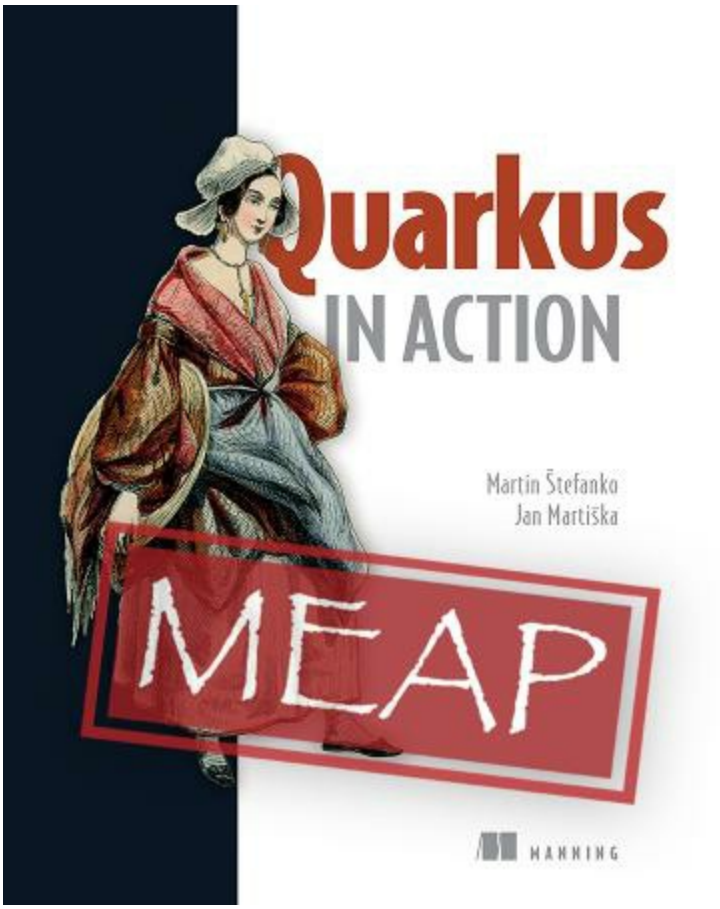
Quarkus IN ACTION

Martin Štefanko
Jan Martiška

MEAP

MANNING

Quarkus in Action MEAP V03



MEAP VERSION 3

MANNING PUBLICATIONS

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 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

In this book

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 of these 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 that targets microservices and serverless system development. Quarkus emerged as an alternative to the existing Java microservices stacks 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, as developers know a technology’s productivity and usability are what is appreciated 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 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 or Scala (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.

1.2 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.

1.2.1 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

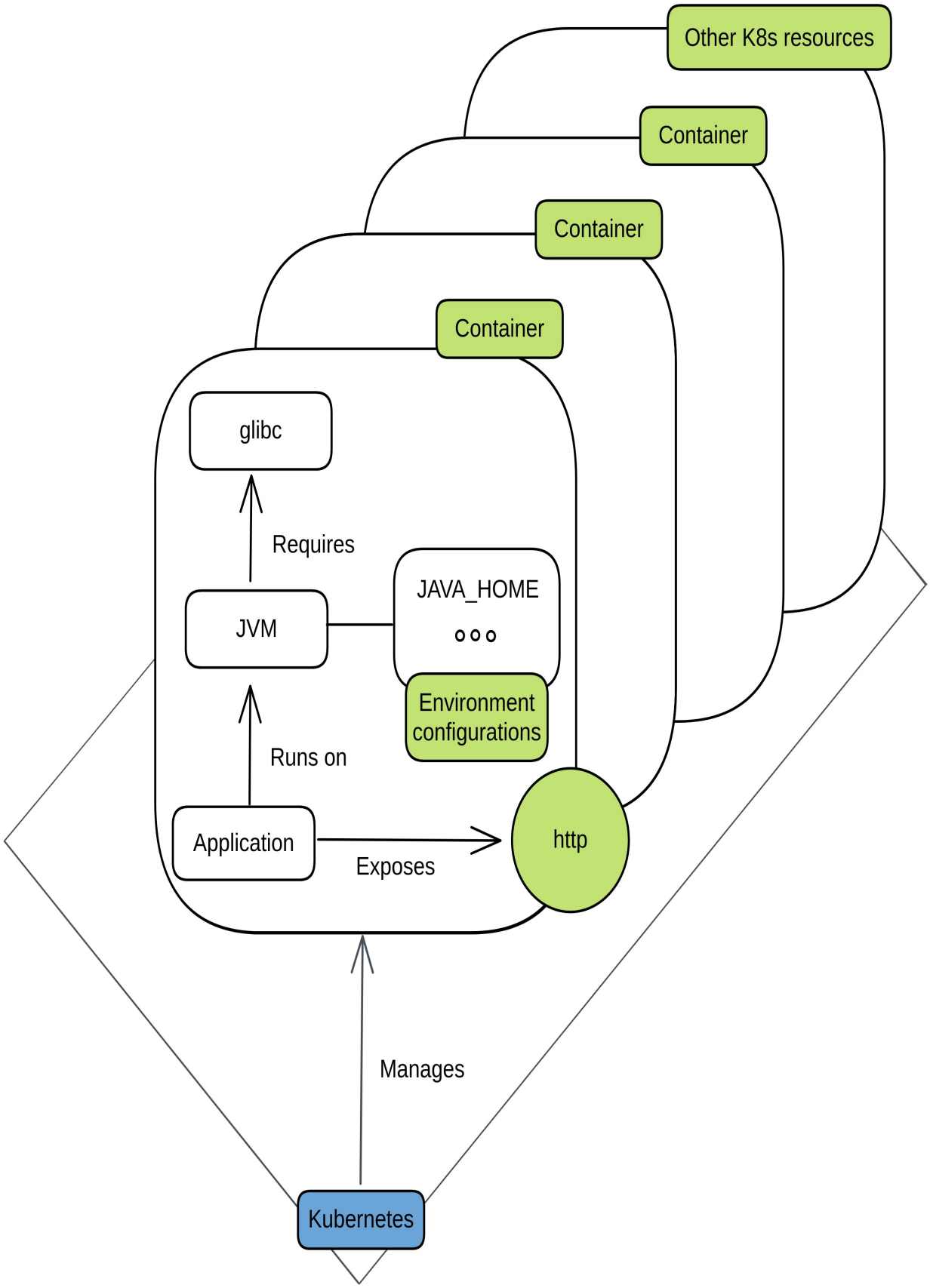
becoming increasingly popular, the need for container management appeared, and that's precisely what Kubernetes is: the de-facto standard container management platform.

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.

In addition to any application exposed entry point (e.g., an HTTP API), the containers can also define some optional hooks that help Kubernetes with their management. For instance, the application can expose a health check endpoints that Kubernetes uses to decide whether the container should be restarted or if it can consume requests (Quarkus provides this functionality for you if you need it). Figure 1.1 depicts a standard JVM container 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.

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 deducted from the container specification as, for instance, the `JAVA_HOME` shown in Figure 1.1 or we can manually pass them to the container 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.

Figure 1.1. Visualizing the relationships between the application, container, and Kubernetes


Other K8s resources
Container
Container
Container
glibc
Requires
JAVA_HOME
ooo
JVM
Environment configurations
Runs on
Application
Exposes
http
Manages
Kubernetes

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 targets are 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 tool-set 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, be small, and Java frameworks typically don't live up to the task. Quarkus, on the other hand, 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 is unique to Quarkus, which 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.

1.2.2 OpenJDK HotSpot and GraalVM

OpenJDK HotSpot refers to the Java Virtual Machine (JVM), the most common runtime for Java applications. GraalVM is a JVM and development kit distribution that brings improved performance, support for multiple languages, and native image 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 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 (as is the case for most alternative frameworks).

1.2.3 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/) or 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 to ease the migration paths for developers that need to adjust to the MicroProfile and Jakarta EE APIs, which might be new to them.

1.3 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

1.3.1 Imperative and reactive programming seamlessly connected together

Most 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.

1.3.2 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 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.

1.4 The Quarkus architecture

Let’s take a closer look at the key components of the Quarkus architecture that is visualized in Figure 1.2. 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.

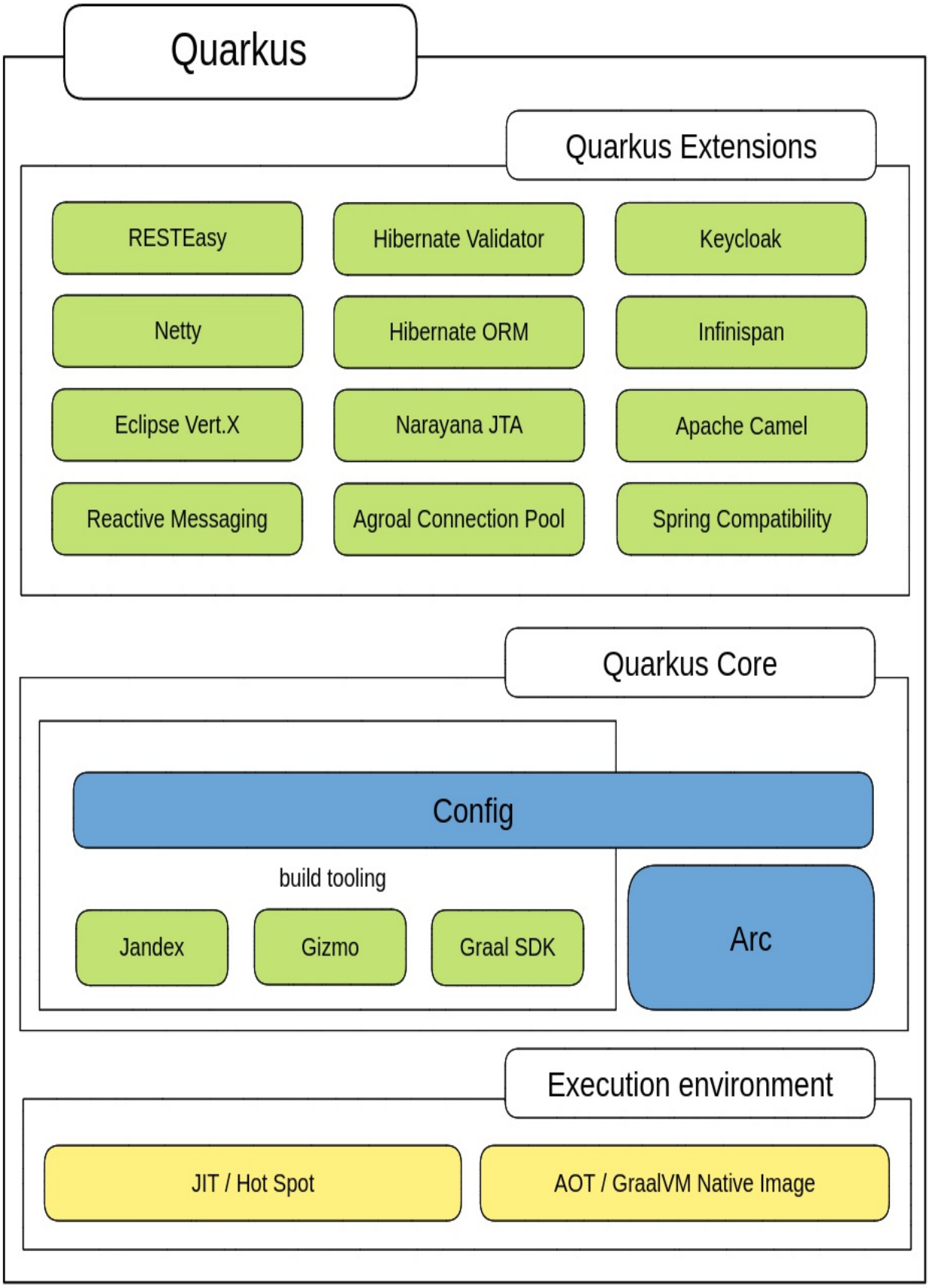
On top of this layer are the critical components used in the Quarkus build tooling. Jandex is a library that provides the indexing of Java classes to a class model representation that is memory efficient and fast to search. It also allows the build tool to do all the heavy lifting related to code metadata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

Gizmo is a bytecode manipulation library. Bytecode is the output of the Java compiler the Java Virtual Machine 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, 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 configurable 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.

Figure 1.2. The overview of the Quarkus architecture


Quarkus
Quarkus Extensions
RESTEasy
Hibernate Validator
Keycloak
Netty
Hibernate ORM
Infinispan
Eclipse Vert.X
Narayana JTA
Apache Camel
Reactive Messaging
Agroal Connection Pool
Spring Compatibility
Quarkus Core
Config
build tooling
Jandex
Gizmo
Graal SDK
Arc
Execution environment
JIT / Hot Spot
AOT / GraalVM Native Image

1.5 Alternative frameworks

This section covers 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.

1.5.1 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 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. Similarly, as in Quarkus, it performs ahead-of-time processing to evaluate decisions that are usually taken at runtime to create a native binary. 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 solves regarding GraalVM compilations are often resolved in Quarkus.

1.5.2 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. Indeed, Micronaut takes the more modern approach called ahead-of-time compilation, which offers noticeable performance gains. Similarly as in Quarkus, Micronaut 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).

1.5.3 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.

1.5.4 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.

1.6 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.

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.

1.6.1 Use cases

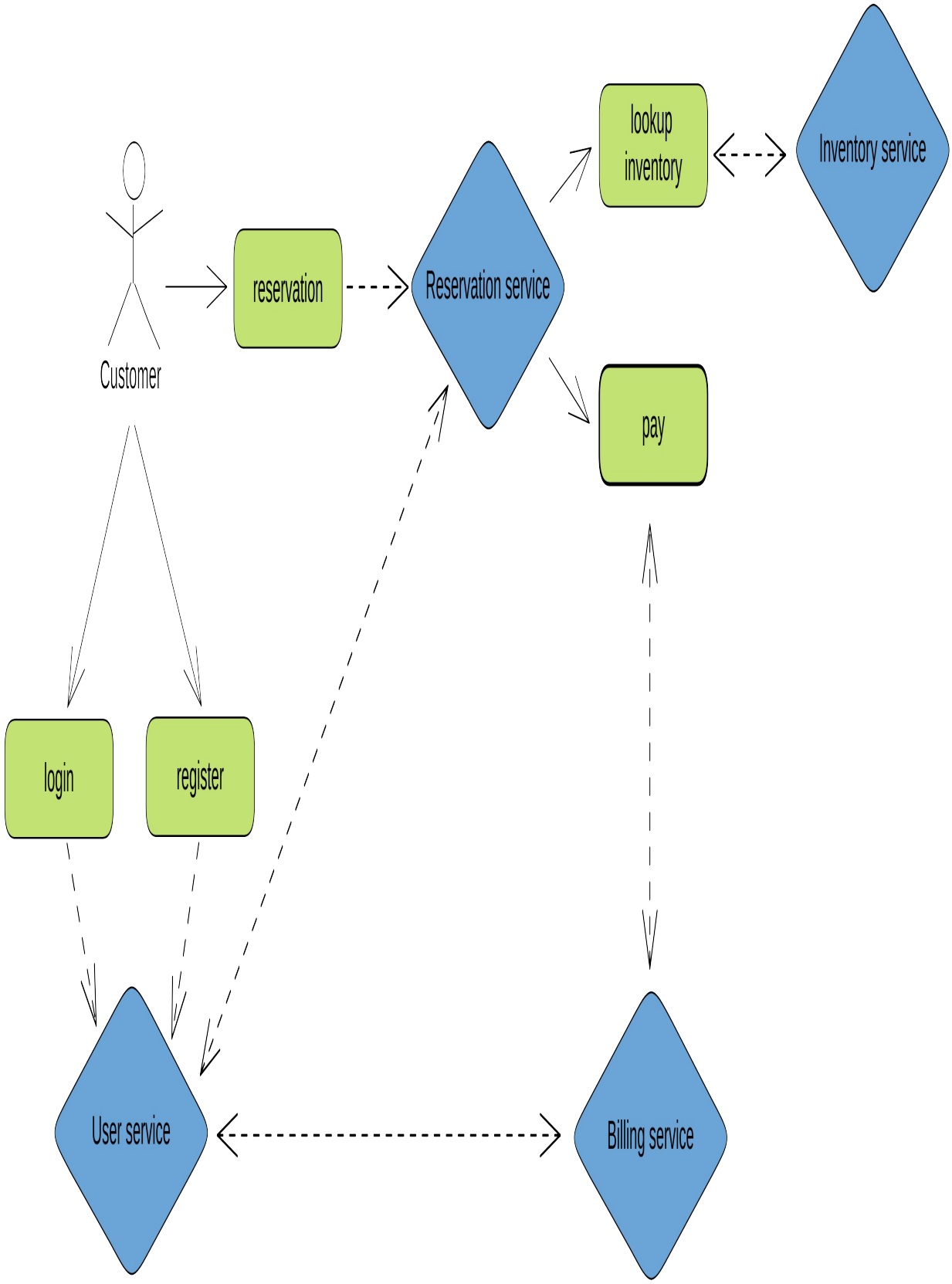
To get an idea of the car rental application's essential features, let's evaluate the primary use cases that cover user management, billing, fleet management, and integration areas. It is important to understand the problems we are solving first before diving into the actual solutions.

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 1.3. 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

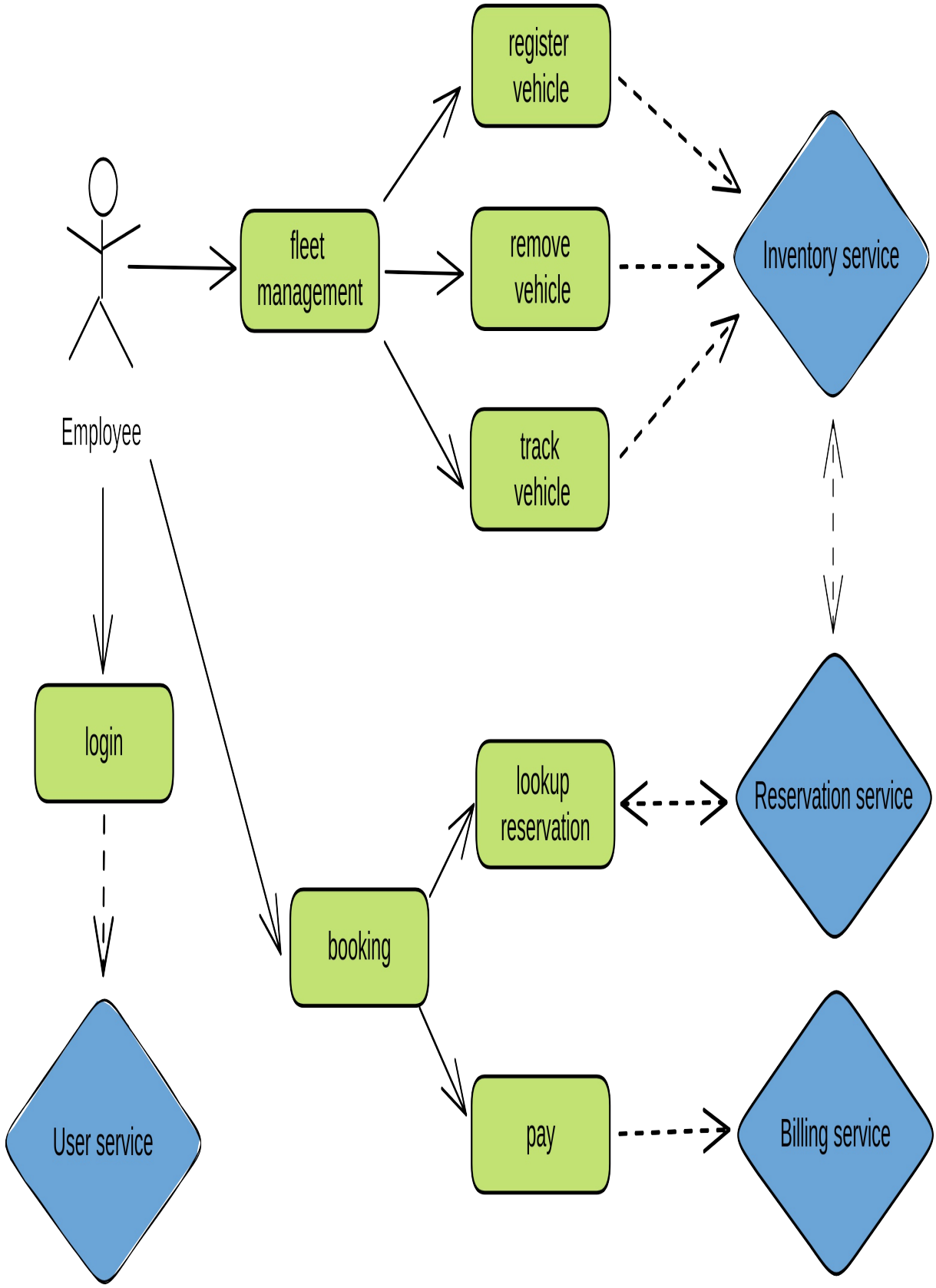
and register options provided by the User service. This service acts as the user entry API that separates the rest of the services. Making the reservation is a more complicated process for which the customers need to log into the system. The Reservation service then provides cars received from the Inventory 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.

Figure 1.3. Modeling actions performed by customers


Customer
reservation
Reservation service
lookup inventory
Inventory service
pay
login
register
User service
Billing service

What about employees? What is their interaction with the system now that most of the functionality is passed directly to the customers? Figure 1.4 provides a visual representation of the employee use cases and how they tie into the core system services. 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. Employees also have access to a console that allows them to track the location of all vehicles provided by the Inventory service.

Figure 1.4. Modeling actions performed by employees


register vehicle
fleet management
remove vehicle
track vehicle
Inventory service
Employee
login
lookup reservation
Reservation service
booking
User service
pay
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.

1.6.2 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, AMQP, and Kafka.

Figure 1.5 shows how the services communicate with each other. Real-life applications usually limit themselves to just a couple of different communication technologies, depending on their needs. The car rental application, however, is designed for teaching purposes and thus will not have such limits. It covers as many communication technologies as possible.

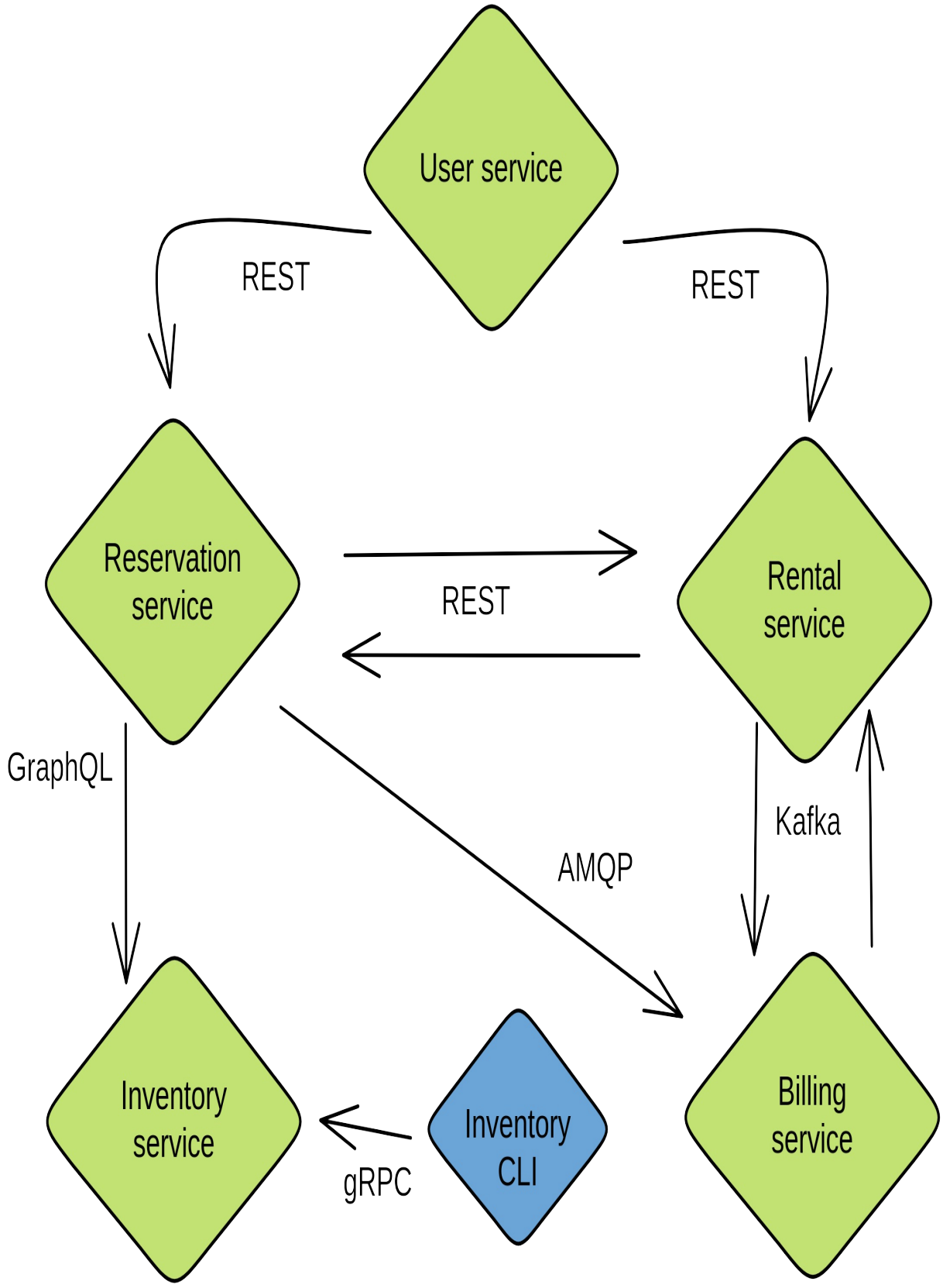
The same applies to database technologies. Each service utilizes different data store ranging from traditional relational to NoSQL (non-SQL) databases.

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 AMQP 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.

Figure 1.5. Car Rental architecture


User service
REST
REST
Reservation service
REST
Rental service
GraphQL
AMQP
Kafka
Inventory service
Inventory CLI
gRPC
Billing service

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.

1.6.3 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 17, which is the latest LTS (Long Term Support) release version available at the time of this writing.

 **Note**

Quarkus 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.

1.7 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.

1.8 Summary

- Quarkus is a full-stack Java framework for developing modern 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.
- Quarkus provides excellent support for building native applications.
- The new development workflow called Dev mode introduced in Quarkus adds to 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. 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.

 **Note**

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 version of the plugin also determines the version of Quarkus used in the generated project. The following `mvn` command invocation will create a new Quarkus project generated in the `quarkus-in-action` directory.

The `resteasy-reactive` extension represents a default REST layer implementation 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.

```
$ mvn io.quarkus.platform:quarkus-maven-plugin: 2.14.1.Final:crea
    -DplatformVersion=2.14.1.Final \ #2
    -DprojectGroupId=org.acme \ #3
    -DprojectArtifactId=quarkus-in-action \ #4
    -Dextensions="resteasy-reactive" #5
```

Important

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.

This command generates a Maven project in the `quarkus-in-action` directory. 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 check all available parameters of the `create` goal with `mvn io.quarkus.platform:quarkus-maven-plugin:2.14.1.Final:help -Ddetail=true -Dgoal=create`

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 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 2.14.1.Final \
  --extension resteasy-reactive
```

 **Note**

The `-P 2.14.1.Final` 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, 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`.

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 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.

The starter website is detailed in Figure 2.1. 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 (`groupId`, `artifactId`, and `version`) 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.

Figure 2.1. Quarkus Generator interface


QUARKUS
2.14
io.quarkus.platform
Platform version
Back to quarkus.io
Available with Enterprise Support
GAV, JDK, and build tool of your project
CONFIGURE YOUR APPLICATION
Group org.acme
Version 1.0.0-SNAPSHOT
Option to generate sample code
Artifact code-with-quarkus
Java Version 17
Build Tool Maven
Starter Code Yes
CLOSE
0
Generate your application (alt + ⏎)
Filters
origin:platform
Extensions list
Web
RESTEasy Reactive [quarkus-resteasy-reactive]
STARTER-CODE
A JAX-RS implementation utilizing build time processing and Vert.x. This extension is not compatible with the quarkus-resteasy extension, or any of the extensions that depend on it.
RESTEasy Reactive Jackson [quarkus-resteasy-reactive-jackson]
Jackson serialization support for RESTEasy Reactive. This extension is not compatible with the quarkus-resteasy extension, or any of the extensions that depend on it

The generator also provides a REST API exposed at https://code.quarkus.io/api. 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" \
--output 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.

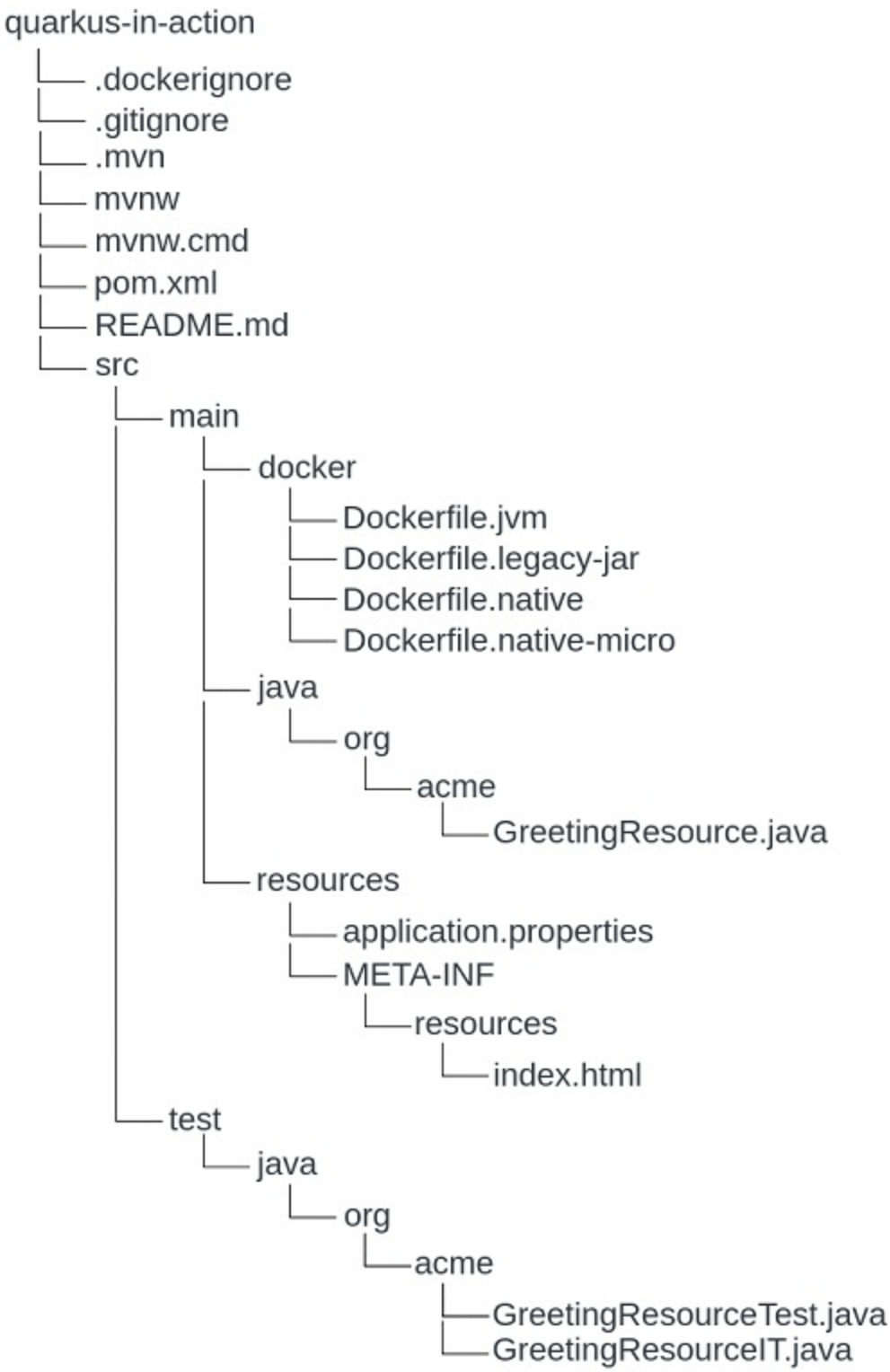
When you learn about new technology, it is best to start with the general overview. If you list the files of the generated `quarkus-in-action` application, the directory structure appears as shown in the Figure 2.2. Depending on your background, this might seem like a lot or maybe not that much. But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

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).

Figure 2.2. Quarkus application directory structure


quarkus-in-action
.dockerignore
.gitignore
.mvn
mvnw
mvnw.cmd
pom.xml
README.md
src
main
docker
Dockerfile.jvm
Dockerfile.legacy-jar
Dockerfile.native
Dockerfile.native-micro
java
org
acme
GreetingResource.java
resources
application.properties
META-INF
resources
index.html
test
java
org
acme
GreetingResourceTest.java
GreetingResourceIT.java

Note

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>` tag listing mostly the Quarkus extensions, all represented as Maven artifacts. If you examine the contents of this tag 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.

Listing 2.1. Quarkus `<dependencies>` tag extension example

```
<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>
  ...
</dependencies>
```

Note

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 `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>
```

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>
```

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. 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 (`javax.ws.rs.*`). The `GreetingResource` exposes a single HTTP GET /hello call by its API.

Listing 2.4. `GreetingResource` class—generated sample JAX-RS endpoint

```
package org.acme;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.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
    }
}
```

Moving on to the `src/main/resouces` 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`.

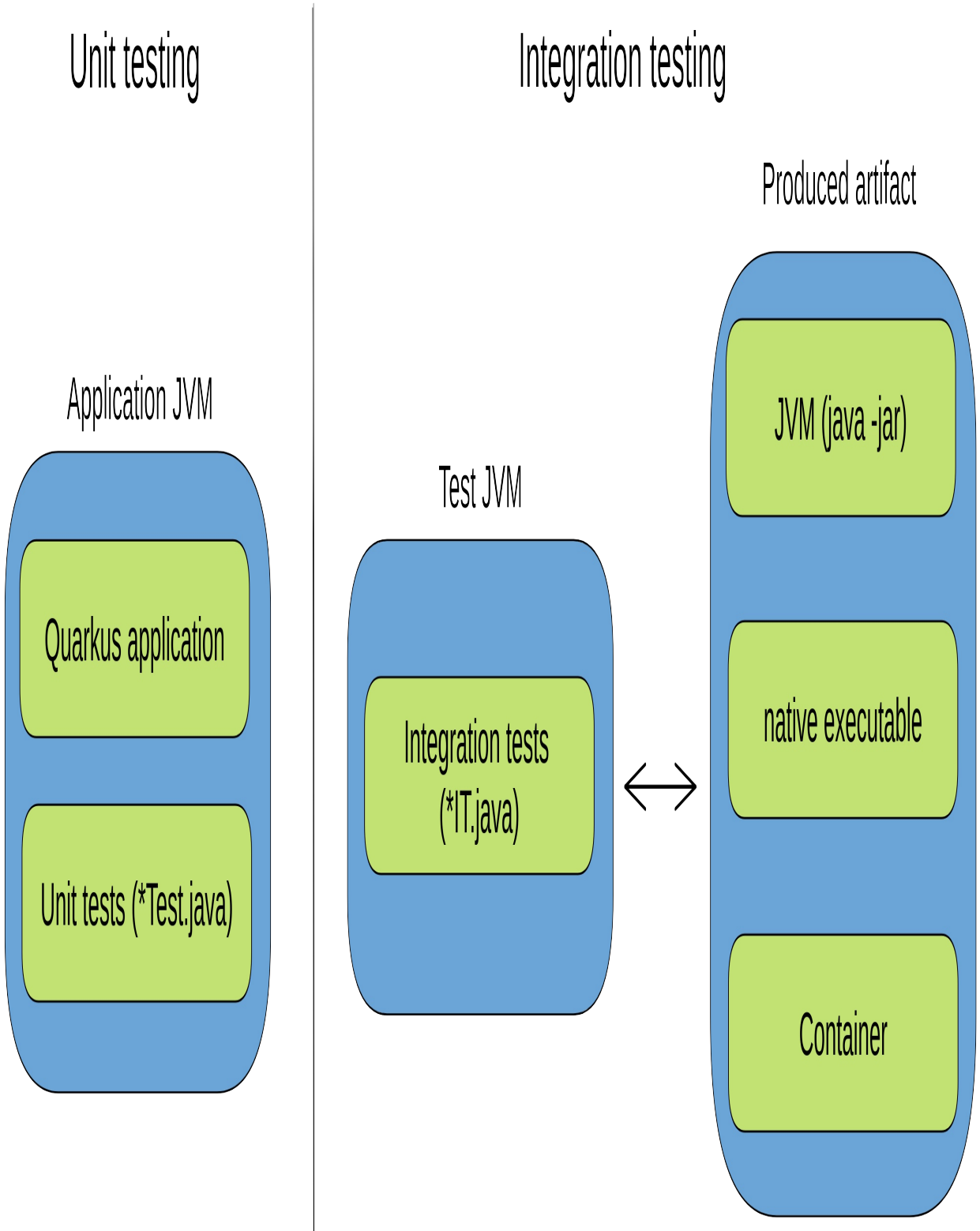
In the only test subdirectory `src/test/java`, which contains the Java test source files, two Java classes are generated in the same package `org.acme`. The `GreetingResourceTest` class provides a sample of the Quarkus test class. Its source code is available in the Listing 2.5. The test class is annotated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

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
    }

}
```

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). 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.

Figure 2.3. Unit versus integration testing with Quarkus


Unit testing
Integration testing
Produced artifact
Application JVM
Quarkus application
Unit tests (*Test.java)
Test JVM
Integration tests
(*IT.java)
JVM (java -jar)
native executable
Container

2.3 Running the Quarkus Application

So far, we have learned how Quarkus functions and what are the contents of the Quarkus application. However, this does not translate to management measurable quotas. So let's run the Quarkus application we generated in the previous sections and investigate where Quarkus excels.

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 compilations 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. When you execute the `quarkus-in-action` application in Dev mode, you can see the output similar to Listing 2.6 in the console.

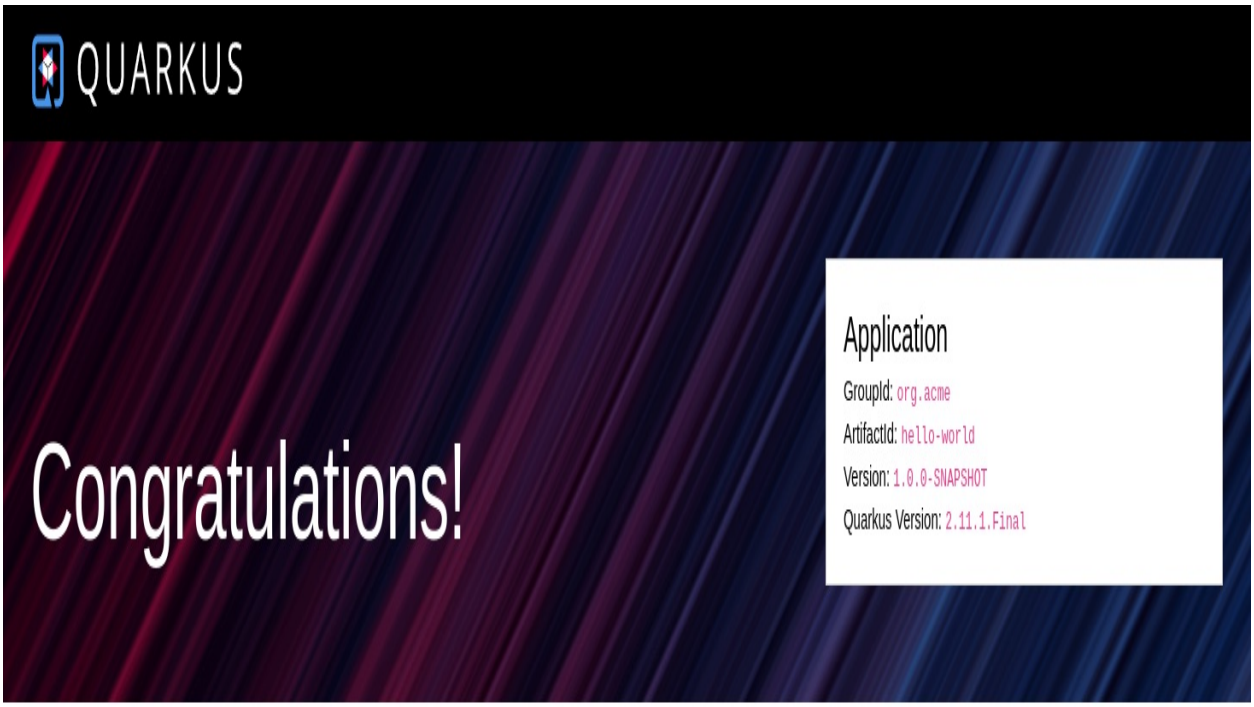
You might notice that we have two additional extensions listed in the "Installed features" list at the end of the output - `smallrye-context-propagation`, responsible for correctly passing contexts between threads and `vertx`, the underlying reactive engine of Quarkus. These extensions are implicitly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

Listing 2.6. The output of the Quarkus application started in Dev mode

```
$ quarkus dev
...
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2022-09-12 11:19:39,034 INFO  [io.quarkus] (Quarkus Main Thread)
quarkus-in-action 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.14.
started in 1.089s. Listening on: http://localhost:8080

2022-09-12 11:19:39,055 INFO  [io.quarkus] (Quarkus Main Thread)
dev activated. Live Coding activated.
2022-09-12 11:19:39,056 INFO  [io.quarkus] (Quarkus Main Thread)
features: [cdi, resteasy-reactive, smallrye-context-propagation,
```

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.

Figure 2.4. Quarkus welcome page



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

Selected extensions

- RESTEasy Reactive (guide)

Documentation

Practical step-by-step guides to help you achieve a specific goal. Use them to help get your work done.

Set up your IDE

Everyone has a favorite IDE they like to use to code. Learn how to configure yours to maximize your Quarkus productivity.

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`) or HTTPie (`http :8080/hello`) to get 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 2.7. The CLI again has the advantage of abstracting from the actual build tool in use.

Listing 2.7. 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` directory which follow the standard (in our case, 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, which can be utilized, for instance, 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 2.8 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 2.8. Running Quarkus with `java -jar`

```
$ java -jar target/quarkus-app/quarkus-run.jar
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2022-09-12 14:10:30,865 INFO  [io.quarkus] (main) quarkus-in-acti
1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.14.1.Final) started i
Listening on: http://0.0.0.0:8080
2022-09-12 14:10:30,888 INFO  [io.quarkus] (main) Profile prod ac
2022-09-12 14:10:30,888 INFO  [io.quarkus] (main) Installed featu
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.

 **Note**

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)

Mandrel is a downstream 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.

Important

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

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 2.9 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 2.9. 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
...
```

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 (section 2.4.3).

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 2.10 depending on the used build tool and preference.

Listing 2.10. 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 2.11 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 2.11. Running Quarkus native executable binary

```
$ ./target/quarkus-in-action-1.0.0-SNAPSHOT-runner
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2022-09-12 15:37:19,641 INFO  [io.quarkus] (main) quarkus-in-acti
1.0.0-SNAPSHOT native (powered by Quarkus 2.14.1.Final) started i
Listening on: http://0.0.0.0:8080
2022-09-12 15:37:19,642 INFO  [io.quarkus] (main) Profile prod ac
2022-09-12 15:37:19,642 INFO  [io.quarkus] (main) Installed featu
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.

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 2.12.

Listing 2.12. 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.NativeImageBuildContainer
Using docker to run the native image builder
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildContainer
Checking image status quay.io/quarkus/ubi-quarkus-native-image:22
```

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]
find the `native-image` in the GRAALVM_HOME, JAVA_HOME and System
Install it using `gu install native-image` Attempting to fall bac
container build.
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildContainer
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.

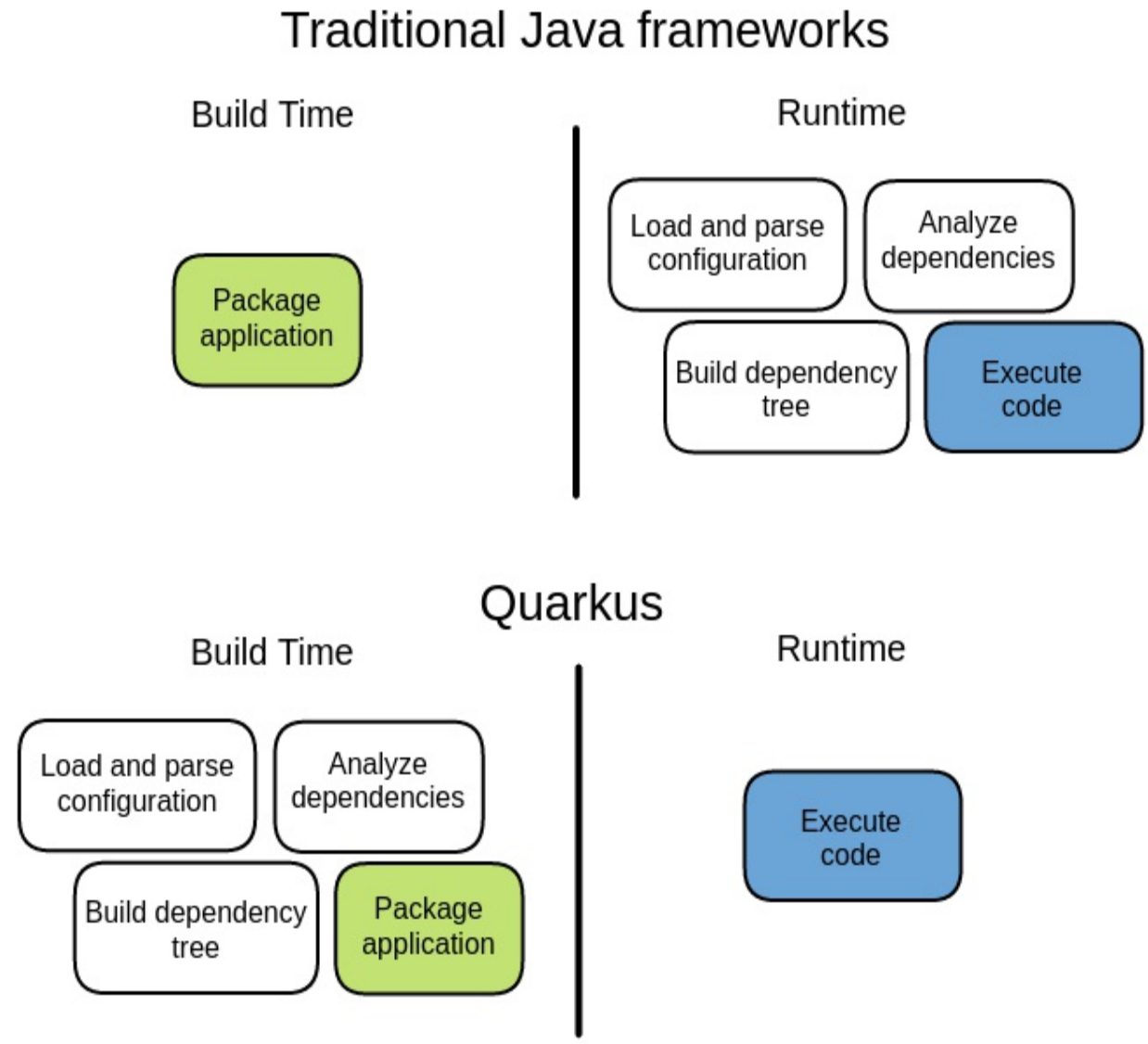
As you probably noticed, Quarkus starts really fast and that’s not the last performance benefit Quarkus provides. So how does Quarkus achieve such a great performance? Quarkus utilizes a concept of build time processing. This means that Quarkus aims to move as much processing as possible to build time (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.

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.

Figure 2.5. Quarkus build time vs runtime processing



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.

Note

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 2.13), we can analyze the startup time logged when the application is starting.

Listing 2.13. Running Quarkus in JVM mode as runnable JAR

```
$ java -jar target/quarkus-app/quarkus-run.jar
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2022-09-13 10:41:28,530 INFO  [io.quarkus] (main) quarkus-in-acti
1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.14.1.Final) started i
Listening on: http://0.0.0.0:8080
```

Wow! Starting any Java application in just 0.557 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. Listing 2.14 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 2.14. Quarkus JVM memory utilization

```
$ ps -eo command,rss | grep quarkus
java -jar target/quarkus-ap 117436
```

Again, a JVM application utilizing only 117 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 2.15 indicates both the startup time and the RSS usage of the generated Quarkus native binary.

Listing 2.15. Quarkus performance measurements of the native executable

```
$ ./target/quarkus-in-action-1.0.0-SNAPSHOT-runner
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2022-03-17 18:44:07,686 INFO  [io.quarkus] (main) quarkus-in-acti
1.0.0-SNAPSHOT native (powered by Quarkus 2.14.1.Final) started i
Listening on: http://0.0.0.0:8080

$ ps -eo command,rss | grep quarkus
./quarkus-in-action-1.0.0-S 38744
```

And now this is really impressive! Just 18 milliseconds to start and only 38 MB of RAM used? These values make Java comparable with generally faster (scripting) languages like Node.js or Go. Adding 18 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?

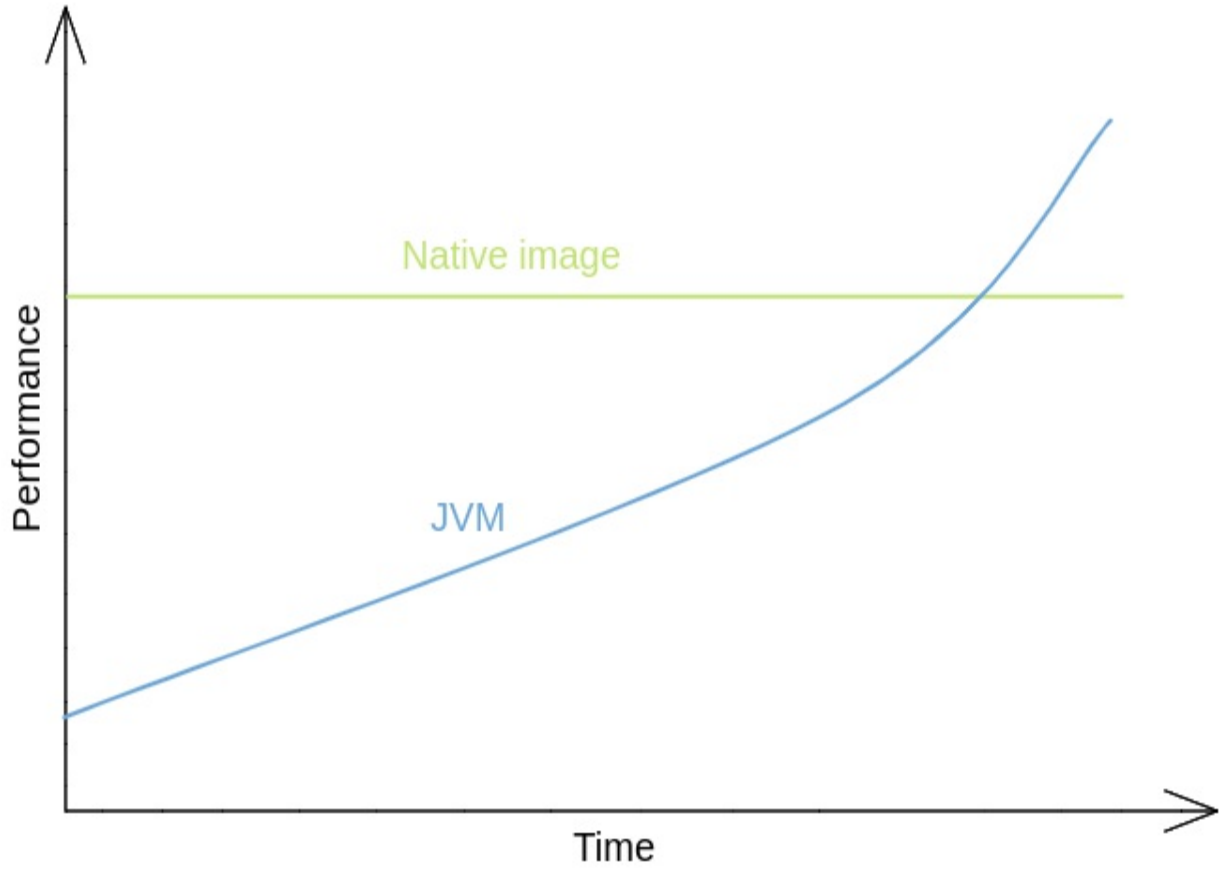
The answer is no. Using a native image or a runnable JAR should always depend on the target use case. Why? Because of the way how JVM (JIT - Just-in-time compiler) optimizes your code when the application runs. Depending on the execution paths of your application, it can eventually outperform the native 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.

 **Note**

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 (or Podman) 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 2.16. 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.

Listing 2.16. `Dockerfile.native-micro` Dockerfile

```
####
# This Dockerfile is used in order to build a container that runs
# Quarkus application in native (no JVM) mode.
# It uses a micro base image, tuned for Quarkus native executable
# It reduces the size of the resulting container image.
# Check https://quarkus.io/guides/quarkus-runtime-base-image for
# 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"]
```

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.

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 2.17. In Quarkus projects, the extensions are managed as Maven dependencies. The `io.quarkus:quarkus-resteasy-reactive` Maven dependency represents the `resteasy-reactive` extension.

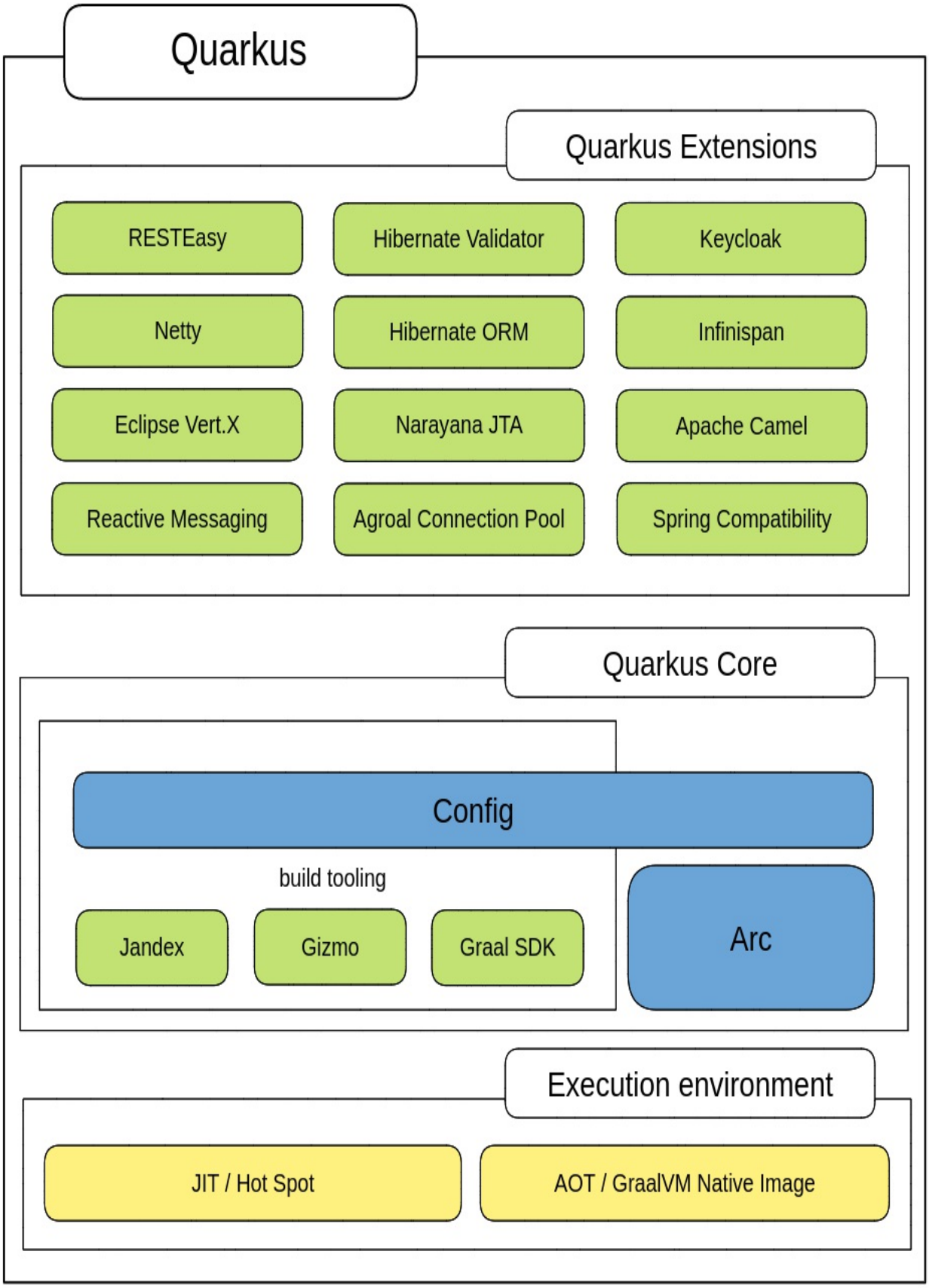
Listing 2.17. Quarkus RESTEasy Reactive extension Maven dependency

```
<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. 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.

Figure 2.7. Quarkus architecture


Quarkus
Quarkus Extensions
RESTEasy
Hibernate Validator
Keycloak
Netty
Hibernate ORM
Infinispan
Eclipse Vert.X
Narayana JTA
Apache Camel
Reactive Messaging
Agroal Connection Pool
Spring Compatibility
Quarkus Core
Config
build tooling
Jandex
Gizmo
Graal SDK
Arc
Execution environment
JIT / Hot Spot
AOT / GraalVM Native Image

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 2.8) there are four extensions in the list - `cdi` (ArC), explicitly added `resteasy-reactive`, and two implicit extensions: `smallrye-context-propagation` and `vertx` which are added as a dependencies of the `resteasy-reactive` extension.

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 2.18 which executed in the `quarkus-in-action` application directory correctly outputs the `quarkus-resteasy-reactive` extension we installed at the project creation.

Listing 2.18. 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 Na
✯ quarkus-resteasy-reactive                               RESTEasy Rea

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 2.19.

Listing 2.19. The installable extensions list

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

✯ ArtifactId                                              Extension Na
✯ blaze-persistence-integration-quarkus                   Blaze-Persis
✯ camel-quarkus-activemq                                  Camel Active
...
```

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 2.20.

Listing 2.20. 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 2.21.

Listing 2.21. Listing all extensions in the messaging category

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

✯ ArtifactId                                              Extension Na
  quarkus-artemis-jms                                     Artemis JMS
```

```
✨ quarkus-google-cloud-pubsub                          Google Cloud
✨ quarkus-kafka-client                                 Apache Kafka
✨ quarkus-kafka-streams                                Apache Kafka
✨ quarkus-qpid-jms                                     AMQP 1.0 JMS
   - Apache Qpid JMS
  quarkus-rabbitmq-client                              RabbitMQ Cli
  quarkus-reactive-messaging-http                      Reactive HTT
   WebSocket Connector
✨ quarkus-smallrye-reactive-messaging                  SmallRye Rea
   Messaging
✨ quarkus-smallrye-reactive-messaging-amqp             SmallRye Rea
   Messaging - AMQP Connector
✨ quarkus-smallrye-reactive-messaging-kafka            SmallRye Rea
   Messaging - Kafka Connector
✨ quarkus-smallrye-reactive-messaging-mqtt             SmallRye Rea
   Messaging - MQTT Connector
✨ quarkus-smallrye-reactive-messaging-rabbitmq         SmallRye Rea
   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.

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-kafk
...
[SUCCESS] ✅  Extension io.quarkus:quarkus-smallrye-reactive-mess
```

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 2.22.

Listing 2.22. `smallrye-reactive-messaging-kafka` extension in `pom.xml` dependency

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-reactive-messaging-kafka</artifact
</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 2.23.

Listing 2.23. Removing Quarkus extension example

```
$ quarkus extension remove "quarkus-smallrye-reactive-messaging-k
...
[SUCCESS] ✔  Extension io.quarkus:quarkus-smallrye-reactive-mess
```

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 2.24. Because the output of this command is too long, we display only the Kafka extension we used previously with columns split per line.

Listing 2.24. Quarkus messaging category list with the `--full` flag

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

✯ ArtifactId quarkus-smallrye-reactive-messaging-kafka

  Extension  SmallRye Reactive Messaging - Kafka Connector

  Version    2.14.1.Final

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

  ...
```

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 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, you will learn what functionalities make Quarkus development such an enjoyable experience. Suppose you think that Quarkus is already impressive. In that case, you might want to wait until you understand how developer productivity is taken to another level when you choose Quarkus as your application engine.

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

- Discovering how Quarkus' Dev mode can speed up application development
- Explaining the configuration of Quarkus applications
- Experimenting with Dev UI to get insights into a running application
- Producing development instances of remote services using Dev Services
- Evaluating continuous unit-testing in your application 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.

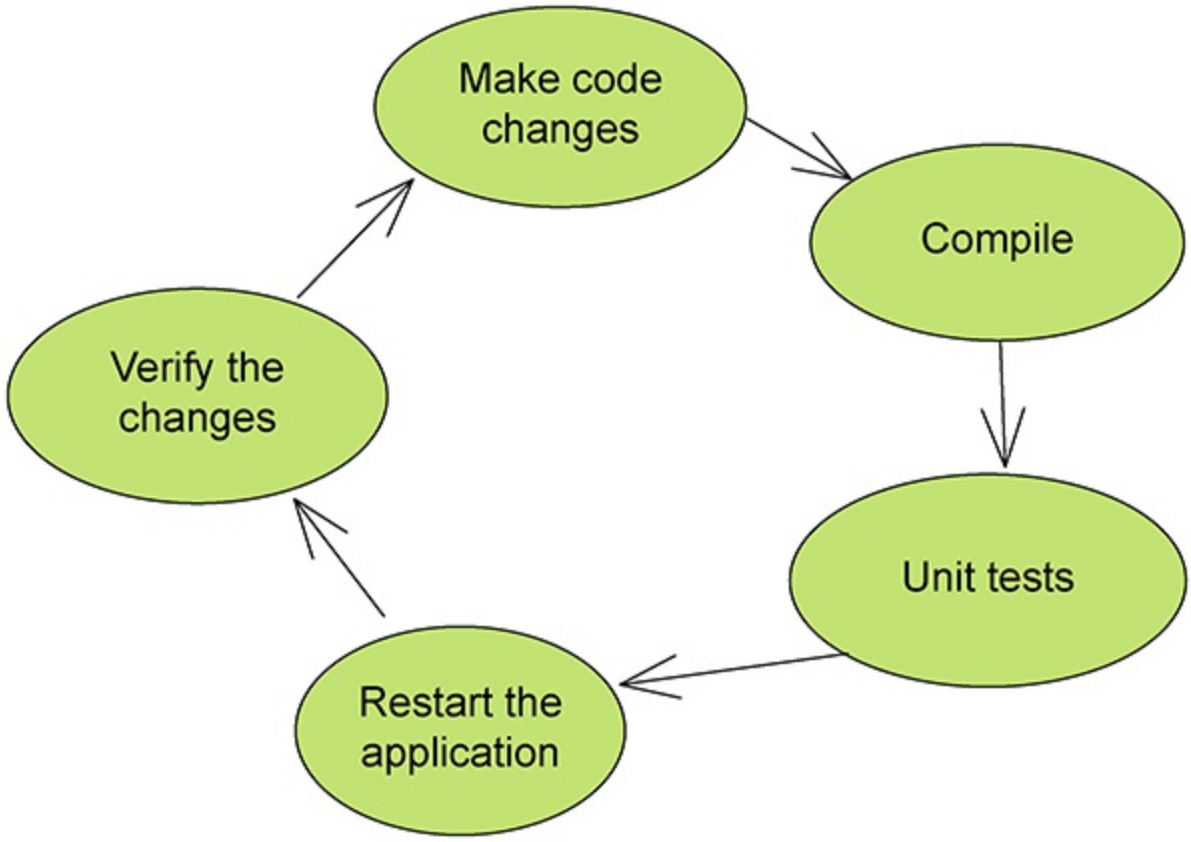
To give you a sneak peek before diving into it, Dev Services is another feature that you will surely enjoy. If you've worked on real-world projects, you've probably used databases a lot. In that case, you're probably familiar with the tediousness of managing databases for development - spinning them 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.

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

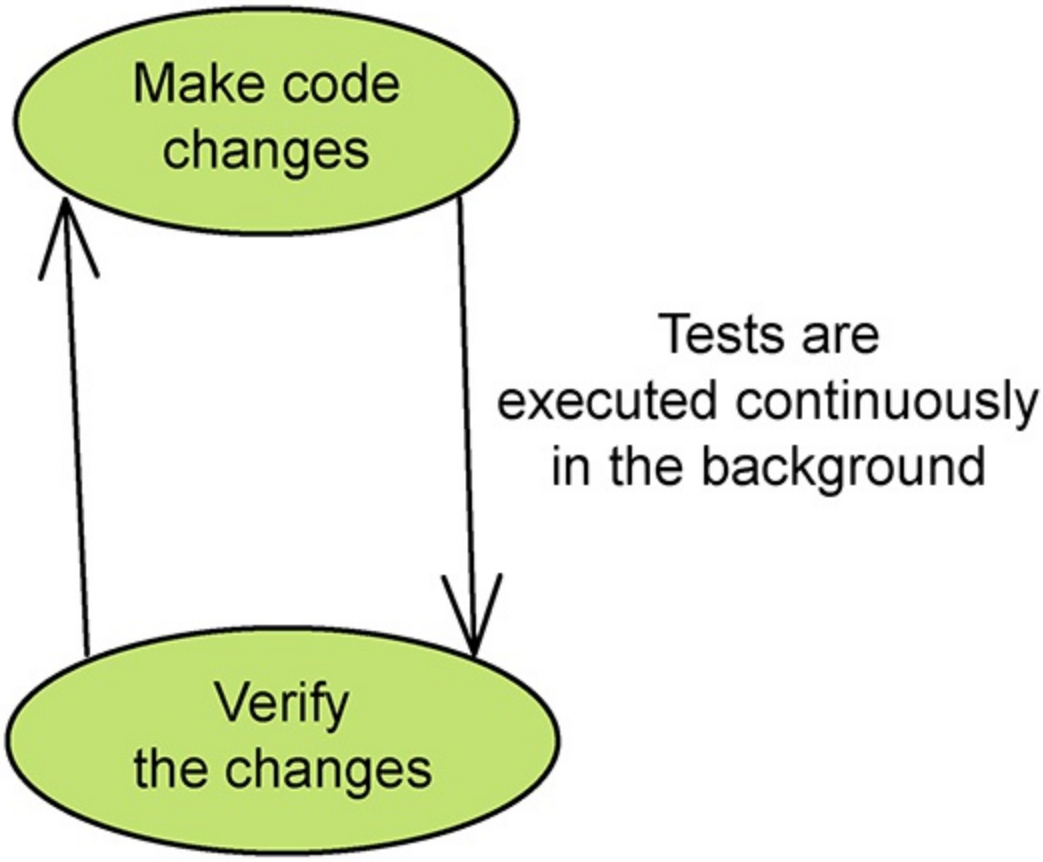
Quarkus' development mode, often referred to as Dev mode, radically changes how we develop Java applications. Traditionally, the development loop workflow has looked something like Figure 3.1. Generally, to test your changes, you need to manually hit recompilation, run tests (at least once in a while), 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
```

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 stoppe
(...)
INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] (vert.x-worker-thr
   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.

 **Note**

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.

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:1
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 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. 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).

 **Note**

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.

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.

 **Note**

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;

@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

 **Note**

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. 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`.

Listing 3.4. Config override with system property

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

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.

 **Note**

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.

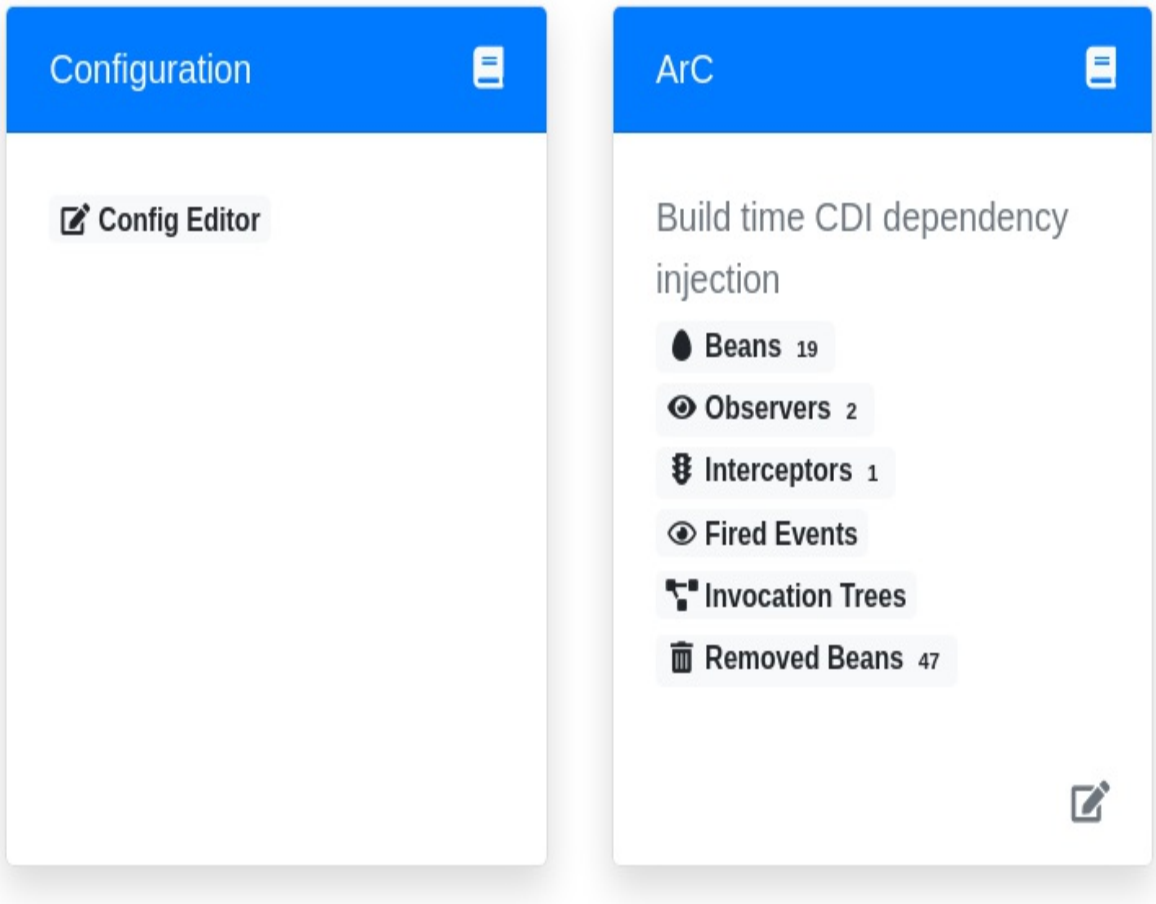
3.3.1 Experimenting with Dev UI

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/ in your web browser manually. The easier way is to press the `d` key while the focus is on the terminal window where your application is running. This should open your browser automatically.

You see panels, and each panel represents one Quarkus extension active in your project. Panels corresponding to extensions that support Dev UI features are blue, and panels for other extensions are gray. Each panel offers a short description of that particular extension, and for extensions with user guides, the top-right corner provides a link to that user guide (the link looks like a book icon).

The `quarkus-in-action` application does not use a lot of extensions. In fact, you probably see only a few blue panels, as shown in Figure 3.3: The main ones are configuration 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, but these two extensions are the ones that we will look at now.

Figure 3.3. Dev UI panels



Changing the configuration

Now, let's explore the Config Editor. Click the link inside the `Configuration` panel. 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 see another table that contains pure JVM system properties and environment variables.

All configuration properties 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.

 **Note**

If you override something in the bottom table (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 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 'tick' 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.

 **Note**

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.

To revert this change, change the value back to 0 and hit Enter (or the tick 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.

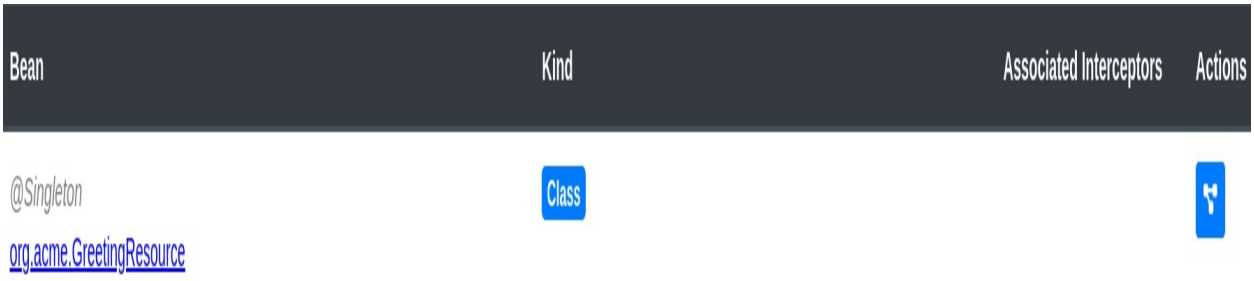
Listing application's CDI beans

Next, let's look at the control panel of the second 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 main page of the Dev UI - click the Quarkus logo at the top left of the page 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). It looks like in Figure 3.4.

- In the first column, you can see that its CDI scope is `@ApplicationScoped`, 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.
- The second column tells you the type of bean. In this case, it's a bean created from a class.
- The third column is a list of interceptors applied to this bean. These are beans intercepting any invocations of your bean.
- The last column called Actions provides and optional link to the Bean Dependency Graph which constitutes a visual representation of bean dependencies.

Figure 3.4. View of the GreetingResource CDI bean in the Dev UI



The Bean Dependency Graph presents a very useful depiction of bean

dependencies which looks like in Figure 3.5.

Figure 3.5. The Bean Dependency Graph of the `GreetingResource` class

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.

 **Note**

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-`	API registry

	`registry-avro`	
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] (buil
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-en
26 minutes ago  Up 26 minutes ago  0.0.0.0:43345->8080/tcp  frost
```

 **Note**

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 3.7.

Listing 3.7. Source of the `GreetingResource#whoAmI` method

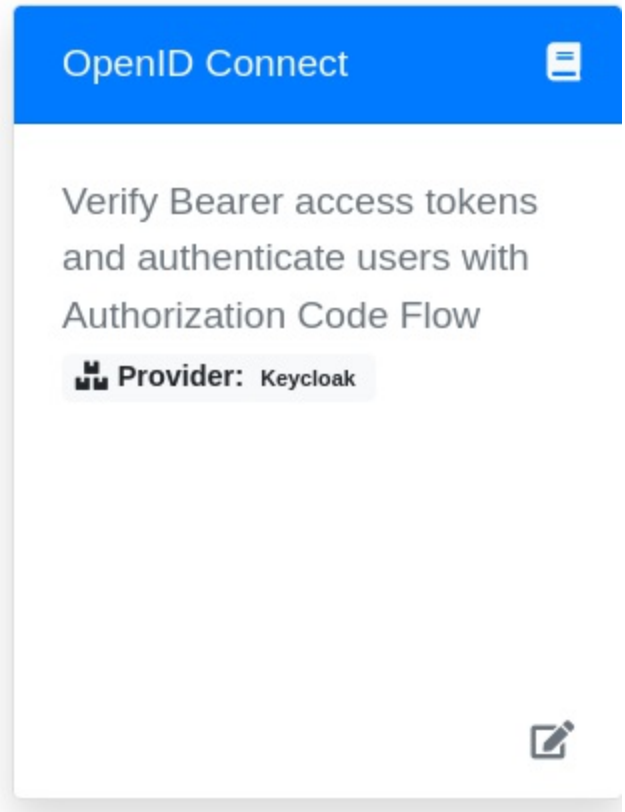
```
// import javax.ws.rs.core.Context;
```

```
// import javax.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`.

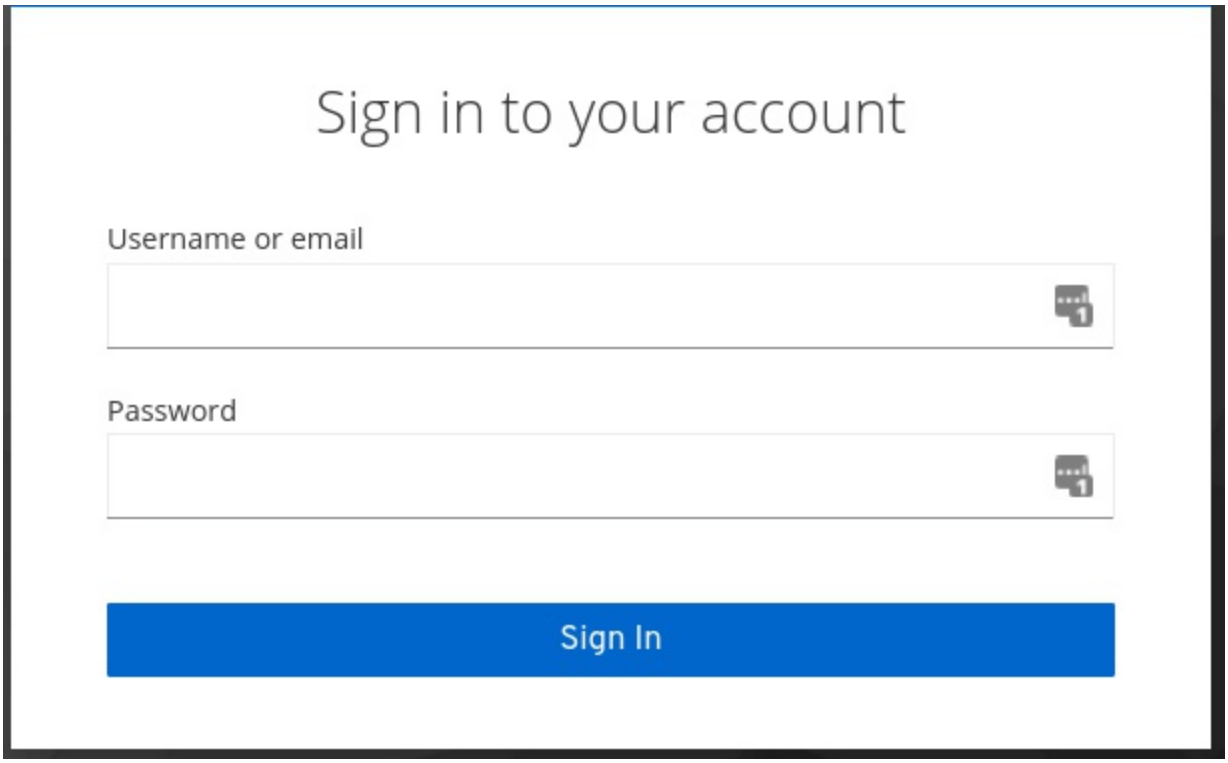
To obtain an authentication token, go to the Dev UI (press `d` with a focus on the Quarkus terminal window), you will see a new panel present: OpenID Connect, as shown in Figure 3.6.

Figure 3.6. 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 3.7.

Figure 3.7. 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.

Note

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/
```

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 3.8.

Listing 3.8. 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.

 **Note**

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 3.9. Status line in terminal shows a failure in continuous testing.

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

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

Listing 3.10. Status line in terminal after shows successful run of continuous testing.

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

 **Note**

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?

 **Note**

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 3.11. 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 3.11. Logs produced by running a test when test output is enabled

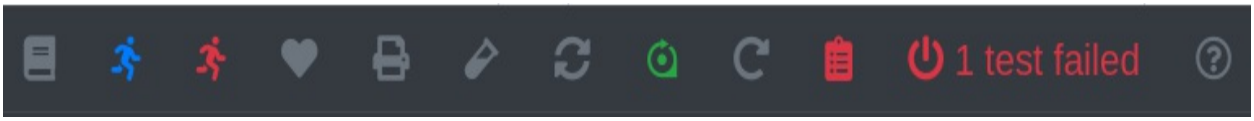
```
2022-03-18 10:09:32,835 INFO  [io.quarkus] (Test runner thread)
   quarkus-in-action 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.
   started in 0.288s. Listening on: http://localhost:8081
2022-03-18 10:09:32,836 INFO  [io.quarkus] (Test runner thread)
   Profile test activated.
2022-03-18 10:09:32,836 INFO  [io.quarkus] (Test runner thread) I
features: [cdi, resteasy-reactive, smallrye-context-propagation,
```

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. With continuous testing enabled, the bottom right corner of the Dev UI page contains a bunch of controls, as shown in Figure 3.8.

Figure 3.8. Controls in the bottom right corner in the Dev UI



The clipboard icon (a bit to the left of the `1 test failed` message) takes you to a page containing test reports and allows you to control other things about the application, like turning live reload on or off. The `Re-run all tests` button (the running blue person) is most notable for testing. Clicking it is equivalent to hitting `r` in the terminal. By the way, hitting `r` when focusing on this browser window still works. After the tests finish, the graphical reports are automatically updated. If you click on a failed test report, you will see the failure's stack trace and the relevant output from the testing Quarkus instance.

 **Note**

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. 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.

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

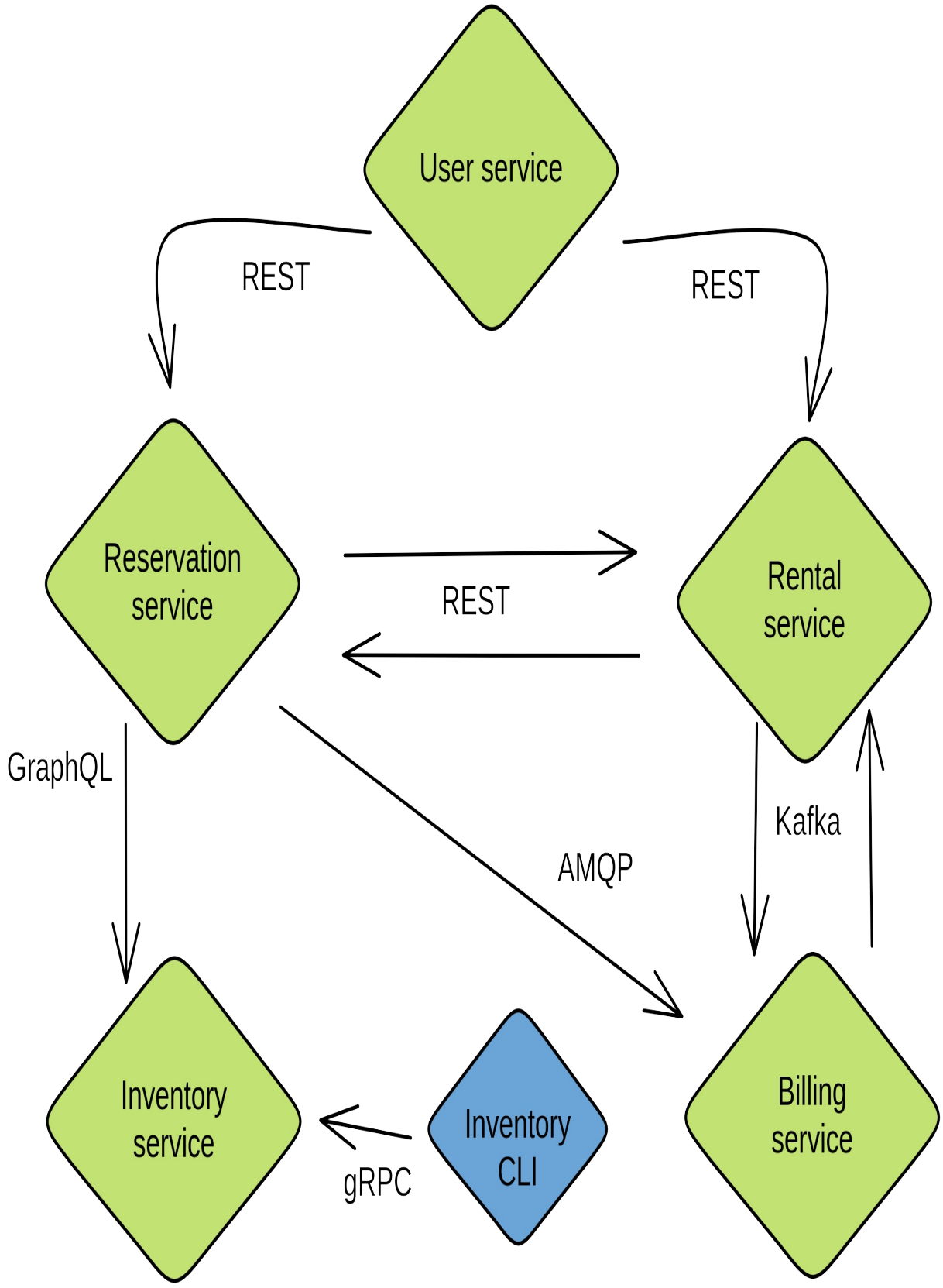
- Learning how to expose and consume APIs using the REST paradigm
- Exploring GraphQL as an alternative to the REST paradigm
- Evaluating gRPC and Protocol Buffer as another way to expose and consume APIs

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

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


User service
REST
REST
Reservation service
REST
Rental service
GraphQL
AMQP
Kafka
Inventory service
Inventory CLI
gRPC
Billing service

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 ubiquitous 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.

 **Important**

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 2.14.1.Final
--extension quarkus-resteasy-reactive-jackson,rest-client-reactiv
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.

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 javax.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;
    }
}
```

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.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, fa
     * otherwise
     */
    public boolean isReserved(LocalDate startDay, LocalDate endDa
        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 javax.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;
    }
}
```

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. You may also delete the generated `GreetingResource` at this point, as we don't need it.

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 javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.acme.reservation.inventory.Car;
import org.acme.reservation.inventory.InventoryClient;
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 reservation
            InventoryClient inventoryClient) {  #3
        this.reservationsRepository = reservations;
        this.inventoryClient = inventoryClient;
    }

    @GET
    @Path("availability")  #4
    public Collection<Car> availability(@RestQuery LocalDate star
            @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.f
        // 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();
    }
}
```

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 javax.ws.rs.Consumes;

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

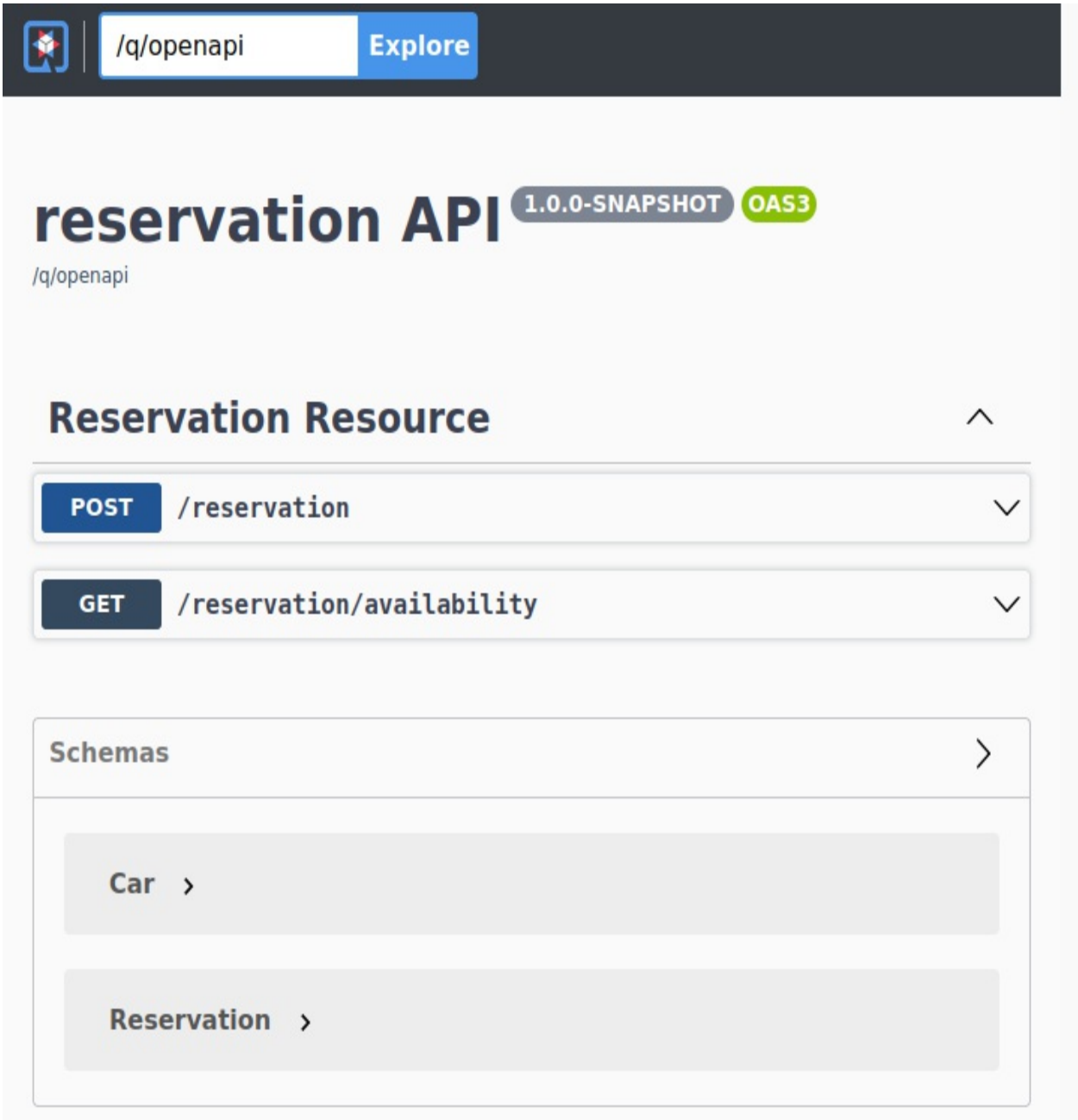
4.2.3 Experimenting with the exposed REST API using the Swagger UI

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.

 **Note**

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 /reservations/availabity` 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.

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/availabity` 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 2.14.1.Final --ex
resteasy-reactive-jackson --no-code
```

 **Note**

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 javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

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

    private final AtomicLong id = new AtomicLong(0);
```

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

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 javax.ws.rs.POST;
import javax.ws.rs.Path;

import org.eclipse.microprofile.rest.client.inject.RegisterRestCl
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);
}
```

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 reservation
            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(reservat
        // 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;
    }

   ...
}
```

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 described the GraphQL data model. The main components of a schema are types and fields. A type which, 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 class. 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.

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 javax.annotation.PostConstruct;
import javax.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<>();
        initialData();
    }

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

    private void initialData() { #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);
    }

}
```

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 javax.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;
        }
    }

}
```

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.

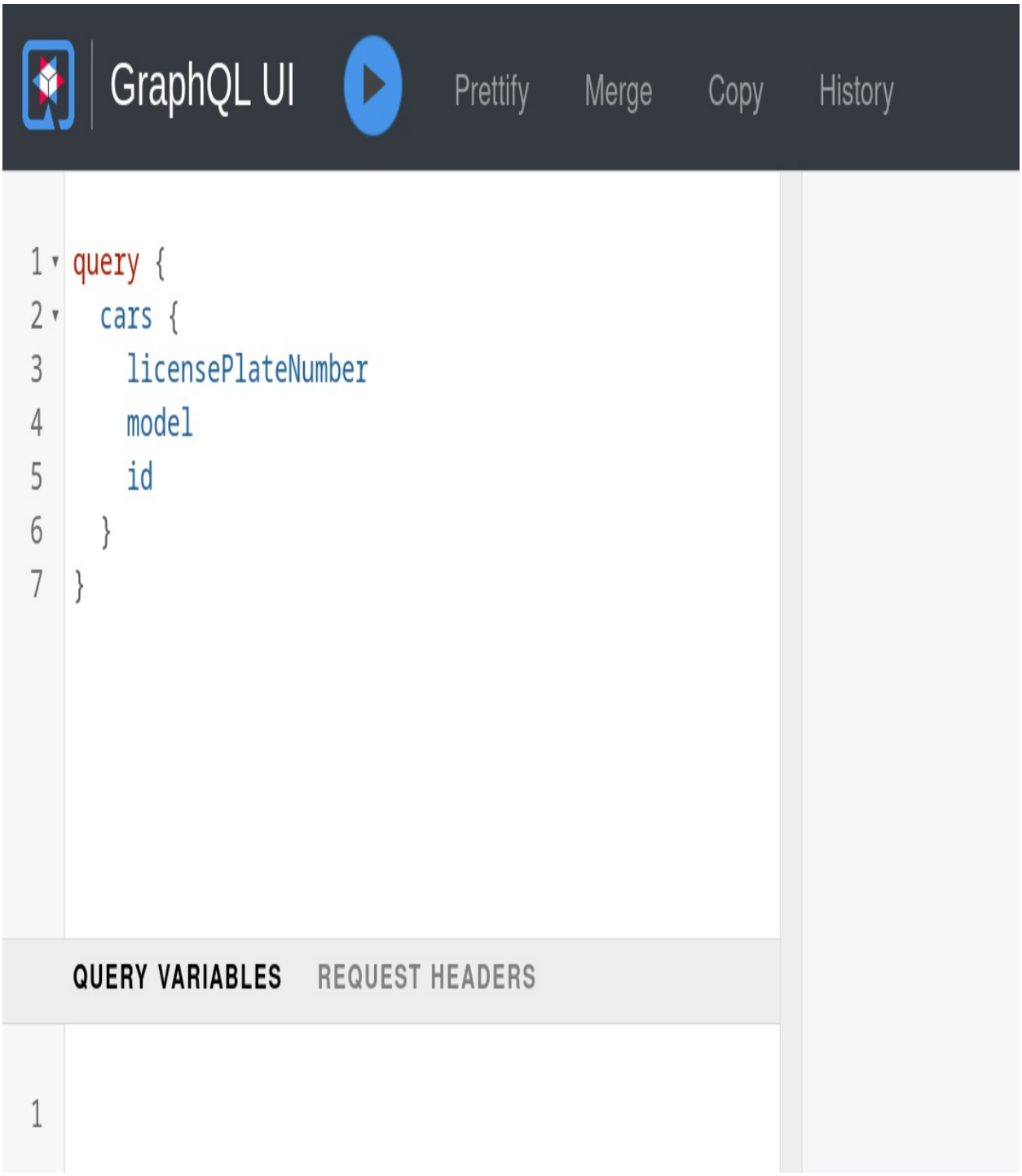
The UI looks like the one presented in Figure 4.3.

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.

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

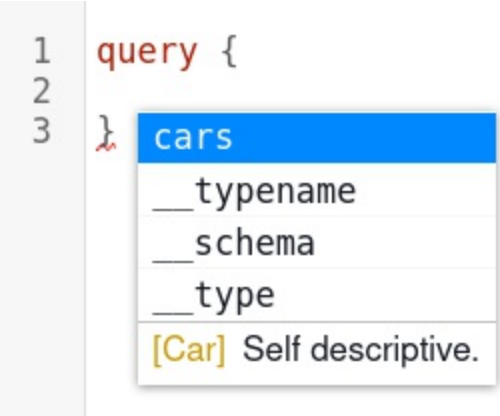
```
query {
  cars {
    id
    licensePlateNumber
    model
    manufacturer
  }
}
```

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 autocompletion suggestions



 **Note**

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: "O123"
    model: "406"
    manufacturer: "Peugeot"
  }) { #3
    licensePlateNumber
    model
    manufacturer
    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": "O123",
      "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
}
```

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();
}
```

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:80
```

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 inventor
      @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.documen
// import static io.smallrye.graphql.client.core.Field.field;
// import static io.smallrye.graphql.client.core.Operation.operat
```

```
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.

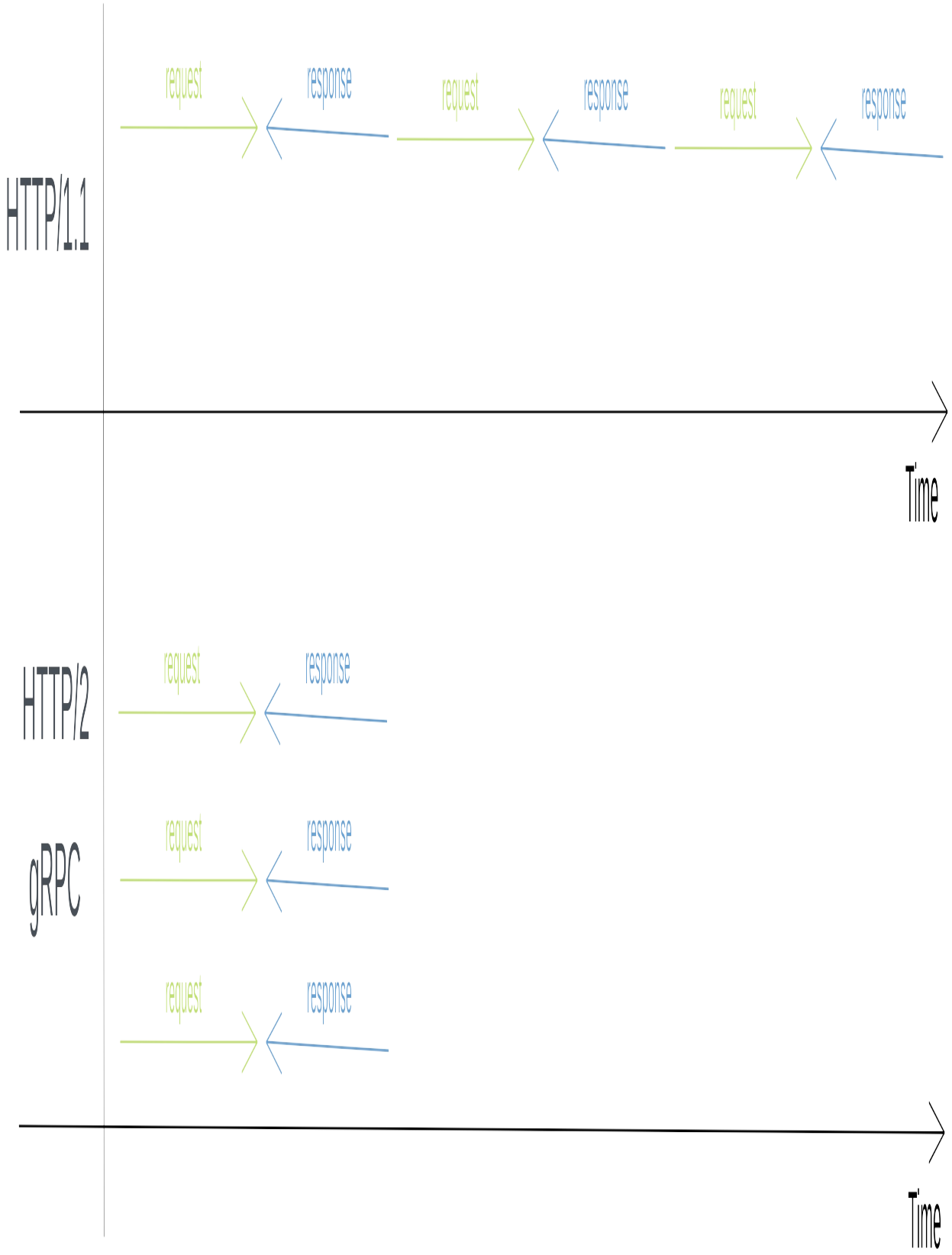
4.6.1 Understanding when to use gRPC

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


HTTP/1.1
request
response
request
response
request
response
Time
HTTP/2
gRPC
request
response
request
response
request
response
Time

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 licensePlateNumb
        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`. 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

```
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) {}
}
```

 **Note**

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 Grpc 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)
}
```

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.licensePlateNum
                .setManufacturer(removedCar.manufacturer)
                .setModel(removedCar.model)
                .setId(removedCar.id)
                .build());
        }
        return Uni.createFrom().nullItem();
    }
}
```

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.

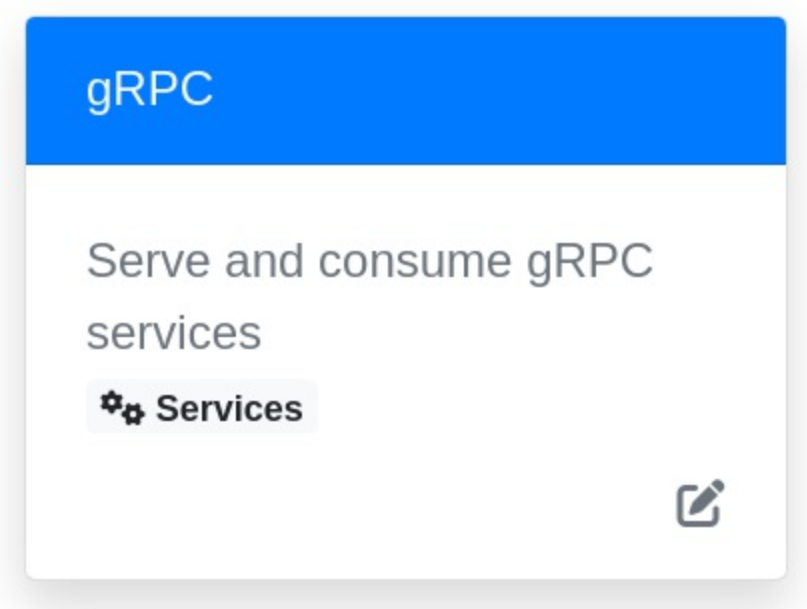
Listing 4.39. Starting the Dev mode with gRPC extension

```
$ quarkus dev
...
INFO  [io.qua.grp.run.GrpcServerRecorder] (vert.x-eventloop-threa
```

```
Server started on 0.0.0.0:9000 [SSL enabled: false]
...
```

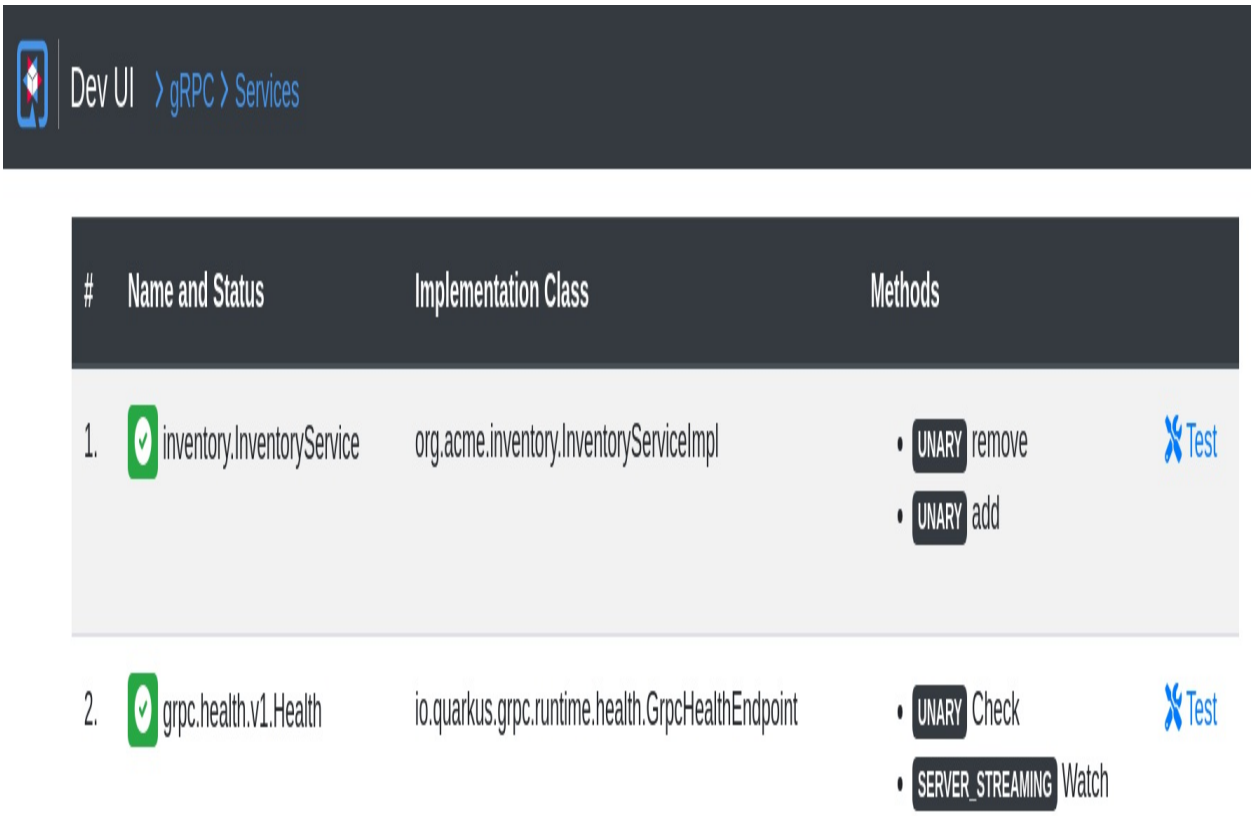
The gRPC server starts using port `9000`. It can be accessed using the command line tools like `grpcurl` (https://github.com/fullstorydev/grpcurl) or the Quarkus Dev UI available at http://localhost:8083/q/dev. Since the Dev UI doesn't require us to install anything, we will focus on it for now. On the 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



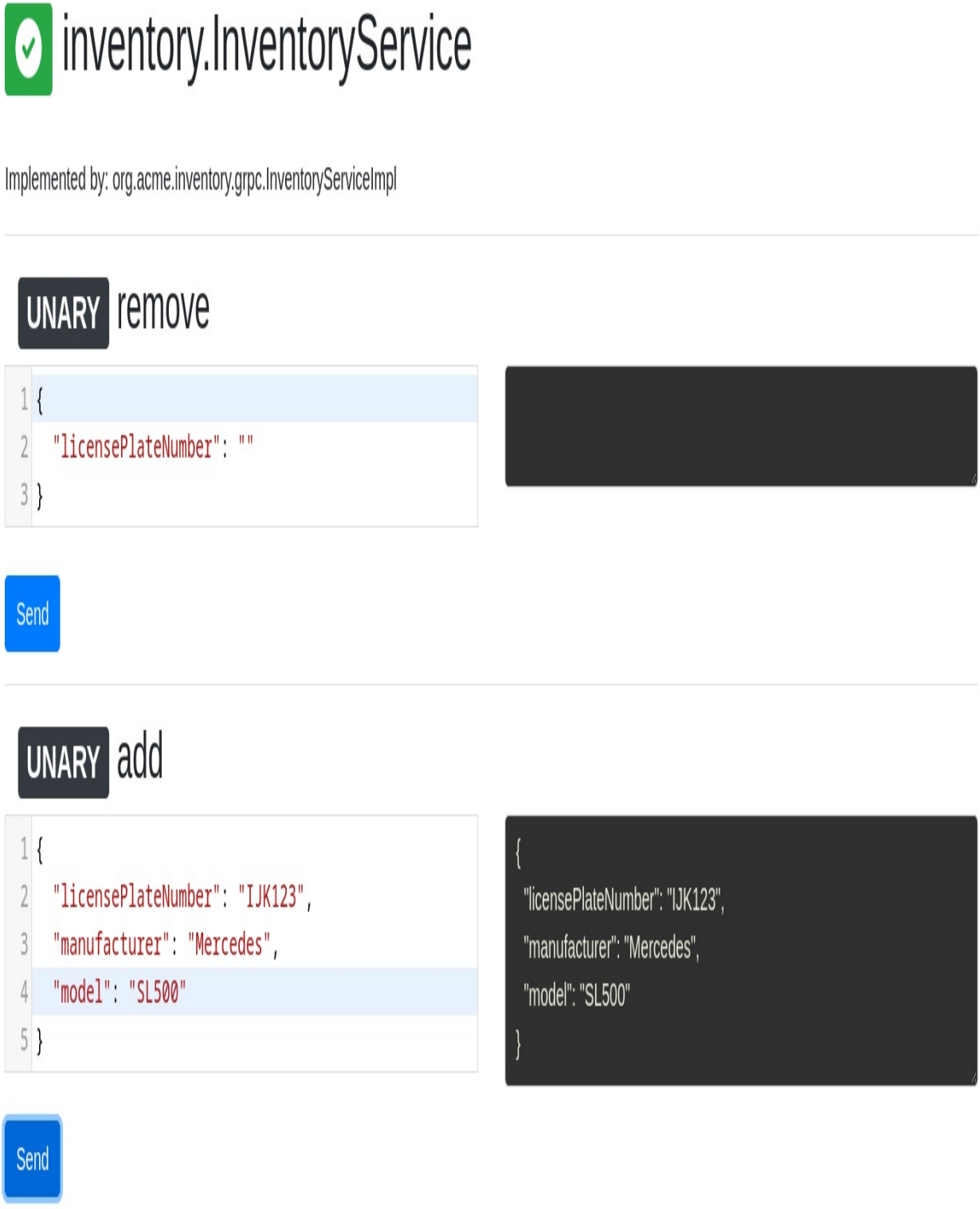
The `Services` button takes you to the gRPC service table, which looks like the one shown in Figure 4.7. The list contains a health endpoint `grpc.health.v1.Health` and our defined `inventory.InventoryService`. Each service has a `Test` link on the right side of the table that points to the respective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

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

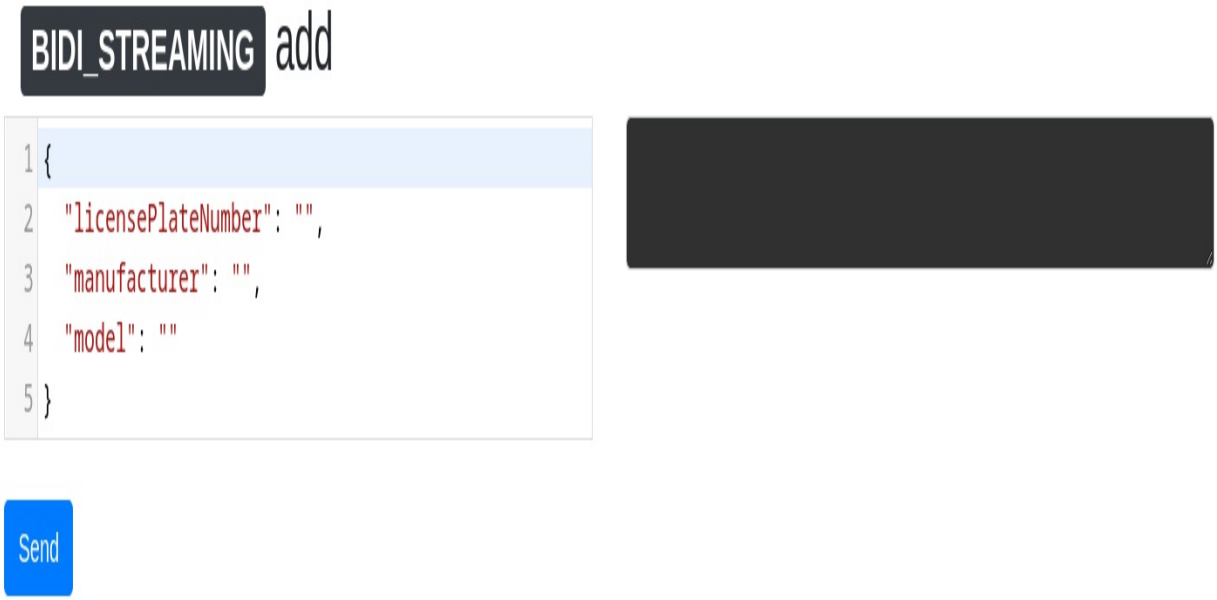
Inventory service to bidirectional streaming so that we can add multiple cars per single connection and directly see how our Quarkus application processes them in the responses. The first step is to update the proto file (`inventory.proto`) and add the `stream` keywords to the `add` method, as shown 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.getLicensePlateNumbe
            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());
}
```

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 manufacture
                    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
    }
}
```

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.

 **Note**

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.quarkus.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 manufacture
                    String model) {
        inventory.add(InsertCarRequest.newBuilder()
            .setLicensePlateNumber(licensePlateNumber)
            .setManufacturer(manufacturer)
            .setModel(model)
            .build())
            .onItem().invoke(carResponse ->
                System.out.println("Inserted new car " + carRespo
            .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
    }
}
```

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
```

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.

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.

For many developers, testing is just a necessary evil. If this is your case, we hope that Quarkus' continuous testing from chapter 3 has already started changing your mind. Let’s see if we can go even further with this! 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 will add things is in the `chapter-04/reservation-service` directory - we suggest you create a copy so that you back up the state of the application after chapter 4. The finished solution is in `chapter-05/reservation-service`.

5.1 Writing tests

To write tests for your applications, Quarkus comes with its lightweight testing framework based on the JUnit 5 library (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.

 **Note**

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 intents 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.

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 way, where you add a test and then execute it using `mvn test`. Both approaches should yield the same result. The test source needs to be in the file named `ReservationRepositoryTest.java` inside the `src/test/java/org/acme/reservation` directory, and 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 javax.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
        reservation.endDay = LocalDate.now().plus(12, ChronoUnit.
        reservation.carId = 384L;
        repository.save(reservation);  #4

        Assertions.assertNotNull(reservation.id); #5
        Assertions.assertTrue(repository.findAll().contains(reser
    }
}
```

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.

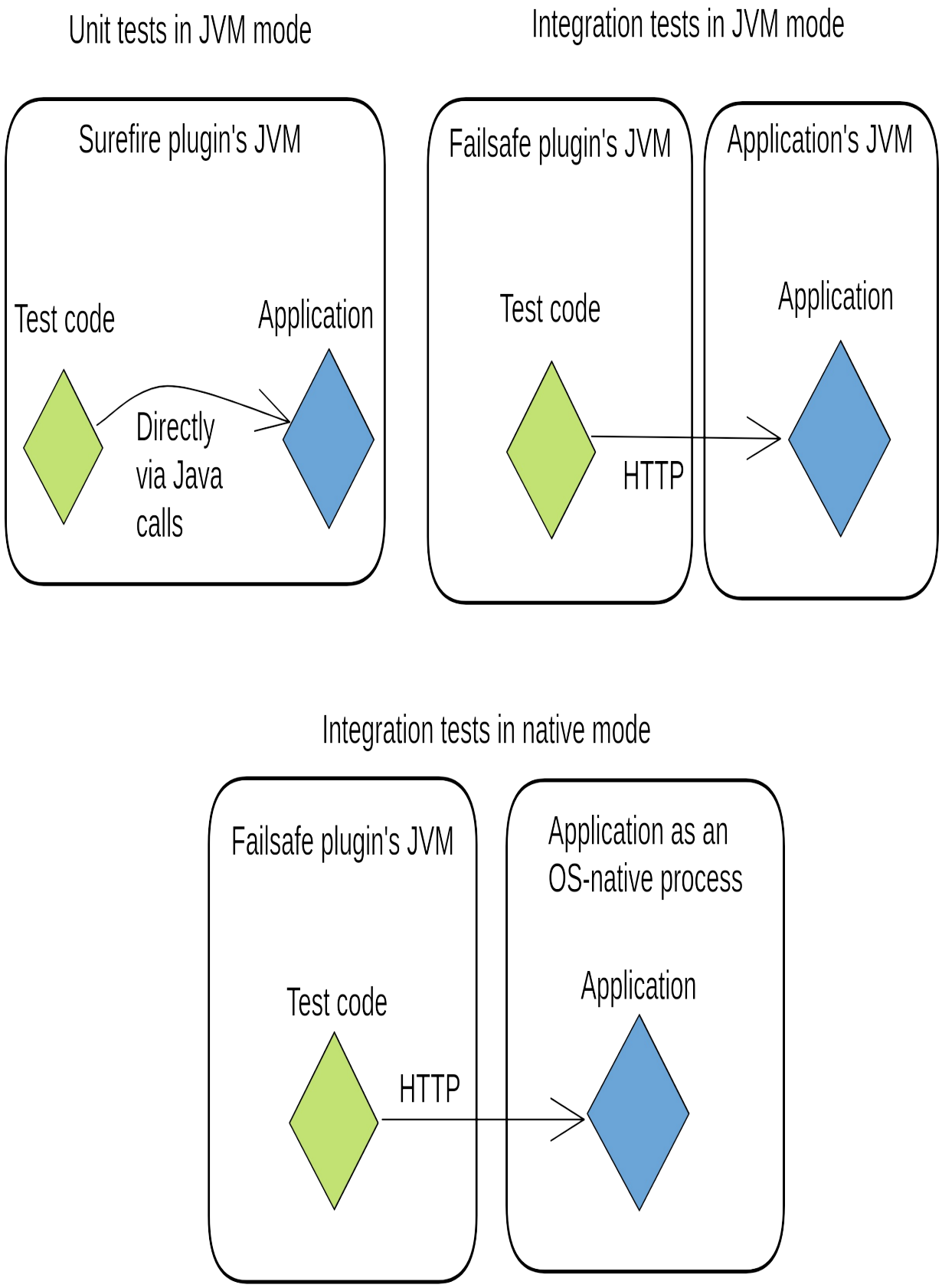
Because native mode tests require compiling a full binary of the application, which takes a lot of time, they also aren't supported for Continuous testing. To execute them, one has to execute a separate Maven or Gradle build.

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


Unit tests in JVM mode
Surefire plugin's JVM
Test code
Application
Directly
via Java
calls
Integration tests in JVM mode
Failsafe plugin's JVM
Test code
HTTP
Application's JVM
Application
Integration tests in native mode
Failsafe plugin's JVM
Test code
HTTP
Application as an
OS-native process
Application

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.

 **Note**

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 test, 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)
    @TestHTTPResource
    URL reservationResource;

    @Test
```

```
    public void testReservationIds() {
        Reservation reservation = new Reservation();
        reservation.carId = 12345L;
        reservation.startDay = LocalDate
            .parse("2025-03-20");
        reservation.endDay = LocalDate
            .parse("2025-03-29");
        RestAssured
            .given()
            .contentType(ContentType.JSON)
            .body(reservation)
            .when()
            .post(reservationResource)
            .then()
            .statusCode(200)
            .body("id", notNullValue());
    }
}
```

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 ReservationResourceTes
}
```

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 in Listing 5.4. 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);
    }
}
```

It’s enough to place this class somewhere in the sources for the tests (`src/test/java`) and it will be picked up automatically.

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.

 **Note**

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. 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.

 **Note**

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

```
@TestHTTPEndpoint(ReservationResource.class)
@TestHTTPResource("availability")
URL availability;

@BeforeAll
public static void setup() {
    GraphQLInventoryClient mock =
        Mockito.mock(GraphQLInventoryClient.class);
    Car peugeot = new Car(1L, "ABC 123", "Peugeot", "406");
    Mockito.when(mock.allCars())
            .thenReturn(Collections.singletonList(peugeot));
    QuarkusMock.installMockForType(mock,
            GraphQLInventoryClient.class);
}

// uses mocks
@DisabledOnIntegrationTest(forArtifactTypes =
  DisabledOnIntegrationTest.ArtifactType.NATIVE_BINARY)
@Test
public void testMakingAReservationAndCheckAvailability() {
    String startDate = "2022-01-01";
    String endDate = "2022-01-10";
    // Get the list of available cars for our requested timeslot
    Car[] cars = RestAssured
        .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();
    reservation.carId = car.id;
    reservation.startDay = LocalDate.parse(startDate);
    reservation.endDay = LocalDate.parse(endDate);
    // Submit the reservation
    RestAssured
        .given()
            .contentType(ContentType.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
        .given()
            .queryParam("startDate", startDate)
            .queryParam("endDate", endDate)
        .when().get(availability)
        .then().statusCode(200)
        .body("findAll { car -> car.id == " + car.id + "}", hasSi
}
```

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
    }
}
```

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., are supported during tests as well as when developing in Dev mode.

6 Exposing and securing web applications

This chapter covers

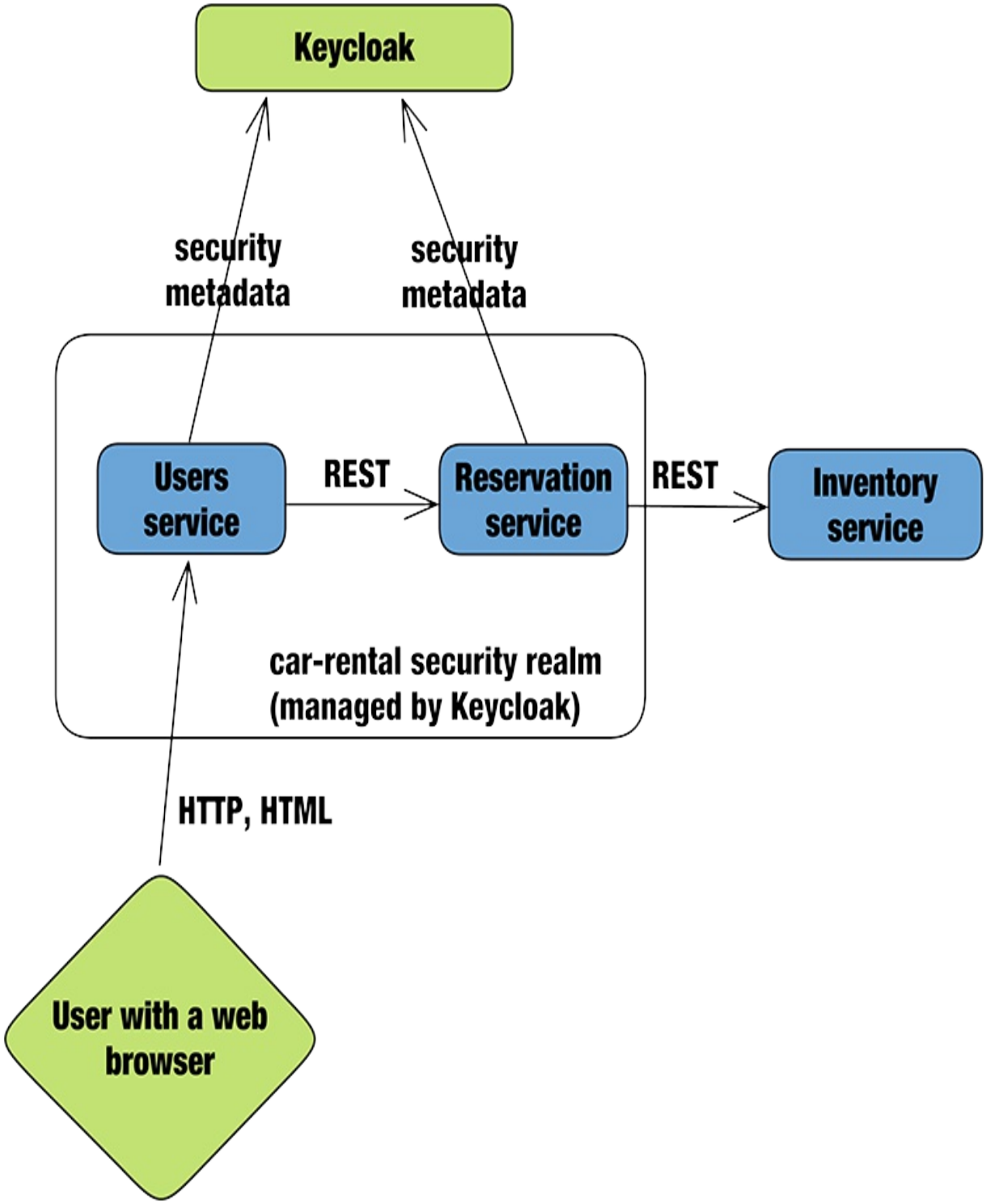
- Developing a basic secured web application with HTML
- Creating a more advanced HTMX-based UI
- Propagating the security context for calls between a web application and a REST service
- Exploring other alternatives for frontend development

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 now use this concept in the car rental project.

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 2.16.0.Final --ext
```

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-client`, because we need the `@AccessToken` annotation for propagating the OIDC ID token from the `Users` service to the `Reservation` service.

 **Note**

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>
```

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.qute.Template;
import io.quarkus.qute.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() != nul
            ? securityContext.getUserPrincipal().getName() : null
        return whoami.data("name", userId);    #3
    }
}
```

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!
```

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.

Important

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,

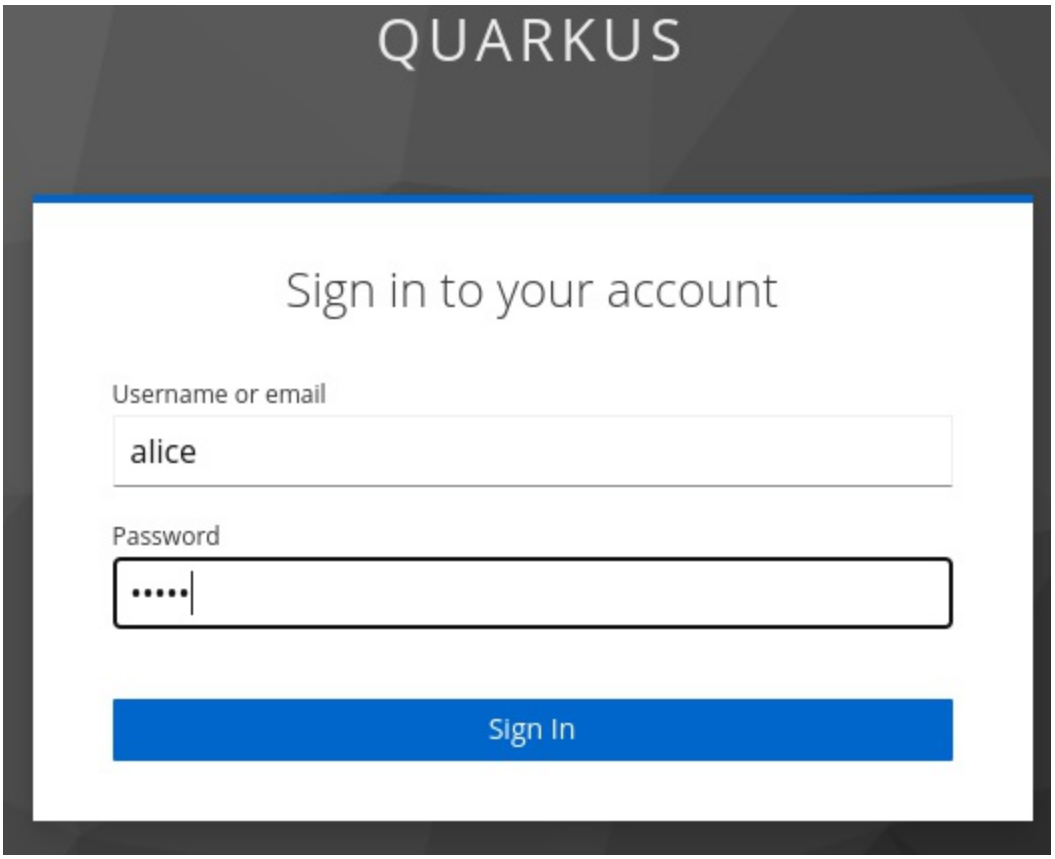
it is possible to customize the Dev Services Keycloak instance via Quarkus configuration properties, if that is necessary). We now configure only the application itself. Add these lines into `application.properties` of the `Users` service:

Listing 6.3. Security-related configuration properties of the `Users` service

```
quarkus.http.auth.permission.all-resources.paths=/*
quarkus.http.auth.permission.all-resources.policy=authenticated
quarkus.oidc.application-type=web_app
quarkus.keycloak.devservices.shared=true
quarkus.oidc.logout.path=/logout
```

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!
```

Note

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!
```

 **Warning**

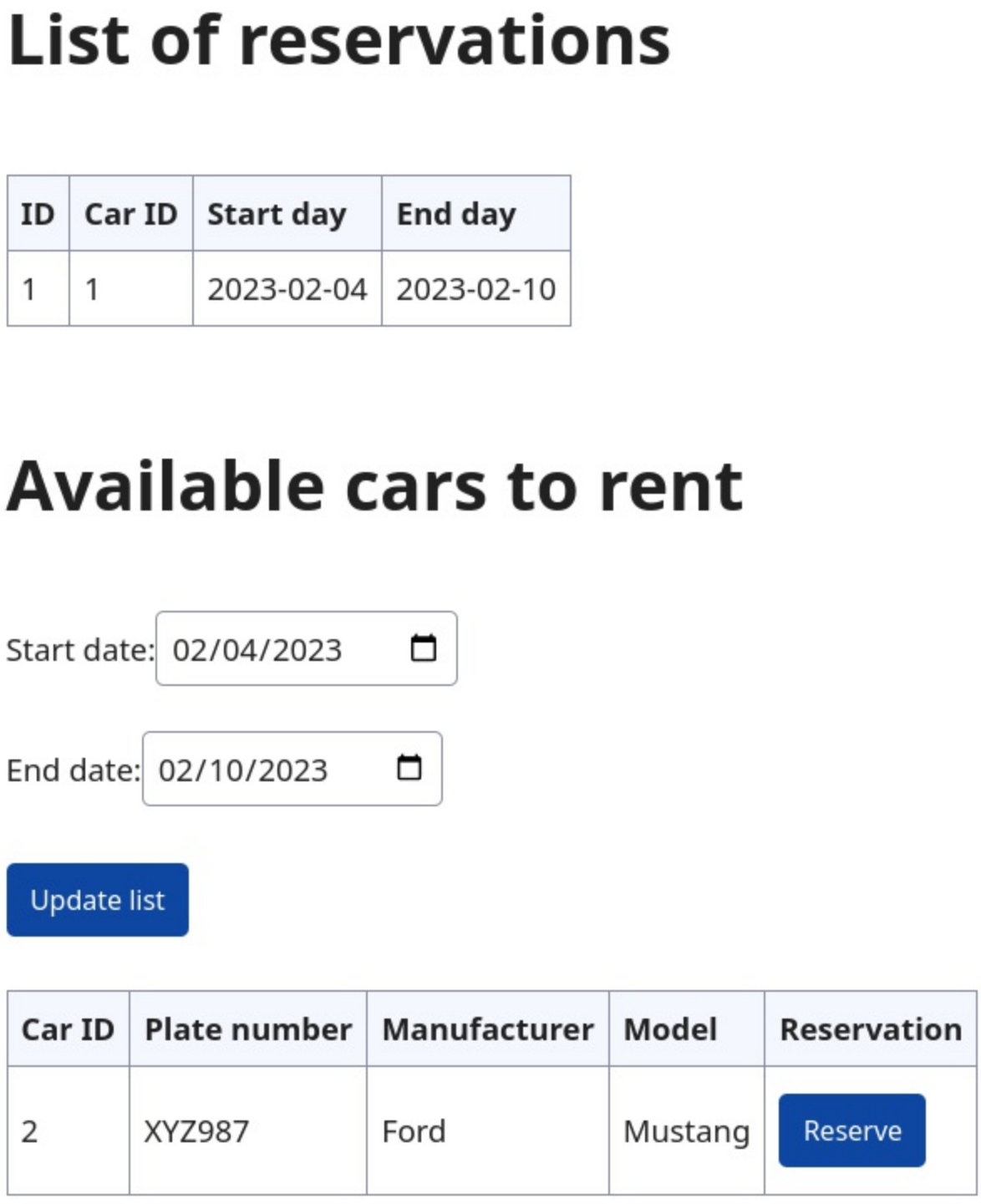
If it happens that you restart the Keycloak instance (for example, by stopping and restarting the whole Quarkus Dev mode - NOT just a live reload of the application) while you're logged in to the application in your browser, it might happen that after refreshing the page, the browser will again pass 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.

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     #1
quarkus.keycloak.devservices.shared=true  #2
```

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;
```

 **Note**

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;
}
```

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)) #1
        .collect(Collectors.toList());
}
```

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.RegisterRestCl
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();

    @POST
    Reservation make(Reservation reservation);

    @GET
    @Path("availability")
    Collection<Car> availability(  #4
        @RestQuery LocalDate startDate,
        @RestQuery LocalDate endDate);
}
```

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 information about the username, and a logout button, as shown in Listing 6.1.

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/htmx.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>
```

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.

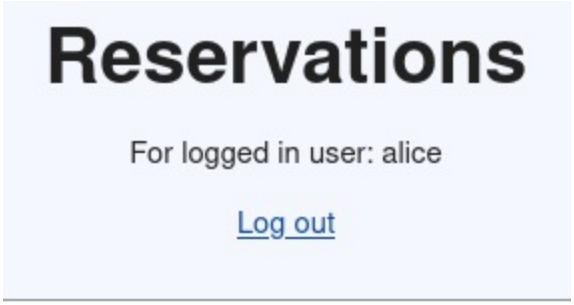
Listing 6.11. Initial version of the REST resource serving the `index.html` page

```
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)                                   #
    public TemplateInstance index(@RestQuery LocalDate startDate,
                                  @RestQuery LocalDate endDate) {
        if (startDate == null) {
            startDate = LocalDate.now().plusDays(1L);
        }
        if (endDate == null) {
            endDate = LocalDate.now().plusDays(7);
        }
        return Templates.index(startDate, endDate,
            securityContext.getUserPrincipal().getName());
    }
}
```

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.

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}                                    #
{@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>
  {#for car in cars}
    <tr>
```

```
    <td>{car.id}</td>
    <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>
    </form>
  {/for}
</table>
</div>
```

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">
    <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"
     hx-trigger="load, update-available-cars-list from:body"
     hx-include="[id='startDateInput'],[id='endDateInput']">
<!-- To be replaced by the result of calling /available -->
</div>
```

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>
```

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>
```

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();
}
```

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.

 **Note**

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-realm.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.j
    command:
      - start-dev
      - --import-realm #2
```

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/realms/c
%prod.quarkus.oidc.client-id=reservation-service
```

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/realms/c
%prod.quarkus.oidc.client-id=users-service
```

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 linchpin 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-client` 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.