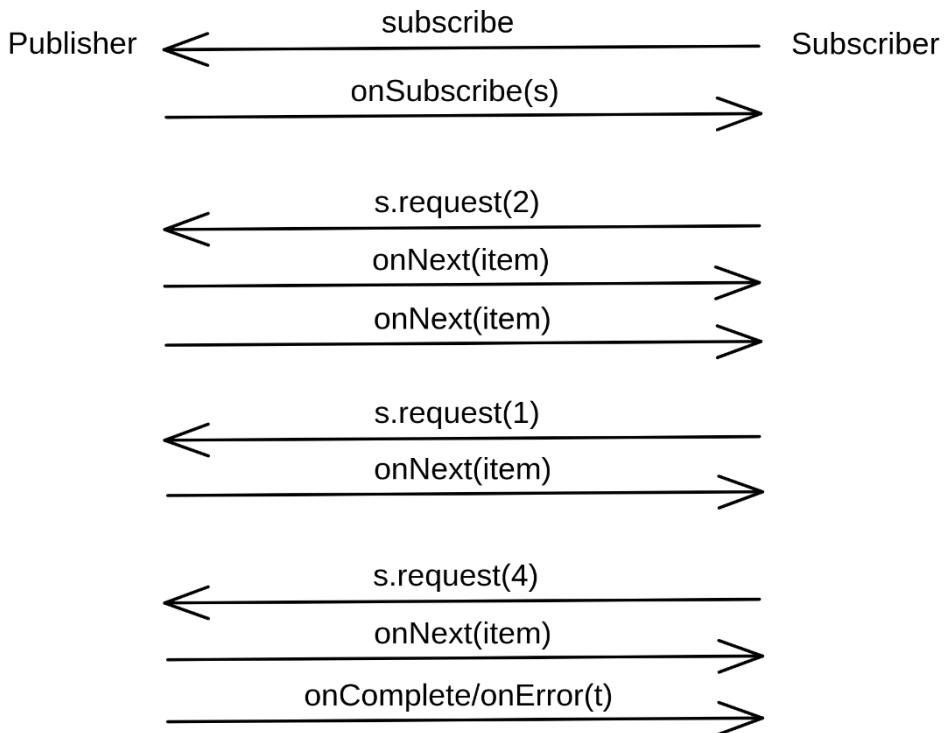


Figure 8.4 The example interaction between Reactive Streams Publisher and Subscriber

In this figure, you can see that the stream processing:

1. The processing initiates with a call to the `subscribe` method of the Publisher, which passes the instance of the Subscriber.
2. In turn, the Publisher calls the `onSubscribe(s)` method of the Subscriber passing in the created `Subscription` object (`s`) representing the link between them.
3. The Subscriber instance then uses the subscription object to request the number of data items it can consume (the `request(n)` method), representing the built-in backpressure. If the Subscriber can't catch up with published items fast enough, it can lower the number of requested items to tell the Publisher to slow down.
4. After the `request(n)` call, the Publisher calls the `onNext(item)` callback on the Subscriber `n` times.

5. This continues until the `Publisher` has no more items to produce or an error occurs, meaning that either `onComplete` or `onError` callbacks are called on the `Subscriber`, respectively.

Even if these interfaces might seem easy to implement at first sight, it is generally not encouraged to implement them directly in your application as the specification with the associated TCK (technology compatibility kit/test suite) is quite complex. Mutiny types implement Reactive Streams APIs (since version 2.0 they implement the JDK Flow variants) so you don't have to.

8.2.2 Mutiny API

Mutiny provides two main types that represent asynchronously computed results: `Uni` and `Multi`:

- `Uni` represents a single result or failure.
- `Multi` represents a stream of produced results that can be infinite.

As we already learned, both `Uni` and `Multi` receive events when the result computes. One important distinction is that since `Uni` can produce precisely one item or failure. It doesn't represent the full publisher as defined in the Reactive Streams specification. The call to the `subscribe` method is enough to express the initiative to get the result, so it directly triggers the computation. Additionally, it can also handle `null` values, which the Reactive Streams specification prohibits.

Mutiny was built with simplicity in mind. Reactive programming can be complex, so having a simple API is essential. Mutiny provides only the mentioned types, `Uni` and `Multi`. Users utilize these types to define reactive pipelines. The pipeline represents a stage in the reactive stream. In this way, the events (including items) flow through the pipeline, and each stage can change these events, filter/drop some of them, or fire new ones. It is also important to remember that the pipeline only starts once a subscription/subscriber requires it to start (by subscribing to it).

Users define the individual stages of the reactive pipelines with a provided fluent API in the `Uni` and `Multi` methods. The idea is to use autocompletion in modern IDEs and JavaDoc to determine your needs in a particular stage quickly. The [Listing 8.1](#) analyzes an example of this API.

Listing 8.1 An example use of the Mutiny API

```
Multi.createFrom().items("a", "b", "c")
    .onItem() #1
        .transform(String::toUpperCase)
    .onItem() #2
        .invoke(s -> System.out.println("Intermediate stage " + s))
    .onItem() #3
        .transform(s -> s + " item")
    .filter(s -> !s.startsWith("B")) #4
    .onCompletion() #5
        .invoke(() -> System.out.println("Stream completed"))
    .subscribe() #6
        .with(s -> System.out.println("Subscriber received " + s));
```

#1 Stage transforming the received items to upper case.

#2 Stage that prints the items to the standard output and passes it downstream.

#3 Another transformation, this time appending a new string.

#4 Filter operation that propagates items only if they don't start with B.

#5 Definition of the completion callback.

#6 The subscription with a simple subscriber is required for the Multi to start producing items.

To better imagine how individual stages flow one to another, the visual representation of the pipeline is available in [Figure 8.5](#). You may notice the similarity to the Streams flow available in listing [Figure 8.4](#).

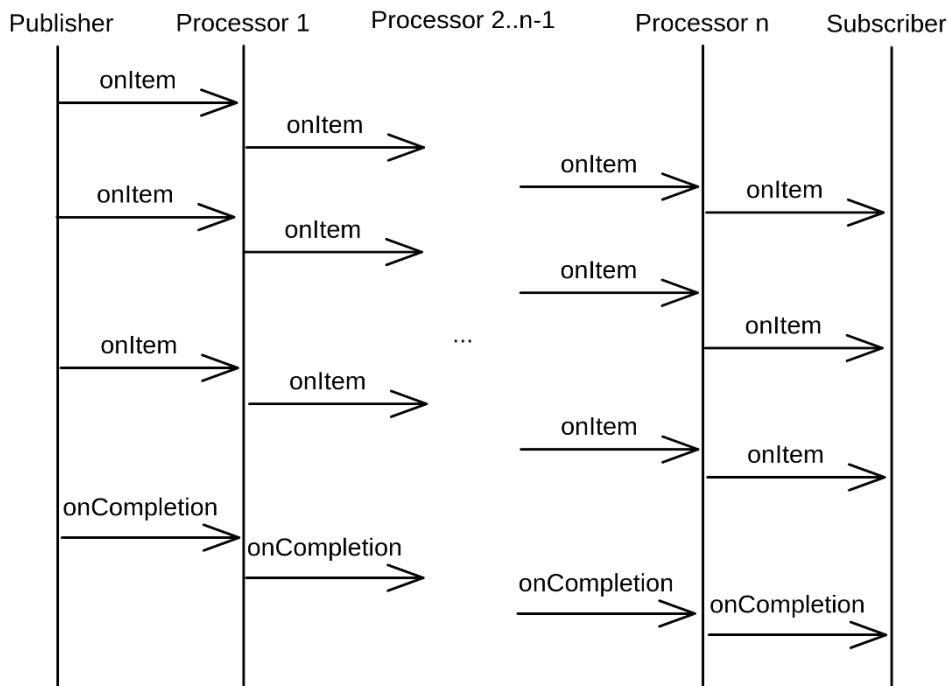


Figure 8.5 The reactive pipeline visualization in Mutiny

Executing this pipeline produces the following result available in [Listing 8.2](#) where we can differentiate individual items and other events as they flow through the defined pipeline.

Listing 8.2 The result of the Mutiny example code execution

```

Intermediate stage A
Subscriber received A item
Intermediate stage B
Intermediate stage C
Subscriber received C item
Stream completed
  
```

On the first line, we can see the first intermediate stage hit with the initially produced item "a". The next stage transforms it to "A item" string and since it doesn't start with "B", it is not filtered out. The subscriber thus receives "A item" and prints it on the second line. The intermediate stage for item "b" is filtered later downstream (the subscriber will never receive it) is listed by line three. Finally, the case for the produced item "c" is the same as for "a". Completion event fires when the `Multi` has no more items to produce (last line).

8.2.3 Using Mutiny in user applications

Asynchronous computations are an essential concept in applications. Because the code utilizes continuations, Mutiny is asynchronous per definition. Consider the following example. Let's consider the `ReservationResource#availability` method from the Reservation service as the only method of this class that is still blocking. In this method, we make a call to the Inventory service to obtain all cars which we can filter later. Later it lists all available reservation in order to check which cars are available for the selected dates. The excerpt of this method is provided in [Listing 8.3](#).

Listing 8.3 Part of the code of `ReservationResource#availability` calling Inventory service and Rental service

```
// public Collection<Car> availability(...) {
// obtain all cars from inventory
List<Car> availableCars = inventoryClient.allCars();

...
// get all current reservations
return Reservation.<Reservation>listAll()
...
```

As you might expect, doing these operations sequentially might take a lot of time in case the number of cars or reservation increases. We first need to fetch all cars from the Inventory service and only after this operation completes, we can start pulling all reservations from the database. However, there is no reason why these two operation could not execute in parallel. So how could we do these calls asynchronously with Mutiny?

The `Reservation.listAll()` method is already in fact reactive (it returns `Uni<List<Reservation>>`), so there is nothing to do. The `InventoryClient` requires a little bit more work since we need to change the return type of the `allCars()` method to now return asynchronous Mutiny's `Uni<List<Car>>`. [Listing 8.4](#) lists all the required changes that we need to make `allCars()` method reactive.

Listing 8.4 Changes required to make InventoryService#allCars method reactive

```
// InventoryClient.java
Uni<List<Car>> allCars();

// GraphQLInventoryClient.java
Uni<List<Car>> allCars();

// MockInventoryClient.java
@Override
public Uni<List<Car>> allCars() {
    Car peugeot = new Car(1L, "ABC 123", "Peugeot", "406");
    return Uni.createFrom().item(List.of(peugeot));
}

// ReservationResourceTest.java
Mockito.when(mock.allCars())
    .thenReturn(Uni.createFrom()
        .item(List.of(peugeot)));
```

The `ReservationResource#availability` can now combine the `Unis` produced by `Reservation.listAll()` and `inventoryService.allCars()` methods as demonstrated in listing [Listing 8.5](#).

Listing 8.5 Making ReservationResource#availability reactive with Mutiny

```

@GET
@Path("availability")
public Uni<Collection<Car>> availability(
    @RestQuery LocalDate startDate,
    @RestQuery LocalDate endDate) {
    Uni<Map<Long, Car>> carsUni = inventoryClient.allCars()
        .map(cars -> cars.stream().collect(Collectors
            .toMap(car -> car.id, Function.identity()))); #1

    Uni<List<Reservation>> reservationsUni = Reservation.listAll();

    return Uni.combine().all()
        .unis(carsUni, reservationsUni) #2
        .asTuple()
        .chain(tuple -> {
            Map<Long, Car> carsById = tuple.getItem1(); #3
            List<Reservation> reservations = tuple.getItem2();

            // for each reservation, remove the car from the map
            for (Reservation reservation : reservations) {
                if (reservation.isReserved(startDate, endDate)) {
                    carsById.remove(reservation.carId);
                }
            }
            return Uni.createFrom().item(carsById.values());
        });
}

```

#1 The asynchronously produced list of cars is mapped to map instance as before.

#2 Combine both unis into a tuple.

#3 After both Uni instances compute values, they can be retrieved from the tuple.

The `availability` method now defines three `Uni` instances. The first one fetches cars list mapped into a map, the second lists all reservations, and the third one combines them. The actual pulling of data from the Inventory service and the database executes at the same time and it happens only after someone subscribes to the produced combined `Uni` after it is returned from this method. Quarkus automatically subscribes to the `Uni` when it is returned from the JAX-RS method. The `availability` method is now reactive and you might verify its functionality with a call to `GET /reservation/availability`.

8.3 Reactive engine

From the beginning, reactive programming was essential to the Quarkus design. Reactive architecture is one of the core principles of Quarkus. Quarkus runs on top of a reactive core created from components like Eclipse Vert.x (<https://vertx.io/>) and Netty (<https://netty.io/>) to provide a set of features (e.g., non-blocking I/O, eventloop threads, or asynchronous APIs like Mutiny) allowing users to create reactive systems.

However, reactive architecture doesn't require users to write reactive applications. Reactive programming is an optional feature that Quarkus applications are free to utilize but are not required to. It is also possible to mix both blocking and reactive code inside the same application. This is possible because moving from already reactive code to blocking is possible (as you might remember with, for instance, `.await().indefinitely()`). Conversely, it's not so easy.

Many Quarkus extensions support reactive programming. The extension description typically mentions if it supports reactive APIs (generally encapsulated in Mutiny integration). However, reactive pipelines can be defined and used even without direct integration with other extensions, as we've seen in the example before ([Listing 8.4](#)).

8.4 Making Car Rental reactive

The reactive nature of Quarkus already took our path in Car Rental microservices in the reactive direction. In this section, we analyze how we have already utilized the reactive APIs and learn how we can possibly extend their use.

8.4.1 Reservation service

REACTIVE PERSISTENCE

The Reservation service is an excellent example of a reactive service. As we wanted to learn about Hibernate Reactive in Chapter 7, we included the `quarkus-hibernate-reactive-panache` extension to utilize the reactive, non-blocking SQL database drivers. This extension allows mapping the database operations to the reactive streams directly used to map the values returned from the database into the `Uni` or `Multi` types. [Listing 8.6](#) provides an example of such execution. The `persist()` database operation returns a `Uni`, which (when processed) is then passed downstream as a return value of the JAX-RS method.

Listing 8.6 Non-blocking version of ReservationResource#make method

```

@Consumes(MediaType.APPLICATION_JSON)
@POST
@WithTransaction
public Uni<Reservation> make(Reservation reservation) {
    ...
    return reservation.<Reservation>persist().onItem() #1
        .call(persistedReservation -> {
            ...
        });
}

```

#1 The persist operation that emits an event item when the reservation is persisted.

The item emitted by this `Uni` represents the persisted reservation received from the database. We continue the stream execution only when the database operation completes. But we are not blocked waiting for the database to finish the insert operation. The `make` method is executed on an I/O thread, and then the thread is released to be used by something else while the remote database is processing the persisting. The `onItem` callback is called after the database operation completes, and can be potentially executed on a different I/O thread than the one that initially started processing the `make` method.

With Mutiny, it's straightforward to define any operation that should be performed when the database fetches the entity data. For instance, if you want to change the entity before we return it from the JAX-RS response, we could chain the `onItem()` calls similarly as demonstrated in [Listing 8.1](#).

REACTIVE REST

Since we returned the entities from the database in the JAX-RS methods directly, we also needed to change the signatures of these methods to return Mutiny types. You might remember that another option is to block the created `Uni` results, but that is not reactive. When Quarkus sees the `Uni` as the return type of the JAX-RS method, it automatically runs the method on the I/O thread. In this way, Quarkus is free to serve other requests in the meantime because the method at hand is no longer blocking and will resume the execution of the current request only when the result is available. This follows the same execution model described in section [8.1.1](#).

As we have learned, Quarkus decides whether the method should run on the non-blocking event Oloop (I/O) thread or the (possibly blocking) worker thread depending on the method's return value. If the method returns the `Uni`, we cannot block it. If you try to do so, Quarkus logs the following warning:

```
2023-05-14 15:50:11,264 WARN [io.ver.cor.imp.BlockedThreadChecker]
  - (vertx-blocked-thread-checker) Thread Thread[vert.x-eventloop-thread-3,
  -5,main] has been blocked for 2054 ms, time limit is 2000 ms: io.vertx
  -.core.VertxException: Thread blocked
  ...
  ...
```

Users can change the thread that executes the operation by annotating the method with the `@Blocking` annotation that tells Quarkus to run the method on the worker (executor) thread instead. The `@NonBlocking` annotation also does the opposite thing for the blocking method (not returning Mutiny types).

You can also use `Multi` in the JAX-RS methods to return a stream of responses. This is particularly useful when streaming data continuously (e.g., logs or binary data) as the SSE (Server Sent Events) media type.

8.4.2 Inventory service

In the Inventory service, we utilized Mutiny and its reactive APIs in the gRPC integration from Chapter 4. The gRPC extension generates stubs for our services that already provide Mutiny return types in their method signatures. It doesn't mean they must be used (the extension also generates a blocking service API). However, it is recommended to develop gRPC services with reactive types.

We learned in Chapter 4 that the gRPC protocol has an integrated notion of a stream which we use in the `inventory.proto` file like in [Listing 8.7](#).

Listing 8.7 The excerpt from the inventory.proto file showing the stream of requests

```
service InventoryService {
  rpc add(stream InsertCarRequest) returns (stream CarResponse) {}
  rpc remove(RemoveCarRequest) returns (CarResponse) {}
}
```

The gRPC stream naturally maps into the use of the Mutiny type `Multi` as it represents a potentially infinite number of events that come in the form of the `InsertCarRequest` or `CarResponse` items, respectively. In the Java implementation, we define the operations executed when a new request comes in like demonstrated in [Listing 8.8](#).

Listing 8.8 The GrpcInventoryService#add method

```

@Override
@Blocking
public Multi<CarResponse> add(
    Multi<InsertCarRequest> requests) { #1
    return requests
        .map(request -> { #2
            Car car = new Car();
            car.setLicensePlateNumber(request.getLicensePlateNumber());
            car.setManufacturer(request.getManufacturer());
            car.setModel(request.getModel());
            return car;
        }).onItem().invoke(car -> {
            Log.info("Persisting " + car);
            QuarkusTransaction.run( () ->
                carRepository.persist(car) #3
            );
        }).map(car -> CarResponse.newBuilder() #4
            .setLicensePlateNumber(car.getLicensePlateNumber())
            .setManufacturer(car.getManufacturer())
            .setModel(car.getModel())
            .setId(car.getId())
            .build());
}

```

#1 Receive a Multi to which we (or Quarkus) can subscribe.

#2 Map the received gRPC class into the model class Car and pass it downstream.

#3 Persist the mapped Car instance into the database.

#4 Map result into an instance of CarResponse, which is returned to stream passed back to the caller.

In fact, if you take a closer look at the methods in this class, you might notice that they are not in really reactive. Notice the `@Blocking` annotations that we need to use to tell Quarkus that even if the methods return reactive types, we cannot execute them on the eventloop thread because they run database I/O operations (the insert/delete of the car). It is a little strange to not use the Hibernate Reactive here as it would naturally fit into the reactive pipeline. But if we want to demonstrate as many functionalities that Quarkus provides as possible, we need to make compromises.

8.5 Virtual threads with project Loom

Reactive programming is generally considered harder to write, maintain and debug than the imperative programming model that we all learn first. However, the performance of the reactive model (with event loops) outperforms the imperative programming style in systems with high levels of concurrency and lots of I/O operations.

This is one of the main reasons why the project *Loom* was created. Project Loom, also known as virtual threads, was introduced as a preview feature in JDK 19, and it became generally available in JDK 21. Virtual thread is a thread managed by the JDK that is mapped to the real platform thread called carrier thread when it runs. Once a blocking operation starts, JDK saves the state of the virtual thread and removes its execution from the carrier thread letting any other available virtual thread to start executing on that carrier thread. Once the blocking operation completes the original virtual thread is again eligible to continue its run on carrier thread (not necessarily the same carrier thread). Virtual threads are more lightweight on resource consumption than platform threads (about 1000x faster to start and about 1000x less memory). This allows potentially huge amounts (millions) of virtual threads to be created, whereas a typical system can only handle several thousand regular threads due to the cost they put on the operating system.

Virtual thread model provides similar concept as event loop. However, it shades users from using continuations and allows them to write regular imperative code, but without adding the burden of actually blocking (expensive) platform threads during runtime. Visually the execution on virtual threads looks like presented in [Figure 8.6](#).

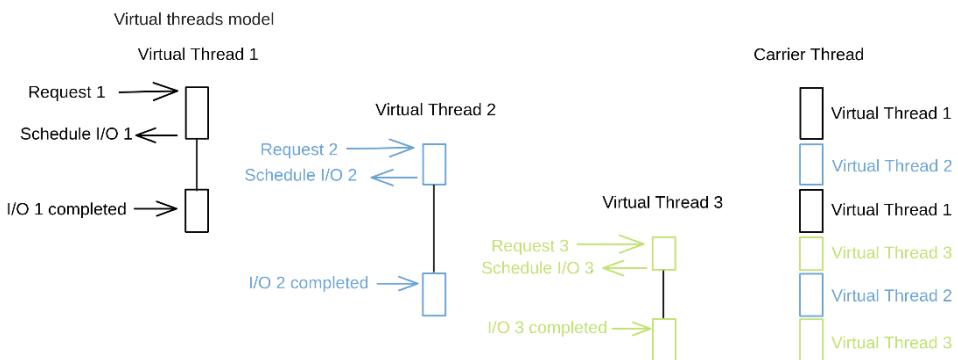


Figure 8.6 Virtual thread execution model with I/O request handling

Virtual threads do everything in the background so the users imperative code doesn't need to change. The platform (JDK) detects the point where you are about to block, and it yields the virtual thread from the carrier thread which allows it to be used by different virtual thread in the meantime. The performance is then comparable with that of the reactive programming model since the thread utilization is similar.

8.5.1 Project Loom in Quarkus

In Quarkus, you might run any method on a virtual thread by simply annotating it with `@RunOnVirtualThread` annotation. You can also place this annotation on a whole class to make all its methods run on virtual threads.

```
@GET
@Path("/virtualThread")
@RunOnVirtualThread
public void virtualThread() {
    Log.info("Running on " + Thread.currentThread().getName());
}
```

And calling this method we can see that we are indeed running it on a new virtual thread every time we call it:

```
... (quarkus-virtual-thread-0) Running on quarkus-virtual-thread-0
... (quarkus-virtual-thread-1) Running on quarkus-virtual-thread-1
... (quarkus-virtual-thread-2) Running on quarkus-virtual-thread-2
```

TIP If you continued running this over and over, you would notice that the ID of the used virtual thread keeps increasing forever and never repeats. This is because virtual threads, unlike platform threads, are not reused. It doesn't make a lot of sense to create pools of reusable virtual threads, simply because they are so cheap to create, start, and destroy (collect). The best practice is to create a new virtual thread for every task where you need it. When the virtual thread finishes its work, it is simply reclaimed by the garbage collector.

You might be probably asking now why we need to annotate every class we want to run on virtual thread manually. Why there isn't a global flag that would run everything on virtual thread? And those are surely good questions.

8.5.2 Problems with project Loom

Nothing in life is free and virtual threads are not an exception. They hide a lot of complexity behind the curtain, so it comes with a few possible problems.

PINNING OF THE CARRIER THREAD

Pinning happens when the virtual thread cannot be unmounted from the carrier thread when it executes a blocking operation. This essentially blocks the carrier thread too meaning it cannot execute other virtual threads. This happens in two cases:

- Virtual thread holds a monitor lock (`synchronized` blocks). This is actually a limitation of the current JDK implementations and might be resolved in the future, but at least in JDK 21, `synchronized` blocks cause thread pinning.
- Virtual thread has a native call (through the Java Native Interface) in its stack.

Without going into too much detail, don't worry if you don't fully understand these concepts. Usually they are not the concern of the end developers. However, they are often utilized in the underlying libraries that your code uses which means that you might not be able to predict when the pinning can happen.

OVERUSE OF VIRTUAL THREADS FOR COMPUTATIONS

The virtual threads scheduler cannot interrupt a running thread (unless it yields by starting an I/O operation). If you execute long-running CPU-based computations (without I/O operations) on the virtual thread, it might clog up the carrier thread which is unavailable for other virtual thread executions. This is also called monopolizing.

TIP This brings us to a very important point. Virtual threads, in the same fashion as reactive programming, are only suitable in applications with a lot of I/O operations. If you use them for CPU-bound computations, both reactive programming AND virtual threads perform worse than classic imperative programming with platform threads, because they add some overhead, and you're not using the main benefit they were introduced for (not blocking expensive threads during I/O operations).

CARRIER THREADS SCALING

If there are too many unscheduled virtual threads (which might be caused by both previous points), the JVM is forced to create new carrier threads. It goes without saying, that this brings both CPU and memory overhead. In the worst cases, it can also lead to applications running out of memory.

THREADLOCALS WITH LARGE OBJECTS

Many libraries utilize `ThreadLocal` values to store information. These values are only accessible from the same thread. This holds true also for virtual threads.

The problem rises when the objects stored in `ThreadLocal` are large. When the number of threads used to be small, it wasn't an issue and these object were often even cached and reused. But with virtual threads, each virtual thread receives its own instance of the object, and so every switch of the virtual thread means a lot of in-memory copying to make sure that each virtual thread has the correct values at all times. All this copying degrades the application's performance. Replacing the usage of thread locals is often not an easy task (e.g., transactions).

JDK proposed a new API called scoped values which is still in preview in JDK 21 (Java Enhancement Proposal (JEP) 446) which solves the issues with `ThreadLocal`. However, it will take time until all libraries that rely on it refactor `ThreadLocal` uses to scoped values.

Summarizing the problem overview in this section, we can see that virtual threads are not a silver bullet, but they surely are a great asset. They have their place but currently, Quarkus cannot rely on project Loom to work correctly in all scenarios and this is why there is no global flag that would move every execution on virtual thread. You need to explicitly request the execution on the virtual thread with the `@RunOnVirtualThread` annotation. It will surely take a few years until all libraries used in Quarkus (but also reused in several different frameworks) catch up on all requirements (e.g., pinning or thread locals). So it still make sense to understand reactive programming since even if you will not use it directly, it will still be used under the hood to give you the performance benefit comparable to project Loom.

8.6 Wrap up and next steps

This chapter explains reactive programming. We learned what reactive programming means and why it is advantageous to consider this architecture.

We learned about Quarkus' reactive engine, which makes it easy to start writing reactive applications with Quarkus. However, developers can choose if they want to adapt the reactive paradigm. The transition from imperative programming can even be gradual if needed because Quarkus allows combining both reactive and imperative code in the same application.

Lastly, we also explained the project Loom and virtual threads. We showed you how you can use virtual threads in Quarkus and also explained the potential problems that they might bring.

In the next chapter, we learn how to externalize events that are either propagated into our applications or outside of them. We follow up on the reactive concepts explained in this chapter into the reactive messaging—a MicroProfile specification that defines a standard way for the asynchronous message exchange that, as we learned in this chapter, is defined as the standing pillar of reactive systems.

8.7 Summary

- Reactive programming is a programming paradigm in which we write asynchronous, non-blocking, and scalable code based on continuations.

- Reactive programming paradigm might be harder to learn but provides better utilization of resources.
- SmallRye Mutiny is a reactive programming library utilized in Quarkus providing a simple API enabling writing asynchronous, event-driven application with integrated backpressure handling.
- Mutiny provides two main types: `Uni` and `Multi` that contain a fluent APIs that build reactive pipelines. You can use autocompletion and JavaDoc to build the pipeline step by step.
- Quarkus has a reactive engine based on Vert.x and Netty, providing a straightforward platform for writing reactive applications. However, a reactive engine doesn't require users to write reactive code, and it is possible to mix both the reactive and imperative code in the same application.
- Quarkus automatically decides whether your code should run on the non-blocking eventloop thread or on possibly blocking worker thread. This can be adjusted with `@Blocking` and `NonBlocking` annotations.
- Virtual threads (Project Loom) are lightweight JVM operated threads that are easy to create in big amounts and are allowed to perform blocking operations without blocking platform threads. But there are still problems that may cause them to limit application performance or even stop its execution if they are not handled correctly.

9

Reactive messaging

This chapter covers

- Learning what is MicroProfile Reactive Messaging
- Defining Reactive systems
- Utilizing reactive messaging in Quarkus
- Exploring Kafka and RabbitMQ broker integrations in Quarkus
- Reactive messaging in Car Rental

In the previous chapter, we learned what reactive programming is and why it can benefit user applications. We evaluated reactive programming as great alternative for applications that have a high resources demand, so they can effectively scale to higher numbers of concurrent requests. But in microservices, where the application consists of many isolated service applications, we often also need a way to let the applications communicate asynchronously without blocking threads while we wait for responses or coupling the applications together too closely. This is where reactive messaging comes in.

In this chapter, we learn about the MicroProfile Reactive Messaging specification that was created with the intent to propagate reactive messages between services. This standard provides a simple, unified API that we will also utilize in a newly created Billing service in our Car Rental application.

We learn about two popular asynchronous messaging alternatives that connect our services - Apache Kafka and RabbitMQ. We use the Quarkus integration of the reactive messaging to demonstrate the asynchronous message passing (using the same API for both Kafka and RabbitMQ) which is the base pillar of reactive systems. Additionally, reactive messaging builds on top of the Reactive Streams API, so we can use the Mutiny APIs we experimented with in the previous chapter.

9.1 Car Rental messaging integrations

This chapter constitutes a significant update to the car rental architecture. [Figure 9.1](#) demonstrates everything that we will add.

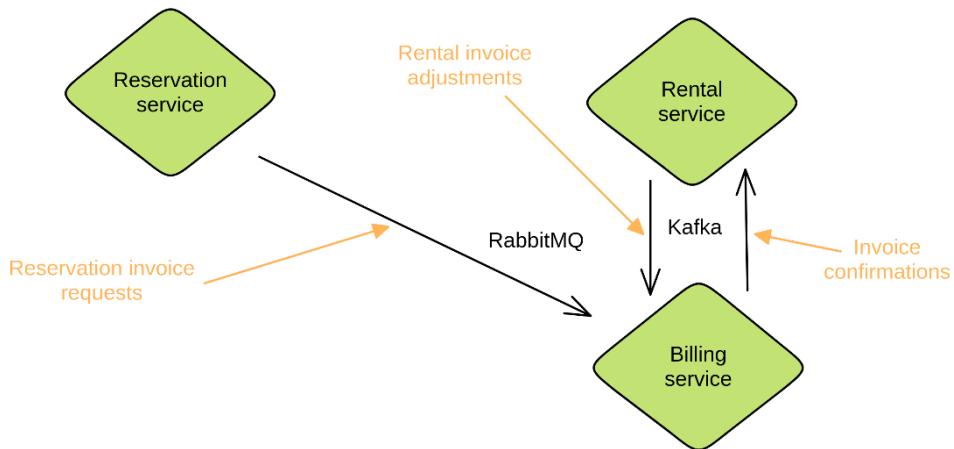


Figure 9.1 The Car Rental architecture changes incorporating reactive messaging channels

The integrations of Apache Kafka and RabbitMQ providers process messages that represent invoice requests and confirmations processed by the newly created Billing service. For simplicity, the Billing service will not make actual invoices. We just stub the invoice generation and consider it paid after some random time. When we finish this chapter implementation, this newly implemented part of the Car Rental system will be totally asynchronous and non-blocking.

9.2 Reactive systems

As we learned in the previous chapter, reactive programming is better at utilizing the application resources than the traditional imperative model. However, it also directly translates to application architecture and design.

In 2014, a group of independent developers created a document called the Reactive Manifesto (<https://www.reactivemanifesto.org/>). In this document, they described the requirements, which translated to the properties the modern distributed systems should have. These properties are responsiveness, resiliency, elasticity, and message passing. If the system has these properties, it is defined as a Reactive System. Putting this together visually, we can see how the properties depend on each other, as shown in [Figure 9.2](#).

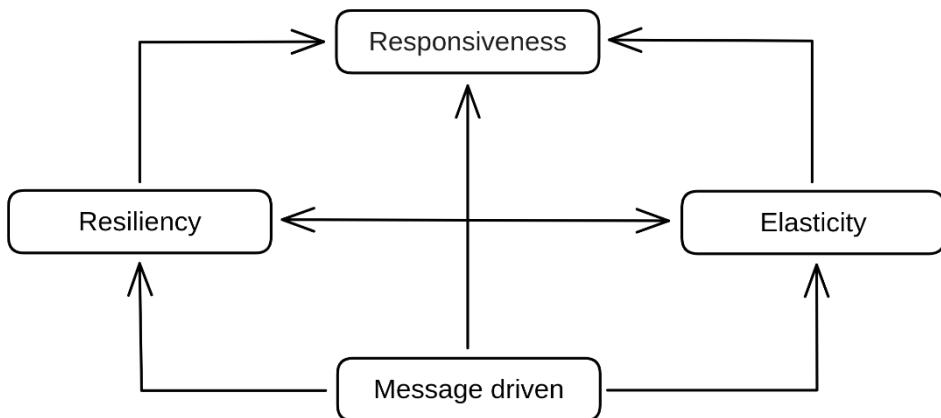


Figure 9.2 The reactive system architecture

- **Responsiveness** — Users expect to measure responses in mere milliseconds. So the application's responsiveness is a building stone of modern enterprise applications. Being responsive means that the system must handle user requests in a timely fashion. It also extends to the ability to identify and deal with failures. Responsiveness is the most important property conditioned by the other properties defined in the manifesto.
- **Resiliency** — The system must be able to stay responsive in case of failures. The application's services should be sufficiently isolated and highly available. In case of a service failure, the transition to another (backup) service should be fast and transparent so the system stays responsive. Users should not notice any service downtime.
- **Elasticity** — Further extends the concept of Resiliency, which we can understand as scaling elastic service to zero. Elasticity is the ability to scale the system up and down horizontally (meaning in the number of service instances) depending on the user demand. As modern applications usually deploy in production environments where they pay for the resources only when utilized, it is paramount to scale the services down when they are not needed to run. Conversely, if there is a demand (e.g., during the holiday season), the systems must be able to scale up for the system to stay responsive.

- **Message driven**—Asynchronous message passing represents the base property that enables all the above properties in the reactive systems. The messages are the only means of communication between individual parts of the application. The system sends the messages to the virtual addresses, meaning that the exact location of the called services is unavailable to the calling service. This enables seamless integration of resiliency and elasticity into the system. Errors also propagate as messages. Together, this allows reactive systems to be more isolated and loosely coupled. Integrating it also with asynchronicity allows for non-blocking communication that provides concepts like backpressure (the consumer's ability to control the received load).

When the system grows, these properties often become requirements. Reactive system principles need to be integrated into the system design. It is better to do this intentionally from the beginning than to try to add them to an already implemented system.

As an exercise, try to apply these properties to the microservices that we already implemented in the Car Rental system. What would happen if the Rental service wouldn't be available when the Reservation service tries to call it? You might also want to think about how they compare to the design of the system you are currently working on.

9.3 MicroProfile Reactive Messaging

The MicroProfile Reactive Messaging specification was created to standardize the way of defining the distributed asynchronous communication. It provides a simple API that the application developers use to propagate messages between CDI beans which can either produce, consume, or process messages. The messages are propagated through the notion of channels which represent the virtual addresses. These CDI beans are contained to only one application. However, MicroProfile Reactive Messaging defines an SPI called connectors which allows to plug external services as channels into the application.

9.3.1 Synchronous vs asynchronous distributed communication

A system based on the microservices architecture consists of separated and isolated services that communicate via remote protocols. Many of these systems utilize some form of the synchronous communication protocols. We covered the most frequently utilized protocols for this purpose in Chapter 4. These protocols require all parts of the applications to be continuously running in order to provide fully functional service. In [Figure 9.3](#) we see what happens when any of these services or the communication between them fail which is unavoidable in any production environment.

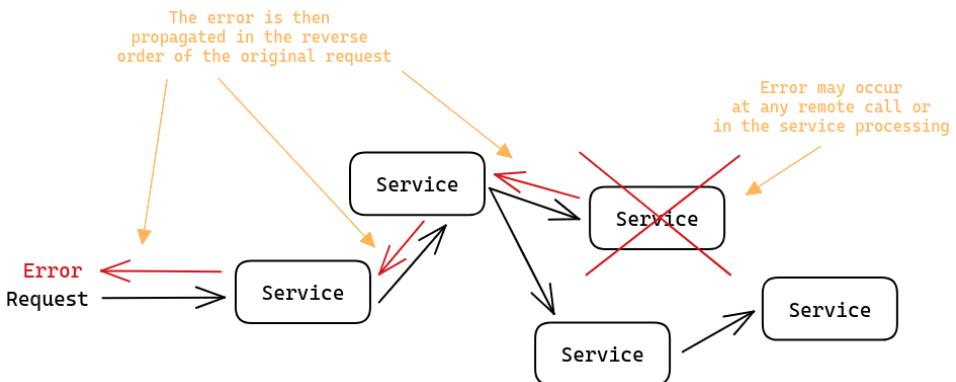


Figure 9.3 The synchronous communication failure propagation in microservices

We can analyze how the failed service error propagates through the calling chain of the services that the original call transmits through. Eventually, if no other service handles the error in a suitable manner (which often cannot be implemented due to the business reasons), the error is received by the original request issuer.

Another option for dealing with the problem of synchronous error propagation is the fault tolerance strategies. In fact, there is a separate MicroProfile Fault Tolerance specification that handles these cases. These strategies include retries of the failed requests, timeouts, or circuit breaker implementations. However, all these options directly affect the request times (time that the system takes to produce the response). And as we learned, responsiveness is the most important property that system should try to achieve in modern production deployments.

TIP In this chapter, we focus on reactive systems. Fault tolerance strategies which are an important concept in synchronous communications which many applications utilize are detailed in the following chapter.

With the asynchronous message passing, the system doesn't rely on services being always available. Instead, the message replication and delivery guarantees are pushed into the channels themselves. If the called service isn't available, the channel can buffer the messages until it comes back online and can continue to pull requests from the channel queue. Since the communication is asynchronous, the service that issues the request isn't blocked and can continue processing new requests if needed.

[Figure 9.4](#) demonstrates the failure in a reactive system based on the asynchronous message passing. The messages are accumulated in the channel until the receiving service can process them.

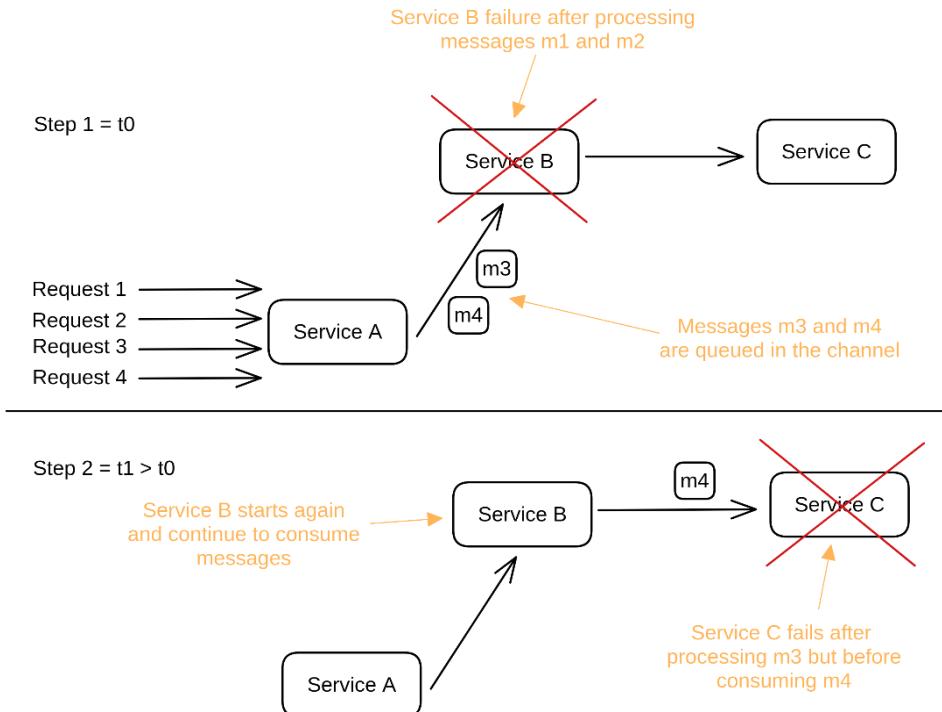


Figure 9.4 The asynchronous communication failure propagation in microservices

In the first step, the service B failed after processing first two messages/requests, so m_3 and m_4 representing the last two requests are buffered in the asynchronous channel. After this service comes back up it starts processing messages from the channel again. However, in the second step, the last service C also fails but only after processing message m_3 . So now the message m_4 is the only one buffered in the second channel, waiting for the service C restart, so it can process it.

9.3.2 Reactive messaging API

MicroProfile Reactive Messaging defines an API that bridges the CDI beans through communication channels. A channel represents a virtual address to which we can send or consume messages. Conceptually, it represents a unique `String` which is defined by the user code.

The flow of messages is based on the reactive streams API that we analyzed in the previous chapter. The CDI beans can define multiple methods that either produce, consume, or process messages. In this way, these methods represent either publishers, subscribers, or processors from the reactive streams definitions. [Figure 9.5](#) provides a visual representation of this processing.

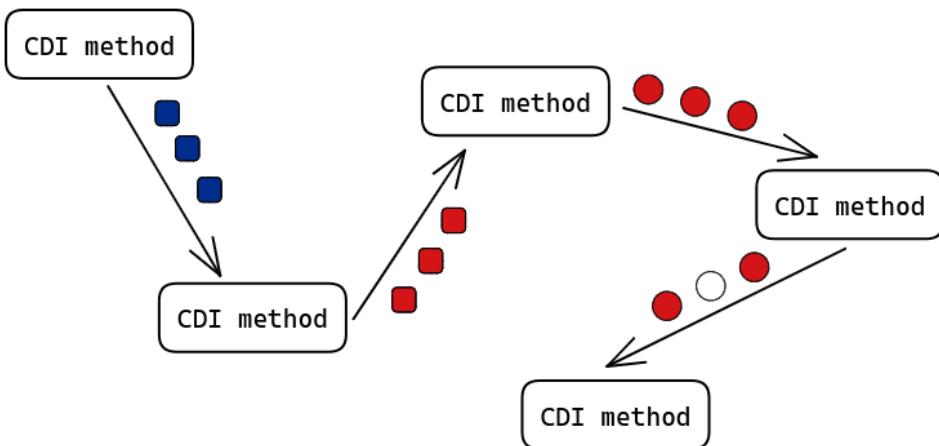


Figure 9.5 The model of the MicroProfile Reactive Messaging processing

The last CDI method acts as a filter on messages while the others act as processors transforming the received messages. The CDI methods don't have to be all in the same CDI bean. The specification allows only `@ApplicationScoped` and `@Dependent` CDI scopes for CDI beans used in reactive messaging.

The API conceptually consists of two main annotations—`@Outgoing` and `@Incoming` which are used to either produce or consume messages respectively. Both annotations take a single `String` argument that represents the name of the channel from which they are consuming or to which they are producing messages. The use of these two annotations on a method specifies how the method acts in the reactive messaging flow:

- Producer (reactive streams `Publisher`)—method annotated with `@Outgoing` annotation:

```

@Outgoing("channel-name")
public String producer() {
    return "hello";
}

```

- Consumer (reactive streams `Subscriber`)—method annotated with `@Incoming` annotation:

```

@Incoming("channel-name")
public void consumer(String payload) {
    System.out.println(payload);
}

```

- Processor (reactive streams `Processor`) – method annotated with both `@Incoming` and `@Outgoing`:

```
@Incoming("channel-name")
@Outgoing("channel-name-2")
public String processor(String payload) {
    return payload.toUpperCase();
}
```

The channel is always established only between a single producer and a single consumer with any number of optional processors in between. In other words, for a channel there is always only one source and one sink.

The flow of the reactive stream established through the channel is fully controlled by the implementation of the MicroProfile Reactive Messaging specification. In case of Quarkus, this implementation is the SmallRye Reactive Messaging library. SmallRye implementation provides many additional features on top of what is required by the specification. For instance, with SmallRye Reactive Messaging, it is possible to broadcast produced messages through a single channel to multiple consumers. Additionally, this implementation also provides several connectors for different external systems that we analyze in the following sections.

The MicroProfile Reactive Messaging specification defines several supported method signatures. Mainly, it also directly supports Reactive Streams types like `Publisher` or `Subscriber` and their subclasses. This means that we can directly utilize the SmallRye Mutiny's type `Multi` we learned about in the previous chapter which is the preferred way when writing reactive applications in Quarkus because it is a subtype of `Publisher`. A comprehensive example of reactive messaging use with Mutiny is provided in [Listing 9.1](#). It utilizes two channels (`ticks` and `times`) that combined produce a timestamp every second for a total of 5 seconds after which the producer stops producing messages.

Listing 9.1 A comprehensive example of MicroProfile Reactive Messaging with Mutiny

```

@Outgoing("ticks") #1
public Multi<Long> aFewTicks() {
    return Multi.createFrom()
        .ticks().every(Duration.ofSeconds(1))
        .select().first(5); #2
}

@Incoming("ticks")
@Outgoing("times") #3
public Multi<String> processor(Multi<Long> ticks) {
    return ticks.map(tick -> Instant.now().toString());
}

@Incoming("times") #4
public void consumer(String payload) {
    System.out.println(payload);
}

```

#1 The original producer pushes messages to the ticks channel.

#2 The created Multi produces 5 items representing clock ticks every second.

#3 Processor consumes the messages from ticks channel and produce a transformed items into the times channel.

#4 Consumer method for the times channel.

Notice the Multi can be used as both the return type and the argument in the processor method.

9.3.3 Message Acknowledgements

Acknowledgments provide the means for the consuming services to provide information back to the sending services. For instance, this is useful to propagate an error if the message cannot be processed.

There are two types strategies that users can choose from for positive acknowledgments (acks) or negative acknowledgments (nacks) processing—explicit or implicit. The @Acknowledgment annotation controls the strategy for a particular method. It receives a single argument of type Strategy with following options:

- MANUAL — strategy requiring explicit ack or nack through provided Message wrapper type.
- PRE_PROCESSING — messages implicitly acked before the method is entered.
- POST_PROCESSING — messages implicitly acked (or nacked on exception) after the method finishes.

- `NONE` — messages are not acknowledged either explicitly or implicitly.
The acknowledgement is handled elsewhere (e.g., different provider).

Acks and nacks are always provided only after the messages sent down the underlying channel in case of producers or processors are acked. In this way, the chain of acks builds in the opposite direction that the messages are sent until it reaches the original producer.

For instance, we can control the manual acknowledgments through the `Message` wrapper as shown in [Listing 9.2](#).

Listing 9.2 A manual acknowledgement example

```
@Incoming("channel-name")
@Acknowledgment(Acknowledgment.Strategy.MANUAL) #1
public CompletionStage<Void> consumer(Message<String> message) {
    if (processMessage(message.getPayload())) {
        return message.ack();
    } else {
        return message.nack(
            new IllegalStateException(
                "Cannot process message " + message.getPayload()));
    }
}
```

#1 Specify that the MANUAL strategy should be used in this method.

The method receives its `String` payload wrapped in the `Message` object that it can use to either call the `ack()` method for the positive acknowledgement or the `nack(Throwable)` method for negative acknowledgement.

The `@Acknowledgment` annotation is not mandatory. The default values of the used strategies differ for the individual combinations of types in the method signatures. You can find the detailed list of these defaults in the specification. We also used implicit acknowledgments in the last example in the previous section ([Listing 9.1](#)). If the signature directly consumes messages, reactive messaging always requests one more item when the item is consumed as it is also automatically acknowledged in the background (the default value for this signature is `POST_PROCESSING`). If we did not manually limit the number of produced items, they would be produced forever.

9.3.4 Integrating imperative code with reactive messaging

At least for now, it is not possible to move to totally reactive architecture in all deployed system services. For instance, the microservices that are processing the frontend requests, (which are typically utilize HTTP calls) still require to expose HTTP or REST APIs. For this reason, MicroProfile Reactive Messaging provides a simple API to bridge the imperative and reactive worlds with the `@Channel` annotation.

The `@Channel` annotation has two use cases—to either produce messages from the imperative code or to consume a channel into an instance of `Publisher` through a CDI injection.

To produce a message from imperative code into a reactive messaging channel, we need to utilize also the `Emitter` class that has a `send` method as showed in [Listing 9.3](#).

Listing 9.3 An Emitter example

```
@Inject
@Channel("requests")
Emitter<String> requestsEmitter; #1

@POST
@Path("/request")
public String request(String body) {
    requestsEmitter.send(body); #2
    return "Processing " + body;
}
```

#1 Use the `Emitter` class to send messages to the channel.

#2 The method `send` sends the message to the channel asynchronously.

The `@Channel` annotation specifies the channel name that can be used then in the `@Incoming` annotation. The `send` method returns a `CompletionStage` that is completed when the message is acknowledged.

The second use case is particularly useful for applications working with server sent events which push the events from server to the usually front end services through an HTTP connection. We can use the `@Channel` annotation to also inject instances of Reactive Streams (or Flow) `Publisher` or its subclasses as in our case the Mutiny's `Multi`. Consuming the produced ticks from the producer introduced in [Listing 9.1](#) into an instance of `Multi` can be done as demonstrated in [Listing 9.4](#). This listing also presents the utilization of the `SERVER_SENT_EVENTS` media type in the GET JAX-RS method which produces this channel.

Listing 9.4 Consuming the reactive messaging channel into a Multi

```

@Inject
@Channel("ticks")
Publisher<Long> ticks;

@GET
@Path("/consume")
@Produces(MediaType.SERVER_SENT_EVENTS)
public Publisher<Long> sseTicks() {
    return ticks;
}

```

If you are not familiar with the server sent events data type, you can easily just open the `/consume` URL in the browser or pass the following command line flags to clients we utilize in this book. In every case, you can see that the server pushed new data every second.

```

$ http :8080/ticks/consume --stream
$ curl -N localhost:8080/ticks/consume

```

9.4 Introducing reactive messaging in Reservation

With what we have learned so far in this chapter, we are able to now include reactive messaging in the Reservation service. For now, we will focus only on the semantics in this single service that we can later transform to remote communication over different protocols supporting asynchronous messaging.

The Reservation service will use reactive messaging to send messages to the Billing service that we haven't created yet. However, we can start integrating reactive messaging in the Reservation by providing a stub for Billing service that we will create in the following section. This is a good example of how we can develop individual microservices independently.

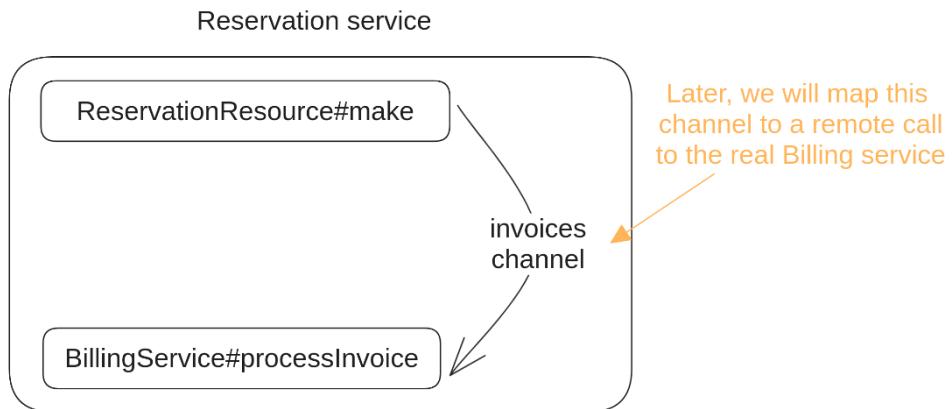


Figure 9.6 The reactive messaging integration in the Reservation service

We start by defining the model for the invoices that we are sending, let's create a new class `org.acme.reservation.billing.Invoice` as demonstrated in [Listing 9.5](#). It contains all required information that needs to be propagated to the Billing service to create the invoice. As always, you can find the full source code of this class in the book resources `chapter-09` directory.

Listing 9.5 The code of the Invoice class

```

package org.acme.reservation.billing;

import org.acme.reservation.entity.Reservation;

public class Invoice {

    public Reservation reservation; #1
    public double price;

    // all-arg constructor, toString
}

```

#1 Keep the registration details so they can be printed on the invoice.

The place where we would like to charge the customer for making a reservation is the `ReservationResource#make` method that creates reservations by persisting them into the database. Before we do that, we should make sure that the customer will be charged for the order. In essence, we can make the request for billing asynchronous, because we can always cancel the reservation later if the invoice isn't paid in time.

Even if we already use several reactive extensions in the Reservation service, none of them uses reactive messaging. We need to add a new extension `quarkus-smallrye-reactive-messaging` which we can for instance do with the `quarkus` CLI as shown in the following snippet:

```
$ quarkus ext add quarkus-smallrye-reactive-messaging
```

The `make` method is a JAX-RS POST method which means that we need to send the message to the reactive channel manually with `Emitter` as described in [9.3.4](#). The name of channel is `invoices` as we are sending invoices to the Billing service. [Listing 9.6](#) provides the updated code with relevant parts of the `ReservationResource`. For simplicity, we use a standard rate per day which is applicable to all cars.

Listing 9.6 ReservationResource changes required to produce messages to the reactive messaging channel

```
public static final double STANDARD_RATE_PER_DAY = 19.99;

@.Inject
@Channel("invoices")
Emitter<Invoice> invoiceEmitter; #1

@Consumes(MediaType.APPLICATION_JSON)
@POST
@WithTransaction
public Uni<Reservation> make(Reservation reservation) {
    ...

    return reservation.<Reservation>persist().onItem()
        .call(persistedReservation -> {
            Log.info("Successfully reserved reservation " +
                persistedReservation);

            invoiceEmitter.send(new Invoice(reservation,
                computePrice(reservation))); #2

    ...
}

private double computePrice(Reservation reservation) {
    return (ChronoUnit.DAYS.between(reservation.startDay,
        reservation.endDay) + 1) * STANDARD_RATE_PER_DAY;
}
```

#1 Inject the Emitter for the invoices channel.

#2 Send the asynchronous message into the invoices channel.

Since we don't have the Billing service yet, we can create a stub that consumes messages for now and just prints them to the console. Create a new class `org.acme.reservation.billing.BillingService` as demonstrated in [Listing 9.7](#).

Listing 9.7 The code of the BillingService, a stub for the real Billing service

```
package org.acme.reservation.billing;

import org.eclipse.microprofile.reactive.messaging.Incoming;

import jakarta.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class BillingService {

    @Incoming("invoices")
    public void processInvoice(Invoice invoice) {
        System.out.println("Processing received invoice: " + invoice);
    }
}
```

Creating a new reservation now correctly produces a new asynchronous message to the invoices channel which is then consumed in the BillingService#processInvoice method. [Listing 9.8](#) shows how we can experiment with this new functionality.

Listing 9.8 The code of the BillingService

```
$ http POST :8081/reservation <<< '{
  "carId": 1,
  "startDay": "3333-01-01",
  "endDay": "3333-01-03"
}'

# in the Reservation Dev mode terminal
2023-06-24 12:09:59,802 INFO  [org.acm.res.res.ReservationResource] (
  -vert.x-eventloop-thread-4) Successfully reserved reservation
  -Reservation{id=1, carId=1, startDay=3333-01-01, endDay=3333-01-03}
Processing received invoice: Invoice{reservation=Reservation{id=1, carId=1,
  -startDay=3333-01-01, endDay=3333-01-03}, price=39.98}
```

The Emitter#send method returns a CompletionStage that is completed when the produced message is acknowledged. If the message is nacked, the CompletionStage is completed exceptionally. To correctly handle the reservation start, we should verify that the invoice request is received by the Billing service, meaning it is acknowledged. We could use the CompletionStage directly as showed in the following snippet:

```

CompletionStage<Void> invoiceCS = invoiceEmitter.send(
    new Invoice(reservation, computePrice(reservation)));

invoiceCS.handle((ack, nack) -> {
    ...
});

invoiceCS.toCompletableFuture().join();

```

The `join()` method would block until the invoice message is acked (or nacked). This could work, but we are writing a reactive service so plugging the invoice acknowledgment into the reactive pipeline would be better. With Mutiny, we can do it easily. Mutiny provides the emitter alternative called `MutinyEmitter` that returns a `Uni`. With it, we can update the `make` method as shown in [Listing 9.9](#).

Listing 9.9 The make method utilizing MutinyEmitter

```

@Inject
@Channel("invoices")
MutinyEmitter<Invoice> invoiceEmitter; #1

@Consumes(MediaType.APPLICATION_JSON)
@POST
@WithTransaction
public Uni<Reservation> make(Reservation reservation) {
    reservation.userId = context.getUserPrincipal() != null ?
        context.getUserPrincipal().getName() : "anonymous";

    return reservation.<Reservation>persist().onItem()
        .call(persistedReservation -> {
            Log.info("Successfully reserved reservation "
        + persistedReservation);

        Uni<Void> invoiceUni = invoiceEmitter.send( #2
            new Invoice(reservation, computePrice(reservation)))
            .onFailure().invoke(throwable -> System.out.printf(
                "Couldn't create invoice for %s. %s%n",
                persistedReservation, throwable.getMessage()));

        if (persistedReservation.startDay.equals(LocalDate.now())) {
            return invoiceUni.chain(() -> #3
                rentalClient.start(persistedReservation.userId,
                    persistedReservation.id)
                .onItem().invoke(rental ->

```

```

        Log.info("Successfully started rental " + rental))
        .replaceWith(persistedReservation));
    }
    return invoiceUni
        .replaceWith(persistedReservation); #4
});
}

```

#1 Injecting MutinyEmitter instead of Emitter.

#2 The send method now returns a Uni.

#3 If needed, we can simply chain the invoice acknowledgment to the Rental service invocation.

#4 Or replace the produced invoice ack with persisted reservation.

Now we have a fully functioning reactive service! The `make` method is reactive. It guarantees that only if the user pays for the rental (the Billing service acknowledges the invoice request), the rental start request is sent to the Rental service.

Hopefully, this isn't too complicated. Definitely, reactive programming might be a complex programming approach to adapt. But once you get the handle of it, it comes quite naturally since it is the way we all think about the world. The Car Rental is now able to process many more reservation requests than it did with the imperative model.

9.5 Connectors

The MicroProfile Reactive Messaging architecture we investigated so far was limited to only one service. However, in the microservices applications we need to be able to communicate with other services. We learned how we can bridge the synchronous protocols like HTTP into reactive messaging with `@Channel` which we could possibly misuse for this mapping but there are also different distributed protocols and projects that are designed for asynchronous message passing that we can directly integrate into reactive pipelines.

9.5.1 What is a connector?

In MicroProfile Reactive Messaging, a connector represents an SPI (Service Provider Interface – intended for the implementations, not users) that presents a way to extend the messaging channels to external messaging providers. From the user point of view, it is fully included only as a configuration. In the application, we define the channels in the exactly same way as we did in the previous section. The configuration then tells the reactive messaging implementation (SmallRye, in the case of Quarkus) to either push the outgoing messages to the external solution or to pull the incoming messages from it. A connector is configured per channel. [Figure 9.7](#) visualizes this concept. The connector represents a boundary of the application that bridges the messages to the remote services while still preserving the reactive messaging (streams) flow.

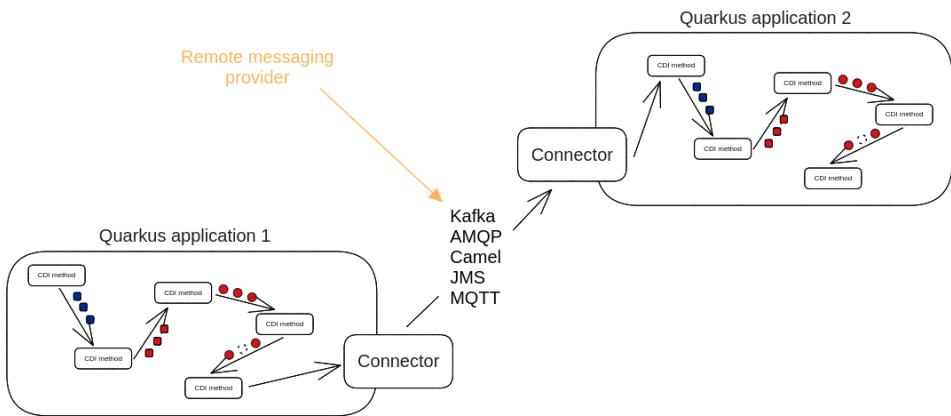


Figure 9.7 The connectors architecture

There are multiple connectors for different remote brokers that users can choose from, including Apache Kafka, RabbitMQ, or JMS (Jakarta Messaging). Of course, they are easily switchable by only changing the application configuration. There is no need to change the reactive messaging code since it is possible to just remap individual channels into different connector in the configuration.

Users refer to connectors by their names which are defined by the implementations. In SmallRye Reactive Messaging, they are prefixed with `smallrye-` prefix following by the underlying technology that the connector bridges (e.g., `smallrye-kafka`, `smallrye-rabbitmq`, or `smallrye-jms`).

9.5.2 Connector configuration

Connectors are configured through the standard configuration mechanisms that we learned about in Chapter 3. MicroProfile Reactive Messaging specification defines the following formats of configuration properties:

```
mp.messaging.incoming.{channel-name}.{attribute-name}=attribute-value
mp.messaging.outgoing.{channel-name}.{attribute-name}=attribute-value
mp.messaging.connector.{connector-name}.{attribute-name}=attribute-value
```

The configuration of the channels (`channel-name`) overrides the global configuration of connectors (`connector-name`). It is important to point out that all configuration for channels is specific to the either `incoming` or `outgoing` channel configuration. This means that if we use the same channel (`channel-name`) for both producing and consuming of messages (i.e, reactive streams' processor), we need to configure each direction separately. Even if it uses the same connector.

The only required configuration for each channel is the `connector` attribute which specifies the name of the connector. However, if you only have one connector extension on the classpath, you don't have to configure it. It is automatically auto-attached to all unconfigured channels. But be careful if you start utilizing any channel in-memory (i.e., you define a consumer in the same application), then in-memory connector takes precedence.

Most of the connectors require additional configuration for the external service connections configuration that they need to set up. For instance, the location of the production instance of Kafka or the JMS queue URL.

TIP Most of the connectors and their respective Quarkus extensions support Dev Services. So it's preferred to specify connection configuration only for the `prod` profile.

As you probably noticed, the channels which are not mapped to any connector, as the channels that we used so far in the previous section, are by default processed in-memory. In this way, the same reactive messaging concepts apply also for the local development as well as for the remote communication (which from the user's view) is easily changeable in the application configuration.

9.6 Reactive messaging with Kafka connector

We are now ready to introduce the last backend service into the Car Rental project—the Billing service. As you might remember from the architecture diagram, this service is connected to other backend services purely through the non-blocking, asynchronous technologies as demonstrated in [Figure 9.8](#). The Billing service is fully reactive microservice based on asynchronous messaging.

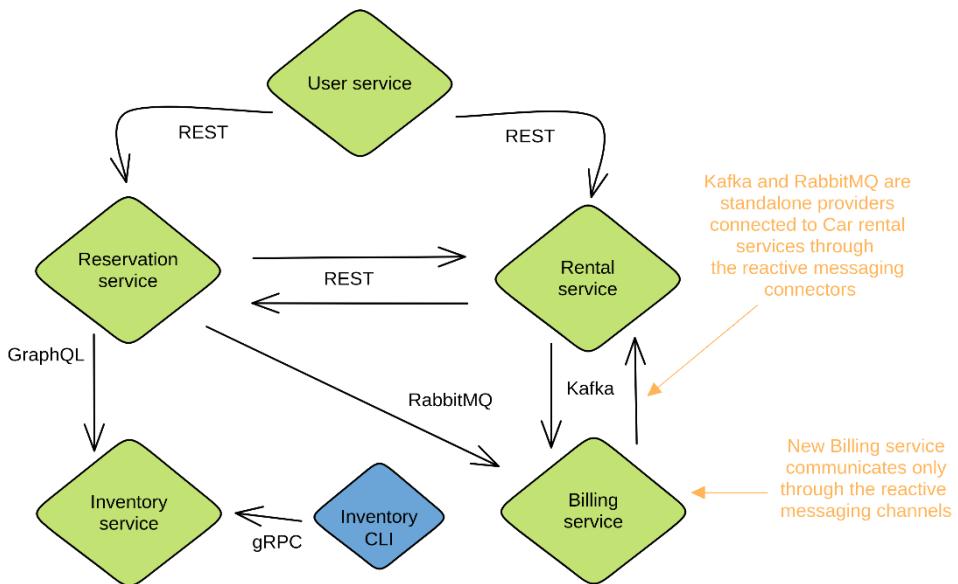


Figure 9.8 The Car Rental architecture diagram detailing the Billing service integration

Since we already prepared Reservation service to send invoices with reactive messaging, let's first focus on the simplest service we created so far, the Rental service. This allows us to demonstrate how we can start integrating remote protocol from scratch.

Rental service communicates with the Billing service through an Apache Kafka broker (<https://kafka.apache.org/>). We are not going to cover Kafka internals here but there is an excellent resource available in the book *Kafka in Action* by Dylan Scott, Viktor Gamov, and Dave Klein (Manning, 2022).

To utilize reactive messaging with Kafka, we need to include a connector that is able to connect to a Kafka broker. SmallRye Reactive Messaging provides such connector called `smallrye-kafka`.

9.6.1 Kafka integration in the Rental service

As with any other Quarkus functionality, there is an extension that sets up the SmallRye Kafka connector named `smallrye-reactive-messaging-kafka`. We can add it to the Rental service with the following command which needs to be executed in the `rental-service` directory:

```
$ quarkus ext add smallrye-reactive-messaging-kafka
```

If you now start the Rental service in Dev mode, you'll notice that this extension comes with packaged Dev Services for Kafka broker which container is started in the background similarly as the database that Rental uses to persist rentals. [Listing 9.10](#) shows the log message that details that the Dev Services were started.

Listing 9.10 Dev Services for Kafka started log message

```
INFO [io.qua.kaf.cli.dep.DevServicesKafkaProcessor] (build-53) Dev
- Services for Kafka started. Other Quarkus applications in dev mode
- will find the broker automatically. For Quarkus applications in
- production mode, you can connect to this by starting your application
- with -Dkafka.bootstrap.servers=OUTSIDE://localhost:39809
```

This message also hints that any other Quarkus applications started in Dev mode on the same computer automatically connect to the same Kafka instance, which is very useful for local testing. We will surely utilize this soon with the Billing service once created. Additionally, we also have a simple flag that we can include if other Quarkus services start in production mode.

TIP The Kafka Dev Services in fact start a more lightweight version of Redpanda platform (<https://redpanda.com>) that is compatible with Kafka API.

The messages that we send to the Billing service represent invoices for the rentals where users return cars sooner or later than originally reserved through the Reservation service, meaning the price for the rental needs to be adjusted. For this reason, we create a single Kafka topic called `invoices-adjust` to which the Rental service sends invoice messages if such user case is detected.

First, we need to define the format of the `InvoiceAdjust` messages that Rental service sends to the Billing service through Kafka. Create a new class `org.acme.rental.billing.InvoiceAdjust` which looks like [Listing 9.11](#).

Listing 9.11 Rental service representation of the InvoiceAdjust message

```

package org.acme.rental.billing;

import java.time.LocalDate;

public class InvoiceAdjust {

    public String rentalId;
    public String userId;
    public LocalDate actualEndDate;
    public double price;

    public InvoiceAdjust(String rentalId, String userId,
                         LocalDate actualEndDate, double price) {
        this.rentalId = rentalId;
        this.userId = userId;
        this.actualEndDate = actualEndDate;
        this.price = price;
    }

    @Override
    public String toString() {
        return "InvoiceAdjust{" +
            "rentalId='" + rentalId + '\'' +
            ", userId='" + userId + '\'' +
            ", actualEndDate=" + actualEndDate +
            ", price=" + price +
            '}';
    }
}

```

Next, we need to set up the emitter that will send the new invoice once the return end date is exceeded. We can update the `RentalResource#end` JAX-RS method like demonstrated in [Listing 9.12](#).

Listing 9.12 The modification of the RentalResource for integration with reactive messaging

```

@Path("/rental")
public class RentalResource {

    public static final double STANDARD_REFUND_RATE_PER_DAY = -10.99;
    public static final double STANDARD_PRICE_FOR_PROLONGED_DAY = 25.99;
}

```

```

@Inject
@RestClient
ReservationClient reservationClient; #1

@Inject
@Channel("invoices-adjust")
Emitter<InvoiceAdjust> adjustmentEmitter; #2

...

@PUT
@Path("/end/{userId}/{reservationId}")
public Rental end(String userId, Long reservationId) {
    Log.infof("Ending rental for %s with reservation %s",
              userId, reservationId);

    Rental rental = Rental
        .findByUserAndReservationIdsOptional(userId, reservationId)
        .orElseThrow(() -> new NotFoundException("Rental not found"));

    Reservation reservation = reservationClient #3
        .getById(reservationId);

    LocalDate today = LocalDate.now();
    if (!reservation.endDay.isEqual(today)) {
        adjustmentEmitter.send(new InvoiceAdjust( #4
            rental.id.toString(), userId, today,
            computePrice(reservation.endDay, today)));
    }

    rental.endDate = LocalDate.now();
    rental.active = false;
    rental.update();
    return rental;
}

private double computePrice(LocalDate endDate, LocalDate today) {
    return endDate.isBefore(today) ?
        ChronoUnit.DAYS.between(endDate, today)
        * STANDARD_PRICE_FOR_PROLONGED_DAY :
        ChronoUnit.DAYS.between(today, endDate)
        * STANDARD_REFUND_RATE_PER_DAY;
}

```

```

    ...
}

#1 Inject the new ReservationClient (we are yet to create) that call the Reservation service through
the exposed REST API.
#2 Inject the invoices-adjust channel emitter that sends Kafka messages representing invoice
adjustments.
#3 Retrieve the reservation from the Reservation service to get the originally promised end day.
#4 If the customer returns the car on a different date than promised in the reservation, charge the
customer accordingly.

```

We also need to retrieve the reservation from the `Reservation` service in order to compare the end date. For simplicity (and because this is not a reactive code), the new `ReservationClient` code is omitted here but with what we learned so far, it shouldn't be hard to implement. Try to implement it as an exercise. Remember that the endpoint that returns the reservation JSON is available at `GET /admin/reservation/{id}`. Note that you also need `quarkus-rest-client-reactive-jackson` if you haven't added it to the `Rental` client before. If you need a hint on how to implement this, the whole code is available in the `chapter-09/rental-service` directory of the book resources.

This is all that we need to start sending invoice adjustments from the code perspective. Since `smallrye-kafka` is the only connector on the classpath, the Quarkus already configures the `invoices-adjust` channel to be mapped into the `invoices-adjust` Kafka topic and adds required serializers for the `InvoiceAdjust` Kafka record that we send to Kafka broker. You can see this in the log message when the Dev mode starts:

```

INFO [io.qua.sma.dep.processor] (build-21) Configuring the channel
  - 'invoices-adjust' to be managed by the connector 'smallrye-kafka'
INFO [io.qua.sma.dep.processor] (build-25) Generating Jackson serializer
  -for type org.acme.rental.billing.InvoiceAdjust

```

However, we don't see the produced messages even if we know now they are produced to the started Kafka Dev service when we send invoice adjustments. Kafka message consumption without the reactive messaging is beyond the scope of this book. So for now, feel free to define an `@Incoming consumer` for the `invoices-adjust` channel with which you can verify that the messages are produced correctly. However, if you have experience with Kafka, you can check the Dev Service Kafka with client tools like the example showed in listing [Listing 9.13](#). But it would be useful to also verify this message production in an automated test.

TIP This verification is totally optional. The `kafka-console-consumer` comes from Apache Kafka tools but if you don't have it already available, you don't need to download it for this example.

Listing 9.13 The `kafka-console-consumer` example consuming the `prolong` topic

```
$ ./kafka-console-consumer.sh --bootstrap-server localhost:46803 --topic \
invoices-adjust --from-beginning
{"rental": {"id": "649837b0d478bc0dca7d0980", "userId": "anonymous",
  "reservationId": 13, "startDate": "2023-06-25", "endDate": "2023-06-25",
  "active": false}, "actualEndDate": "2023-06-25", "price": 25.99}
```

9.6.2 Testing reactive messaging with Kafka

Quarkus utilizes the Kafka Companion Java library which is also provided by the SmallRye Reactive Messaging. According to their definition (<https://smallrye.io/smallrye-reactive-messaging/4.7.0/kafka/test-companion>) "It is not intended to mock Kafka, but to the contrary, connect to a Kafka broker and provide high-level features". This library provides an easy integration with the already started Kafka Dev Service, which is available during the test execution.

To use the Kafka Companion API, Quarkus provides a wrapper extension `quarkus-test-kafka-companion` which can be included in the `Rental` service in the `test` scope as demonstrated in [Listing 9.14](#). We also use Rest Assured and Mockito in the new test. Add these dependencies if they are not already present to the `Rental` service.

Listing 9.14 `quarkus-test-kafka-companion` extension in the test scope

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-test-kafka-companion</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-junit5-mockito</artifactId>
  <scope>test</scope>
</dependency>
```

[Listing 9.15](#) contains the test source code created in `org.acme.rental.RentalResourceTest` class. You can find the full source code (with all imports) in the directory `chapter-09` of the book resources.

Listing 9.15 The source code of the `RentalResourceTest` test

```
@QuarkusTest
@QuarkusTestResource(KafkaCompanionResource.class) #1
public class RentalResourceTest {

    @InjectKafkaCompanion
    KafkaCompanion kafkaCompanion;

    @Test
    public void testRentalProlongedInvoiceSend() {
        // stub the ReservationClient call
        Reservation reservation = new Reservation();
        reservation.endDay = LocalDate.now().minusDays(1);

        ReservationClient mock = Mockito.mock(ReservationClient.class);
        Mockito.when(mock.getId(1L)).thenReturn(reservation);
        QuarkusMock.installMockForType(mock, ReservationClient.class,
            RestClient.LITERAL);

        // start new Rental for reservation with id 1
        given()
            .when().post("/rental/start/user123/1")
            .then().statusCode(200);

        // end the with one prolonged day
        given()
            .when().put("/rental/end/user123/1")
            .then().statusCode(200)
            .body("active", is(false),
                "endDate", is(LocalDate.now().toString()));

        // verify that message is sent to the invoices-adjust Kafka topic
        ConsumerTask<String, String> invoiceAdjust = kafkaCompanion
            .consumeStrings().fromTopics("invoices-adjust", 1)
            .awaitNextRecord(Duration.ofSeconds(10)); #2

        assertEquals(1, invoiceAdjust.count());
        assertTrue(invoiceAdjust.getFirstRecord().value()
            .contains("\\"price\":" +
```

```

        RentalResource.STANDARD_PRICE_FOR_PROLONGED_DAY));
    }
}

```

#1 KafkaCompanionResource sets up the test context and allows us to inject KafkaCompanion.

#2 Adjustable consumption of Kafka records.

This test uses `KafkaCompanion` that can be easily injected. It first stubs the `ReservationClient` with Mockito, since we can't make the remote REST call to the Reservation service in our test. Then it creates and ends a new rental through the exposed REST API. Next, it sets up a consumer listener for the `invoices-adjust` Kafka topic to which the `end` method should have pushed the invoice adjustment message (because the mock returns `endDay` one day before today). After that, it can verify that the message is indeed produced to the Kafka broker.

You can execute this test with:

```

$ mvn test
...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
  ▶ 20.983 s - in org.acme.rental.RentalResourceTest
...

```

The Kafka companion provides a very simple integration testing for our Kafka broker integration, and it proves that we are indeed sending the invoice adjustment messages to the Kafka topic. Now we only need to consume them in the Billing service.

9.7 Reactive messaging with RabbitMQ

RabbitMQ (<https://www.rabbitmq.com/>) is one of the most popular message brokers. It provides a lightweight platform supporting multiple messaging protocols and also streaming. The default protocol is AMQP (Advanced Message Queuing Protocol) 0-9-1. The messages are sent and consumed from queues. If you want to learn more about this broker, you can check the *RabbitMQ in Depth* book by Gavin M. Roy (Manning, 2017).

In Quarkus, applications utilize RabbitMQ broker through the SmallRye Reactive Messaging RabbitMQ connector called `smallrye-rabbitmq`. We can use the connector in the exactly same way as the Kafka connector introduced in the previous section since both utilize reactive messaging.

TIP By default, RabbitMQ utilizes AMQP 0-9-1 which is very different from the latest version AMQP 1.0. RabbitMQ can be configured to use AMQP 1.0 via separate plugin. SmallRye Reactive Messaging also provides a SmallRye AMQP connector which supports AMQP 1.0.

In this section, we are going to expand the reactive messaging setup in the Reservation service that we prepared in section [9.4](#). Since Quarkus utilizes reactive messaging in the same way whether we use connector to pass messages outside the application's JVM or we utilize the built-in in-memory channels, the required changes in the Reservation service are minimal.

As you are probably already used to, our first step is to include the Quarkus extension that brings the `smallrye-rabbitmq` connector. This extension is called `quarkus-smallrye-reactive-messaging-rabbitmq` and it can be included in the Reservation service as showed in [Listing 9.16](#).

Listing 9.16 Adding the smallrye-rabbitmq connector extension

```
$ quarkus ext add smallrye-reactive-messaging-rabbitmq
```

In section [9.4](#), we already prepared the reactive messaging to produce invoices when the reservation is created in the `ReservationResource` class. An excerpt of this class sending the invoices is provided in [Listing 9.17](#).

Listing 9.17 A portion of the ReservationResource class that sends invoices to the invoices channel

```
@Inject
@Channel("invoices")
MutinyEmitter<Invoice> invoiceEmitter;

...
Uni<Void> invoiceUni = invoiceEmitter.send(
    new Invoice(reservation, computePrice(reservation)))
    ...
```

Similarly to the Kafka connector, if the `smallrye-rabbitmq` connector is the only connector on the classpath, it is automatically configured as the outgoing connector for the `invoices` channel which is mapped to the `invoices` RabbitMQ queue. The RabbitMQ connector extension also provides the RabbitMQ compatible broker that is started as a Dev Service. In the Dev mode, you can now check that the RabbitMQ broker container starts and that our application is already configured to connect to it by pressing `c` in the terminal where the Dev mode runs:

```
smallrye-reactive-messaging-rabbitmq
  Container:      d0766964b1b5/focused_solomon docker
→ .io/library/rabbitmq:3.12-management
  Network:        351e3fe3-3077-492e-87c0-165354f9a41d (tc-1kRVRWLn,
→ d0766964b1b5) - null:38639->5672/tcp ,null:37759->15672/tcp
  Exec command:   podman exec -it d0766964b1b5 /bin/bash
  Injected config: - rabbitmq-host=localhost
                    - rabbitmq-http-port=37759
                    - rabbitmq-password=guest
                    - rabbitmq-port=38639
                    - rabbitmq-username=guest
```

The invoices are currently just consumed and printed to the standard output in the `BillingService` class. Since we already have this consumer available, let's investigate how it can consume messages from the RabbitMQ broker. Of course, the `BillingService` is not yet connected to the RabbitMQ broker since both producer and consumer for the `invoices` channel are in the same application (it uses the in-memory connector). Let's change the name of the incoming channel in the `BillingService` to demonstrate this as shown in [Listing 9.18](#).

Listing 9.18 Modification of the incoming channel name to `invoices-rabbitmq` in `BillingService`

```
@ApplicationScoped
public class BillingService {

    @Incoming("invoices-rabbitmq")
    public void processInvoice(Invoice invoice) {
        System.out.println("Processing received invoice: " + invoice);
    }
}
```

When the Reservation service starts now in Dev mode, it produces the following log messages stating it is connecting to the address `invoices-rabbitmq` (they don't need to be in succession):

```
INFO [io.qua.sma.dep.processor] (build-12) Configuring the channel
→ 'invoices-rabbitmq' to be managed by the connector 'smallrye-rabbitmq'
INFO [io.sma.rea.mes.rabbitmq] (Quarkus Main Thread) SRMSG17036: RabbitMQ
→ broker configured to [localhost:38639] for channel invoices-rabbitmq
INFO [io.sma.rea.mes.rabbitmq] (Quarkus Main Thread) SRMSG17007: Connection
→ with RabbitMQ broker established for channel `invoices-rabbitmq`
INFO [io.sma.rea.mes.rabbitmq] (vert.x-eventloop-thread-1) SRMSG17000:
→ RabbitMQ Receiver listening address invoices-rabbitmq
```

But of course, we are only producing messages to the queue derived from the channel declaration in the `ReservationResource` ([Listing 9.17](#)) which is `invoices`. This means that nothing is received in the consumer method anymore.

To map the `invoices-rabbitmq` channel to the correct RabbitMQ queue `invoices` we could also modify the `@Channel` annotation in the `ReservationResource`, but we can also modify the `application.properties` like presented in [Listing 9.19](#) which gives us more flexibility. The exchange is the RabbitMQ concept that is responsible for the message routing. It also defaults to channel name if not specified.

Listing 9.19 Changing the RabbitMQ queue name for a channel in configuration

```
mp.messaging.incoming.invoices-rabbitmq.queue.name=invoices
mp.messaging.incoming.invoices-rabbitmq.exchange.name=invoices
```

When the application now restarts the `invoices-rabbitmq` channel consumes from the `invoices` RabbitMQ queue. But there is one catch. Since the messages sent from RabbitMQ are encoded as JSON object, we can't consume the `Invoice` object in the `@Incoming` method directly since there is no default mapper that would transform it (as there was with in-memory channel). However, it is really simple to map this object to the `Invoice` instance again as showed in [Listing 9.20](#).

Listing 9.20 The final version of BillingService consuming messages from RabbitMQ

```
import io.vertx.core.json.JsonObject;
...
@ApplicationScoped
public class BillingService {

    @Incoming("invoices-rabbitmq")
    public void processInvoice(JsonObject json) {
        Invoice invoice = json.mapTo(Invoice.class);
        System.out.println("Processing received invoice: " + invoice);
    }
}
```

Of course, since we are just printing the invoice to the standard output, we could just directly print the received `JsonObject`. But learning how to easily map the `JsonObject` to the domain objects is surely useful.

Repeating request for making a new reservation now correctly produces a new invoice message into the RabbitMQ broker which is then consumed in the `BillingService` and printed to the output. However, now it goes through the RabbitMQ broker.

9.7.1 Testing reactive messaging with RabbitMQ

In the RabbitMQ testing, we don't have a similar helper library as we had with Kafka (`quarkus-test-kafka-companion`). We will utilize the reactive messaging directly to receive posted message from the `invoice` queue in the test. As you may remember, Quarkus test is also a CDI bean, so we define the `@Incoming` method similarly as we did in the `BillingService`. In fact, we can move the `BillingService#processInvoice` as is into the newly created test because we will no longer need it in the application itself.

Since we produce messages to RabbitMQ broker asynchronously, we need a mechanism that allows us to wait for the message to be received before we start asserting its contents. For this reason, we use the Awaitility (<http://www.awaitility.org/>) library. You can add it to the Reservation service as the following dependency. Notice the version is not defined since it is defined in the Quarkus BOM.

```
<dependency>
  <groupId>org.awaitility</groupId>
  <artifactId>awaitility</artifactId>
  <scope>test</scope>
</dependency>
```

The code of the new `org.acme.reservation.ReservationInvoiceProducerTest` is available in [Listing 9.21](#). Create this class in the `src/test/java` directory.

Listing 9.21 The source code of the ReservationInvoiceProducerTest in Reservation service

```

@QuarkusTest
@ApplicationScoped
public class ReservationInvoiceProducerTest {

    private final Map<Integer, Invoice> receivedInvoices = new HashMap<>();
    private final AtomicInteger ids = new AtomicInteger(0);

    @Incoming("invoices-rabbitmq")
    public void processInvoice(JsonObject json) {
        Invoice invoice = json.mapTo(Invoice.class);
        System.out.println("Received invoice " + invoice);

        receivedInvoices.put(ids.incrementAndGet(), invoice);
    }

    @Test
    public void testInvoiceProduced() throws Throwable {
        // Make a reservation request that sends the invoice to RabbitMQ
        Reservation reservation = new Reservation();
        reservation.carId = 1L;
        reservation.startDay = LocalDate.now().plusDays(1);
        reservation.endDay = reservation.startDay;

        given().body(reservation).contentType(MediaType.APPLICATION_JSON)
            .when().post("/reservation")
            .then().statusCode(200);

        Awaitility.await().atMost(15, TimeUnit.SECONDS)
            .until(() -> receivedInvoices.size() == 1);

        // Assert that the invoice message was received in this consumer
        Assertions.assertEquals(1, receivedInvoices.size());
        Assertions.assertEquals(ReservationResource.STANDARD_RATE_PER_DAY,
            receivedInvoices.get(1).price);
    }
}

```

In this test, we define an `@Incoming(`invoice-rabbitmq`)` method as a test consumer that verifies that the received message. The test method just makes an HTTP call for a new reservation that should produce a new invoice message into the `invoices` channel and verifies that the consumer method `processInvoice` was called with the correct data.

TIP Always remember to clean your Maven project (`mvn clean`) when you are removing classes.

We can now delete the `BillingService` as we no longer need it. Actually, we have to delete it because it is also sink that would conflict with the sink in the test. Also, in `application.properties` we now can prefix the messaging properties with `%test.` prefix, so they are picked only in the test mode. In other modes we default to `invoices` queue and exchange names.

```
%test.mp.messaging.incoming.invoices-rabbitmq.queue.name=invoices
%test.mp.messaging.incoming.invoices-rabbitmq.exchange.name=invoices
```

The test can be executed either in the Dev mode by pressing `r` with the focus in the terminal window where the Dev mode is running or simply by running `mvn clean test`. A RabbitMQ broker container is started in the background. The invoice is processed through the queue `invoices` the Reservation service creates once it connects to it.

9.8 Car Rental Billing service

Now that we're already producing invoices from both Reservation (RabbitMQ) and Rental (Kafka) services, it's time to create the actual consumer which is the Billing service.

Billing service is the last backend service in Car Rental. It's been a while since we created the last new Quarkus application, but you can probably remember that we can simply run the command available in [Listing 9.22](#). We also specify the required extensions that we will utilize to start working on billing functionality.

Listing 9.22 Creating new Billing service

```
$ quarkus create app org.acme:billing-service --no-code --extension
  -resteasy-reactive-jackson,
  -smallrye-reactive-messaging-kafka,smallrye-reactive-messaging-rabbitmq,
  -quarkus-mongodb-panache
```

We still want to expose REST API which is why we need `resteasy-reactive-jackson` extension. Next two extensions are required because Billing service consumes asynchronous messages through Kafka and RabbitMQ reactive messaging integrations. The last extension sets up MongoDB database integration.

We implement a simple model that allows persisting of the invoices and exposes them through HTTP. The following code is explained in detail in previous chapters, so we don't dive into specifics. The finished code of the Billing service is available in the `chapter-09/billing-service` directory.

The final representation of the Billing service can be summarized as:

- It has a single `org.acme.billing.model.Invoice` entity (`id` (`ObjectId`), `totalPrice` (`double`), `paid` (`boolean`), `reservation` (`Reservation`)).
 - Reservation can be a (public static final) inner class containing all parameters as in the Reservation service (`id` (`Long`), `userId` (`String`), `carId` (`Long`), `startDay` (`LocalDate`), `endDay` (`LocalDate`))
- The `org.acme.billing.InvoiceResource` exposes sole HTTP GET endpoint at `/invoice` supplying all available invoices.

One last thing is to set up the correct port for the Billing service which is expected to be running on port `8084` and the name of the MongoDB database. We can do this in the `application.properties`:

```
quarkus.http.port=8084
quarkus.mongodb.database=billing
```

You should be by now familiar with all of this code. If that's not the case, feel free to revisit previous chapters.

9.8.1 Reactive messaging in the Billing service

Billing service acts as a consumer for the messages coming from the Reservation service through the RabbitMQ queue called `invoices` and also as a consumer for the messages coming from the Rental service through the `invoices-adjust` Kafka topic. It also produces asynchronous messages to the Rental service that are also propagated through the Apache Kafka via the topic called `invoices-confirmations`. Visual representation of this communication is provided in [Figure 9.9](#). It contains the names of the Kafka topics and RabbitMQ queue, some of which we already implemented.

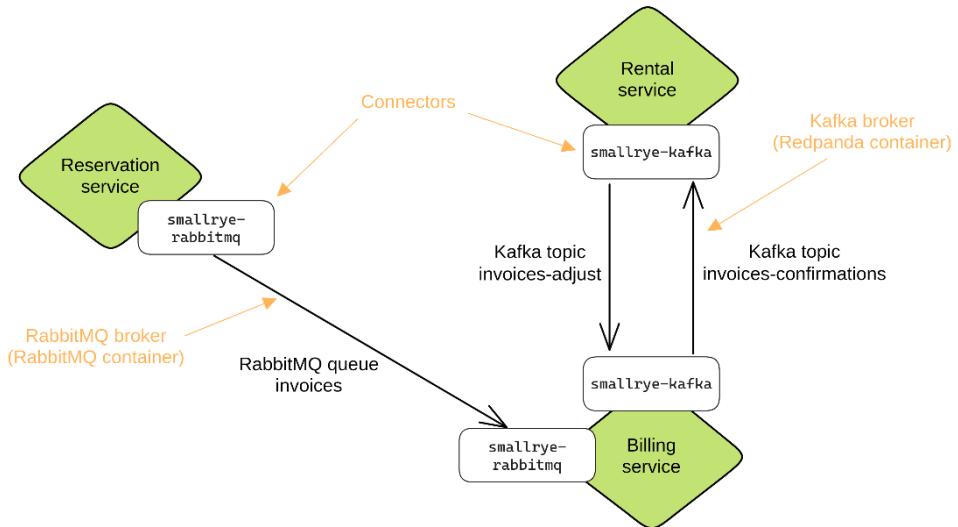


Figure 9.9 The asynchronous messaging architecture in the car rental system

The Kafka topic `invoices-confirmed` contains the messages that the Rental service can use to verify at the rental end that the reservation/rental has been paid for.

9.8.2 Consuming RabbitMQ messages from the Reservation service

The Reservation service sends invoice requests when the reservations are created to the RabbitMQ queue called `invoices`. To consume these messages in the Billing service, we need to first connect it to the same RabbitMQ broker and implement the consumer that processes the received invoice message and requests the payment.

The connection to the same broker is easy as long as we are using Dev Services since (similarly as with Kafka) any other Quarkus application started in Dev mode on the same machine automatically connects to the Dev Service instance of the RabbitMQ broker started by the first Quarkus application. Starting the Billing service in Dev mode, it automatically connects to RabbitMQ broker started by the Reservation service or vice versa.

We don't need to implement the actual logic that will request payments in our imaginary car rental system. Instead, we will just sleep for a random time period after which we consider the invoice paid. After the invoice is paid, we produce a new confirmation message to the new `invoices-confirmed` channel that maps into the Kafka topic.

Because the request payment operation is a blocking process (we need to wait for the user to pay or in our case for the sleep to end), we need to split the processing through one more channel called `invoices-requests` which allows us to move this blocking operation out of the non-blocking I/O thread. This in-memory channel consumes the produced invoices, request payments, and sends the confirmation messages if the payment is successful.

We start by defining the consuming message data class `org.acme.billing.data.ReservationInvoice`. As demonstrated in [Listing 9.23](#), it is exactly same as the invoice sent from the Reservation service.

Listing 9.23 The ReservationInvoice data class

```
package org.acme.billing.data;

import org.acme.billing.model.Invoice;

public class ReservationInvoice {

    public Invoice.Reservation reservation;
    public double price;
}
```

Next we create class `org.acme.billing.InvoiceProcessor` that uses consumes the `ReservationInvoice` messages. It looks as demonstrated in the [Listing 9.24](#).

Listing 9.24 The source code of the InvoiceProcessor class

```

@ApplicationScoped
public class InvoiceProcessor {

    @Incoming("invoices")
    @Outgoing("invoices-requests")
    public Message<Invoice> processInvoice(Message<JsonObject> message) {
        ReservationInvoice invoiceMessage = message
            .getPayload().mapTo(ReservationInvoice.class);
        Invoice.Reservation reservation = invoiceMessage.reservation;
        Invoice invoice = new Invoice(invoiceMessage.price,
            false, reservation);

        invoice.persist();
        Log.info("Processing invoice: " + invoice);

        return Message.of(invoice,
            () -> message.ack()); #1
    }
}

```

#1 Manually acknowledge the received message only after downstream ack.

The code of this class is straightforward. We define a processor method `processInvoice` that consumes the invoice request messages from the RabbitMQ connector (queue `invoices`). This method received the `Message` wrapper of the JSON object that represents our message. This wrapper allows us to manually acknowledge the message only after it is persisted into the database. This is a safeguard that guarantees we don't lose any invoices.

TIP The asynchronous messages in reactive messaging don't automatically trigger live reload of the Quarkus application running in Dev mode. When you finish your changes you need to manually restart the application by pressing `s` in the Dev mode terminal or with an HTTP request.

Because we have both connectors for Kafka and for RabbitMQ on the classpath (`smallrye-kafka` and `smallrye-rabbitmq` extensions), we need to also configure which connector should be used for the `invoices` channel in `application.properties`:

```
mp.messaging.incoming.invoices.connector=smallrye-rabbitmq
```

If you would now try to start a new reservation, you would not see anything happening in the Billing service because we didn't configure the `invoices-requests` sink yet. But the messages are already being consumed.

Let's implement the payment request processing. Create a new class `org.acme.billing.PaymentRequester` as showed in [Listing 9.25](#).

Listing 9.25 The source code of the PaymentRequester class

```
@ApplicationScoped
public class PaymentRequester {

    private final Random random = new Random();

    @Incoming("invoices-requests")
    @Outgoing("invoices-confirmations")
    @Blocking #1
    public InvoiceConfirmation requestPayment(Invoice invoice) {
        payment(invoice.reservation.userId, invoice.totalPrice, invoice);

        invoice.paid = true;
        invoice.update();
        Log.infof("Invoice %s is paid.", invoice);

        return new InvoiceConfirmation(invoice, true);
    }

    private void payment(String user, double price, Object data) {
        Log.infof("Request for payment user: %s, price: %f, data: %s",
            user, price, data);
        try {
            Thread.sleep(random.nextInt(1000, 5000));
        } catch (InterruptedException e) {
            Log.error("Sleep interrupted.", e);
        }
    }

    /* uncomment in order to consume confirmation here
    @Incoming("invoices-confirmations")
    public void consume(InvoiceConfirmation invoiceConfirmation) {
        System.out.println(invoiceConfirmation);
    }
    */
}
```

#1 We are executing a blocking operation in this method, so we must make sure it executes on the worker thread.

This processor sleeps for a random period between 1 and 5 seconds to simulate payment operations. After the sleep is done, it produces an `InvoiceConfirmation` message to the `invoices-confirmations` channel.

We also need the last data class called `InvoiceConfirmation` that we can send to the Rental service. Create this class in the `org.acme.billing.data` package.

Listing 9.26 The source code of the `org.acme.billing.data.InvoiceConfirmation` class

```
package org.acme.billing.data;

import org.acme.billing.model.Invoice;

public class InvoiceConfirmation {

    public Invoice invoice;
    public boolean paid;

    // all-arg constructor, toString
}
```

The information whether the invoice is paid or not is also contained in the invoice itself. But for the demonstration of the processor, we want to showcase returning a different data class than the one that is received.

The `invoices-confirmations` channel is not mapped into any connector for now. If you would like to test the functionality, you can uncomment the `consume` method in the `PaymentRequester` to see the invoices printed to the console when you create new reservations in the Reservation service.

9.8.3 Producing confirmations to Rental service

As demonstrated in the [Figure 9.9](#), we are expected to produce the invoice confirmations to the Rental service once the payment is processed. After last section, we are already producing messages into the `invoices-confirmations` channel. In order to send them to Apache Kafka, we need to configure this channel to use the `smallrye-kafka` connector.

If you are not already running the Rental service in Dev mode, start it. Depending on whether the Billing service is already running or not, either Rental service now automatically connects to the started Redpanda (Kafka) container or vice versa. However, we recommend starting the Billing service first, so it manages both Dev Services for RabbitMQ and for Kafka. It makes it easier for us to manage the Dev Services instances if the origin is in the same Dev mode.

It would be good to stop now a while and think about how much effort and tedious tasks the Quarkus's Dev mode takes off our shoulders. Simply by running the Dev mode, Dev Services automatically wire three of our services through two separate remote brokers that we don't need to manage. Can you think about how long it would take you to create this set up manually?

Now that both services are connected to the same Kafka instance, changing the production of the invoice confirmation to be sent to Kafka is as simple as setting the connector for the `invoices-confirmations` channel in the `application.properties` of the Billing service. Remember to also remove the optional consumer in the `PaymentRequester#consume` method if you opt to implement (uncomment) it since we can't consume in two sinks.

```
mp.messaging.outgoing.invoices-confirmations.connector=smallrye-kafka
```

Note the `outgoing` keyword in the configuration. This is important since we are producing messages to Kafka.

Billing service is now acting as a real invoice processing service. The last missing functionality is to integrate it also with the Rental service.

9.8.4 Rental service invoice confirmations

Rental service communicates with the Billing service in two ways. It consumes the invoice confirmations for created reservations and it also produces the invoice adjustment requests. All of this communication utilizes the Apache Kafka broker.

If the Rental service runs in Dev mode, it is already connected to the same Kafka instance as the Billing service. We can start integrating the invoice confirmations.

Because all of our invoice processing is asynchronous, we verify that the customer paid for the rental only when it is ending. At this point, we also request any invoice adjustments if the original rental isn't ended on the perceived day.

We start by adding the `paid` attribute to the `Rental` entity which we can use to track the payment confirmations. Modify the `org.acme.rental.entity.Rental` class as demonstrated in [Listing 9.27](#). The rest of the class doesn't need any changes.

Listing 9.27 Adding the paid attribute to the Rental entity

```
public class Rental extends PanacheMongoEntity {

    public boolean paid;
    ...
}
```

We can start consuming messages from the invoices-confirmations topic in the Kafka broker. Create a new class org.acme.rental.invoice.InvoiceConfirmationService as showed in [Listing 9.28](#).

Listing 9.28 The source code of the InvoiceConfirmationService which consumes messages from Kafka

```
@ApplicationScoped
public class InvoiceConfirmationService {

    @Incoming("invoices-confirmations")
    @Blocking #1
    public void invoicePaid(InvoiceConfirmation invoiceConfirmation) {
        Log.info("Received invoice confirmation " + invoiceConfirmation);

        if (!invoiceConfirmation.paid) {
            Log.warn("Received unpaid invoice confirmation - "
                    + invoiceConfirmation);
            // retry handling omitted
        }
    }

    InvoiceConfirmation.InvoiceReservation reservation =
        invoiceConfirmation.invoice.reservation;

    Rental.findByNameAndReservationIdsOptional(
        reservation.userId, reservation.id)
        .ifPresentOrElse(rental -> {
            // mark the already started rental as paid
            rental.paid = true;
            rental.update();
        }, () -> {
            // create new rental starting in the future
            Rental rental = new Rental();
            rental.userId = reservation.userId;
            rental.reservationId = reservation.id;
            rental.startDate = reservation.startDay;
            rental.active = false;
            rental.paid = true;
            rental.persist();
        });
    }
}
```

#1 We need the @Blocking annotation because we are executing database operations.

This consumer method first performs a check that tests if the invoice is really paid and if not, it logs a warning message. Retrying of the failed payments is beyond the scope of our example application. Next, we check if the rental covered by this invoice is already started by looking it up in the database. If it is already persisted, then it is already started without payment. In this case, we can just mark it as paid. If it is not in the database, then we received the invoice confirmation before the start of the rental. We can then persist it here as it represents an already paid rental.

We also need the data classes that can be mapped to the received invoices. To save some space, let's create a class `InvoiceConfirmation` in the `org.acme.rental.invoice.data` package that contains all data classes. Notice that we are not parsing all the data that is sent in the invoice.

- `InvoiceConfirmation (invoice (Invoice), paid (boolean))`
- `inner class Invoice (paid (boolean), reservation (InvoiceReservation))`
- `inner class InvoiceReservation (id (Long), userId (String), startDay (LocalDay))`

The full source code of this class is available in the `chapter-09` directory. We renamed the `reservation` class to `InvoiceReservation` so it can't be confused with the `Reservation` data class we use in the `ReservationClient`. But also note that the field must be named as `reservation` to correctly parse sent confirmations from the Billing service. We also included `toString` methods for each class.

Now we can adjust the `RentalResource` class to take into account already confirmed rentals. [Listing 9.29](#) details the `start` method changes that now need to check if the rental for a selected reservation is already persisted because the invoice confirmation was received before or if the new rental needs to be started as before.

Listing 9.29 Modifications in the RentalResource#start method to accommodate invoice confirmations

```

@Path("/start/{userId}/{reservationId}")
@POST
@Produces(MediaType.APPLICATION_JSON)
public Rental start(String userId,
                     Long reservationId) {
    Log.infof("Starting rental for %s with reservation %s",
              userId, reservationId);

    Optional<Rental> rentalOptional = Rental #1
        .findByUserAndReservationIdsOptional(userId, reservationId);

    Rental rental;
    if (rentalOptional.isPresent()) {
        // received confirmed invoice before
        rental = rentalOptional.get();
        rental.active = true;
        rental.update(); #2
    } else {
        // rental starting right now before payment
        rental = new Rental();
        rental.userId = userId;
        rental.reservationId = reservationId;
        rental.startDate = LocalDate.now();
        rental.active = true;
        rental.persist(); #3
    }
    return rental;
}

```

#1 Try to find the rental in the database.

#2 If found, update the active attribute.

#3 If not found, persist a new record that will be paid for in the future.

The `end` method is simpler. We check that the rental is paid, otherwise we let the customer know that there is a problem. The payment error processing is left out for simplicity.

Listing 9.30 Modifications in the RentalResource#end method to check payments

```

public Rental end(String userId, Long reservationId) {
    ...

    Rental rental = Rental
        .findByUserAndReservationIdsOptional(userId, reservationId)
        .orElseThrow(() -> new NotFoundException("Rental not found"));

    if (!rental.paid) {
        Log.warn("Rental is not paid: " + rental);
        // trigger error processing
    }

    Reservation reservation = reservationClient
        .getById(reservationId);
    ...
}

```

It was a lot of code, but we are now correctly handling the invoice confirmations in the Rental service. If you wish to take this a little bit further, you can write tests that verify the correct scenario handling as an exercise. In every case, you can try the functionality now by creating a reservation with the start day equal to today and with the one in the future. In both cases, eventually all rentals (GET :8082/rentals) are updated as paid.

9.8.5 Invoice adjustments in the Billing service

The last missing piece is to consume invoice adjustments sent from the Rental service in the Billing service if the customer returns the car on an unexpected date. The Rental service is already sending these messages to the Kafka topic called `invoices-adjust` which we implemented in section [9.6.1](#).

By now, you should know what needs to be done in the Billing service to consume these messages. So if you feel like it, try it first yourself as an exercise. You can always look back into this section if you need.

In the Billing service, we will save the adjustment invoices into a separate table. Let's create a new entity (`extends PanacheMongoEntity`) class `InvoiceAdjust` in the `org.acme.billing.model` package. We also recommend to add the `toString` method since we will use this entity in some log messages.

- `InvoiceAdjust (id (ObjectId), rentalId (String), userId (String), actualEndDate (LocalDate), price (double), paid (boolean))`

This time, we keep all the attributes that the Rental service sends for the tracking purposes in the Billing service. The invoice adjustments are the sole responsibility of the Billing service. Rental service doesn't need to be involved in the payment processing after it sends the adjustment message.

The processing of the invoice adjustment is similar to the invoice processing. We can add the new method `requestAdjustment` the `PaymentRequester` as presented in [Listing 9.31](#).

Listing 9.31 `PaymentRequester#requestAdjustment` method source code

```
@Incoming("invoices-adjust")
@Blocking
@Acknowledgment(Acknowledgment.Strategy.PRE_PROCESSING) #1
public void requestAdjustment(InvoiceAdjust invoiceAdjust) {
    Log.info("Received invoice adjustment: " + invoiceAdjust);

    payment(invoiceAdjust.userId, invoiceAdjust.price, invoiceAdjust);
    invoiceAdjust.paid = true;
    invoiceAdjust.persist();
    Log.infof("Invoice adjustment %s is paid.", invoiceAdjust);
}
```

#1 Change acknowledgment strategy in order to not block producer waiting for payment to be processed.

Lastly, we need to configure the new `invoices-adjust` channel to pull messages from the Kafka broker. Add the following configuration properties to the `application.properties` file in Billing service:

```
mp.messaging.incoming.invoices-adjust.connector=smallrye-kafka
mp.messaging.incoming.invoices-adjust.auto.offset.reset=earliest
```

The `earliest` offset allows us to reload unprocessed messages from Kafka in a case of crashes.

The Billing service is now able to consume invoice adjustments. Feel free to experiment a little with the provided APIs. Try creating reservations starting today and in the future. Also try to set the rental end date in future and end it manually in the Rental service to see the adjustment processing in the Billing service.

The log statements nicely demonstrate the flow of asynchronous messages. When you create a reservation, you can see in the Billing service how the payment is being processed. Once finished, you can see the messages in the Rental service received either invoice confirmation or adjustment.

So why would you choose to implement your application with reactive messaging? The whole payment backend processing is asynchronous and extremely reliable. Try to push several reservations in quick succession or try to stop any service (that is not owning the Dev Service instances!) and push some messages to other running services. Rental service is a good choice because it is at the end of the message chain. Use the start day in future because Reservation also calls Rental via HTTP if the start day is today. This shows the nice comparison to asynchronous message passing in this scenario. You will see that both RabbitMQ and Kafka keep track of unprocessed (not yet acknowledged) messages. When the service comes back up, it can still process the messages that were sent while it was down.

9.9 Wrap up and next steps

This chapter introduced the MicroProfile Reactive Messaging specification together with the SmallRye Reactive Messaging library. We defined the reactive systems and their properties that are becoming necessary for any modern enterprise application with the asynchronous message passing as the base pillar of such systems.

Next, we investigated the APIs provided by the MicroProfile Reactive Messaging specification. We learned about connectors and how they allow us to connect to different messaging providers.

Lastly, we created a new Billing service that communicates with Reservation service through the RabbitMQ broker via the AMQP 0-9-1 protocol. It also exchanges asynchronous messages with the Rental service through the Apache Kafka platform.

There was a lot of coding in this chapter. However, it was fun! We learned that plugin a reactive pipeline as a connection between two Quarkus services is rather easy. With two annotations stating the same channel name, a little bit of configuration, and the power of Dev services, you can write reactive services with ease.

9.10 Summary

- The reactive manifesto defines reactive systems as applications that are responsive, elastic, resilient, and based on asynchronous message passing.
- MicroProfile Reactive Messaging is a specification defining a simple API for the asynchronous message passing based on reactive streams. Message flow through channels defined as methods in CDI beans using `@Incoming` and `@Outgoing` annotations.
- Quarkus uses SmallRye Reactive Messaging library that implements the specification, but it also provides useful features which are not in the specification yet.
- Connectors are SPI bridges that allow production and consumption of messages from remote services as for instance, Kafka or RabbitMQ broker.

- SmallRye Reactive Messaging connectors are packaged in separate extensions. They also provide Dev Services for the remote brokers which the connector bridges.
- Applications started in Dev mode automatically connect to the remote brokers started as Dev Services. Development of Quarkus applications utilizing reactive messaging doesn't require user intervention, but it is possible to change the configuration if needed.

10

Cloud-native application patterns

This chapter covers

- Introducing the MicroProfile set of specifications for cloud-native Java applications
- Monitoring the status and health of applications
- Collecting and visualizing application metrics
- Tracing requests within and across services
- Adding fault tolerance capabilities to applications

We've repeated multiple times that Quarkus is a framework for building cloud-native applications. But what does it mean exactly? What are the characteristics of cloud-native applications? What are the patterns that are commonly used to build them?

In this chapter, we'll answer these questions and show you how to implement these patterns in Quarkus applications. We also look into the related tools Quarkus offers to make these integrations easier. In the next chapter, we will expand on what we learn here by actually deploying the full Acme Car Rental in the cloud and seeing how all these functionalities neatly work together.

The main characteristics that we discuss in this chapter are:

- **Health:** Applications should expose information about their health so that various kinds of their problems can be detected (either by a human operator or automatically). An application is unhealthy if some of its components or external connections don't work as needed - for example, if a database connection isn't working, a deadlock is detected, or the memory heap is full. Human operators or automated tools can then respond to this information and take appropriate action, such as restarting the application.
- **Metrics:** Metrics are similar to health checks in that they convey information about the status of an application. However, unlike health checks whose outcome is binary (good or bad), metrics are numeric values. They can tell how much memory is used, how many threads in a thread pool are usually busy, how many requests are being processed per minute, and so on. You may create automated alerts for metrics, for example to receive an email when an application's memory usage exceeds a certain threshold.
- **Tracing:** Tracing allows you to follow (visually) a request as it flows through multiple components and services. This is useful for troubleshooting purposes and can help to find performance bottlenecks.
- **Fault tolerance:** Fault tolerance allows applications to gracefully deal with failures. For example, it can retry a request to a failing external service instead of propagating the failure to the caller.
- **Service discovery:** Service discovery is a way of dynamically finding the location of services that your application needs to interact with, allowing you to decouple the location of those services from your configuration. This makes deployments more robust and adaptable to changes.

TIP Health, metrics, and tracing (and also logging) are very often together called *observability*. This term refers to characteristics of an application that make it possible to observe its state while it's running.

The practical sections of this chapter focus on updating the Inventory service to add the capabilities that were outlined in the bullet list.

But first, we should talk about MicroProfile. MicroProfile is a set of specifications that define a programming model for building cloud-native Java applications, including the above-mentioned capabilities. We have already worked with a few MicroProfile specifications in previous chapters, such as the REST Client, GraphQL and OpenAPI, and we explore more of them in this chapter.

TIP Health, Tracing, Metrics, and Fault Tolerance are part of MicroProfile APIs. Service Discovery is not part of it at the time of writing, but may well be in the future.

10.1 MicroProfile, SmallRye and Quarkus

As we said, MicroProfile (<https://micropattern.io/>) is a set of specifications. It belongs under the Eclipse Foundation umbrella, and it's a community-driven effort. All specifications are available as open source on GitHub (under the Eclipse organization: <https://github.com/eclipse/>).

Under the MicroProfile umbrella, you will actually find two sets of specifications. The first set is the so-called MicroProfile platform. Just like the Jakarta EE platform, this is a set of specifications that are released together under a single version number (each specification has its own versioning though, and a platform release bundles a particular set of specification versions together).

The second set is the individual specifications that aren't part of the platform, usually referred to as "standalone" specifications. This includes, for example, the GraphQL specification that we've used already in previous chapters.

As of finishing this book, the latest available version of the MicroProfile platform is 6.0. Quarkus 3.2 aligns with this MicroProfile version, with one exception (Metrics, we will discuss this in the relevant section later).

The MicroProfile 6.0 platform contains these specifications:

- Telemetry 1.0
- OpenAPI 3.1
- REST Client 3.0
- Config 3.0
- Fault Tolerance 4.0
- Metrics 5.0
- JWT Authentication 2.1
- Health 4.0

Quarkus 3.2 also supports these standalone specifications:

- LRA 2.0
- GraphQL 2.0
- Context Propagation 1.3
- Reactive Messaging 3.0
- Reactive Streams Operators 3.0
- OpenTracing 3.0

Quarkus' compatibility with each specification is verified using a so-called Technology Compatibility Kit (TCK). This is a set of tests that verify that the implementation behaves as specified. The TCKs are provided as part of the MicroProfile specifications, and Quarkus runs them as part of its own build.

10.1.1 What is SmallRye?

SmallRye (<https://smallrye.io/>) is a set of implementations of the MicroProfile specifications. We said that Quarkus is also a MicroProfile implementation, so this begs for a bit of explanation. Each SmallRye project implements one of the MicroProfile specifications. Quarkus then provides an extension per each SmallRye project that integrates the SmallRye implementation into Quarkus and allows you to use it in your applications.

The Quarkus extension makes sure that the project is well integrated into the overall Quarkus architecture, for example by providing relevant `quarkus.*` configuration properties, and by contributing some related logic into the build-time application initialization mechanism.

10.2 Monitoring application health

In cloud deployments (and in computing in general), all kinds of things can go wrong. A network connection can fail, a database can become unavailable, a disk can fill up. It's important to be able to detect these problems and react to them. MicroProfile Health is one of the tools that you may use for this. It allows you to expose information about the status of your application via an HTTP endpoint. That endpoint can then be periodically monitored by various tools. For example, in a Kubernetes deployment, it typically serves as the endpoint for the Kubernetes liveness, readiness, and startup probes. These probes run periodically (in configured intervals) and when the application reports problems, Kubernetes can react to that, for example by restarting the pod, or refraining from sending traffic to it until health problems are resolved.

With MicroProfile Health, there are 5 different kinds of health checks.

- **Liveness**—This is usually used as the target of Kubernetes liveness probes. When an application's liveness check fails, it signifies that the application has had an error it is not expected to recover from, like running out of memory, or detecting a deadlock. Kubernetes reacts by restarting the pod.
- **Readiness**—This is usually used as the target of Kubernetes readiness probes. When a readiness check fails, it signifies that the application is not ready to receive traffic, for example because a required database connection isn't working, or because the application has detected that the amount of current active tasks is above a certain threshold. Kubernetes will react by holding off sending further traffic to the pod until the condition is resolved.

- **Startup** — This is usually used as the target of Kubernetes startup probes. The startup check should start passing when the application is initialized and ready to start receiving traffic. This check is only used during the initial startup of the application, and after it passes once, it is not called anymore - the liveness and readiness checks take over its responsibility.
- **Wellness** — This is actually not a part of the MicroProfile Health specification, but it is added as an extra feature by SmallRye Health. The wellness check conveys potential health problems that might affect the application, but are not severe enough to warrant a restart or a traffic hold-off. This can be useful for reporting data to custom monitoring tools, or manual monitoring by human operators.
- **Custom health groups** — These are any custom user-defined health groups (types of checks) that can have arbitrary use. They are also not yet defined in the MicroProfile Health specification.

10.2.1 Monitoring health of Car Rental

Let's see some ways to monitor the health of the Car Rental applications. We start by verifying the functionality of the Inventory service's database connection - this is a check that Quarkus provides automatically, so it is very easy. After that, we will learn how we can add a custom health check.

DATABASE CONNECTION HEALTH CHECK

When your application uses a JDBC data source, Quarkus automatically adds a readiness health check that verifies that the database connection is working. When the database is unavailable, the readiness check fails, saying that the application can't process requests right now. Let's try it out with the Inventory service with its MySQL connection.

Navigate to the `inventory-service` project's directory and add the `quarkus-smallrye-health` extension. Depending on whether you're using the Maven plugin or the CLI:

```
$ quarkus extension add smallrye-health
```

or

```
$ mvn quarkus:add-extension -Dextensions=smallrye-health
```

Run the `inventory-service` project in Dev mode. This automatically starts a MySQL database instance via Dev Services as a container.

To see the health check's status, we have two options. One is to view the Health UI which is part of the Dev UI. Navigate to it either by opening <http://localhost:8083/q/dev-ui> in a browser, or by pressing the 'd' key in the terminal window where you started the application. In the Dev UI, locate the card titled "SmallRye Health" and click the "Health UI" link inside that card (see [Figure 10.1](#)).

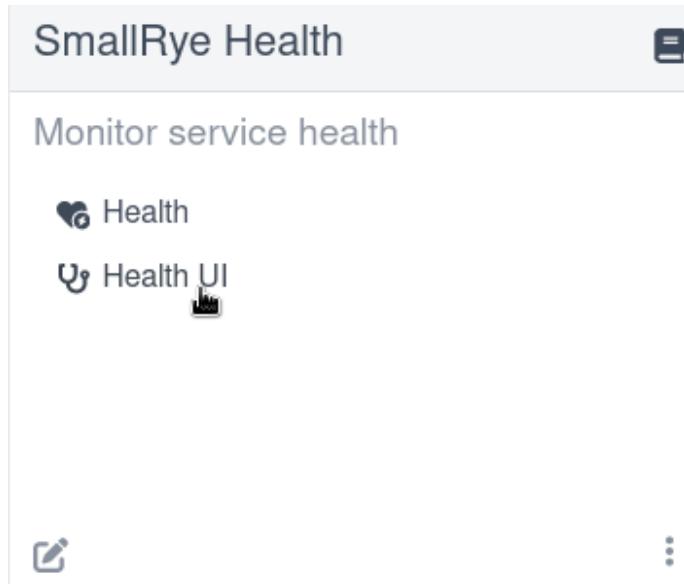


Figure 10.1 SmallRye Health card in the Dev UI

In the Health UI, you will see a health check with a title "Database connections health check" with a status "UP", just like in [Figure 10.2](#). There is probably also at least one other check that is related to the gRPC server that is embedded in the application. This page shows all health checks together, regardless of their type.

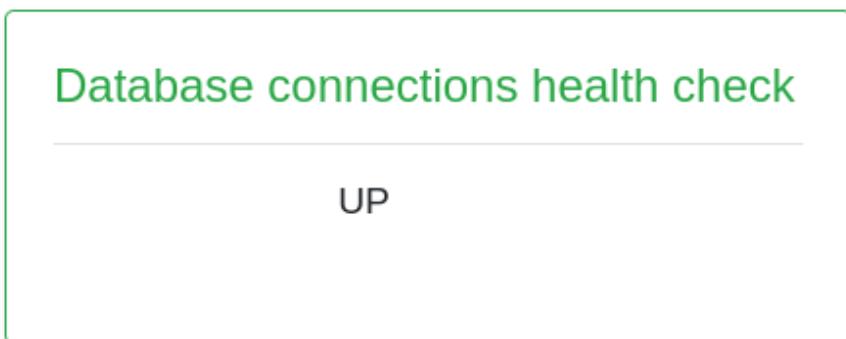


Figure 10.2 Health UI card for a passing database health check

Let's also try to access the check via the HTTP endpoint directly. Open <http://localhost:8083/q/health/live>, either via a browser, or your favorite command-line tool. You should see the raw JSON output of all liveness checks, like in [Listing 10.1](#).

Listing 10.1 JSON output from the endpoint for liveness checks

```
{
  "status": "UP",
  "checks": [
    {
      "name": "gRPC Server",
      "status": "UP",
      "data": {
        "grpc.health.v1.Health": true,
        "inventory.InventoryService": true
      }
    },
    {
      "name": "Database connections health check",
      "status": "UP",
      "data": {
        "<default>": "UP" #1
      }
    }
  ]
}
```

#1 Each data source is listed separately. We have only one data source, and it is named "<default>".

TIP By appending `/live` at the end, we specify that we're only interested in liveness checks. <http://localhost:8083/q/health> would return all checks grouped by their type.

Now let's deliberately crash the database. Use your container runtime tooling to find the ID of the running MySQL container, and stop the container. For example, with podman:

```
$ podman ps --format "{{.ID}} {{.Image}}"
90187f3f0451 docker.io/library/mysql:8.0
$ podman stop 90187f3f0451
```

Now request the health check again. If you do it in the Dev UI, the card of the check turns red, as shown in [Figure 10.3](#).

Database connections health check

Unable to execute the validation
check for the default DataSource:
Socket fail to connect to
host:address=(host=localhost)
(port=44461)(type=primary).
Connection refused

Figure 10.3 Health UI card for a failing database health check

On the HTTP endpoint, you will see the equivalent in raw JSON, as shown in [Listing 10.2](#). The gRPC check was omitted from this listing for brevity.

Listing 10.2 JSON output with a failing database check

```
{
  "status": "DOWN",
  "checks": [
    {
      "name": "Database connections health check",
      "status": "DOWN",
      "data": {
        "<default>": "Unable to execute the validation check
- for the default DataSource: Socket fail to connect to host:address=
-(host=localhost)(port=44461)(type=primary). Connection refused"
      }
    }
  ]
}
```

To bring the database back up, execute (replace the container ID with the same that you used above):

```
$ podman start 90187f3f0451
```

The health check should now pass again.

CUSTOM HEALTH CHECK

Now let's add a custom health check to the Inventory service. We will use a very simple wellness check that shows a status of "DOWN" if it is detected that the car inventory is empty. An empty inventory does not necessarily mean that the application itself is broken, but it surely is a sign that something is wrong, so a wellness check is probably the correct type for this check. Add the class that implements the check, it will be named `CarCountCheck` and reside in the `org.acme.inventory.health` package:

Listing 10.3 Wellness check that reports DOWN if the car inventory is empty

```
@Wellness
public class CarCountCheck implements HealthCheck {

    @Inject
    CarRepository carRepository;
    @Override
    public HealthCheckResponse call() {
        long carsCount = carRepository.findAll().count();
        boolean wellnessStatus = carsCount > 0;
        return HealthCheckResponse.builder()
            .name("car-count-check")
            .status(wellnessStatus) #1
            .WithData("cars-count", carsCount) #2
            .build();
    }
}
```

#1 Set the status to UP if there is at least one car.

#2 Report the number of cars as additional data.

After your application starts up, there should actually be two cars imported automatically, so the check should pass, as you can verify by viewing the Health UI, or calling <http://localhost:8083/q/health/well> (the `/well` suffix means that you only want to see wellness checks). The output should look like in [Listing 10.4](#).

Listing 10.4 JSON output of the wellness check

```
{
  "status": "UP",
  "checks": [
    {
      "name": "car-count-check",
      "status": "UP",
      "data": {
        "cars-count": 2
      }
    }
  ]
}
```

To make the check fail, you need to delete both cars from the database. You may use either the GraphQL endpoint at <http://localhost:8083/graphqI-ui/> or the gRPC CLI that we created in Chapter 4. For the CLI, navigate to the `inventory-cli` project's directory, make sure the project is built (`mvn package`) and execute:

```
$ java -jar target/quarkus-app/quarkus-run.jar remove ABC123
$ java -jar target/quarkus-app/quarkus-run.jar remove XYZ987
```

Now request the wellness check again. You should see the output as shown in [Listing 10.5](#).

Listing 10.5 JSON output of the wellness check when the inventory is empty

```
{
  "status": "DOWN",
  "checks": [
    {
      "name": "car-count-check",
      "status": "DOWN",
      "data": {
        "cars-count": 0
      }
    }
  ]
}
```

If you want to re-add the initial two cars into the database, force a hot reload by pressing 's' in the terminal where the Inventory service is running.

10.3 Application metrics

Metrics provide additional insight into the well-being of a running application. Compared to health checks, which have a binary outcome (up or down), metrics are numeric values, and thus can be used for plotting graphs, for example.

The typical usage of metrics in cloud deployments is to have applications periodically report their metrics to a central monitoring system, such as Grafana, which then can plot graphs for visualization, and raise automatic alerts when certain defined thresholds are exceeded, for example if an application starts using too much memory, or its response time becomes too high.

Generally, we can distinguish two kinds of metrics: framework metrics and application-specific (business) metrics. Framework metrics are usually provided by the runtime automatically, and they track general information about the running application, such as:

- The number of requests received by the application per time unit
- Heap memory consumption
- Number of active database connections
- Time that it takes to handle one request
- Occupancy of threads in a thread pool
- Number of messages waiting in a queue to be processed

Business metrics are provided by the application itself, and usually track something specific from the domain of the application. For example, in a web shop, you might want to track the number of orders placed per time unit.

10.3.1 Micrometer

The recommended way to work with metrics in Quarkus is to use the Micrometer extension. Micrometer (<https://micrometer.io/>) is a library that provides a common API for working with metrics, and it supports submitting metrics to many different monitoring backends, such as Prometheus, Graphite, SignalFX or Datadog.

TIP The MicroProfile 6.0 platform, which Quarkus 3.2 is aligned with, contains the MicroProfile Metrics 5.0 specification. This is one exception that Quarkus does not support, and instead, it supports Metrics 4.0 only. The reason is that the Quarkus developers have decided to use a slightly different approach, and to prefer the Micrometer extension as the way for working with metrics. There is a built-in SmallRye Metrics extension that is compatible MicroProfile Metrics 4.0, but it's not recommended to use it. Instead, the Micrometer extension is preferred as the way forward.

10.3.2 Metric types

Micrometer supports these types of metrics:

- **Timer**—Measures the time that it takes to execute a certain operation, and generates a histogram of the measured times
- **Counter**—A simple counter that can only be incremented, it counts the number of occurrences of a particular event.
- **Gauge**—A single numeric value that can change in any direction over time. An example would be the heap memory usage of the application.
- **Distribution summary**—A histogram of numeric values, just like a timer, but not necessarily tied to time-related measurements.

Counters and gauges are single-value metrics, while timers and distributions consist of multiple values. By default, they contain three values - `max` for the maximum value in, `count` for the number of values, and `sum` for the sum of all values. They can also be configured to, for example, publish individual percentiles.

TIP In context of Micrometer, and in its documentation, the term "meter" is more commonly used than "metric". On the other hand, in the MicroProfile space, "metric" is the more common term. We will use the term "metric" in this book.

10.3.3 Metric dimensionality

To allow for more flexibility in querying and generating useful insights from metrics, they can be dimensional. This means that a metric is not identified by its name only, but also by a set of key-value pairs called tags. Metrics that are related to each other can share the same name, but have different tags. For example, to count the HTTP requests coming to a server, you can have a set of counters under the name `http_requests` with these tags:

- `method`: the HTTP method used for the request, such as `GET` or `POST`
- `path`: the path of the request, such as `/api/orders`
- `status`: the status code of the response, such as `200` or `404`

Each possible combination of tag values is associated with a separate counter. With this naming strategy you can, for example:

- Query the number of requests for a particular HTTP method, by aggregating together all values where the `method` tag contains the desired method name.
- Query the number of requests for a particular path that resulted in a particular status code, by aggregating all metrics where both the `path` and `status` tags have the desired values.

You will see that this is what Quarkus does for most of its built-in metrics, but of course it's possible (and recommended) to add dimensions to your business metrics too, when applicable.

10.3.4 The programming model

We said that there are two types of metrics - framework and business (application-specific) metrics, and that framework metrics are built-in to the runtime, so they don't need any coding, only configuration. So, how do we work with business metrics, how can we collect them?

With the Micrometer library and Quarkus, there are two possible approaches - imperative (programmatic) and declarative (using annotations). The annotation-based approach is much easier and less verbose, on the other hand the imperative approach is more flexible and allows for some more advanced use cases.

To declare a simple counter metric with an annotation, the only thing that you need to do is to put a `@io.micrometer.core.annotation.Counted` annotation on a method of a CDI bean:

```
@Counted(name = "orders", description = "Number of orders placed")
public void orderPlaced(OrderDetails details) {
    // order processing logic
}
```

This will automatically register a counter metric named `orders` and attach an interceptor to your method, and that interceptor will take care of incrementing the counter's value by one whenever the method is called.

To do the equivalent using the imperative approach, you would need to do the following. The `MeterRegistry` imports from the `io.micrometer.core.instrument` package:

```
@Inject
MeterRegistry meterRegistry;

public void orderPlaced(OrderDetails details) {
    // order processing logic
    meterRegistry.counter("orders").increment();
}
```

The method `meterRegistry.counter(...)` has a get-or-create semantic, so it will automatically create a counter of the given name if it doesn't exist yet, and then return it. The `MeterRegistry` is the global registry of all metrics that are present in your application, and allows dynamically adding and removing metrics.

TIP In the previous example, we are not specifying the description of the metric. If you want to do that, you need to use the `Counter.builder(...)` method to create the metric manually with all its associated metadata, and then register it in the registry. This needs to happen just once, before any updates to the metric start happening. For example, you may do this inside a `@PostConstruct` method of a `@Singleton` bean (a `@Singleton` bean is a CDI bean that creates only one instance and that instance is created automatically immediately when the application starts).

10.3.5 Prometheus and Grafana

So now that we have explained the programmatic side of collecting metrics, we need to discuss what happens to them after they are collected. Generally, metrics from running applications are periodically fed into a monitoring system that stores them, processes them and allows humans to query them, potentially also by visualizing them in a dashboard. There are many monitoring systems available, one of the most popular solutions in the cloud-native world is Prometheus (<https://prometheus.io/>) along with Grafana (<https://grafana.com/>). This is the combination that we will use in this book.

Prometheus is an engine for collecting, storing and querying metrics. Grafana is a dashboarding tool that can query the metrics from Prometheus and display them in a dashboard.

Prometheus works by periodically scraping metrics from the application, that means it periodically makes HTTP requests to an application's endpoint. Prometheus prescribes a certain format that the endpoint needs to return. In the context of Quarkus, this endpoint is served by the `micrometer` extension and is available (by default) at `/q/metrics`. The endpoint contains a separate line for each metric value.

The complete architecture of a deployment with Prometheus and Grafana looks like presented in [Figure 10.4](#).

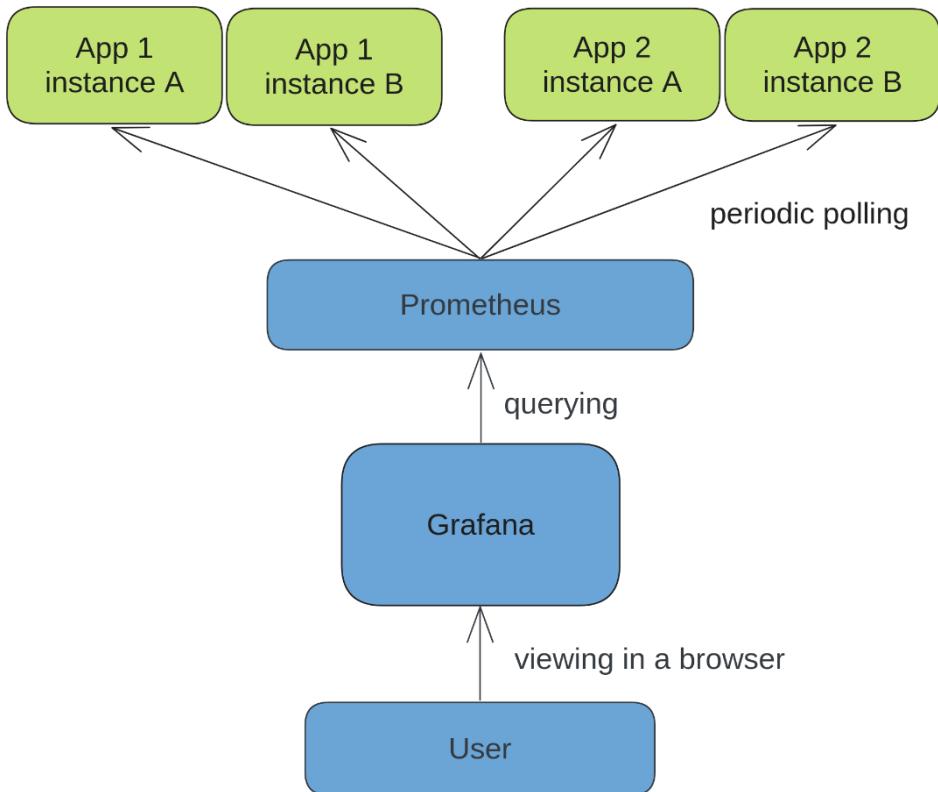


Figure 10.4 Typical deployment for monitoring metrics with Quarkus, Prometheus and Grafana

10.3.6 Monitoring metrics of Car Rental

Enough theory, let's jump to the practical part and collect some metrics related to the inventory service. First, let's look at metrics that Quarkus provides out-of-the-box and how to visualize them with Grafana, and then, we will create some custom ones.

The first thing that we need to do is add the `quarkus-micrometer` extension to the Inventory service. Go to the `inventory` directory and run either:

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

or

```
$ quarkus extension add micrometer
```

From now on, whenever you start the application, it collects and exposes some metrics. Let's try it out. Start the application in dev mode. Then call the `/q/metrics` endpoint that presents all registered metrics along with their values. If you're using cURL, you can do it like this:

```
$ curl http://localhost:8083/q/metrics
```

Another option is to open the Dev UI and click the 'Prometheus' link inside the panel titled 'Micrometer metrics'.

In the output, you will see a lot of metrics (it should be well over 100 lines). These are all the metrics that are collected by Quarkus by default when Micrometer is enabled. Some lines are prefixed with `# TYPE`, these are metadata lines that describe the type of the metric (counter, gauge, etc.). Some lines are prefixed with `# HELP`, these are metadata lines that show the description of the particular metric, if it has one. Metrics that don't start with `#` are the actual metric data.

As just one example, these lines are related to the metric describing how long the JVM process has been running:

Listing 10.6 Lines related to the process.uptime metric

```
# HELP process_uptime_seconds The uptime of the Java virtual machine
# TYPE process_uptime_seconds gauge
process_uptime_seconds 128.22
```

VISUALIZING IN GRAFANA

Now that we have some metrics, let's visualize them in Grafana, because reading them in the raw form is not very convenient. To run Prometheus and Grafana, we have prepared a Docker Compose file that should set up everything you need, and we won't dive into the details how this is all configured. You can find the file in the Git repository as `chapter-10/inventory-service/docker-compose-metrics.yml`. Start it by executing:

```
docker-compose -f docker-compose-metrics.yml up
```

If everything succeeds, Grafana is available at <http://localhost:3000/> and Prometheus at <http://localhost:9090/>. We don't need to look into the Prometheus UI, Prometheus runs just to serve as the metric storage engine that is used under the hood by Grafana. So, now open your browser and navigate to <http://localhost:3000/>. At the login screen, use `admin` as both the username and the password. If you are asked to change the password, you can press 'Skip' to ignore this prompt.

Open the main menu on the left, and click 'Dashboards' to view a list of dashboards that are currently registered in Grafana. See [Figure 10.5](#) for a screenshot of the main menu.

TIP To make the screenshots more suitable for including in a book, we are using the light theme of Grafana. The default theme is dark, so your UI might look different. To switch to the light theme, open the main menu, then 'Administration', then 'Default preferences', then select 'Light' from the 'UI Theme' selector, and then hit the 'Save' button.

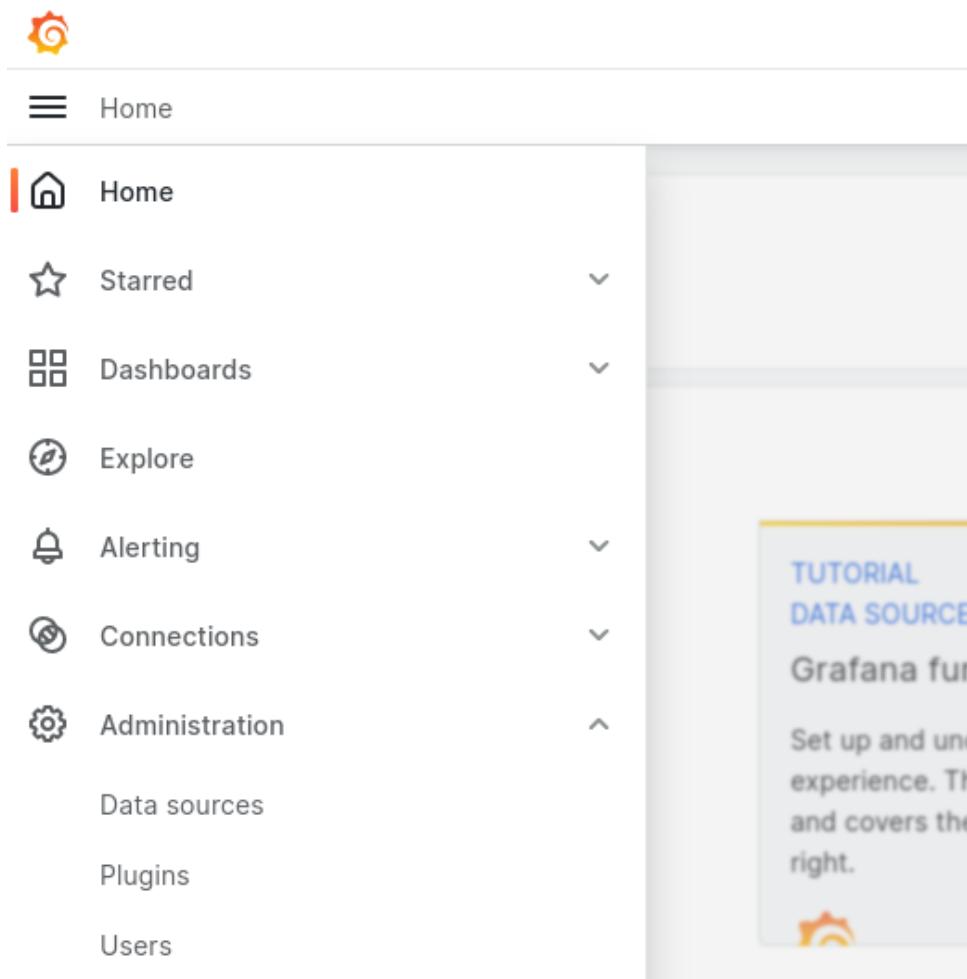


Figure 10.5 Navigating the main menu in Grafana

In the list of dashboards, there should be one dashboard, and it's named 'Inventory service'. Click on it to open it. For this example, we have prepared for you a dashboard that contains three metrics:

- Car registrations: a line chart with the number of cars that have been added into the inventory using the GraphQL endpoint (the `register` mutation). We haven't implemented this metric yet, so the chart probably isn't showing any data for you right now.
- Process uptime: a single-value panel that shows the value of the `process.uptime` metric that we saw earlier.
- Process CPU usage: a line chart showing the values of the `process.cpu.usage` metric over time.

A screenshot of the dashboard is shown in [Figure 10.6](#).

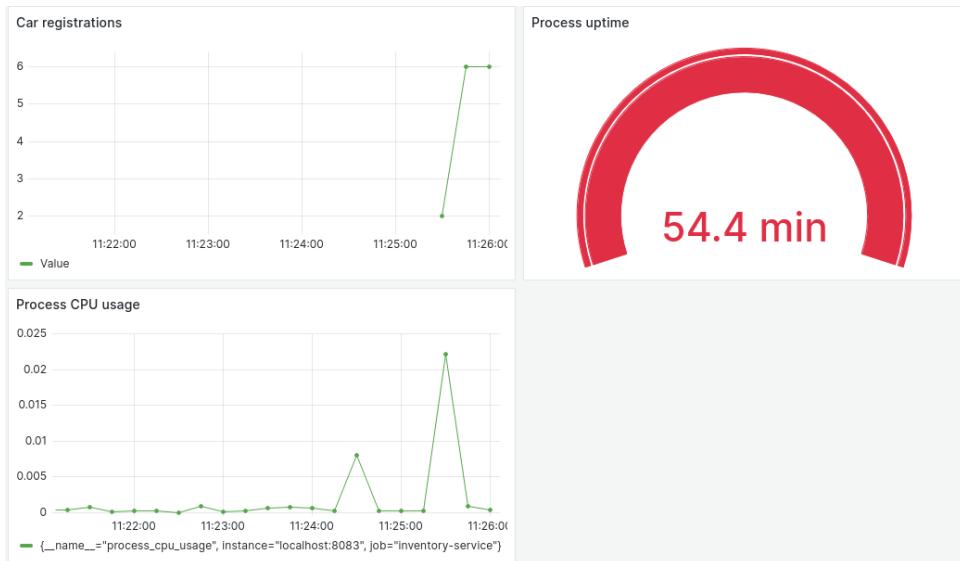


Figure 10.6 The Inventory service dashboard in Grafana

All the charts get updated automatically over time. Grafana is configured to query Prometheus for the current data every 10 seconds, and this is the same interval that Prometheus is configured to query the application.

Try playing around with the dashboard. For example, in the top-right corner, there is a dropdown allowing you to select the time range from which you want to see data. The default is 'Last 5 minutes', but you can select any time range in the past (if there's data for it). Existing visualizations can be rearranged with your mouse. New visualizations can be added under the blue 'Add' button in the top right corner. Grafana knows what metrics are exposed by your application, so it will offer you a list that you can choose from. If you choose a multi-dimensional metric, the generated chart will contain multiple data series, one for each dimension. An example is `jvm_memory_used_bytes` that contains the size of used memory in each JVM memory pool (Eden space, Old gen, Survivor space,...) separately. As an exercise, you may try to create a visualization of the `jvm_memory_used_bytes` that might look like in [Figure 10.7](#). Under the options of the time series, select the unit (bytes) to let Grafana know how to convert the values automatically to something more convenient, like megabytes instead of bytes.

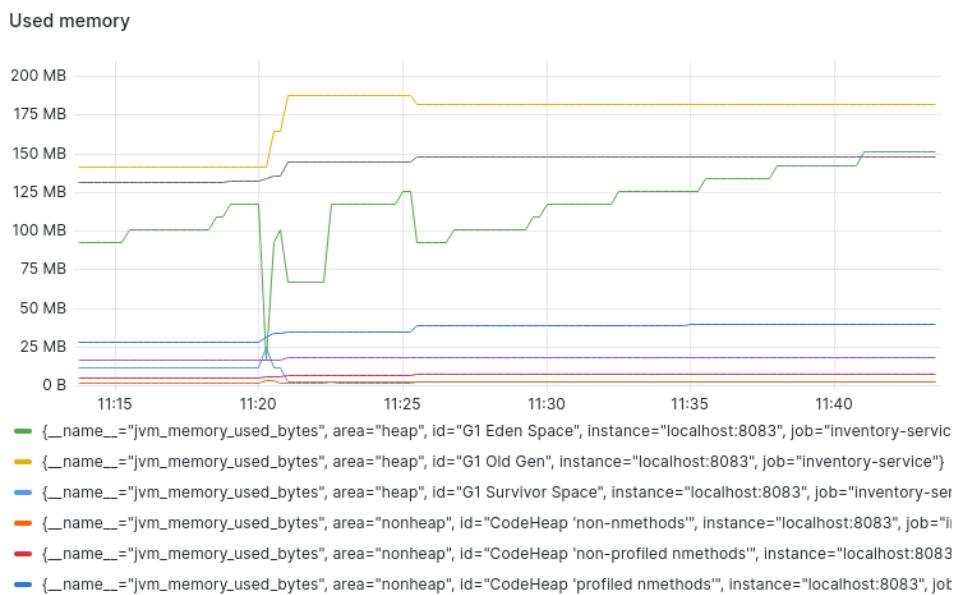


Figure 10.7 Visualizing JVM memory usage in Grafana

CUSTOM METRICS

Now let's get back to the metric for car registrations. This is not a built-in metric provided by Quarkus out-of-the-box, we have to define it manually. As we already discussed, for application-specific metrics, we have two approaches - programmatic (using the `MeterRegistry` object) and declarative (with annotations). Given that in this case, it's just a matter of counting invocations of a single method, we can use the annotation approach.

Open the class with the GraphQL endpoint for managing the inventory, `org.acme.inventory.service.GraphQLInventoryService`. Now simply annotate the `register` method with:

```
@Counted(description = "Number of car registrations")
```

The correct import for this annotation is `io.micrometer.core.annotation.Counted`. And that's it. But before you call the `/q/metrics` endpoint to view it, you have to call the `register` method at least once, because the counter is registered lazily when the owning bean is first instantiated. Open the GraphQL UI at <http://localhost:8083/q/graphql-ui/> and execute the following mutation:

```
mutation {
  register(car: {
    licensePlateNumber: "123"
  }) {
    id
  }
}
```

The metric should now appear in the `/q/metrics` endpoint as well as start showing some actual values in the Grafana chart. The raw line to look for, if you're looking at the `/q/metrics` endpoint, starts with `method_counted_total` prefix. It's actually a multidimensional metric, every counter created via a `@Counted` annotation will correspond to multiple data series. The dimensions are:

- `class`: the fully qualified name of the class where the counted method resides
- `method`: the name of the counted method
- `exception`: the name of the exception that was thrown from the counted method
- `result`: the result of the counted method (success or failure)

So, in our example, successful invocations of the `register` method will count towards the data series that is defined as

```
method_counted_total{class="org.acme.inventory.service.
-GraphQLInventoryService",exception="none",
-method="register",result="success"}
```

10.4 Tracing

One of the challenges of microservices is the complexity of the flows between multiple services. A single HTTP request from a client might trigger a relatively large chain of invocations between services. When something goes wrong in that chain (be it a failure or a performance issue), it's not exactly easy to trace the flow of the request and find out where the problem is. This is where distributed tracing comes in. Distributed tracing allows users to follow the flow of a request inside the components of a single service (allowing you to see which methods are invoked), as well as between services, and visualize it for easier troubleshooting. Not only you can see where the request goes, but also how long each part of the processing takes, so it's easier to spot performance issues too.

This is achieved by adding a unique identifier to each request - a trace ID. Each incoming request into a system corresponds to one trace. A trace consists of spans, where a span represents a single unit of work within a trace. A span typically corresponds to a method invocation when tracing the inside of a single service, it can also be a database invocation, or a remote request when tracing communication between services. Just like a trace, each span also has its unique ID. Spans are typically nested, so they form a tree.

An example of a trace is shown in [Figure 10.8](#).

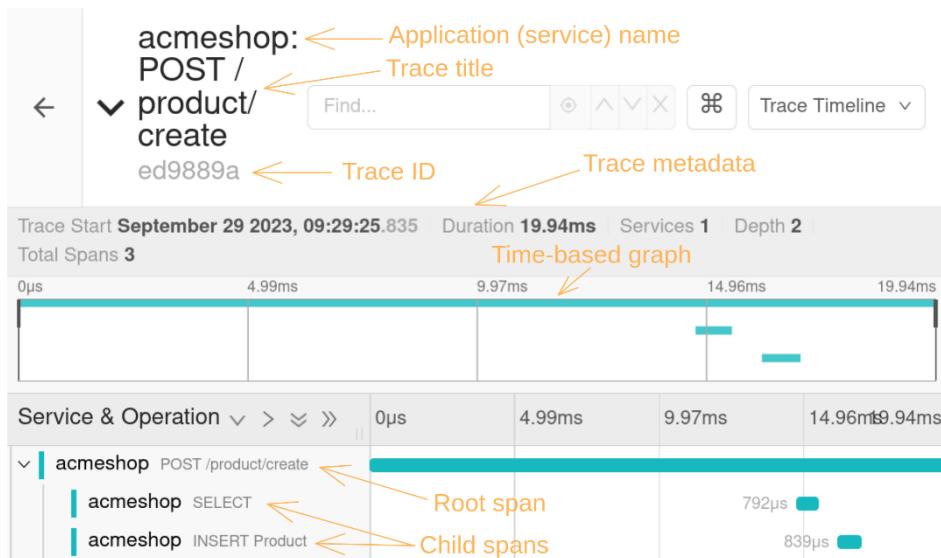


Figure 10.8 A simple trace example

This is a simple example that only involves a single application. It is a trace from a single invocation of a REST operation that adds a new product definition into a shop. From the screenshot, it is easy to deduce that the trace ID in this case is ed9889a. The trace consists of 3 spans, the root span (the first line in the table) represents the REST call, and then there are two database calls. One is a SELECT to generate the next ID from a database sequence, the other one is the actual INSERT of the new product definition.

The trace ID is propagated between services (only those that support distributed tracing, of course), so that it is possible to correlate spans from different services that belong to the same trace. This adds a requirement on the underlying protocol to support passing this metadata somehow. For the HTTP protocol, trace propagation is done using HTTP headers.

So now that we know that individual services are instrumented to generate span data, how are spans collected and visualized? This depends on the tracing library, and the chosen visualization tool. In our case, we will be using OpenTelemetry as the tracing library, which is currently the most popular tracing library in the cloud-native world. Quarkus has first-class support for OpenTelemetry. For visualizing the traces, we will use Jaeger, which is the tool from which the example screenshot in [Figure 10.8](#) is. Let's now talk briefly about OpenTelemetry.

10.4.1 OpenTelemetry

OpenTelemetry (<https://opentelemetry.io/>), commonly referred to as OTel for short, is a collection of various tools and APIs that are used to collect and export telemetry data from applications. OpenTelemetry doesn't only cover tracing. It also has capabilities for collecting metrics and logs, but these were not yet considered ready and stable, and thus not supported by Quarkus at the time of writing this book, so we will focus only on the tracing part in this book.

Usage of OpenTelemetry in the context of cloud-native Java applications is further specified by the MicroProfile Telemetry specification. Quarkus is an implementation of this specification.

OpenTelemetry was conceived by merging two former projects, OpenTracing and OpenCensus. It is a vendor-neutral and programming language-agnostic. It provides a common protocol (named OTLP) for exporting telemetry data from applications to a tool named OpenTelemetry Collector, which then centrally processes the data and forwards it to a backend.

Usage of an OpenTelemetry collector is in most cases optional. Application frameworks may support exporting telemetry data directly to a backend, without going through an OpenTelemetry Collector. For smaller-scale deployments, this is easier to set up and maintain, but for larger deployments, it is advised to use an OpenTelemetry Collector, because it can centrally add features like filtering, batching and retrying in case of failures. In our practical examples, we will be using Jaeger, which supports receiving OTLP data directly from applications, so we will actually need to use a separate collector (even though it is possible).

[Figure 10.9](#) depicts the topology when using a collector.

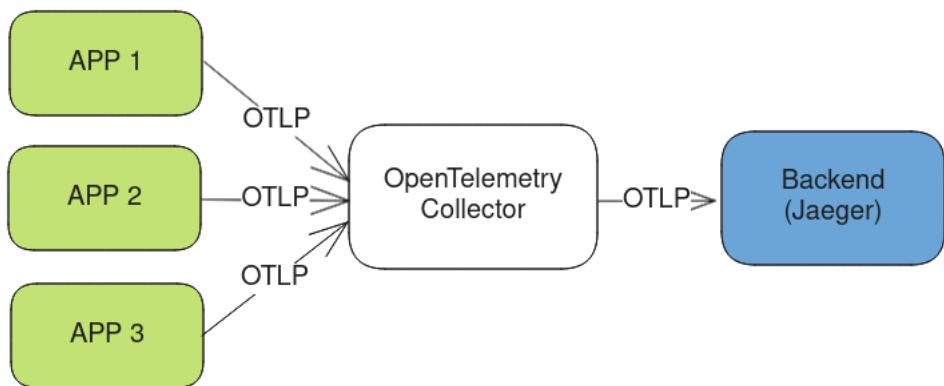


Figure 10.9 OpenTelemetry tracing topology with OpenTelemetry Collector

[Figure 10.10](#) depicts the topology when not using a collector.

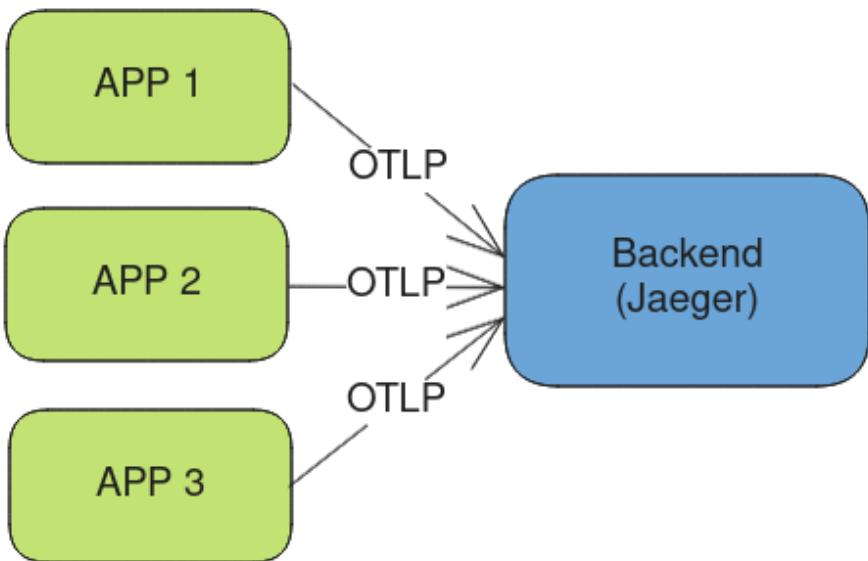


Figure 10.10 OpenTelemetry tracing topology without OpenTelemetry Collector

10.4.2 Adding tracing capabilities to Car Rental

Let's try to add tracing to our Car Rental application. In this exercise, we use three existing services (Users, Reservation, and Inventory) to show how tracing works between multiple services. We don't add any new business functionality and there are zero changes to the application code - everything is achieved by simply adding the OpenTelemetry extension to each of these services. The solution, as always, can be found in the `chapter-10` directory in the book's git repository.

Go to the root directories of all the mentioned services (`users-service`, `reservation-service`, and `inventory-service`) and add the `quarkus-opentelemetry` extension for each of them:

```
$ quarkus extension add opentelemetry
```

or

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

RUNNING JAEGER

Now that we've added the extension, Quarkus will automatically start collecting tracing data. But to be able to store and visualize the data, we need Jaeger. We will make use of the fact that in recent versions, Jaeger supports receiving OpenTelemetry data directly, without needing a separate collector. We have again prepared a ready-to-use docker-compose file that starts everything. The file can be found as `chapter-10/docker-compose-tracing.yml` in the git repository. It's really short, so we will also list it here in [Listing 10.7](#).

Listing 10.7 Using docker-compose to start Jaeger and the OpenTelemetry Collector

```
version: "3"
services:
  jaeger-all-in-one:
    image: docker.io/jaegertracing/all-in-one:1
    ports:
      - "16686:16686" # Jaeger UI
      - "4317:4317"   # OTLP receiver
```

It's actually just a single container that contains Jaeger, and an OpenTelemetry collector is conveniently included with Jaeger. You can see that the Jaeger UI is exposed on port 16686, and the OpenTelemetry Collector listens on port 4317 for incoming data. What's convenient is that `localhost:4317` is also the default export target for the OpenTelemetry Quarkus extension, so we don't need to configure anything in our applications.

Start the docker-compose file by running:

```
podman-compose -f docker-compose-tracing.yml up
```

Another way is to use your container runtime (Podman, Docker, etc.) directly, for example with Podman:

```
podman run --rm -p 16686:16686 -p 4317:4317
→ docker.io/jaegertracing/all-in-one:1
```

GENERATING SOME TRACING DATA AND VISUALIZING IT

Now it's time to start the three applications (Users, Reservation, and Inventory) in Dev mode. After that, let's access the Users service by opening <http://localhost:8080> in a browser. After logging in as `alice` with the same password, you should see the old familiar UI of the Users service from chapter 6, with some cars available for reservation. The difference is that this time, by opening this page, we have generated some tracing data.

Let's go to the Jaeger UI at <http://localhost:16686> and see what we can find.

In the left-hand menu, you see a dropdown item with services that Jaeger knows about - services that have submitted some tracing data. There should be four services: `users-service`, `reservation-service`, `inventory-service` and `jaeger-query` (which is an internal service that traces requests to Jaeger itself and is always present), as shown in [Figure 10.11](#).

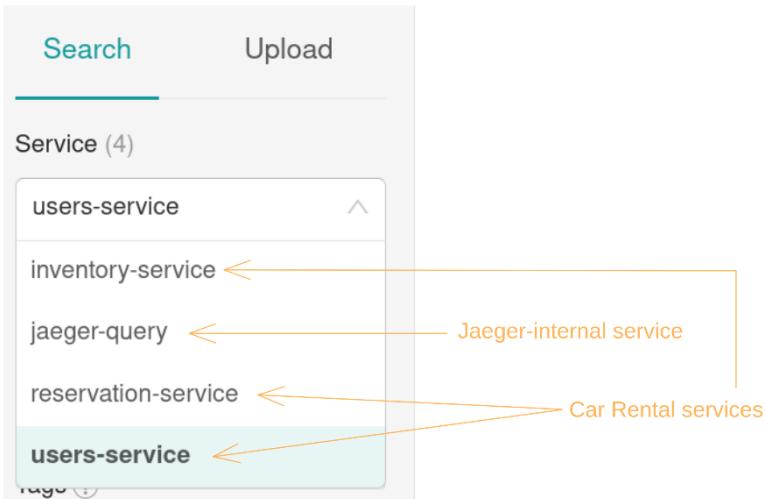


Figure 10.11 Jaeger UI - service dropdown

If you can't see all four, it is possible that some services haven't submitted their data yet. Data is collected periodically, so perhaps a refresh in a few seconds will fix it.

By selecting a service from this list and clicking the `Find Traces` button below, we ask Jaeger to show us all known traces that involve that service. Select the `users-service` and click `Find Traces`. Depending on how many HTTP requests you've made to the `users` service, your results may look a bit different, but you should be able to find a set of three traces that look like in [Figure 10.12](#):

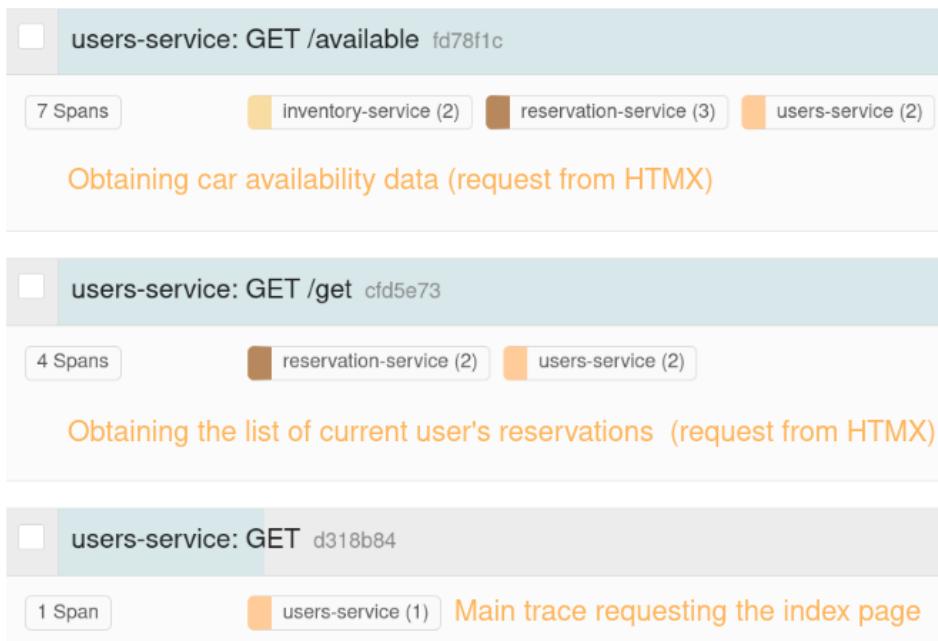


Figure 10.12 Jaeger UI - traces

These three separate traces correspond to what had to be done to render the page at `localhost:8080`.

The first trace from the bottom corresponds to the `GET` request itself that was made by your browser to the `users-service`. This request was handled by simply rendering the `index.html` page which was returned to the browser. That's why it contains only one span. If you click on that trace and then open the single span, you are able to see the tags added to that span, such as in [Figure 10.13](#). Here you can see various metadata for the request, such as the client's IP address, response's content length, status code, etc.

GET	
▼ Tags	
code.function	index
code.namespace	org.acme.users.ReservationsResource
http.client_ip	127.0.0.1
http.method	GET
http.response_content_length	1150
http.scheme	http
http.status_code	200
http.target	/
internal.span.format	proto
net.host.name	localhost
net.host.port	8080
otel.library.name	io.quarkus.opentelemetry
span.kind	server
user_agent.original	Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0

Figure 10.13 Jaeger UI - trace tags

But the server simply sending the HTML page isn't everything that was necessary to fully render the page with available cars and user's reservations. Remember that we used the HTMX library to make asynchronous HTTP requests. And to fully render the page, there have to be two such requests. One is to get the list of available cars, and the other is to get the list of user's reservations. These two requests represented by the second and third trace in [Figure 10.12](#). Click the left-pointing arrow at the top left of the page to go back to the list of traces. Let's look at the trace that has 7 spans - it should look like [Figure 10.14](#).



Figure 10.14 Jaeger UI - trace for getting available cars

This trace corresponds to the request to the `/available` endpoint exposed by the inventory service and was performed by the browser and the HTMX code. See the file `src/main/resources/META-INF/resources/templates/ReservationsResource/index.html` in the `users-service` if you need to refresh your memory.

To render the table of available cars, multiple services had to be involved (Reservation and Inventory as well), that's why you can see spans from three services inside the trace. After clicking the trace, you will see a graph showing the spans, their relationships and a time axis, as shown in [Figure 10.15](#).

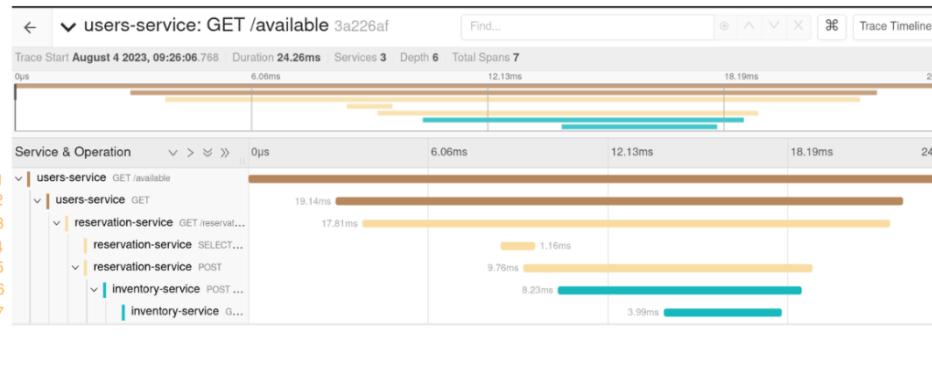


Figure 10.15 Jaeger UI - graph of spans from calling the car availability endpoint

Let's explain each span and what it represents, going from top to bottom:

1. The `GET /available` span is the AJAX request made by the browser to `localhost:8080/available`.
2. The second `GET` span corresponds to the client part of the request made by the `users-service` to the `reservation-service`. After opening the tags of the span, you should see that the `http.url` tag shows `http://localhost:8081/reservation/availability?` followed by date arguments.
3. The third span is the server-side part of that request - it was generated by the `reservation-service`. Again, look through the tags to see what exactly was actually called.
4. This span corresponds to a database call made by the `reservation-service` to obtain a list of reservations from its PostgreSQL database instance. In the `db.statement` tag, you can see the whole SQL query.
5. Client-side span of the GraphQL request that the `reservation-service` made to the `inventory-service` to obtain the list of cars in the inventory.
6. Server-side span created for the HTTP request received by the `inventory-service`.
7. Span for the GraphQL operation itself. You can see that the `inventory-service` created two spans, one for the received HTTP request itself, and one for the `cars` GraphQL query that was executed while handling the request.

We highly recommend you spend some time exploring the spans and their metadata to get an understanding what piece of work each span corresponds to.

We have only looked at spans that are generated automatically by Quarkus, but you can generate custom spans for all kinds of operations. For example, any method of a CDI bean can be marked with a `@io.opentelemetry.instrumentation.annotations.WithSpan` annotation so that any invocations of the bean will be recorded as spans. For more complex scenarios, it is also possible to inject an instance of `io.opentelemetry.api.trace.Tracer` to generate any custom spans programmatically using a builder API.

10.5 Fault Tolerance

In distributed systems, very often we have to deal with unexpected failures. If the system is not built with fault tolerance in mind, a single failure can easily have a cascading effect and bring down large parts of the system. Fault tolerance is a feature of a system that allows it to continue working with as little impact as possible when some of its components face failures. For this reason, Quarkus implements the MicroProfile Fault Tolerance API. MicroProfile Fault Tolerance (<https://github.com/eclipse/microprofile-fault-tolerance>) is a specification and an annotation-based Java API that allows developers to build more resilient applications. For Quarkus, the implementation of these strategies is provided by the `quarkus-smallrye-fault-tolerance` extension that brings in the SmallRye Fault Tolerance library.

Fault tolerance is achieved through various fault tolerance patterns that can be applied on specific parts of applications:

- **Retry**—If an operation fails, retry it a certain number of times before propagating the failure back to the client.
- **Fallback**—If an operation fails, execute a fallback instead, that is an alternative operation that is less likely to fail.
- **Bulkhead**—Limit the number of concurrent invocations of a certain operation. This prevents overloading the system.
- **Circuit breaker**—If an operation fails a certain number of times, stop re-attempting it for some amount of time (instead, throw an error when the operation is requested). This helps prevent cascading failures.
- **Timeout**—If an operation takes too long, abort it. This prevents the system from getting stuck and unresponsive.

Everything is achieved declaratively using annotations, so the API is very easy to use and doesn't pollute business code with fault tolerance logic. SmallRye Fault Tolerance also offers an imperative (programmatic) API, but we don't discuss that in this book. You can find more about it in the official SmallRye documentation - <https://smallrye.io/docs/smallrye-fault-tolerance/6.2.6/index.html>.

10.5.1 Adding Fault Tolerance to Car Rental

Let's try to add some Fault Tolerance capabilities to Car Rental. We keep it simple and only add two fault tolerance policies, both apply to calls from the `reservation-service` to the `inventory-service` when `reservation-service` requests the list of available cars to be able to return a list of cars available for the given dates (we're talking about the `/availability` endpoint of the `reservation-service`). We assume that the `inventory-service` can be unavailable at times, and we allow some time for it to come back up before failing the `/availability` request - this is achieved using a `Retry` policy. We also add a `Fallback` policy in case that the `inventory-service` is down for too long and the `Retry` policy gives up - in that case, we return an empty list of cars available for the given dates, instead of throwing an exception.

RETRY POLICY

Start by adding the `smallrye-fault-tolerance` extension to the `reservation-service`.

```
quarkus extension add smallrye-fault-tolerance
```

or

```
./mvnw quarkus:add-extension -Dextensions="smallrye-fault-tolerance"
```

TIP We are adding the extension to the `reservation-service`, not the `inventory-service`. This is because the fault tolerance policies are applied to the client side of the request, not the server side - we want the `reservation-service` to be resilient to failures of the `inventory-service`. The `inventory-service` can have its own fault tolerance policies, for example for accessing the database, but that's not our goal here.

Open the `reservation-service` in your IDE and then open the `ReservationResource` class. Annotate the `availability` method with:

```
@Retry(maxRetries = 25, delay = 1000)
```

The correct import is `org.eclipse.microprofile.faulttolerance.Retry`. This way, we have established a `Retry` policy that makes the `availability` method get re-executed up to 25 times, with a 1-second delay between each attempt. The method uses a REST client internally to call the `inventory-service`, that is the most likely place for the failure to happen. If all 25 attempts fail, only then the failure propagates back to the client. This also means that when the `inventory-service` is down, the `availability` method will wait for up to 25 seconds before failing the request.

TIP When using the `@Retry` annotation, make sure that the method behaves idempotently, meaning it can be executed multiple times without causing inconsistencies (side effects). This generally shouldn't be a problem if it only accesses transactional resources, because for each invocation, a new transaction will start, and then get rolled back on failure. Nevertheless, caution has to be taken when using `@Retry` on methods that modify any data in a non-transactional manner.

Now, let's try this out by calling the `/availability` endpoint while the `inventory-service` is down. Run the `reservation-service`, but **not** the `inventory-service`. Call the `/availability` endpoint, for example by opening the Swagger UI at <http://localhost:8081/q/swagger-ui> and invoking the `/reservation/availability` operation. Or, simply call the endpoint directly by sending a GET request to

```
http://localhost:8081/reservation/availability?  
-endDate=2022-03-10&startDate=2022-03-10
```

Change the dates to anything else if you want. The dates aren't important right now.

If the `inventory-service` isn't running, the request will hang for up to 25 seconds, during which you have time to bring the `inventory-service` up (we suggest Dev mode, as always). If 25 seconds isn't enough, feel free to tinker with the arguments in the `@Retry` annotation and increase that time. After the `inventory-service` is up, the request will succeed. If it doesn't come up during that time, the request will fail.

FALLBACK POLICY

Let's take it a little step further by adding a Fallback policy. We will tell the `ReservationResource` that when the `inventory-service` is unavailable for more than 25 seconds, it should return an empty list of cars available for the given dates instead of returning an error.

TIP As you might have guessed, the `Retry` policy has a higher priority than the `Fallback` policy, so the `Fallback` policy (returning an empty list) will only come into effect only after all configured retries have failed.

Add the new method to the `ReservationResource` that serves as the fallback for the `availability` method, as shown in [Listing 10.8](#). The return type of the fallback needs to be the same as the original method.

Listing 10.8 Fault tolerance availability fallback

```
public Uni<Collection<Car>> availabilityFallback(LocalDate startDate,
                                                    LocalDate endDate) {
    return Uni.createFrom().item(List.of());
}
```

Annotate the original `availability` method to use the fallback:

```
@Fallback(fallbackMethod = "availabilityFallback")
```

The correct import is `org.eclipse.microprofile.faulttolerance.Fallback`.

Now let's try this out by, again, calling the `availability` endpoint while the `inventory-service` isn't running. After the configured retries have been exhausted, the call returns, but this time with an empty list of cars. Again, if you don't feel like waiting 25 seconds, adjust the numbers in the `@Retry` annotation as you wish.

10.5.2 Alternative Fault Tolerance approaches

HYSTRIX

Hystrix (<https://github.com/Netflix/Hystrix>) is a fault tolerance library developed by Netflix. It is no longer in active development, but the MicroProfile Fault Tolerance specification was heavily inspired by it.

ISTIO

Istio (<https://istio.io/>) is a service mesh implementation for Kubernetes. It provides tools for managing observability, traffic management and security for applications deployed in Kubernetes clusters. It can also be used to add fault tolerance patterns. Compared to using the MicroProfile Fault Tolerance specification, the fault tolerance policies are applied at the service mesh level, that is by injecting proxy containers, called sidecars, next to application pods.

Moving the fault tolerance logic to this level has its advantages as well as disadvantages. The main advantage is the centralization of the fault tolerance policies configuration and the possibility to dynamically change them dynamically without having to rebuild the application. The main disadvantage is that you lose the ability to apply policies on a per-method basis. It's also harder for applications to watch the status of the fault tolerance components, because they are exposed on a higher level.

10.6 Service discovery

Another challenge that is commonly faced in cloud deployments is discovering the location of other services. Given that instances of a service can be dynamically created and destroyed, and that their IP addresses are not known in advance, it may be necessary to use a service discovery mechanism to allow services to find one another. In Kubernetes in particular, this is usually not very difficult, because Kubernetes employs the notion of Services, where a Service is a logical abstraction that groups a set of Pods and provides network access to them. Using the service name as the URL when establishing a connection to another service automatically connects you to the service. It also optionally provides load balancing between service instances to evenly spread the load between its instances. This is handled on the Kubernetes level. The communication is depicted in [Figure 10.16](#).

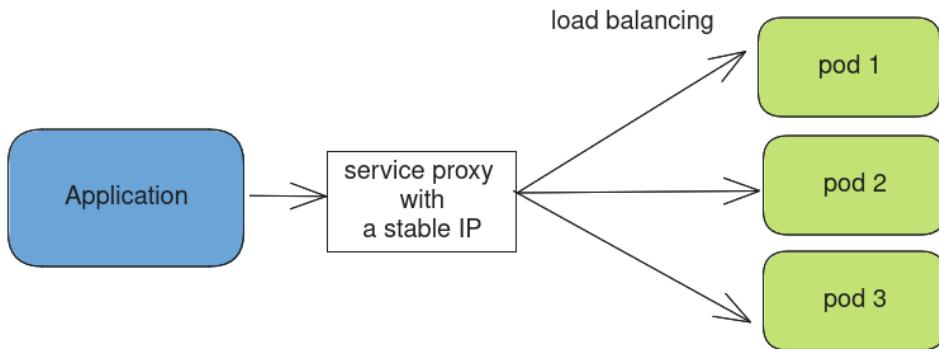


Figure 10.16 Communication between pods via a Kubernetes service proxy

However, if you can't, or you don't want to use the Kubernetes Service discovery mechanism, Quarkus provides SmallRye Stork, a library that implements an abstraction over different service discovery mechanisms, allowing you to easily swap between them. Kubernetes is one of the supported underlying mechanisms, too, so you can use it together with Stork if you want the flexibility to switch to a different mechanism (for example, Consul).

At the time of writing, Stork supports the following mechanisms:

- Consul
- DNS
- Kubernetes

- KNative
- Eureka
- Composite (a combination of multiple mechanisms)
- Static list (a hardcoded list of URLs)
- Custom mechanism implemented as a Java class

Stork also provides configurable load balancing strategies to more evenly spread the load between multiple instances. We won't dive into Stork here, since our production will be in Kubernetes/OpenShift. This section is just to let you know the basic concept. For more information about Stork, visit <https://smallrye.io/smallrye-stork>.

10.7 Wrap up and next steps

In this chapter, we learned about patterns that are commonly used when running microservices in cloud environments. One of the most important aspects is observability (which includes health checks, metrics, logging and tracing), because when running in the cloud, monitoring how the applications are working is generally harder than with traditional server-based deployments. Luckily, Quarkus offers tools to make all observability components easier.

We explored health checks using the MicroProfile Health specification. These serve not only for manual monitoring, but fit in well with probes built into Kubernetes, so they allow automatically restarting unhealthy pods, detecting whether the application has already started, etc.

We also learned about metrics and the Micrometer project and how to visualize them as charts in Grafana. Metrics grant insight into application behavior and are used for troubleshooting performance issues, or simply for collecting statistics about how users utilize the application.

We also learned about distributed tracing and how to use OpenTelemetry to collect tracing data from applications and visualize it in Jaeger. Distributed tracing is useful for troubleshooting issues that span multiple services, because it allows us to see the whole picture of what happens in the application when processing requests. It also allows us to see how long each service took to process each operation, so we can identify bottlenecks.

Then we learned about how to apply fault tolerance patterns to make applications more resilient to failures in other services they depend on, because failures in complex cloud deployments are very hard to avoid. Using the MicroProfile Fault Tolerance utilities mitigates the impact of such failures.

We also briefly mentioned service discovery mechanisms that are used in cloud deployments to allow services to find one another without having to hard-code the addresses of other services.

10.8 Summary

- MicroProfile is a set of specifications for building cloud-native applications in Java.
- MicroProfile Health provides a way to expose health checks for applications over HTTP endpoints. Health check types are Liveness (whether the application can continue to run), Readiness (whether the application can accept requests at the moment), Startup (whether the application has finished initializing) and Wellness (for reporting less serious problems that don't require immediate automated action). Wellness is added by SmallRye on top of the MicroProfile specification.
- Quarkus provides some health checks out of the box (for example, whenever you use a data source, a readiness check is automatically created for it) and allows you to create your own custom health checks by implementing the corresponding `HealthCheck` interface.
- Health checks are monitored by calling an HTTP endpoint. For development purposes, Quarkus also offers a way to view the status of health checks in the Dev UI.
- Micrometer is a library for collecting metrics from Java applications. It offers a declarative and a imperative API and supports a wide variety of backends to export the metrics to. One of the most popular solutions is to use Prometheus for storing metrics and Grafana for their visualization and alerting.
- Metrics with Micrometer are defined either by using annotations, or programmatically. They use dimensions (tags) to allow for more advanced querying and aggregation.
- Quarkus also provides some metrics out of the box. For example, whenever you use a data source, metrics can be automatically created for it, if you specify a configuration property for it.
- OpenTelemetry is a collection of tools and APIs for collecting telemetry data from applications. In this book, we focused on its distributed tracing capabilities. Distributed tracing allows you to trace requests as they flow in a distributed system, and visualize the traces to understand the performance of the system and to debug problems.
- Jaeger is one of the popular tools for visualizing distributed tracing data. A trace consists of spans, which represent individual operations. Spans inside a single trace form a tree structure.
- Fault tolerance is the ability of a complex system to limit the impact of failures in its components. With Quarkus, this is achieved by leveraging various fault-tolerant patterns declared via annotations from MicroProfile Fault Tolerance API. The patterns include Retry, Fallback, Bulkhead, Circuit Breaker and Timeout.

- SmallRye Stork is a project that allows you to decouple your services and avoid hard-coding addresses of services that another service depends on. It's an abstraction over different service discovery mechanisms and it allows you to easily switch between them without changing your application's code.
- Stork also provides client-side load balancing, allowing to spread the load between multiple instances of a service more evenly.

11

Quarkus applications in the cloud

This chapter covers

- Running Acme Car Rental services in production mode
- Containerizing Quarkus applications
- Deploying Quarkus in the cloud (Kubernetes and OpenShift)
- Creating serverless application deployments with Quarkus
- Constructing Acme Car Rental production in the OpenShift platform

After completing chapter 10, we are finished with the development of the Car rental microservices. Now, it's time to move from Dev mode to production. This chapter explores several deployment scenarios ranging from a local test environment with Docker to a fully deployed system in the public cloud.

As everybody in the software industry knows, more and more applications are moving to cloud environments. This is the general direction that the software industry has been heading towards for years now. It allows for more flexibility, scalability, reliability, and ultimately, significant cost saving. In the previous chapter, we focused on the program aspects (capabilities of the application itself) needed for cloud deployments — mainly observability and reliability, and how Quarkus makes it all very easy. In this chapter, we focus on the operational aspects — once your application is written, how do you deploy and manage it? Quarkus has a plethora of tools and integrations to make this easy. We start by running our car rental services in Quarkus production mode, then continue with their containerization, and then we deploy the full Acme Car Rental system to the cloud.

11.1 Car Rental production

Let's first repeat what our production system precisely consists of. Throughout the book, we have developed five car rental microservices: Reservation, Rental, Inventory, Users, and Billing services. However, we also utilize a bunch of third-party remote providers or brokers (handled mainly by Dev Services) such as databases, Kafka, or Prometheus. [Figure 11.1](#) describes everything we need to run for our Acme Car rental system production to function properly.

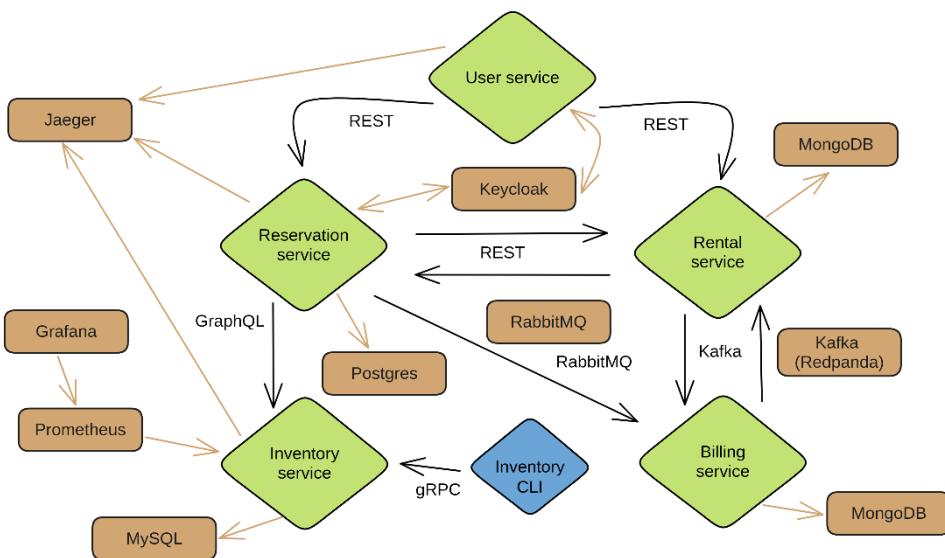


Figure 11.1 The Acme Car Rental production with Dev Services

Apart from the five car rental microservices and additional Inventory CLI, we also have ten containers that are either started as Dev Services (e.g., Postgres or RabbitMQ) and three services that we need to start manually since they don't have Dev Services (e.g., Jaeger that we tackled in Chapter 10).

To run a full Acme Car Rental deployment on your system with Dev mode and Dev Services, you can follow these simple steps:

1. Go into the `chapter-11` directory of the book resources.
2. Start containers that don't have Dev Services:

```
$ docker-compose -f ./docker-compose-ch10.yml up
```

This starts the first three containers: Jaeger, Prometheus, and Grafana.

3. In a new terminal, navigate to the `chapter-11/users-service` directory and start Dev mode (`quarkus dev`). Dev Services start the Keycloak container.

4. In a new terminal, navigate to the `chapter-11/reservation-service` directory and start Dev mode (`quarkus dev`). Dev services start Postgres and RabbitMQ containers.
5. In a new terminal, navigate to the `chapter-11/rental-service` directory and start Dev mode (`quarkus dev`). Dev services start MongoDB and Redpanda containers.
6. In a new terminal, navigate to the `chapter-11/inventory-service` directory and start Dev mode (`quarkus dev`). Dev services start the MySQL container.
7. In a new terminal, navigate to the `chapter-11/billing-service` directory and start Dev mode (`quarkus dev`). Dev services start the second MongoDB container for Billing service.

Wow, that's a lot of services currently running! For verification, [Listing 11.1](#) shows all containers running in addition to the five Dev modes of our car rental services.

Listing 11.1 All provider Docker containers running in local car rental production

```
$ docker ps --format "table {{.ID}} {{.Image}}"
CONTAINER ID   IMAGE
0e9ffffa49bf9   docker.io/prom/prometheus:v2.44.0
aec2f2f994b1   docker.io/grafana/grafana:9.5.2
b8bee99668be   docker.io/jaegertracing/all-in-one:1
284d873941cd   quay.io/keycloak/keycloak:23.0.0
11440c040d54   docker.io/library/rabbitmq:3.12-management
c0a22db4dea5   docker.io/library/postgres:14
b1ccc937076e   docker.io/library/mongo:4.4
7b528cb6a84f   docker.io/vectorized/redpanda:v22.3.4
3a2551e53ff8   docker.io/library/mysql:8.0
0124e344a8d3   docker.io/library/mongo:4.4
```

Now we can verify that the system is running by opening <http://localhost:8080> in any browser that redirects us to Keycloak, where we can log in (alice:alice or bob:bob) and land on our main page. You can make a few reservations if you feel like it and see how all of our five microservices propagate messages.

It is relatively easy to get the 15 services (5 applications and 10 supporting containers) running locally. However, in production, we won't be able to use Dev Services. So next, we need to externalize the Dev Services and run individual containers manually.

11.1.1 Externalizing providers for independent deployment

Externalizing every provider we use in Acme Car Rental is relatively simple. We need to extract the containers from [Listing 11.1](#) to a Docker Compose file. This file is too long to be inlined in the book, so the full version is available in the book resources at `chapter-11/docker-compose-dependent-services.yml`. You might notice that it starts more containers than we had with the Dev Services since the production versions sometimes require additional resources (e.g., Zookeeper for Kafka). The local production deployment changes as demonstrated in [Figure 11.2](#).

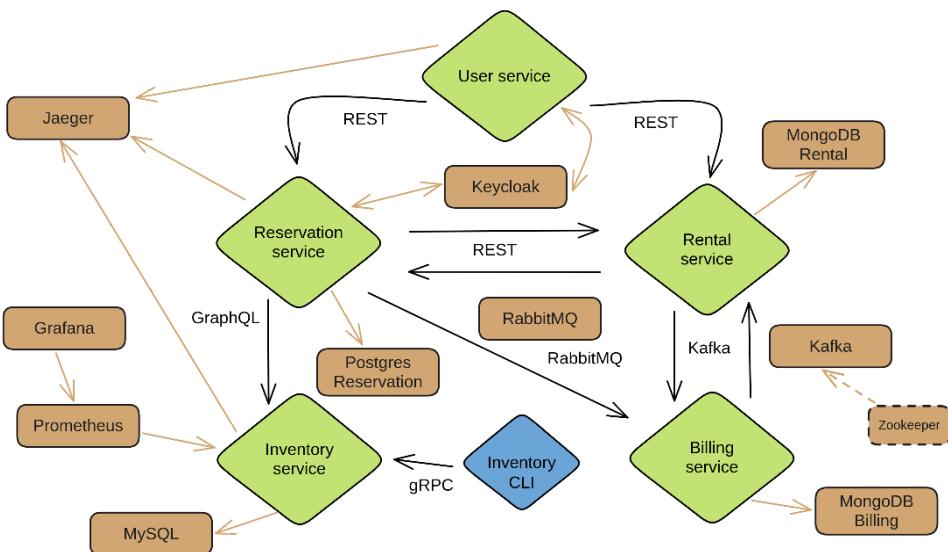


Figure 11.2 The Acme Car Rental production with all dependent providers

Overall, this brings us to eleven containers. But they can all be started and managed together by running:

```
docker-compose -f docker-compose-dependent-services.yml up
```

Sometimes, not all containers stop when you kill the above process. In such case, run `docker-compose -f docker-compose-dependent-services.yml down`, which stops and removes all defined containers.

The hard part is that now we need to provide all the configuration that the Dev Services handled manually. We take this opportunity to also prepare our services for the scenario where they won't all run on a local, single machine. Let's review all car rental services to see what changes we must make.

USERS SERVICE

Users service calls the Reservation service to get and create reservations. The location of the Reservation resource is currently hardcoded in the `org.acme.users.ReservationsClient` interface, as shown in the [Listing 11.2](#).

Listing 11.2 The original code of ReservationsClient in Users service

```
@RegisterRestClient(baseUri = "http://localhost:8081")
@AccessToken
@Path("reservation")
public interface ReservationsClient {
    ...
}
```

Hardcoding the URL in the annotation was the quick-and-easy approach, but it won't really work in production. We will have to configure the REST client using regular Quarkus configuration mechanisms to allow for more flexibility. Change the `ReservationsClient` as demonstrated in [Listing 11.3](#).

Listing 11.3 ReservationsClient update with the configuration key

```
@RegisterRestClient(configKey = "reservations")
@AccessToken
@Path("reservation")
public interface ReservationsClient {
    ...
}
```

Now, we can adjust the configuration file `application.properties` to utilize the newly defined configuration key:

```
quarkus.rest-client.reservations.url=http://localhost:8081
```

TIP It's also possible to override the REST client's URL without specifying a custom config key - the default config key is the fully qualified name of the interface. In that case, the property would be `quarkus.rest-client."org.acme.users.ReservationsClient".url`.

The Users service also utilizes Keycloak for authentication. We already created the Keycloak prod mode setup in Chapter 6, so we can reuse it here. In the `application.properties`, we can find the following configuration that connects to the Keycloak started on `localhost:7777`:

```
%prod.quarkus.oidc.auth-server-url=http://localhost:7777/realmss/car-rental
%prod.quarkus.oidc.client-id=users-service
%prod.quarkus.oidc.token-state-manager.split-tokens=true
```

Hardcoded localhost won't obviously work in the cloud, but we will keep it for now, while we're still running everything locally. The chapter-11/docker-compose-dependent-services.yml file configures Keycloak to be available at localhost:7777.

RESERVATION SERVICE

The Reservation service calls the Rental service to start rentals. We have the same problem with the hardcoded REST client URL, so change the org.acme.reservation.rental.RentalClient like this:

```
- @RegisterRestClient(baseUri = "http://localhost:8082")
+ @RegisterRestClient(configKey = "rental")
```

And add the following configuration:

```
quarkus.rest-client.rental.url=http://localhost:8082
```

Reservation service was also prepared already for the production Keycloak in Chapter 6. The properties are the same as those in the Users service. Additionally, it also manages its own Postgres database, and it connects to RabbitMQ. Now, let's add the Postgres configuration:

```
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = quarkus
quarkus.datasource.password = quarkus
%prod.quarkus.datasource.reactive.url =
  -> vertx-reactive:postgresql://localhost:5432/quarkus
```

Notice that except for the last property, we can define all other properties without the %prod. prefix since they are applicable in any mode (we use the same credentials as Dev Service). However, the last %prod.quarkus.datasource.jdbc.url should be defined only in prod mode if we want Dev Services to start in Dev mode.

RabbitMQ only requires connection information to the broker, which is also defined only in prod mode to keep the Dev Services working:

```
%prod.rabbitmq-host=localhost
%prod.rabbitmq-port=5672
%prod.rabbitmq-http-port=15672
%prod.rabbitmq-username=guest
%prod.rabbitmq.password=guest
```

RENTAL SERVICE

The last REST client we need to adjust is in the Rental service's `org.acme.rental.reservation.ReservationClient`. The changes are the same as in the previous two services, so we summarize them in the following snippet:

```
- @RegisterRestClient(baseUri = "http://localhost:8081")
+ @RegisterRestClient(configKey = "reservation")

# In application.properties
+ quarkus.rest-client.reservation.url = http://localhost:8081
```

We utilize MongoDB to store the created rentals. The Rental service also sends and receives data from the Kafka broker. To get the MongoDB and Kafka connections set up, add the following configuration:

```
%prod.quarkus.mongodb.connection-string = mongodb://localhost:27017
%prod.kafka.bootstrap.servers=localhost:9092
```

INVENTORY SERVICE

The Inventory service exposes the GraphQL UI we used to access from the Dev UI. Dev UI isn't available in prod mode, but the GraphQL UI can still be added explicitly. To make the GraphQL UI available in production mode, add the following config property, and it will again be available on <http://localhost:8083/graphq1-ui>:

```
quarkus.smallrye-graphql.ui.always-include=true
```

The Inventory service only utilizes the MySQL database, which is similar to the Reservation service integration with Postgres integrated through the Panache ORM, so the relevant configuration is very similar:

```
quarkus.datasource.db-kind = mysql
quarkus.datasource.username = quarkus
quarkus.datasource.password = quarkus
%prod.quarkus.datasource.jdbc.url = jdbc:mysql://localhost:3306/quarkus
```

The Inventory service also contains some starting data that we load into the database with the `import.sql` script in Dev mode. Since it would be helpful for us to begin with some cars available in the DB also in prod mode, we can add this configuration that will load this SQL script also in prod mode:

```
quarkus.hibernate-orm.sql-load-script=import.sql
```

TIP The default behavior is that in Dev mode, `import.sql` is used (if it exists). In prod mode, loading SQL scripts is disabled by default, so here we explicitly enabled it by setting the file name. If we wanted to disable it in Dev mode, we would set `%dev.quarkus.hibernate-orm.sql-load-script=no-file`.

BILLING SERVICE

The Billing service is the most complex since it connects to RabbitMQ, Kafka, and its instance of MongoDB. But it's not something we couldn't manage:

```
%prod.quarkus.mongodb.connection-string = mongodb://localhost:27018
%prod.kafka.bootstrap.servers=localhost:9092
%prod.rabbitmq-host=localhost
%prod.rabbitmq-port=5672
%prod.rabbitmq-http-port=15672
%prod.rabbitmq-username=guest
%prod.rabbitmq.password=guest
```

Note that we need to change the MongoDB port to 27018 so it doesn't conflict with the MongoDB instance of the Rental service (which runs on port 27017).

This is all that we need to adjust and configure to get the Acme Car Rental production running. We can now follow these steps to run the Docker compose and start our services in prod mode:

1. Change your current directory to `chapter-11` directory.
2. Start containers for remote providers:

```
$ docker-compose -f ./docker-compose-dependent-services.yml up
```

3. Open the following directories, each in a new terminal, and run `quarkus build (mvn clean package)` followed by `java -jar target/quarkus-app/quarkus-run.jar`:
 - a. chapter-11/users-service
 - b. chapter-11/reservation-service
 - c. chapter-11/rental-service
 - d. chapter-11/inventory-service
 - e. chapter-11/billing-service

Congratulations! This is our complete Acme Car Rental production running on your system. Together, these sixteen services provide the functionality we've implemented throughout the book.

TIP The configuration of dependent services is purposely simplified to ease the full system deployment (e.g., Keycloak's `start-dev` command for easy realm integration), and it shouldn't be used in real production systems. We focus on writing Quarkus applications in this book, so we don't have the space to cover much of the details of all dependent services.

Feel free to experiment with the system as you like (UI is at <http://localhost:8080>). Next, we move to the containerization of our car rental services as a required step before we can move all of this workload to the cloud.

11.2 Building and pushing container images

We already touched on the containerization of Quarkus applications in Chapter 2 when we explained the Dockerfiles generated by default when you create a new Quarkus application. However, Quarkus also provides four extensions that allow the building and pushing of images. It supports the following technologies:

- **Jib** — extension `quarkus-container-image-jib`. *Jib* (<https://github.com/GoogleContainerTools/jib>) is a tool that provides fast, reproducible, and daemonless (without the Docker daemon) image builds compatible with Docker and *OCI* (Open Container Initiative, outside the scope of this book). It automatically splits the application into several layers, doing the same thing as we saw with the generated Dockerfiles. It moves your Quarkus application dependencies (`target/lib` folder) to a separate layer with the expectation that the dependencies don't change as often as the application, which allows faster image rebuilds. Note that Docker (or Podman) is still needed if you build the image locally because Jib registers the built image with it to make the image available in your local registry.

- **Docker**—extension `quarkus-container-image-docker`. This extension uses the generated Dockerfiles in the `src/main/docker` directory to perform normal Docker builds.
- **BuildPack**—extension `quarkus-container-image-buildpack`. *Buildpacks* (<https://buildpacks.io/>) are a toolset that transforms an application's source code into a runnable image. It automatically detects everything your application needs to run. Quarkus utilizes the Docker daemon for the actual build. While there are buildpack alternatives to Docker, this extension only supports Docker as of the time of writing. It also purposely doesn't define the default base build image, so the user needs to provide it manually through the configuration:

```
quarkus.buildpack.jvm-builder-image=... # JAR
quarkus.buildpack.native-builder-image=... # native
```

- **OpenShift**—extension `quarkus-container-image.openshift`. This extension uses OpenShift-specific binary builds. Such a build takes your built artifact (and dependencies) and uploads it to the OpenShift cluster. OpenShift then builds the application image automatically in the OpenShift build system. This is also called *s2i* (source to image). From the user's point of view, you don't need to do anything. The extension generates everything required for your s2i build to complete successfully.

These are only the choices of the underlying technology that builds the images with your Quarkus application. The manipulation with all of these extensions is the same.

11.2.1 Building images with Quarkus without extensions

We don't need to add the extensions mentioned above to our Quarkus application to start building images. Quarkus CLI `quarkus` already ships with built-in commands that we can use. For instance, to create an image with Jib, you can use the command in [Listing 11.4](#) that we can test, for example, in the Reservation service.

Listing 11.4 Build image using the Jib containerizer (in Reservation service)

```
$ quarkus image build jib -Dquarkus.jib.base-jvm-image=
→registry.access.redhat.com/ubi8/openjdk-21-runtime:1.18

...
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Starting
→ (local) container image build for jar using jib.
...
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Created
→ container image <username>/reservation-service:1.0.0-SNAPSHOT
→ (sha256:<sha>)
```

We need to explicitly set the `quarkus.jib.base-jvm-image` because it uses the JDK 17 image by default, and we build on JDK 21. The `quarkus image build` command takes one of the four arguments that correspond to the available containerization options—`jib`, `docker`, and `buildpack`. `openshift` requires a connection to the OpenShift cluster and the manifests (that are generated by either `openshift` or `kubernetes` extensions). We will come back to OpenShift builds later in this chapter.

If we want to push the built image, we can do it with `quarkus image push`. It uses the `docker.io` registry (Docker Hub) by default. In the background, this is a standard `docker push`, so if your local username is the same as your Docker username and you're authenticated, this will work out of the box. We will explain how to change these defaults soon.

11.2.2 Building and pushing images with Quarkus extensions

We can also add any containerization extensions mentioned above directly to our Quarkus projects. In that case, we can integrate the building of the images directly in our build pipeline (both Maven and Gradle).

For instance, let's add the `quarkus-container-image-docker` extension to the Reservation service. For example, with `quarkus` CLI:

```
$ quarkus ext add quarkus-container-image-docker
```

The following instructions are the same for all four container extensions. To build the image, we can use the `quarkus` CLI (`quarkus image build`) or specify the configuration property `quarkus.container-image.build=true`. The property is a standard Quarkus configuration, so if we define it in `application.properties` we create an image with every application build. More preferably, we should only set it on the command line build command to apply it only when we need to build the image as shown in [Listing 11.5](#).

Listing 11.5 Build image using the included extension in Reservation service

```
# same as ./mvnw clean package -Dquarkus.container-image.build=true
$ quarkus image build
...
[INFO] [io.quarkus.container.image.docker.deployment.DockerProcessor]
→ Starting (local) container image build for jar using docker.
...
[INFO] [io.quarkus.container.image.docker.deployment.DockerProcessor]
→ Built container image <username>/reservation-service:1.0.0-SNAPSHOT
```

Similarly, if you want to push the created image, the CLI command is `quarkus image push` or the property is `quarkus.container-image.push=true`. Utilizing it during the application build will push the image to the `docker.io` registry by default, as shown in [Listing 11.6](#).

Listing 11.6 Pushing image using the included extension to the Docker registry

```
# same as ./mvnw clean package -Dquarkus.container-image.push=true
$ quarkus image push
...
[INFO] [io.quarkus.container.image.docker.deployment.DockerProcessor]
→ No container image registry was set, so 'docker.io' will be used
[INFO] [io.quarkus.deployment.util.ExecUtil] The push refers to
→ repository [docker.io/<username>/reservation-service]
...
[INFO] [io.quarkus.container.image.docker.deployment.DockerProcessor]
→ Successfully pushed docker image <username>/reservation-service:
→1.0.0-SNAPSHOT
```

If you include multiple containerization extensions in the same Quarkus application, Quarkus won't allow you to build or push images. But if you have such a use case, you can add `quarkus.container-image.builder` configuration property, which defines which of the extensions should be used (e.g., `docker` or `jib`).

If you are logged into the Docker Hub, you can see that the image was pushed to <https://hub.docker.com/>. You may need to push to a different registry, or the username on your local system doesn't match the username you chose on Docker Hub. So, let's look into how we can change the specifics of the created images.

11.2.3 Image customizations

Another very popular alternative to Docker Hub, which we also use for our services in this book, is Quay (<https://quay.io>). [Listing 11.7](#) demonstrates the individual configuration properties that change the parameters of the created images.

Listing 11.7 Customizing the images created with Quarkus container extensions

```
quarkus.container-image.registry=quay.io # defaults to docker.io
quarkus.container-image.group=quarkus-in-action # defaults to username
quarkus.container-image.name=reservation-service # defaults to artifactId
quarkus.container-image.tag=1.0 # defaults to project version
quarkus.container-image.image=registry/username/image-name:tag
```

Note that `quarkus.container-image.image` takes precedence in case of conflicts. In the above case, the produced image would be `registry/username/image-name:tag`. Another useful property is `quarkus.container-image.additional-tags`, which allows you to create multiple tags from the same image build. Quarkus also enables you to override the connection information to the registry with `quarkus.container-image.[username|password|insecure]` if needed.

11.2.4 Pushing Car rental images to quay.io

We are now ready to create our production images. For the rest of this chapter, we will be deploying our car rental services to cloud platforms. For some of these use cases, we must have images of our services publicly available. For the actual deployments, you can either build and push your own images for all five of our services or use the provided ones (deployed under `quay.io/quarkus-in-action`). Therefore, the following steps are optional. If you decide to skip pushing your own images, you can jump straight to [11.3](#).

Now let's demonstrate how to push the images to your own quay.io registry. To be able to do this, you need to create an account at <https://quay.io/>. An account with the free tier should be sufficient. Later, when logging in using the `docker login` command, we recommend using the encrypted CLI password stored in your filesystem as a file that you can download from your profile page (see quay.io for exact instructions, as these may change). One more important thing to note is that the quay repositories that you push to should be public to allow OpenShift to pull from them without additional configuration (which we won't get into here).

TIP For the following tasks, if you use the prepared solutions from the chapter-11 of the book's resources, all projects have `quarkus.container-image.group` set to `quarkus-in-action`. You will have to either change this directly in all the relevant `application.properties` files, or always pass the `-Dquarkus.container-image.group=<your-quay-org>` parameter to override it to your own quay.io organization.

The following series of steps allow us to deploy the Reservation service to the `quay.io`:

1. If you haven't already, add the `quarkus-container-image-docker` extension:

```
$ quarkus ext add quarkus-container-image-docker
```

2. Add the following configuration to the `application.properties`:

```
quarkus.container-image.registry=quay.io
# replace below with your quay.io group!
quarkus.container-image.group=quarkus-in-action
quarkus.container-image.tag=1.0.0
```

3. Log in to the `quay.io`:

```
$ docker login quay.io
```

4. Build and push the image:

```
# ./mvnw clean package -DskipTests -Dquarkus.container-
image.build=true
# -Dquarkus.container-image.push=true
$ quarkus image push --also-build
```

After the command finishes, we can find the pushed image in our Quay account (in the reference repository, it is at <https://quay.io/repository/quarkus-in-action/reservation-service>). If you are pushing your images, follow the same steps for all remaining car rental services. Otherwise, you can use the provided set of images available at <https://quay.io/organization/quarkus-in-action>.

Now, we are ready to consume these images in our production environments. If you want to test our complete production car rental system, we provide `docker-compose-full-car-rental.yml` containing all car rental services and infrastructure. Together, that makes it sixteen containers. If you've pushed your own images, please change the image definitions (organization id) in this file accordingly. Once you're ready, you can start car rental production with a single command, as demonstrated in [Listing 11.8](#).

Listing 11.8 Running car rental production with Docker compose

```
$ docker-compose -f ./docker-compose-full-car-rental.yml up
```

Give it a minute to start since it's a lot of computations. Once all containers initialize, you can verify you see seventeen containers as shown in [Listing 11.9](#). You can also test the functionality of our system at <http://localhost:8080>.

Listing 11.9 List of car rental containers in full production

```
$ podman ps
CONTAINER ID NAMES
64a6aec6562 inventory-service
d7dc156754b2 postgres-reservation
401cd1a548b5 rabbitmq
6096a5b27b97 mongodb-rental
2b692d2a7566 mongodb-billing
2ccd054ba274 zookeeper
09cebc797321 mysql
7608724bed4a prometheus
07b6ce6a3d49 grafana
075a14ad5b99 jaeger
435f9656de3a keycloak
265f1e1df6af kafka
2a028aaef995 users-service
6414151aa86f rental-service
095bb7e1df62 reservation-service
91a6a246c99f billing-service
```

If some containers don't stop when you stop your Docker Compose process, run `docker-compose -f ./docker-compose-full-car-rental.yml down` that removes all containers defined in the file.

11.3 Kubernetes and OpenShift integration

Now that we have all services ready to be deployed to the cloud, we can start with the operations "fun"! We know many developers (including authors) don't like writing long YAMLs to operate their applications in clouds, and Quarkus understands this too. In this section, we explain how Quarkus simplifies the deployment to platforms like Kubernetes and OpenShift (a Red Hat-branded version of Kubernetes that we will use for our deployments), which will allow us to experiment with the deployment of the car rental system in these various platforms later in this chapter.

11.3.1 Generating Kubernetes resources

Since writing long YAML is a tedious and error-prone task, Quarkus provides extensions to do this work for us in the background. It utilizes a library called *Dekorate* (<https://dekorate.io/>) that generates Kubernetes manifests. It generates sensible defaults for modern Java applications and provides easy customization options (e.g., with properties or annotations). Additionally, the generated manifests can be combined with predefined ones (if you have resources you need to deploy together).

Quarkus generates Kubernetes manifests with the following extensions that target three leading platforms:

- `quarkus-kubernetes` - original Kubernetes
- `quarkus-openshift` - OpenShift-specific manifests
- `quarkus-knative` - serverless platform (see section [11.5](#))

We will look into each of them in this chapter. These extensions generate resources in the `target/kubernetes` directory (or `build/kubernetes` for Gradle). The names of the generated files are derived from the name of the extension that created them (e.g., `kubernetes.yml`). The generated manifests are complete Kubernetes resources, so we can directly apply them to the target cloud cluster. Quarkus can even use these resources automatically.

If we also include any of the image extensions we learned about in the section [11.2](#), the generated manifests will be automatically adjusted to point to our build (and/or pushed) images. This hugely simplifies the integration with the target Kubernetes platform, as the images often need to be available before we can apply manifests.

11.3.2 Customizing generated manifests

The sensible defaults are a great start, but we often need to tweak the generated resources for our specific deployments. Quarkus does this through the configuration properties. Quarkus extensions allow you to customize almost every aspect of the Kubernetes resources they generate. The list is too long for the book, but here are a few examples of often-utilized configurations (parts written in capital letters are placeholders that you should replace with names of your choice):

```

# The application resource type
quarkus.kubernetes.deployment-kind=Deployment|StatefulSet|Job|CronJob

# The number of replicas
quarkus.kubernetes.replicas=5

# The namespace for the resources
quarkus.kubernetes.namespace=custom-namespace

# Annotations to add to the resources
quarkus.kubernetes.annotations.ANNOTATION_KEY=annotation-value

# Labels to add to the resources
quarkus.kubernetes.labels.LABEL_KEY=label-value

# Environment variables, env-var-key translates to ENV_VAR_KEY variable
quarkus.kubernetes.env.vars.ENV_VAR_KEY=env-var-value

# Volume mounts
quarkus.kubernetes.mounts.MOUNT_KEY.name=name-of-the-volume-to-mount
quarkus.kubernetes.mounts.MOUNT_KEY.path=/path/where/to/mount

```

You can find the complete list of customization options in the Quarkus documentation. Typically, the property's name directly corresponds to the configured value in the manifest.

All of the `quarkus.kubernetes` properties represent standard Quarkus configuration properties and are processed when the image and manifests are being built. Changing them later (when deploying the manifests to Kubernetes, or even overriding them on the application's command line during start) won't have any effect.

The generated manifest can also be enhanced by already existing manifests that you, as the application developer, provide. This is useful if you already have some manifests you must base on or apply for your application to function correctly. An example can be a database that deploys together with the Quarkus application (which we will see in the next section). We can do this by placing a file named `kubernetes.yml` (or `openshift.yml`, `knative.yml`, or their JSON variants) to the `src/main/kubernetes` directory. When such file is detected during build, Quarkus combines it with the manifest generated by the relevant extension.

With Quarkus' Kubernetes extensions, we have an excessive arsenal of functionalities at our disposal. So, let's see it in action!

11.3.3 Deploying Quarkus applications on Kubernetes

Since our Car rental application is highly complex, we will stick to one service deployment for a time now and explaining it in more detail, to later return to the entire system deployment by the end of the chapter (section [11.6](#)) with slightly less detail. We chose the Inventory service since its only external dependency is the MySQL database. Feel free to experiment with other services in these sections if you want a small challenge.

TIP This and the following section (OpenShift) are optional. We don't cover the Kubernetes cluster setup nor describe how to download the `kubectl` tool. But if you would like to experiment, we recommend lightweight clusters like *Kind* (<https://kind.sigs.k8s.io/>), *K3s* (<https://k3s.io/>), or *Minikube* (<https://minikube.sigs.k8s.io/docs/>). But provided online (often paid) Kubernetes platform is also an option.

TIP In the following section, we will rewrite the changes from this section (deploying to Kubernetes) by using OpenShift instead. If you prefer just using OpenShift, skip to [11.3.4](#). The solution in the book's resources (chapter11/inventory-service directory) contains the final state after finishing the OpenShift section.

Let's start by adding the `quarkus-kubernetes` extension to the Inventory service project. Change your working directory to the `inventory-service` subdirectory and execute the following `quarkus` command or add it directly to `pom.xml`.

```
$ quarkus ext add kubernetes
```

We now generate Kubernetes manifests every time we build the Inventory service:

```
$ quarkus build
```

The build generates two new files in the `target/kubernetes` folder called `kubernetes.yml` and `kubernetes.json`. Both files represent the same Kubernetes resources. We stick to YAML since it's more commonly used for Kubernetes resource definitions. We cannot fit the whole file here, so the [Listing 11.10](#) contains only the essential parts of it. You can find it for reference in the book resources in the `chapter-11/kubernetes/kubernetes-default.yml` file.

Listing 11.10 Simplified version of the generated kubernetes.yml

```

apiVersion: v1
kind: Service
...
spec:
  ports:
    - name: grpc
      port: 9000
      protocol: TCP
      targetPort: 9000
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8083
---
apiVersion: apps/v1
kind: Deployment
...
spec:
  ...
  spec:
    containers:
      - env:
          ...
        image: quay.io/quarkus-in-action/inventory-service:1.0.0
        imagePullPolicy: Always

```

The default configuration generates two Kubernetes resources—*Service* (network exposure) and *Deployment* (declarative application definition). Notice that our configuration from the Inventory service has been propagated to the individual parts of the manifest. For instance, under the Service definition, we can find the following target port configuration, which correctly points to ports 9000 and 8083 on which we run the Inventory service (it was detected that gRPC is used in the application, so an entry to expose the port 9000 was automatically added).

The most important part of the Deployment resource definition is the location of the image. Quarkus Kubernetes already read the configuration of the Inventory service and correctly set to the target image identifier to the one that the application is configured to produce and push to quay.io.

Both resources contain a lot of additional configurations like various annotations, labels, or the number of replicas (spawned pods). Notice also the automatic addition of liveness, readiness, and startup probes that point to the endpoints exposed by the smallrye-health extension we added in Chapter 10.

MANUAL DEPLOYMENT WITH KUBECTL

The generated file is already ready to be deployed to Kubernetes. However, the Inventory service needs a working connection to the MySQL database. For simplicity, we already provided a ready-to-deploy MySQL Kubernetes manifest in chapter-11/kubernetes/mysql.yml. We could now apply the MySQL resources followed by the generated manifest of the Inventory service, but there is still one more catch. We hardcoded the location of the MySQL database to localhost in application.properties:

```
%prod.quarkus.datasource.jdbc.url = jdbc:mysql://localhost:3306/quarkus
```

In Kubernetes, our MySQL runs at mysql://mysql:3306 (defined by the Service resource), so we must change this configuration when we deploy to the Kubernetes cluster. With the Quarkus Kubernetes extension, we can do it through the environment variable configuration. We can add the following oneliner from [Listing 11.11](#) to the Inventory service's application.properties.

Listing 11.11 Changing the location of the MySQL database in Kubernetes

```
quarkus.kubernetes.env.vars.quarkus-datasource-jdbc-url =
→ jdbc:mysql://mysql:3306/quarkus
```

When you now rerun the build in the inventory-service directory, the regenerated manifest contains the QUARKUS_DATASOURCE_JDBC_URL environment variable correctly set to the URL of MySQL in Kubernetes.

We are now ready to deploy the Inventory service to Kubernetes. Normally it would be two-step process—deploy MySQL and then Inventory. But as was mentioned in the previous section, Quarkus Kubernetes extension allows combining existing manifests with the generated ones. All we need to do is to copy the mysql.yml manifest and save it as inventory-service/src/main/kubernetes/kubernetes.yml, and Quarkus will take care of the rest. Rerunning the build will produce one combined manifest that contains both the Inventory service and MySQL. Now, we can deploy both services with a single command, as shown in [Listing 11.12](#). Note the expected restarts since MySQL is slow to start. We use a simple port-forward command to expose our service on port 31013.

Listing 11.12 Applying generated Kubernetes manifest

```
$ kubectl apply -f inventory-service/target/kubernetes/kubernetes.yml
service/mysql created
service/inventory-service created
deployment.apps/mysql created
deployment.apps/inventory-service created

$ kubectl get pods #1
NAME                      READY   STATUS    RESTARTS   AGE
inventory-service-9cccd459b-pl6dr  1/1     Running   2 (44s ago)   50s
mysql-6b4f8fd9d5-4kdcc        1/1     Running   0          50s

$ kubectl port-forward deployment/inventory-service 31013:8083
Forwarding from 127.0.0.1:31013 -> 8083
Forwarding from [::1]:31013 -> 8083
```

#1 Verify the pods are created and running.

We can now access the running Inventory service at <http://localhost:31013/q/graphq1-ui> or check its logs with `kubectl logs deployment/inventory-service`. Except for having to define YAML for the MySQL resources (which would be most likely provided for us in the actual production environment), we have a working instance of our application running in Kubernetes without writing a single line of YAML!

AUTOMATING KUBERNETES DEPLOYMENTS

We already saw in the previous section how we can automate image pushing with the application build. Quarkus Kubernetes extension extends it also to the (re)application of generated manifests. With a single build time flag `quarkus.kubernetes.deploy=true`, we can in one command:

1. Build our Quarkus application and its image.
2. Push the built image to the registry.
3. Apply/reapply the generated manifest to the Kubernetes cluster.

This significantly simplifies the general development workflow. So let's try it in the Inventory service. We need to do two things before this flag can work—log into an image registry (if you're not logged in already) and configure the Kubernetes namespace.

Log into a Docker registry like quay.io. If you followed the previous section, you should still be logged in. If not, you can do it with `docker login quay.io`. To configure the Kubernetes namespace, we can utilize the Kubernetes extension configuration, which we must add to `application.properties`. Use the name of your namespace.

```
quarkus.kubernetes.namespace=default
```

Now, we can run the following command, which builds the Inventory service, builds its image, pushes the image to quay.io, and redeploys our manually deployed Inventory service to Kubernetes:

```
$ ./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

Note that this command doesn't consume the MySQL services. But that is also intended since we typically deploy dependent services once and only make changes to our application. You can also check your quay.io to see the new version of the deployed image. As an exercise, make some changes and rerun the above command to see that they propagate to the latest version of your deployment in your Kubernetes.

Quarkus Kubernetes extension is compelling. We only touched the surface. Many configurations can plug into the generated manifests that simplify real Kubernetes work. Next, we move to OpenShift, which gives us access to even more options.

11.3.4 Deploying Quarkus applications on OpenShift

Openshift (<https://www.redhat.com/en/technologies/cloud-computing/openshift>) is Red Hat branded version fork of Kubernetes that provides additional features, easier management, and (opinionated) better usability. Most of the features we covered for Kubernetes in the previous sections apply also to OpenShift. We decided to focus on OpenShift because it comes with new resources, intuitive UI, and, most importantly, a free public OpenShift instance called Sandbox that you can use for your testing! We don't need to install anything (besides the client-side tool, `oc`) on your local systems to experiment with Quarkus on OpenShift!

In this section, we also focus on the deployment of only a single service (Inventory) in more detail, and we will get back to deploying the whole system with less detailed explanations in section [11.6](#). The exercise in this section is also optional. Section [11.6.1](#) provides the instructions for setting up OpenShift Sandbox environment. You might choose to use Sandbox also here, but if you would like to try OpenShift on your local system, you can use the *CodeReady Containers* (CRC, <https://developers.redhat.com/products/openshift-local/overview>). CRC provides functionality similar to Minikube for Kubernetes. No matter which option you choose, you must have the `oc` (openshift client) executable locally available (see the instructions in section [11.6.1](#)).

We continue with the Inventory service that we already prepared for Kubernetes deployment. Quarkus Openshift extension (`quarkus-openshift`) is an alternative that provides the same functionality for OpenShift. Let's remove Kubernetes and add OpenShift extension in the `inventory-service`:

```
$ quarkus ext remove kubernetes
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✅ Extension io.quarkus:kubernetes has been uninstalled
$ quarkus ext add openshift
[SUCCESS] ✅ Extension io.quarkus:quarkus-openshift has been installed
```

However, if you try to build the Inventory service now, it will fail because, as you might remember, the OpenShift extension is also an alternative to image build extensions (section [11.2](#)). For this reason, we also need to remove the Docker image extension, or we can also provide the following configuration to use openshift by default:

```
quarkus.container-image.builder=openshift
```

OpenShift extension behaves similarly to Kubernetes, but in addition to the original Kubernetes files (`kubernetes.yml/json`), which didn't change, it also generates files called `openshift.yml` and `openshift.json`. All files are in the same directory (`target/kubernetes` for Maven or `build/kubernetes` for Gradle). We focus on `openshift.yml`. The following listings show a simplified version of the generated `openshift.yml`. The full version is available for reference in the `chapter-11/openshift/openshift-default.yml` file of the book resources.

Listing 11.13 Simplified version of the generated openshift.yml

```
---  
apiVersion: v1  
kind: Service  
...  
  name: inventory-service  
...  
---  
apiVersion: image.openshift.io/v1  
kind: ImageStream  
...  
  dockerImageRepository: registry.access.redhat.com/ubi8/openjdk-17  
...  
---  
apiVersion: image.openshift.io/v1  
kind: ImageStream  
...  
  name: inventory-service  
  namespace: default  
spec:  
  dockerImageRepository: quay.io/quarkus-in-action/inventory-service  
...
```

This first part starts with the `Service` definition, which is the same as in `kubernetes.yml`. The following two resources are called `ImageStream`. `ImageStream` is a custom OpenShift resource that provides an abstraction of container images. This will be important later as we can use it to redeploy our application automatically. There are two defined `ImageStreams`, one for JDK and one for our Inventory service image. They are utilized by the `BuildConfig` resource, which we can see in [Listing 11.14](#).

Listing 11.14 Continuation of the simplified generated openshift.yml

```

---
apiVersion: build.openshift.io/v1
kind: BuildConfig
...
  name: inventory-service
  namespace: default
spec:
  output:
    to:
      kind: ImageStreamTag
      name: inventory-service:1.0.0-SNAPSHOT
  source:
    binary: {}
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: openjdk-17:1.17
---
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
...
  name: inventory-service
  namespace: default
spec:
  ...
  spec:
    containers:
      ...
        - name: JAVA_APP_JAR
          value: /deployments/quarkus-run.jar
    image: quay.io/quarkus-in-action/inventory-service:1.0.0
    imagePullPolicy: Always
  ...
triggers:
  - imageChangeParams:
      automatic: true
      containerNames:
        - inventory-service
    from:
      kind: ImageStreamTag

```

```
name: inventory-service:1.0.0
type: ImageChange
```

This is where OpenShift excels. The BuildConfig configuration defines a source to image (s2i) build of the Inventory service based on the JDK image we saw in [Listing 11.13](#). We can understand JDK-based s2i for our applications in this way:

1. We build a JAR locally.
2. The JAR is pushed to OpenShift.
3. OpenShift builds the application Docker image in the OpenShift build.
4. The new image is run as deployment.

From our perspective, we build the project, and OpenShift takes care of everything else. The last part, the *DeploymentConfig*, is the last custom OpenShift resource that puts everything together. It defines the `inventory-service:1.0.0` quay.io image as the image it runs from. However, it also defines a trigger that automatically redeloys our application when there is an image change in the `inventory-service:1.0.0-SNAPSHOT` ImageStreamTag that is produced by the BuildConfig. So every time we trigger a s2i build, it automatically redeploys the deployment.

MANUAL DEPLOYMENT WITH GENERATED OPENSHIFT.YML

Similarly, as with Kubernetes, the generated `openshift.yml` would deploy Inventory service right away, but we would still miss the MySQL connection. We can also include the `mysql.yml` manifest in the `src/main/kubernetes` directory to fix it. It just has to be named `openshift.yml` as every file in this directory is applied to the resource of the same name. We can copy `kubernetes.yml` to `openshift.yml` in this directory without additional changes.

Furthermore, since we now use the OpenShift extension, we must also adjust the configuration that creates the environment variable that points to MySQL in `application.properties`. We can also create an OpenShift custom resource called *Route*, which exposes our deployment to the outside world. Also, we must remove or adjust the Kubernetes namespace to correspond to our project in OpenShift if it differs.

```
quarkus.openshift.env.vars.quarkus-datasource-jdbc-url =
  → jdbc:mysql://mysql:3306/quarkus

# Expose created service to the world
quarkus.openshift.route.expose=true
```

After building the application and manifests, we can now directly apply the generated `openshift.yml`, which produces a bunch of resources with the `oc` tool like this:

```
$ oc apply -f target/kubernetes/openshift.yml
```

And now we can verify that the Inventory service runs in Openshift with:

```
$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
inventory-service-1-bmprt   1/1     Running   0          16m
inventory-service-1-deploy   0/1     Completed  0          16m
mysql-6b4f8fd9d5-8lqh4      1/1     Running   0          16m

$ oc get routes
NAME    HOST/PORT  PATH  SERVICES  PORT  TERMINATION  WILDCARD
inventory-service <your-url>  inventory-service  http  None
```

OpenShift also comes with a bundled web console. If you don't know where it runs, you can find it with this command:

```
$ oc whoami --show-console
```

It looks as presented in [Figure 11.3](#).

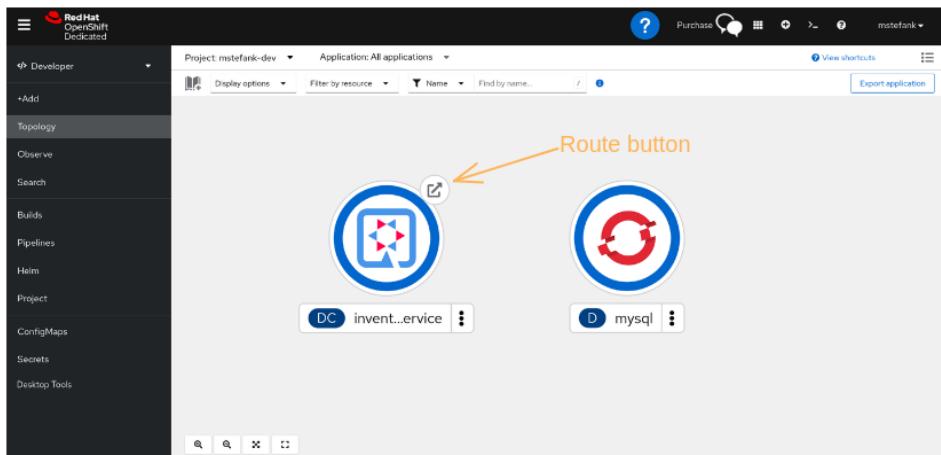


Figure 11.3 OpenShift's web console

We can navigate to the exposed Inventory service route URL by going to `<your-url>` or clicking the route button as shown in the [Figure 11.3](#). Change the path to `http://<your-url>/q/graphql-ui`, and you will see the familiar UI of the Inventory service now running in OpenShift! We didn't configure SSL, so the route exposes HTTP (`http://`). Routes might open on HTTPS instead, so we need to adjust it.

TIP Modern browsers might come with a preconfigured *HSTS* (HTTP Strict Transport Security) that forces *HTTPS*. To workaround this policy, you can configure your browser to turn off this setting, but you can also open *HTTP* URLs in incognito window.

AUTOMATING OPENSOURCE DEPLOYMENT WITH S2I

We prepared the DeploymentConfig to trigger application redeploy when the s2i build finishes. However, because we configure `quarkus.container-image.*` properties, Quarkus automatically picks them to align generated manifests to point to the deployed image. If we want to utilize the image built with s2i, we can remove them. To remove the need to have them specified every time we build or push images (if we decide to do so again), we can define a custom `s2i` configuration profile that removes these properties as shown in the [Listing 11.15](#).

Listing 11.15 The s2i configuration profile removing container image properties

```
%s2i.quarkus.container-image.registry=
%s2i.quarkus.container-image.group=
%s2i.quarkus.container-image.tag=
```

If you now regenerate `openshift.yml` with the `s2i` profile active, it points to `inventory-service:1.0.0-SNAPSHOT` image, which corresponds to the ImageStreamTag produced by our `BuildConfig`. To trigger the `s2i` build, we can run the following command that sets the required flags to use our new config profile (in addition to the `prod` profile) and to trigger an `s2i` build:

```
$ ./mvnw clean package -Dquarkus.profile=prod,s2i
- -Dquarkus.openshift.deploy=true
```

This will take a minute. In the terminal, you can see that the running OpenShift s2i build consuming logs from OpenShift (you can also find it in the OpenShift web console). You can follow how it builds the final Inventory service image.

TIP You can remove all deployed resources from OpenShift with `oc delete all --all`.

After it finishes, you can see a different pod replaced our original pod. Note the suffix of the `inventory-service` pod.

```
$ oc get pods --field-selector status.phase=Running
NAME                  READY   STATUS    RESTARTS   AGE
inventory-service-2-41zp9   1/1     Running   0          77s
mysql-6b4f8fd9d5-c5vwx     1/1     Running   0          79s
```

Our OpenShift workflow is now complete. In section [11.6](#) we look into what we need to do to get the full car rental deployment running in OpenShift. But the hard part is now over. If you understand these basics OpenShift concepts, we can start our journey to get our production running in OpenShift.

11.4 Kubernetes and OpenShift clients

So far, we have utilized various management tools (`kubectl` or `oc`) that have allowed us to manage our Kubernetes or OpenShift platforms. While there are other ways to deploy Quarkus applications (e.g., OpenShift UI or `oc new-app` command), they all require manual steps. Quarkus also allows us to define our deployment workflow in Java directly in the Quarkus application. For this reason, Quarkus provides two extensions — `quarkus-kubernetes-client` and `quarkus-openshift-client` that integrate *Fabric8* clients (<https://github.com/fabric8io/kubernetes-client>). Fabric8 library provides programmatic access to the Kubernetes and OpenShift REST API.

These extensions allow us to CDI inject an instance of either `KubernetesClient` or `OpenShiftClient`. They provide a custom *DSL* (domain-specific language) for accessing Kubernetes or OpenShift, respectively. We don't dive into details here since this is specific to your cloud platform. But for example, if we would like to list all running pods in our OpenShift instance from the last section (`oc get pods --field-selector status.phase=Running`), we can do it as shown in [Listing 11.16](#).

Listing 11.16 Example usage of the quarkus-openshift-client extension

```
@Inject
OpenShiftClient openShiftClient;

@GET
@Path("/pods")
public List<Pod> pods() {
    return openShiftClient.pods().inNamespace("mstefank-dev")
        .withField("status.phase", "Running").list().getItems();
}
```

This DSL provides access to the complete Kubernetes/OpenShift API. If we chose to, we could write our full deployment pipelines and processing in Java with Quarkus cloud client extensions (compared to the traditional approach with generated manifests and CLI tools, it would allow more flexibility, but it would most likely be more complicated and require more code).

11.5 Serverless application with Quarkus

At the beginning of the book, we introduced Quarkus as a framework for microservices and serverless applications. So far, we have covered microservices extensively, but we only touched a little on Quarkus's serverless story. It was intentional since Manning has a separate book *Serverless in Action* focusing on all available Quarkus' serverless integrations. For this reason, the coverage of this topic here is reduced to demonstrating the very basics of how to start with Quarkus serverless, but we won't dive into deeper details.

11.5.1 Serverless architecture

Serverless architecture consists of smaller application deployments that can independently scale up or down depending on user demand. This application scaling means that we can adjust the running application resources according to the user requests, which means that we don't need to pay for unused resources (when there is no demand). When the demand is high, the serverless platform automatically spins up new instances, and when the traffic decreases, it can stop some instances to scale the application down again.

Naturally, application scaling extends to the scale to zero functionality where there are no instances of a particular application running. When a user request comes in, the serverless platform starts a new instance of the user application that can handle the request (aka cold start). The application's startup time adds to the time handling of the (first) request, which means that the serverless application must start fast (nobody would like to wait several dozens of seconds for a page load).

This is where "serverless" comes in since many traditional servers are big and slow to start. Especially in Java, the platforms were designed to run for months (or years) at the time, so starting in a few minutes or consuming a few GBs of memory was fine. Java was less popular for serverless models. But the technologies like Quarkus have started to change it. Quarkus's focus on low memory footprint and ahead-of-time compilation makes it possible to write Java serverless applications. Compilation to GraalVM native images extends this further to make even smaller and faster-to-start applications. Since serverless applications typically don't run for long (by default, one minute; see below), native binaries are an excellent fit for serverless application deployment. As we already saw with Quarkus, creating a native executable is just a matter of a build time flag.

11.5.2 Funqy

There are several serverless platforms (aka *FaaS*, Function as a Service) on the market, e.g., Knative, AWS Lambda, Google Cloud Functions, or Azure Functions. Each provides its own Java API that can be utilized in your Quarkus application. But each API is different, meaning using such API locks your application to a particular platform.

Quarkus's serverless strategy bases on the library called *Funqy*. *Funqy* provides a unified API that abstracts all FaaS platforms mentioned above. It represents a very simple API that we can use to develop Quarkus serverless applications.

TIP Even if *Funqy*, as we use it in this book, is exposed through HTTP in a similar way to REST services, it doesn't serve as a replacement for REST. It doesn't expose all REST features. But if you need complete REST, you can still use *Funqy* together with the REST extensions provided by Quarkus.

The core of the API consists of an annotation called `@Funq`. It defines a method that the extension invokes when the user calls the function. The [Listing 11.17](#) presents an example of this API.

Listing 11.17 The example use of the Funqy API

```
import io.quarkus.funqy.Funq;

public class Function {

    @Funq
    public String process(String input) {
        return "Processed: " + input;
    }
}
```

We can also use POJOs (plain old Java objects) as inputs/outputs (they need to have a default constructor and getter+setters), use reactive Mutiny types, or use dependency injection. But this is it. This simplicity allows us to port this kind of function to different FaaS providers without issues because Quarkus takes care of the various integrations.

11.5.3 Car statistics application with Knative

In this section, we develop a new but totally optional service called `car-statistics` that we deploy as a serverless function with Knative. We chose Knative as the easiest platform to utilize since it is preinstalled already in the OpenShift Sandbox (refer to section [11.6.1](#) for setup instructions) in its Red Hat product version called OpenShift Serverless. It calls our deployed Inventory service through GraphQL to return some statistics of currently tracked cars. Since a service like this doesn't need to run all the time, it's a perfect target for serverless because at times when we don't need the data, we don't need to keep the application running.

TIP The development part in this section is optional. If you're using the OpenShift sandbox, you can start right away. If you're using custom OpenShift or Kubernetes, you must install Knative manually (<https://knative.dev/docs/install/>).

Let's create a new Quarkus `car-statistics` application:

```
$ quarkus create app org.acme:car-statistics -P 3.6.0 --extension
→ funqy-http,quarkus-smallrye-graphql-client,openshift --no-code
```

We need three extensions — Funqy API, GraphQL client to call the Inventory service, and OpenShift integration for seamless Knative deployment to Openshift. Open the project in the IDE.

We start by copying the `org.acme.reservation.inventory` package (three classes) from the Reservation service to the `org.acme.statistics.inventory` package in `car-statistics` since the logic of calling Inventory service through its exposed GraphQL API is the same. Feel free to implement it from scratch as an exercise. You can always verify it with the code in the Reservation service.

Regarding configuration, we need to add the following configuration properties to `application.properties`:

Listing 11.18 The car-statistics configuration.

```
# Needed to generate Knative resources
quarkus.kubernetes.deployment-target=knative

# Inventory service URL on localhost
quarkus.smallrye-graphql-client.inventory.url=http://localhost:8083/graphql
# Inventory service location in OpenShift
quarkus.knative.env.vars.quarkus-smallrye-graphql-client-inventory-url=
-http://inventory-service/graphql

# Needed to configure where Knative pulls the image
quarkus.container-image.registry=
-image-registry.openshift-image-registry.svc:5000
quarkus.container-image.group=<your-openshift-namespace>

# Max concurrent requests to a single pod
quarkus.knative.revision-auto-scaling.container-concurrency=2
```

TIP If you're using the prepared solution from the book's resources, don't forget to change the quarkus.container-image.group to your OpenShift namespace.

The quarkus.kubernetes.deployment-target comes from the underlying Kubernetes extension (since Knative is not specific to OpenShift). It specifies what resources Quarkus generates. The possible values include kubernetes, openshift, knative, minikube, and kind or any combination of these values. We also need to configure the name of the image included in the generated Knative resources. Since we will utilize s2i, we point it to the internal OpenShift registry and your namespace/project. But this could also be configured to the external image we could pull (we also pushed the image to quarkus-in-action quay.io). Lastly, we configure the maximum concurrent requests per single pod/container. It also allows us to try scaling to multiple application instances if there are three or more concurrent requests.

Now, we can create our Funqy function. Create a new class org.acme.statistics.CarStatistics as presented in [Listing 11.19](#).

Listing 11.19 The code of the CarStatistics class utilizing Funq API

```

package org.acme.statistics;

import io.quarkus.funq.Funq;
import io.smallrye.graphql.client.GraphQLClient;
import io.smallrye.mutiny.Uni;
import jakarta.inject.Inject;
import org.acme.statistics.inventory.GraphQLInventoryClient;

import java.time.Instant;

public class CarStatistics {

    @Inject
    @GraphQLClient("inventory")
    GraphQLInventoryClient inventoryClient;

    @Funq
    public Uni<String> getCarStatistics() {
        return inventoryClient.allCars()
            .map(cars -> ("The Car Rental car statistics created at %s. " +
                "Number of available cars: %d")
            .formatted(Instant.now(), cars.size()));
    }
}

```

And now we can test our function in Dev mode. You will also need a running instance of Inventory service (Dev mode is sufficient). When ready, call <http://localhost:8080/getCarStatistics> to get these simple car statistics.

11.5.4 Car Statistics Knative deployment

The final step is to push our function to the cloud. We use the OpenShift Sandbox that comes with Knative preinstalled. With Quarkus, we can do this in a single command:

```
$ ./mvnw clean package -Dquarkus.knative.deploy=true
```

This command triggers the s2i build of our car-statistics application that we configured our Knative function to point to in [Listing 11.18](#). You can verify this in the generated knative.yml/json files in the target/kubernetes directory.

After the s2i build finishes, we can see the car-statistics Knative service deployed in our OpenShift instance, as shown in [Figure 11.4](#).

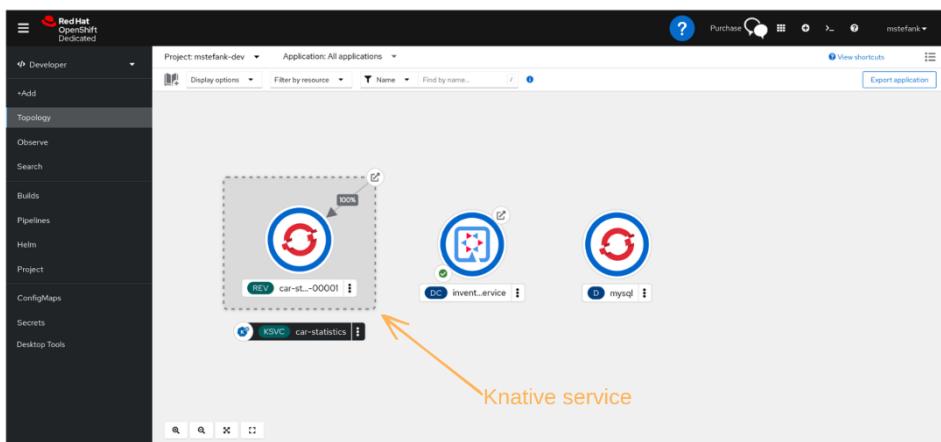


Figure 11.4 The deployed car-statistics Knative service in OpenShift

You can now freely call the exposed route to get back the statistics of our currently tracked cars. For example, by clicking the route button or by http:

```
$ http http://<your-route>/getCarStatistics
...
"The Car Rental car statistics created at 2024-01-22T14:25:25.257426983Z.
→ Number of available cars: 2"
```

By default, we have the scale-to-zero functionality enabled. If you don't call our car-statistics Knative application for one minute, Knative automatically scales it to zero replicas by stopping the running pod instance. [Figure 11.5](#) demonstrates this scenario.

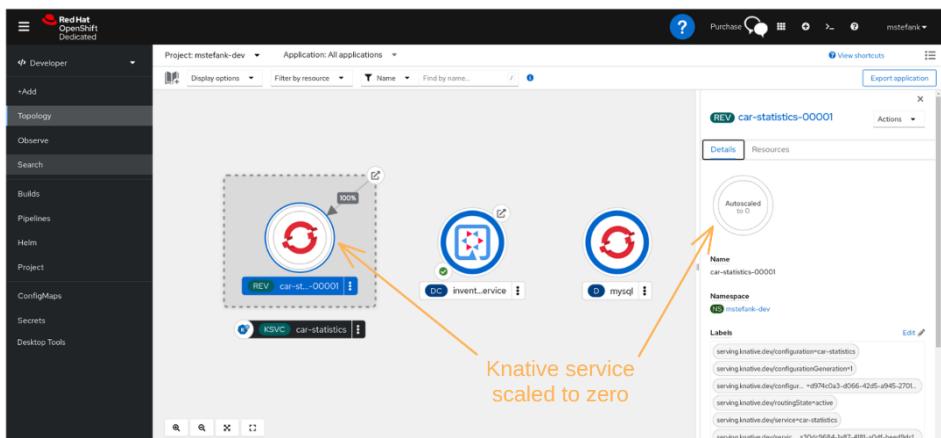


Figure 11.5 The car-statistics Knative deployment scaled to zero

But if we now access the exposed route again (in the browser or terminal), a new pod is started and handles our request! Notice how fast it is (only a few seconds). Remember, this is still Quarkus in JVM. It could be even faster if we compiled `car-statistics` to native. You can easily try this, it's just a matter of adding `-Dnative` to the build command of the `car-statistics` application. The native build will take longer, but then the startup will be extremely quick.

As an exercise, try to invoke the deployed function with two or more concurrent requests to see how the `car-statistics` application scales up when it is not able to keep up with the requests load (if you know how to make concurrent requests; e.g., repeating calls in three concurrent loops). When you call it concurrently, you can see multiple pods being created as shown in [Figure 11.6](#). If executing concurrent requests is hard because the application is too fast to respond, just add an artificial `Thread.sleep` into the `getCarStatistics` method to slow down the response.

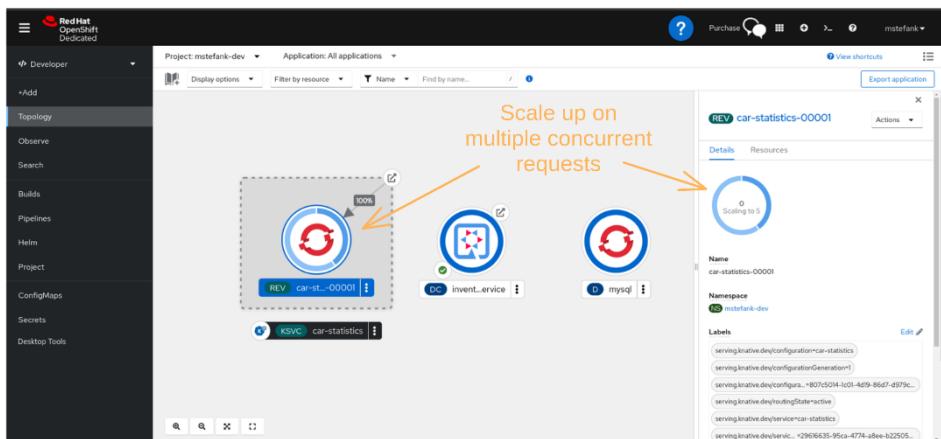


Figure 11.6 The `car-statistics` scale up with multiple concurrent requests

This is the serverless magic. We are running our `car-statistics` application only when we have requests, saving resources when we don't need to call it.

11.6 Deploying Car rental in the cloud

We are now ready to deploy our Car rental application to the cloud! We use the OpenShift Sandbox that we learn how to set up in the following section. The Sandbox is a free, testing, development-targeted, and shared OpenShift cluster with 14 GB of RAM and 40 GB storage for 30 days. It also comes with everything we need for our deployment (e.g., Knative operator) preinstalled.

11.6.1 Setting up OpenShift Sandbox

To start with OpenShift Sandbox, navigate to <https://dn.dev/rhd-sandbox> and click the Start your sandbox for free button as shown in [Figure 11.7](#).

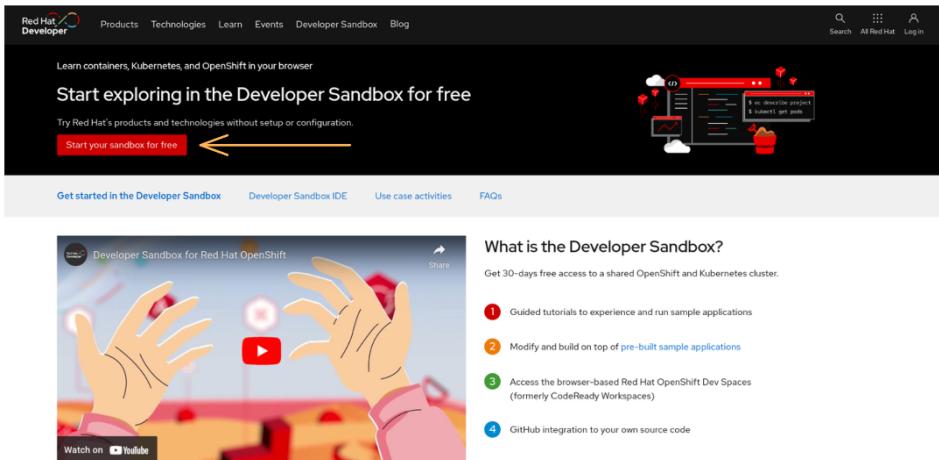


Figure 11.7 OpenShift Sandbox landing page

It redirects you to authenticate with your Red Hat Developer account. If you don't already have one, you can create it. It is completely free and without the need for credit card information.

TIP We will need a password later to log into the OpenShift instance. If you create an account with an external provider (Google, Microsoft, or GitHub), you must set up (reset) your password later. Preferably, create the password with registration.

After you log in, you land in the Hybrid Cloud Console, where you need to click the Get started button. The console might require to verify you with an SMS. However, afterward, you can see the Launch button for Openshift as shown in [Figure 11.8](#) that opens your OpenShift.

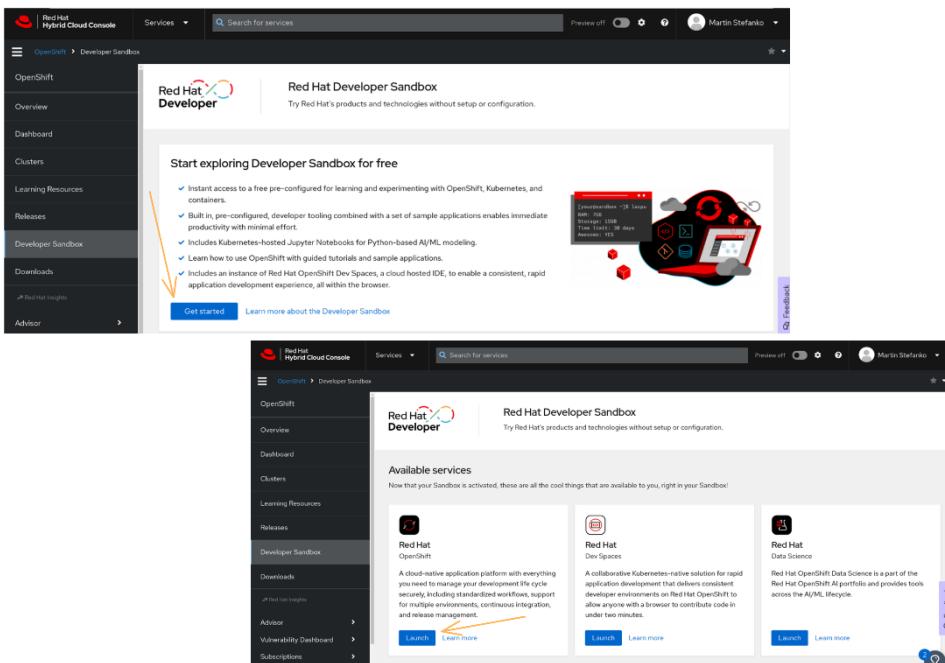


Figure 11.8 Hybrid Cloud Console verifications

We are now in the OpenShift. In Sandbox, we always log in with the *DevSandbox* account into the `<your-username>-dev` project (namespace). After you log in (by clicking the *DevSandbox* button), you land in the OpenShift UI as demonstrated in [Figure 11.9](#).

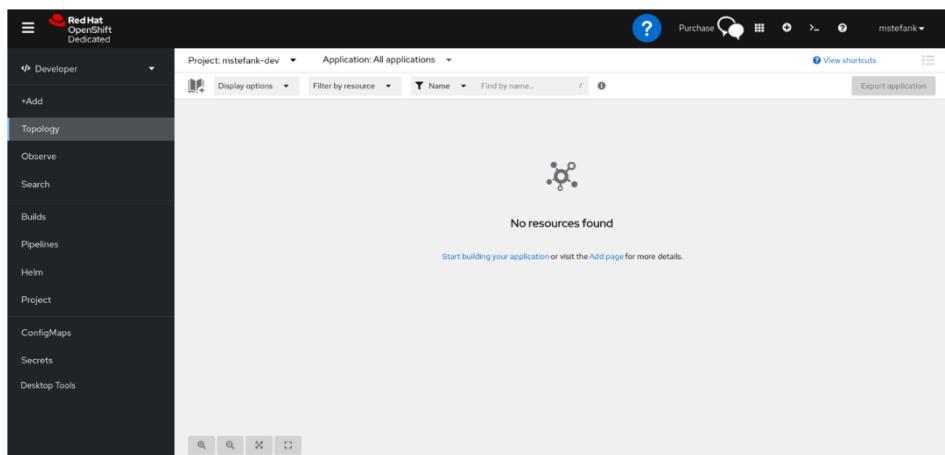


Figure 11.9 OpenShift UI/web console

For now, we focus on the top right corner, where we can find the relevant tools and information that we need to log in to this OpenShift also from local machines. For local login, we need the `oc` (OpenShift client) tool, that you can download from the question mark button, as demonstrated in [Figure 11.10](#).

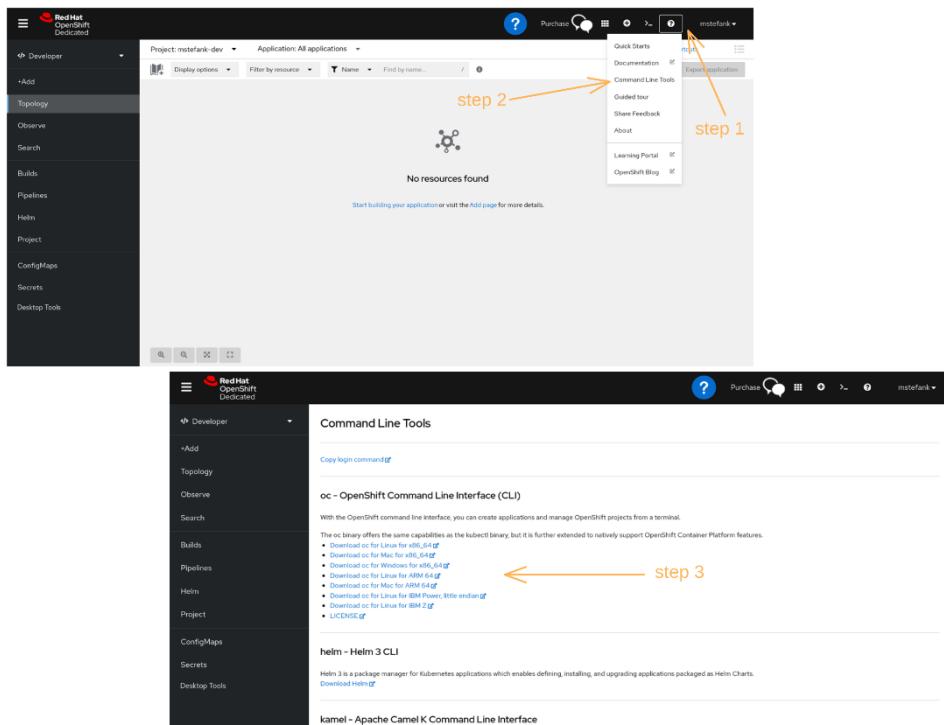


Figure 11.10 Downloading `oc` tool from OpenShift UI

Download the binary for your local system and put it into an executable path. On the last page, you can also see the `Copy login command`, but there is also a more straightforward way accessible from everywhere by opening your profile in the top right corner ([Figure 11.11](#)).

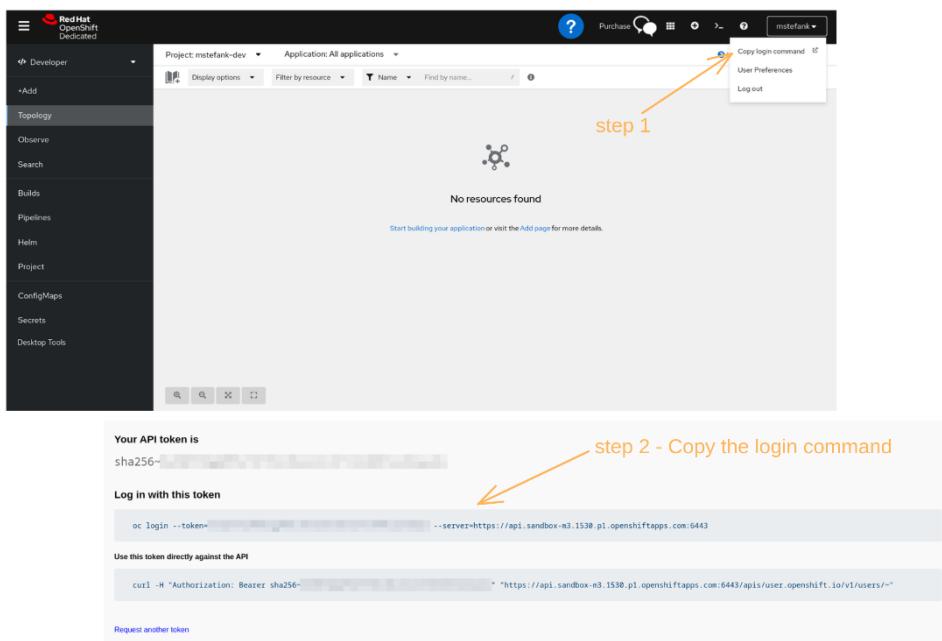


Figure 11.11 Logging into OpenShift from the local system

After you click the `Copy login command` button, a new browser tab again asks you to authenticate with a `DevSandbox` account. Then click the `Display Token` button on the next page, which takes you to the actual token page, also presented in [Figure 11.11](#). Copy the `oc login ...` command and paste it into your local terminal:

```
$ oc login --token=TOKEN --server=SERVER
Logged into "SERVER" as "<username>" using the token provided.
```

The `oc` tool is now connected to your OpenShift sandbox! Let's now deploy the full Acme Car Rental system to your OpenShift.

11.6.2 Car rental infrastructure deployment with OpenShift Sandbox

The Acme Car Rental in OpenShift presents similar challenges as we overcame for the local Docker Compose production deployment. We will still deploy all sixteen car rental containers to OpenShift, but since the OpenShift communication schematics are different (e.g., we are no longer on localhost), we need to adjust our services accordingly.

We start by deploying our infrastructure services. We provided a single, more than three thousand lines long OpenShift manifest that deploys all eleven services we need in one step. You can run it with the `oc` tool as presented in [Listing 11.20](#).

Listing 11.20 Deploying dependent car rental services in OpenShift

```
$ oc apply -f openshift-dependent-services.yml
```

After the command finishes, you can check the OpenShift console that shows all deployed containers running. [Figure 11.12](#) presents a (reorganized) view of the finished deployment.

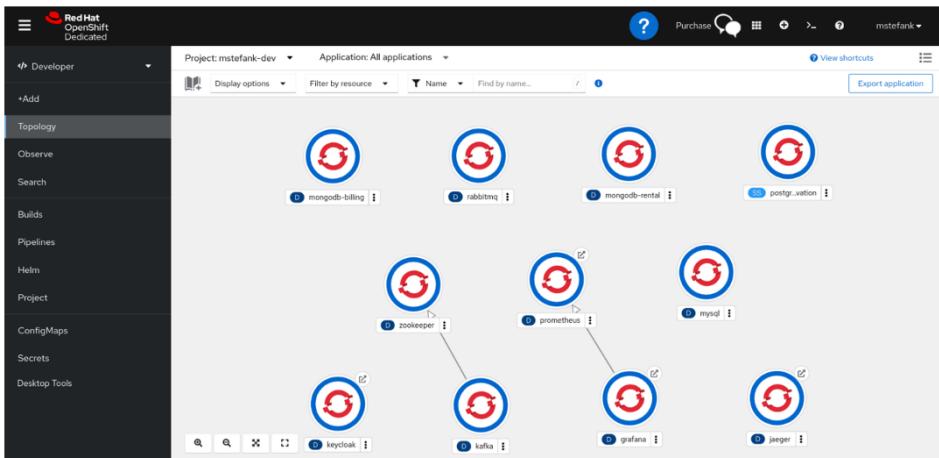


Figure 11.12 Reorganized view of the deployed dependent car rental services

Except for the eleven infrastructure services that you can see in the figure above, we also created four *ConfigMaps* (Kubernetes configuration resource) that host various configuration files for them. You can check them in the `ConfigMaps` tab.

Now, we are ready to start deploying our Quarkus Car rental services. If you need to remove all created resources, [Listing 11.21](#) shows how you can do it with the `oc delete` command utilizing the same manifest.

Listing 11.21 Deleting all created resources with manifest

```
$ oc delete -f openshift-dependent-services.yml
```

ConfigMaps are specific resources, which means they are not removed even with `oc delete all --all`. However, when deleting using the manifest as demonstrated in [Listing 11.21](#), ConfigMaps defined in the manifest will get deleted too.

11.6.3 Car rental services deployment with OpenShift Sandbox

We've come to the final stage of our deployment. Only the five Quarkus car rental services remain to be deployed to the OpenShift Sandbox. We go step-by-step over all services, utilizing the features explained in section [11.3.4](#) to deploy them all to OpenShift Sandbox.

The first thing we need to do is to add the `quarkus Openshift` extension. Run the following `quarkus` command in all Quarkus Car rental services. It might already be in Inventory service if you followed the previous sections.

```
$ quarkus ext add openshift
```

In each service, we also need to apply the same minimum configuration as we did in the Inventory service. Namely, choose `openshift` as image builder extension, define the `s2i` profile that removes the container image specifications, and expose a public route, so we can access our services from outside OpenShift:

```
%s2i.quarkus.container-image.registry=
%s2i.quarkus.container-image.group=
%s2i.quarkus.container-image.tag=

quarkus.container-image.builder=openshift
quarkus.openshift.route.expose=true
```

As in the OpenShift/Kubernetes environment, Deployment resources communicate with each other via the Service resources that assign hostnames, we need to adjust the URL locations of our services. We need to remove all `localhost` references as they don't work in OpenShift.

TIP Quarkus car rental services generate Service resources such that container ports (808x) are mapped to port 80 externally, which means that we need to call (for example) the Reservation service using `reservation-service:80` instead of `reservation-service:8081`. 80 is also the default HTTP port, so it can be omitted, and you can simply use `reservation-service` (just the name of the Service resource) for the URL.

As we learned in the section [11.3.2](#), we can utilize the configuration options of the `quarkus Openshift` extension that allow us to define the environment variables that propagate to the generated manifests to override the configuration hard-coded in the images (inside `application.properties`). Let's adjust the configuration files in the individual car rental services, as demonstrated in the following snippets.

USERS SERVICE

In OpenShift, we must configure Keycloak to its exposed route because any request accessing our application is redirected to this URL. So, it must be accessible from outside the inner cluster. You can find the Keycloak's route with `oc get routes`. The Reservation service is exposed on port 80, configured by its generated Service resource. We also need to adjust the Jaeger URL for OpenTelemetry traces.

```
quarkus.openshift.env.vars.quarkus-otel-exporter-otlp-endpoint=
→http://jaeger:4317
# TODO: Don't forget to insert your Keycloak route here
quarkus.openshift.env.vars.quarkus-oidc-auth-server-url=
→http://<your-keycloak-route>/realms/car-rental
quarkus.openshift.env.vars.quarkus-rest-client-reservations-url=
→http://reservation-service
```

TIP Even if you're using the prepared solutions from the book's GitHub repository, in this case you have to manually update the `quarkus.openshift.env.vars.quarkus-oidc-auth-server-url` because this property is specific to each reader's deployment, and we can't provide a value that will work for everybody. Obtain the route using `oc get route keycloak` after you have deployed the resources from the `openshift-dependent-services.yml` file.

RESERVATION SERVICE

Reservation is a little more complex. We need to change the URLs for Inventory and Rental services, Postgres database, Keycloak (again using your own route the same way as in the Users service), Jaeger, and RabbitMQ.

```
quarkus.openshift.env.vars.quarkus-otel-exporter-otlp-endpoint=
→http://jaeger:4317
quarkus.openshift.env.vars.quarkus-smallrye-graphql-client-inventory-url=
→http://inventory-service/graphql
quarkus.openshift.env.vars.quarkus-datasource-reactive-url=
→vertx-reactive:postgresql://postgres-reservation:5432/quarkus
# TODO: Don't forget to insert your Keycloak route here
quarkus.openshift.env.vars.quarkus-oidc-auth-server-url=
→http://<your-keycloak-route>/realms/car-rental
quarkus.openshift.env.vars.quarkus-rest-client-rental-url=
→http://rental-service
quarkus.openshift.env.vars.rabbitmq-host=rabbitmq
```

RENTAL SERVICE

Similarly, in the Rental service, we need to adjust the locations for the Reservation service, MongoDB, and Kafka.

```
quarkus.openshift.env.vars.quarkus-rest-client-reservation-url=
→http://reservation-service
quarkus.openshift.env.vars.quarkus-mongodb-connection-string=
→mongodb://mongodb-rental:27017
quarkus.openshift.env.vars.kafka-bootstrap-servers=kafka:9092
```

INVENTORY SERVICE

If you followed section [11.3.4](#), Inventory is already ready for the OpenShift deployment. But we still need to update the Jaeger URL for telemetry. We can now remove the provided MySQL manifests from `src/main/kubernetes` since MySQL is deployed with dependent services.

```
quarkus.openshift.env.vars.quarkus-otel-exporter-otlp-endpoint=
→http://jaeger:4317
quarkus.openshift.env.vars.quarkus-datasource-jdbc-url=
→jdbc:mysql://mysql:3306/quarkus
```

BILLING SERVICE

Lastly, the Billing service needs to adjust URLs for its MongoDB, Kafka, and RabbitMQ, as shown in the following snippet. Since we don't need to deal with port conflicts, MongoDB exposes port 27017, the default MongoDB port port.

```
quarkus.openshift.env.vars.quarkus-mongodb-connection-string=
→mongodb://mongodb-billing:27017
quarkus.openshift.env.vars.kafka-bootstrap-servers=kafka:9092
quarkus.openshift.env.vars.rabbitmq-host=rabbitmq
```

We can start deploying these services to the OpenShift Sandbox. For each service, we either need to use the s2i build to create images in the OpenShift sandbox or use `oc apply` with the pushed images (either provided by authors or yours). In every case, remember to log in to your `oc` tool as shown in section [11.6.1](#) before you continue.

For our convenience, since this is a lot of repeating commands, we created shell and bat scripts (`car-rental-s2i.sh/bat`) that automate these tasks in all Car rental services at the same time (adjust the commands for Windows by replacing `/` with `\` and `.sh` with `.bat`). These scripts should be executed from the `chapter-11` directory.

If you get stuck, you can remove all deployed services with `oc delete all --all`. Note that this doesn't remove created ConfigMap objects (for this, you need to run `oc delete -f `openshift-dependent-services.yml``).

DEPLOYING CAR RENTAL SERVICES WITH S2I

As we learned in [11.3.4](#), we can trigger s2i builds with the `-Dquarkus.openshift.deploy=true` flag included in the build command. The entire deployment consists of two steps:

1. Change the directory to the individual service — `cd users-service`.
2. Build with s2i — `quarkus build -Dquarkus.profile=prod,s2i -Dquarkus.openshift.deploy=true` (or `./mvnw clean package` with same flags).

Alternatively, you can deploy all car rental services together with the `car-rental-s2i.sh`/`bat` script:

```
$ ./car-rental-s2i.sh
```

Note that this script takes a few minutes since it runs all tests and runs s2i builds. But after it finishes, you have deployed a complete Acme Car rental system to OpenShift!

DEPLOYING CAR RENTAL SERVICES WITH MANIFESTS

To manually apply generated OpenShift manifests, we need to execute the following sequence of commands in every Car rental service:

1. Build the project — `quarkus build` or `./mvnw clean package` to rebuild the manifests. If you want to push your custom images, then also run `quarkus push` and make sure you've also defined the `quarkus.container-image.group` in all five projects to point at your own quay.io organization. If you use the value `quarkus-in-action`, OpenShift will use the reference images that the authors provided with this book, and thus your deployment will reflect your local changes in manifests, but not in the application code.
2. Apply the generated manifest using `oc apply -f target/kubernetes/openshift.yml`.

As this is, again, quite a tedious task, we provide scripts in the `chapter-11` directory that automate it. We have two scripts, one for the case where you want to use the provided reference images (`car-rental-oc-deploy-reference.sh`), and the other for the case where you want to use your custom images (`car-rental-oc-deploy-custom.sh`).

TIP If the `car-rental-oc-deploy-custom.sh` complains that it can't find the `quarkus` command, maybe you have it defined as an alias in your shell. In that case, run the script using `bash -i car-rental-oc-deploy-custom.sh` instead to make your aliases available for the script.

```
$ ./car-rental-oc-deploy.sh
# ... or:
# make sure you're logged into quay.io with your Docker client!
$ ./car-rental-oc-deploy-custom.sh
```

Alternatively, we have provided a manifest that consumes all prepared images from the `quay.io/quarkus-in-action` organization can also deploy the complete system (including all dependent services, but without applying your custom changes in manifests) with a single command:

```
$ oc apply -f car-rental.yml
```

Whether we use s2i or manual manifests, when the scripts finish, the final Acme Car Rental production runs in the OpenShift sandbox! [Figure 11.13](#) presents our final deployed project running in the cloud.

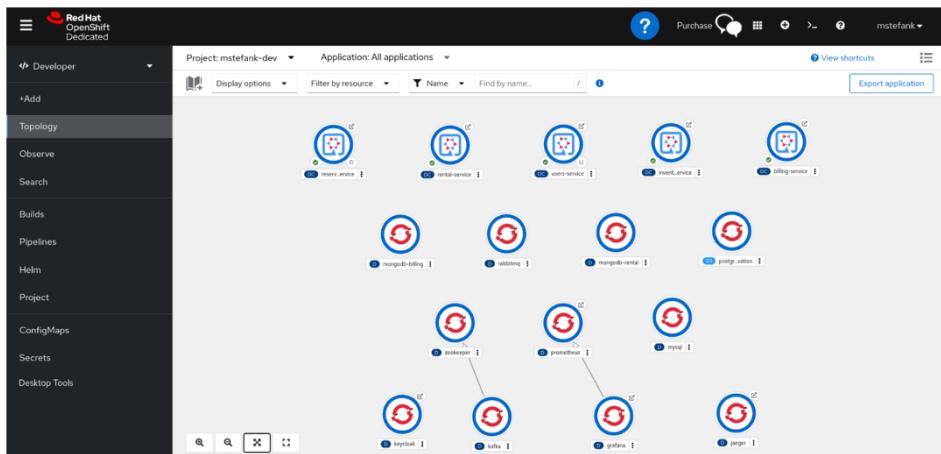


Figure 11.13 Final fully deployed Acme Car Rental system in the OpenShift Sandbox

To tear it down, we can run the following set of commands:

```
$ oc delete -f car-rental.yml # delete all Car Rental resources
$ oc delete all --all # clean remaining build resources (if used s2i)
```

This is the entire Acme Car Rental system that we developed throughout the book in Action! Together, sixteen containers are now running in the cloud. Quarkus puts all building blocks in place to develop, test, and deploy our Quarkus microservices easily.

11.7 Wrap up and next steps

In this chapter, we learned how to use Quarkus to package our applications into Docker images and how to easily deploy them in various deployment environments. We defined the full Car rental production deployment as sixteen applications, of which five are Quarkus car rental services, and eleven are infrastructure-supporting services such as Kafka or Prometheus.

We explained how we can build and deploy Docker images to public registries as quay.io both manually and declaratively with extensions. We were then able to run car rental production locally with Docker Compose.

The Quarkus cloud-native integrations excelled when we started deploying the car rental to various cloud providers. We learned how Quarkus's Kubernetes and OpenShift extensions simplify the deployment of images to the cloud. We also learned about serverless technology and how Quarkus extensions (like Funqy) make switching from microservice to serverless deployment easy.

Lastly, we focused on the OpenShift Sandbox deployment, combining all knowledge accumulated in this chapter to deploy Acme Car Rental production in various ways (s2i, manifests). Looking at the full deployment of sixteen services working in unison is undoubtedly a good reward for everyone following the book's development. This is Quarkus in Action!

11.8 Summary

- Acme Car Rental production consists of sixteen services, where five are Quarkus services we develop throughout the book.
- Quarkus provides four different image-building options (Jib, Docker, BuildPack, and OpenShift). Quarkus CLI allows us to build and push images without requiring extensions, but extensions provide a way to integrate image processing into the build pipeline.
- Quarkus can generate all required Kubernetes, OpenShift, or Knative resources (manifests) without the need to develop them manually. All extensions provide configuration hooks that can customize the final generated resources, which allows for keeping deployment specifications in the configuration.

- OpenShift extension provides two different options of deployments – s2i and manual manifest application. S2i is an OpenShift-specific mechanism that builds an image in the OpenShift provided a copy of a locally packaged application (JAR or native).
- Serverless strategy in Quarkus focuses on the Funqy library. Funqy provides a very simplified API interoperable with many different FaaS platforms.
- OpenShift Sandbox is a free OpenShift cluster that anyone can utilize for development and test purposes. Because of the Quarkus cloud-native integrations, the full Acme Car Rental production deployment rollout to this platform can be fully automated.