



Cloud Native Spring IN ACTION

With Spring Boot and Kubernetes

Thomas Vitale



MEAP Edition
Manning Early Access Program
Cloud Native Spring in Action
With Spring Boot and Kubernetes

Version 12

Copyright 2022 Manning Publications

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

welcome

Thank you for purchasing the MEAP for *Cloud Native Spring in Action*.

To get the most benefit from this book, you'll want to have established skills in Java programming, with experience in building web applications and basic knowledge of the Spring core features. I'll assume you're familiar with Git, object-oriented programming, distributed systems, databases, and testing.

In the last few years, it has become clear that many organizations have had to choose the cloud to keep the business alive. A traditional approach based on application servers and on-premise "snowflake" infrastructure can hardly ensure the requirements of modern applications: scalability, resilience, zero-downtime, reliability, short feedback cycles, and frequent releases. To fully leverage the cloud's capabilities, it wasn't enough to move traditional applications to the new infrastructure. Something else was needed. Applications had to become *cloud native*.

This book will take you on an exciting journey in the cloud native world, from development to production, using the latest features from Spring, Docker, and Kubernetes. You'll have the chance to apply every new idea along the journey by progressively building a full cloud native system for an online bookshop. By the end of the book, you'll have it deployed to a Kubernetes cluster in a public cloud.

The book is divided into four parts. Part 1 will guide you through the cloud native landscape, defining what it means for an application to be cloud native, and establishing tools, technologies, and patterns you'll bring on your journey.

Part 2 will get you started with the fundamentals of cloud native development. You'll work with RESTful Spring application services and data persistence, understand how to manage configuration across environments, define automation for your continuous integration process, learn how the Docker containerization works, and finally deploy your apps to a local Kubernetes cluster.

In Part 3, your journey gets to a whole new level. You'll move from a single app to a distributed system in the cloud. You'll learn about reactive programming, scalability, resilience, and edge services. You'll then explore the features of event-driven architecture and secure your system with authentication, authorization, and encryption.

In Part 4, you'll ensure the observability of your applications, dealing with logging, tracing, and monitoring. Finally, you'll get to the long-wished-for destination: production. You'll learn how to deploy your application to a Kubernetes cluster in a public cloud and automate the delivery and deployment process. A new journey starts in production. You'll get familiar with the next steps in your cloud native adventure and learn some tips for migrating your existing applications.

Please let me know your thoughts on what's been written so far and what you'd like to see in the rest of the book. I welcome your comments in the [liveBook Discussion forum](#) - your feedback will be invaluable in improving *Cloud Native Spring in Action*.

You can find all the code examples used in the book in the accompanying repository on [GitHub](#).

Thanks again for your interest and for purchasing the MEAP!

—Thomas Vitale

brief contents

PART 1: CLOUD NATIVE FUNDAMENTALS

- 1 Introduction to cloud native*
- 2 Cloud native patterns and technologies*

PART 2: CLOUD NATIVE DEVELOPMENT

- 3 Getting started with cloud native development*
- 4 Externalized configuration management*
- 5 Persisting and managing data in the cloud*
- 6 Containerizing Spring Boot*
- 7 Kubernetes fundamentals for Spring Boot*

PART 3: CLOUD NATIVE DISTRIBUTED SYSTEMS

- 8 Reactive Spring: Resilience and scalability*
- 9 API gateway and circuit breakers*
- 10 Event-driven applications and functions*
- 11 Security: Authentication and SPA*
- 12 Security: Authorization and auditing*

PART 4: CLOUD NATIVE PRODUCTION

- 13 Observability and monitoring*
- 14 Configuration and secrets management*
- 15 Continuous delivery and GitOps*
- 16 Serverless, GraalVM and Knative*

APPENDICES

- A Setting up your development environment*
- B Kubernetes in production with DigitalOcean*

Introduction to cloud native

This chapter covers

- What are the cloud and the cloud computing model
- The definition of cloud native
- Characteristics of cloud native applications
- Culture and practices supporting cloud native
- When and why you might consider the cloud native approach
- Topologies and architectures for cloud native applications

Cloud native applications are highly distributed systems that live in the cloud and are resilient to change. Systems are made up of several services, which communicate through a network and are deployed in a dynamic environment where everything keeps changing.

Before diving into technologies, it's fundamental to define what cloud native is first. Like other buzzwords in our field (such as *Agile*, *DevOps*, or *Microservices*), *Cloud Native* is sometimes misunderstood and is a source of confusion because it means different things to different people.

In this chapter, I'll provide you with the conceptual tools you need for the rest of the book. I'll start by defining what cloud native means and what it takes for an application to be called as such. I'll explain the properties of cloud native applications, examine the characteristics of the cloud computing model, and discuss when and why you might want to move to the cloud. I'll also present some fundamental concepts of cloud native topologies and architectures. Figure 1.1 shows an overview of all the different elements I'll cover in this chapter to define and qualify cloud native systems. At the end of this chapter, you'll be ready to start your journey building cloud native applications with Spring and deploying them to Kubernetes.

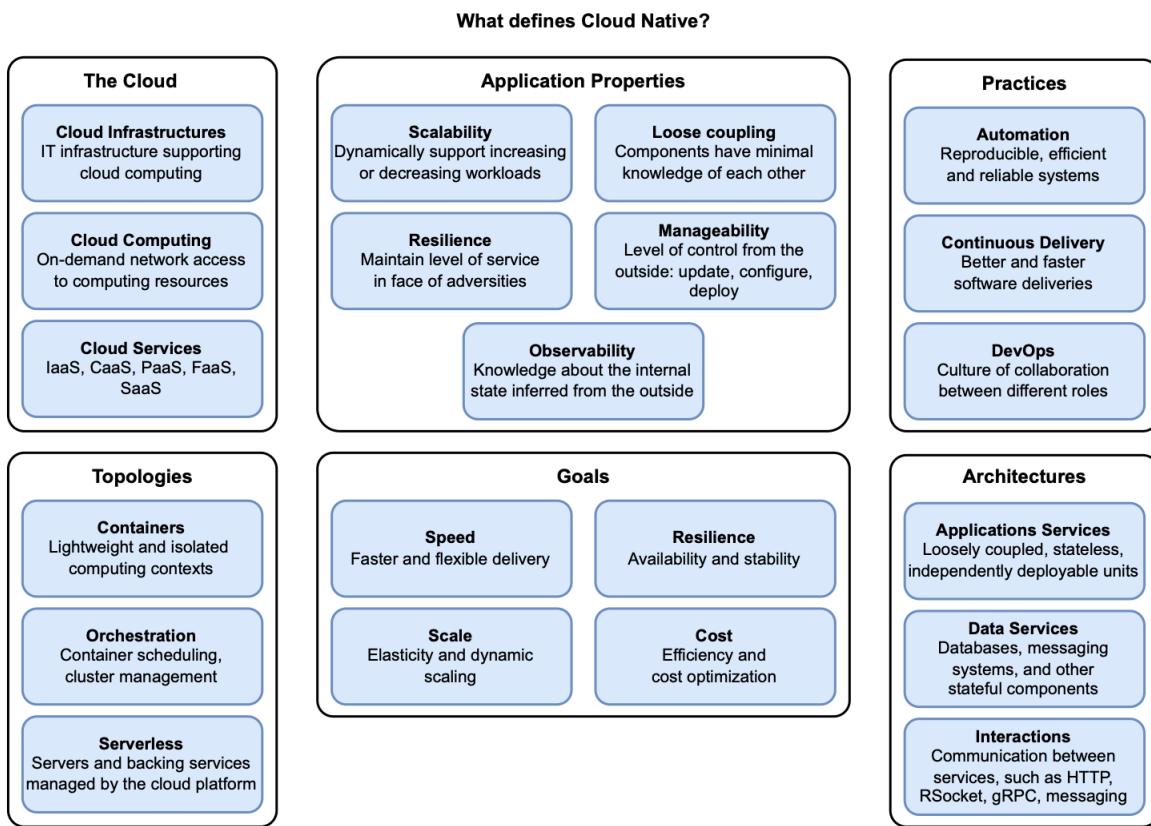


Figure 1.1 Cloud native is an approach to application development aiming at leveraging cloud technologies.

1.1 What is cloud native?

On the 25th of May in 2010, Paul Fremantle, a veteran of the cloud industry, wrote a post on his blog titled: "Cloud Native".¹ He was among the first to use the term *cloud native*. In a period where concepts and technologies like microservices, Docker, DevOps, Kubernetes, or Spring Boot didn't exist yet, Fremantle discussed what it takes for "applications and middleware to work well in a cloud environment", to be *cloud native*, with his team at WSO2.

The key concept explained by Fremantle is for applications to be specifically designed for the cloud and have properties that take advantage of the new environment where they run and of the cloud computing model. You can move a traditional application (designed to run *on the ground*) to the cloud, an approach commonly referred to as "*lift and shift*", but that doesn't make the application *native* to the cloud. Let's see what does.

1.1.1 The Three Ps of Cloud Native

What does it mean for applications to be designed specifically for the cloud? The Cloud Native Computing Foundation (CNCF) answers that question in its cloud native definition.²

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

From this definition, I identify three groups of information that I like calling *The Three Ps of Cloud Native*:

- **Platforms.** Cloud native applications run in platforms based on dynamic, distributed environments: the clouds (public, private, or hybrid).
- **Properties.** Cloud native applications are designed to be scalable, loosely coupled, resilient, manageable, and observable.
- **Practices.** Practices around cloud native applications include robust automation combined with frequent and predictable changes: automation, continuous delivery, and DevOps.

SIDE BAR

What is the Cloud Native Computing Foundation?

The Cloud Native Computing Foundation (CNCF), part of the Linux Foundation, "builds sustainable ecosystems and fosters communities to support the growth and health of cloud native open-source software". The CNCF hosts many cloud native technologies and projects to enable cloud portability without vendor lock-in. If you want to discover the many projects addressing any cloud native aspect, I recommend checking out the CNCF Cloud Native Interactive Landscape.³

In the following sections, I'll examine those concepts further. However, I'd first like you to notice how the definition of cloud native is not tied to any specific implementation detail or technology. The CNCF mentions some in its definition, like containers and microservices, but they are just examples. One of the common misconceptions when starting the migration to the cloud is that you have to adopt a microservices architecture, build containers, and deploy them to Kubernetes. That is not true. Fremantle's post in 2010 is proof of that. He didn't mention any of those because they didn't exist. Yet, the applications he described not only are still considered cloud native, but they also comply with the definition given by the CNCF eight years later.

1.2 The cloud and the cloud computing model

Before focusing on the main characters, the cloud native applications, I'd like to set the scene by describing where the journey takes place, the context in which cloud native applications run: the *cloud* (figure 1.2). In this section, I'll define the cloud and its main characteristics. After all, if cloud native applications are designed to work well in a cloud environment, we should know what kind of environment that is.

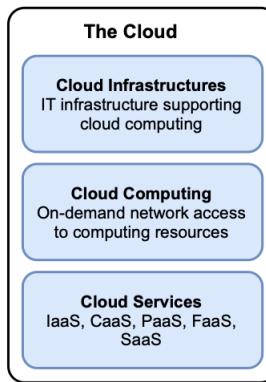


Figure 1.2 The cloud is an IT infrastructure characterized by different computing models and offered as a service by providers according to the degree of control consumers need.

The cloud is an IT infrastructure that supports the delivery of computing resources to consumers according to the cloud computing model. The National Institute of Standards and Technology (NIST) defines cloud computing as follows:⁴

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Just like you get electricity from a provider rather than generating it on your own, with the cloud you can get computing resources (for example, servers, storage, and networks) as a commodity.

The cloud provider manages the underlying cloud infrastructure, so the consumer doesn't need to worry about physical resources like machines or networks. Companies moving to the cloud can get all the computing resources they need via a network (usually the Internet) through a set of APIs that allows them to provision and scale resources as they need on an on-demand, self-service base.

Elasticity is one of the main characteristics of this model: computing resources can be provisioned and released dynamically, depending on the demand.

*Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.*⁵

Traditional IT infrastructures weren't able to provide elasticity. Companies had to calculate the maximum computing capabilities needed and set up an infrastructure that would support that, even if most of it was only required sometimes. With the cloud computing model, the usage of computing resources is monitored, and consumers only pay for what they actually use.

There is no strict requirement about where the cloud infrastructure should be or who should manage it. There are several deployment models to deliver cloud services. The main ones are private cloud, public cloud, and hybrid cloud.

- **Private cloud.** Cloud infrastructure provisioned to be used by a single organization. It can be managed by the organization itself or by a third party, and it can be hosted on-premises or externally. A private cloud is usually the preferred option for organizations dealing with sensitive data or highly-critical systems. It is also a common choice for having complete control over the infrastructure compliance with specific laws and requirements like the General Data Protection Regulation (GDPR) or the California Consumer Privacy Act (CCPA). For example, banks and healthcare providers are likely to set up their own cloud infrastructure.
- **Public cloud.** Cloud infrastructure provisioned for public use. It is usually owned and managed by an organization, the *cloud provider*, and hosted on the provider's premises. Examples of public cloud service providers are Amazon Web Services (AWS), Microsoft Azure, Google Cloud, Alibaba Cloud, and DigitalOcean.
- **Hybrid cloud.** Composition of two or more distinct cloud infrastructures belonging to any of the previous types bound together and offering services as if they were one single environment.

The cloud computing model offers five main service models, depending on which capabilities are provided to the consumer. The decision about which service model to choose should be driven by the degree of control the consumer needs over the infrastructure and which type of computing resources they need to manage.

Figure 1.3 describes the leading cloud computing service models, what is provided by the platform in each model, and which abstractions are provided to the consumer. For example, with the Infrastructure as a Service model (IaaS), the platform provides and manages computing, storage, and networking resources, whereas the consumer provisions and manages virtual machines.

Cloud Computing Service Models				
Infrastructure Platform	Container Platform	Application Platform	Serverless Platform	Software Platform
IaaS	CaaS	PaaS	FaaS	SaaS
Platform: provides computing, storage, and networking resources.	Platform: provides container engine, orchestrator, and underlying infrastructure.	Platform: provides development and deployment tools, APIs, and underlying infrastructure.	Platform: provides the runtime, the whole infrastructure needed to run functions, and autoscaling.	Platform: provides both the software and the whole infrastructure needed to run it.
Consumer: provisions, configures, and manages servers, network, and storage.	Consumer: builds, deploys, and manages containerized workloads and clusters.	Consumer: builds, deploys, and manages applications.	Consumer: builds and deploys functions.	Consumer: consumes a service via a network.

Figure 1.3 The cloud computing service models. They differ by the level of abstraction they provide and who is responsible for managing which level (the platform or the consumer).

1.2.1 Infrastructure as a Service

In the *Infrastructure as a Service* (IaaS) model, consumers can directly control and provision resources like servers, storage, and networks. For example, they can provision virtual machines and install software like operating systems and libraries. Even though this model has been used for a while, it was in 2006 that Amazon made it popular and widely accessible with Amazon Web Services (AWS). Examples of IaaS offerings are AWS Elastic Compute Cloud (EC2), Azure Virtual Machines, Google Compute Engine, Alibaba Virtual Machines, and DigitalOcean Droplets.

1.2.2 Container as a Service

Using the *Container as a Service* (CaaS) model, consumers cannot control primitive virtualization resources. Instead, they provision and manage containers. The cloud provider takes care of provisioning the underlying resources needed to fulfill the needs of those containers, for example, by starting new virtual machines and configuring networks to make them accessible through the Internet. Docker Swarm, Apache Mesos, and Kubernetes are examples of tools used to build container platforms. All major cloud providers offer a managed Kubernetes service, which has become the de-facto technology for CaaS offerings: Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), Google Kubernetes Engine (GKE), Alibaba Container Service for Kubernetes (ACK), and DigitalOcean Kubernetes.

1.2.3 Platform as a Service

In the *Platform as a Service* (PaaS) model, the platform provides infrastructure, tools, and APIs that developers can use to build and deploy applications. For example, as a developer, you can build a Java application, package it as a Java Archive (JAR) file, and then deploy it to a platform working according to the PaaS model. The platform provides the Java runtime, other required middleware and can also offer extra services like databases or messaging systems. Examples of PaaS offerings are Cloud Foundry, Heroku, AWS Elastic Beanstalk, Azure App Service, Google App Engine, Alibaba Web App Service, and DigitalOcean App Platform. In the past few years, vendors have been converging to Kubernetes for building a new PaaS experience for developers and operators. Examples of this new generation of services are VMware Tanzu Application Platform and RedHat OpenShift.

1.2.4 Function as a Service

The *Function as a Service* (FaaS) model relies on serverless computing to let consumers focus on implementing the business logic of their applications (often in the form of functions), whereas the platform takes care of providing servers and the rest of the infrastructure. Serverless applications are triggered by events, such as HTTP requests or messages. For example, you might code a function that analyzes a data set whenever available from a message queue and computes results according to some algorithms. Examples of commercial FaaS offerings are Amazon AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, and Alibaba Functions Compute. Examples of open-source FaaS offerings are Knative and Apache OpenWhisk.

1.2.5 Software as a Service

The service with the highest abstraction is *Software as a Service* (SaaS). In this model, consumers access applications as users, while the cloud provider manages the whole stack of software and infrastructure. Many companies build their applications, use a CaaS or PaaS model to run them, and then sell their usage to the end customers as SaaS. The consumers of SaaS applications typically use thin clients like web browsers or mobile devices to access them. Examples of applications available as SaaS are Proton Mail, GitHub, Plausible Analytics, and Microsoft Office 365.

SIDE BAR**Platform vs PaaS**

Platform is a term that might generate some confusion in a cloud native discussion. Let's clarify. In general, a platform is an operating environment you use to run and manage your applications. So, Google Kubernetes Engine (GKE) is a platform that offers cloud services according to the CaaS model. Microsoft Azure Functions is a platform that provides cloud services following the FaaS model. At a lower level, if you deploy your applications directly on an Ubuntu machine, that will be your platform. In the rest of the book, whenever I use the term platform, I mean the broader concept just explained, unless specified otherwise.

1.3 Properties of cloud native applications

The scene is set: you are in the cloud. How should you design applications to take advantage of its characteristics? The CNCF identifies five main properties that cloud native applications should have. In itself, the idea is that cloud native is a methodology to build and run applications that exhibit those properties. Cornelia Davis sums it up by stating that "cloud-native software is defined by how you compute, not about where you compute".⁶ In other words: the cloud is about *where*, cloud native is about *how*.

I have already covered the *where* part: the cloud. Let's now go ahead and explore the *how*. For a quick reference, figure 1.4 lists the properties and a short description.

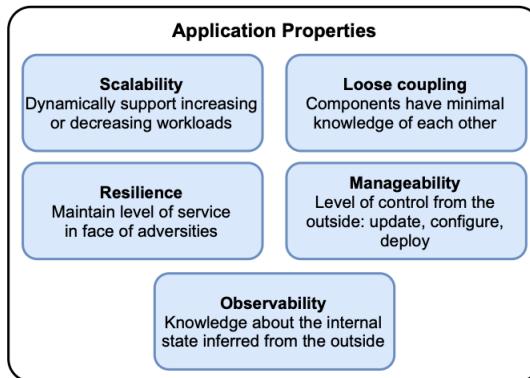


Figure 1.4 The main properties of cloud native applications.

1.3.1 Scalability

Cloud native applications are designed to *scale*, meaning that they can support increasing workloads if provided with additional resources. Depending on the nature of those extra resources, we distinguish between vertical scalability and horizontal scalability.

- **Vertical scalability.** Scaling vertically, or scaling up/down, means adding/removing

hardware resources to/from the computing node, like CPU or memory. This approach is limited since it's not possible to keep adding hardware resources. On the other hand, applications don't need to be explicitly designed to be scaled up or down.

- **Horizontal scalability.** Scaling horizontally, or scaling out/in, means adding more computing nodes or containers (for example, when using Kubernetes) to the system. This approach doesn't have the same limits as vertical scalability, but it requires applications specifically to be scalable.

Traditional systems would usually adopt vertical scalability in case of increasing workloads. Adding CPU and memory was a common approach for an application to support more users without (re)designing it for scalability. And it's still a good option in specific scenarios, but we need something else for the cloud.

In the cloud, where everything is dynamic and in constant change, horizontal scalability is preferred. Thanks to the abstraction levels offered by the cloud computing models, it's straightforward to spin up new instances of your applications rather than increasing the computational power of the machines already running. Since the cloud is elastic, we can scale out and in application instances in a short time and dynamically. I discussed elasticity as one of the main characteristics of the cloud: computing resources can be provisioned and released proactively, depending on the demand. Scalability is a prerequisite for elasticity.

Figure 1.5 shows the difference between vertical and horizontal scalability. In the first case, we scale by adding more resources to the existing virtual machine. In the second case, we add another virtual machine to help the existing one process the extra workload.

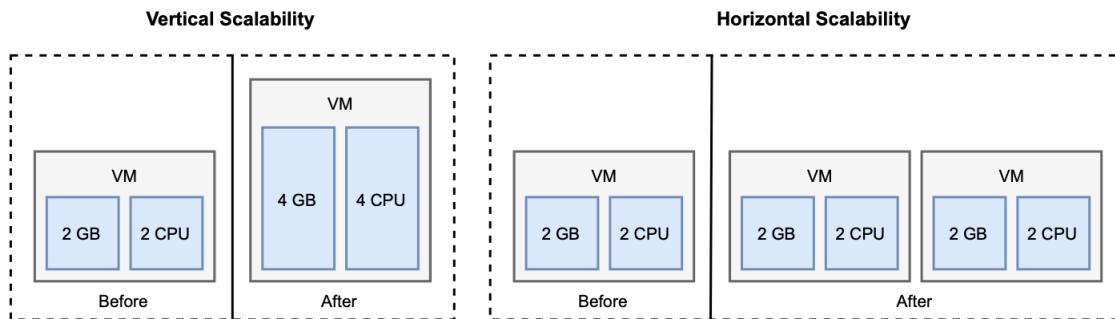


Figure 1.5 When we need to support increasing workloads, the vertical scalability model will add hardware resources to the computing node, while the horizontal scalability model will add more computing nodes.

As you'll see when discussing Kubernetes, the platform (it can be CaaS, PaaS, or something else) takes care of scaling in and out applications dynamically according to the definition above.

As developers, it's our responsibility to design applications that can be scaled. The big obstacle to scalability is the application state, which will eventually become a matter of whether an

application is stateful or stateless. Throughout the book, I'll cover techniques to build stateless applications and let them scale without issues. Among the other things, I'll show you how to push the application state from Spring to data stores like PostgreSQL and Redis.

1.3.2 Loose coupling

Loose coupling is an essential property of a system where parts have as little knowledge as possible of each other. The goal is to evolve each piece independently so that when one is changed, the others don't need to change accordingly.

Coupling and its twin concept *cohesion* have played an essential role in software engineering for decades. It's a good design practice to decompose a system into modules (*modularization*) with minimal dependencies on the other parts (loose coupling) and encapsulate code that changes together (high cohesion). Depending on the architectural style, a module can model a monolithic component or a standalone service (for example, a microservice). Either way, we should aim at achieving a proper modularization with loose coupling and high cohesion.

Parnas identified three benefits of modularization:⁷

- **Managerial.** Since each module is loosely coupled, the team responsible for it should not need to spend much time in coordination and communication with other teams.
- **Product flexibility.** The overall system should be flexible since each module is evolved independently of the others.
- **Comprehensibility.** People should be able to understand and work with a module without having to study the whole system.

The benefits above are usually among the ones associated with microservices. The truth is that you don't need microservices to achieve them. In the last few years, many organizations have decided to migrate from monoliths to microservices. Some of them failed because of a lack of proper modularization. A monolith made up of tightly coupled, non-cohesive components, when migrated, produces a tightly coupled, non-cohesive microservice system, which sometimes is referred to as a *distributed monolith*. If you ask me, it is not a good name because it implies that monoliths are made up of tightly coupled, non-cohesive components by definition. That's not true. The architectural style doesn't matter: a bad design is a bad design. Indeed, I like the *modular monolith* term proposed by Simon Brown to increase awareness that monoliths can also promote loose coupling and high cohesion, and that both monoliths and microservices can end up being "big balls of mud".

Throughout the book, I'll go through some techniques to enforce loose coupling in our applications. In particular, we'll adopt a service-based architecture and focus on building services with clear interfaces to communicate with each other, minimal dependencies with other services and high cohesion.

1.3.3 Resilience

When building cloud native systems, our goal should be to guarantee that our applications are always available, whether there is a failure in the infrastructure or in our software.

A system is resilient if it provides its services even in the presence of faults or environmental changes. Resilience is "the capability of a hardware-software network to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation".⁸

Cloud native applications run in a dynamic environment where everything keeps changing, and faults can and will occur. It cannot be prevented. In the past, we used to consider changes and faults as exceptions. But for highly distributed systems like the cloud native ones, changes are not exceptions: they are the rule.

When discussing resilience, it's worth defining three essential concepts: fault, error, and failure.⁹

- **Fault.** A fault is a defect that produces an incorrect internal state either in the software or the infrastructure. Example: a method call returns a null value, even if its specification mandates that a non-null value is returned.
- **Error.** An error is a discrepancy between the expected behavior of a system and the actual one. Example: due to the fault above, a `NullPointerException` is thrown.
- **Failure.** When a fault is triggered and results in an error, a failure might occur, making the system unresponsive and unable to behave according to its specifications. Example: if the `NullPointerException` is not caught, the error provokes a failure: the system responds to any request with a 500 response.

Faults can become errors, which may provoke failures, so we should design applications to be *fault tolerant*. An essential part of resilience is ensuring that a failure will not cascade to other components of the system but stay isolated while it gets fixed. We also want the system to be *self-repairing* or *self-healing*, and the cloud model can actually enable that.

Throughout the book, I'll show you some techniques to tolerate faults and prevent their effects from propagating to other parts of the system and spreading the failure. For example, we'll use patterns like circuit breakers, retries, timeouts, and rate limiters.

1.3.4 Observability

Observability is a property that comes from the world of control theory. If you consider a system, observability is a measure of how well you can infer its internal state from its external outputs. In the software engineering context, the system is a single application or a distributed system as a whole. The external outputs can be data like metrics, logs, and traces. Figure 1.6 shows how observability works.

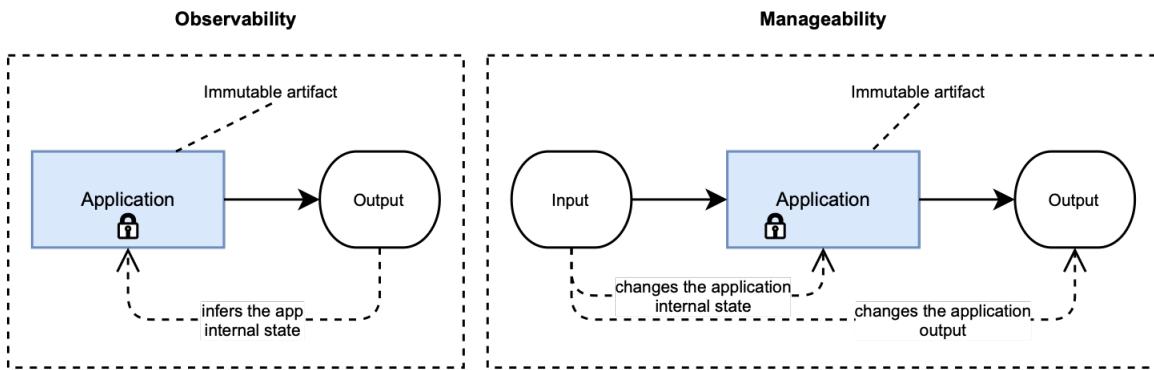


Figure 1.6 Observability is about inferring the internal state of an application from its external outputs. Manageability is about changing the internal state and output from external inputs. In both cases, the application artifact is never changed. It's immutable.

The Observability Engineering team at Twitter identifies four pillars of observability:¹⁰

- **Monitoring.** Monitoring is about measuring specific aspects of an application to get information on its overall health and identify failures. In this book, you'll familiarize yourself with the useful monitoring features of Spring Boot Actuator and integrate Prometheus with Spring to export relevant metrics about the application.
- **Alerting/visualization.** Collecting data about the state of a system is useful only if used to take some action. When a failure is identified while monitoring an application, an alert should be triggered, and some action should be taken to handle it. Specific dashboards are used to visualize the data collected and plot them in relevant graphs to provide a good picture of the system's behavior. This book will show how to use Grafana to visualize data collected from cloud native applications.
- **Distributed systems tracing infrastructure.** In a distributed system, it's not enough to track every single subsystem's behavior. It's essential to trace the data flowing through the different subsystems. In this book, you'll integrate Spring with OpenTelemetry and use Grafana Tempo to collect and visualize the traces.
- **Log aggregation/analytics.** Keeping track of the main events in an application is critical to infer the software's behavior and debug it if something goes wrong. In a cloud native system, logs should be aggregated and collected to provide a better picture of the system behavior and have the possibility of running analytics to mine information from those data. Throughout the book, I'll talk more about logs. You'll use Fluent Bit, Loki, and Grafana to collect and visualize logs, and learn the best practices for logging in a cloud native context.

1.3.5 Manageability

In control theory, the dual concept of observability is controllability, which represents the ability of external input to change the state or the output of a system in a finite time interval. Such a concept leads us to the last of the main properties of cloud native: manageability.

Drawing from the control theory again, we can say that manageability measures how easily and efficiently an external input can change the state or the output of a system. In less mathematical terms, it's the ability to modify an application's behavior without the need to change its code. It

is not to be confused with *maintainability*, which measures how easily and efficiently you can change a system from the inside by changing its code. Figure 1.6 shows how manageability works.

One aspect of manageability is deploying and updating applications while keeping the overall system up and running. Another element is configuration, which I'll address in depth throughout the book. We want to make cloud native applications configurable to modify their behavior without changing their code and building a new release. It's common to make configurable settings like data sources URLs, service credentials, and certificates. For example, depending on the environment, we can use a different data source: one for development, one for testing, and one for production. Other types of configurations could be feature flags to decide whether a specific feature should be enabled at runtime. I'll show you different strategies to configure applications throughout the book, including Spring Cloud Config Server, Kubernetes ConfigMaps and Secrets, and Kustomize.

Manageability is not only about the change itself but also about how easily and efficiently you can apply that change. Cloud native systems are complex, so it's essential to design applications that can adapt to changing requirements regarding functionality, environment, and security. Due to the complexity, we should aim at managing as much as possible through automation, which leads us to the last of the Three Ps of Cloud Native: practices.

1.4 Culture and practices supporting cloud native

This section will focus on the last sentence of the definition of cloud native technologies provided by the CNCF: "Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil." I'll discuss three concepts: automation, continuous delivery, and DevOps (figure 1.7).

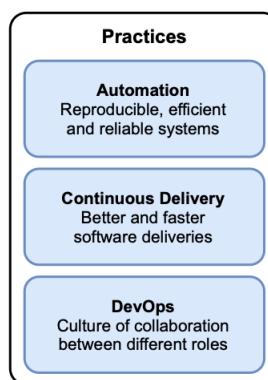


Figure 1.7 Culture and practices for cloud native development.

1.4.1 Automation

Automation is a core tenet of cloud native. The idea is to automate repetitive manual tasks to accelerate the delivery and deployment of cloud native applications. Many different tasks can be automated, from building applications to deploying them, from infrastructure provisioning to configuration management. The most important advantage of automation is that it makes processes and tasks repeatable, and overall systems more stable and reliable. Manually executing a task is error-prone and costs money. By automating it, we can get a result that is both more reliable and more efficient.

The cloud computing model is characterized by the provisioning of computing resources happening in an automated, self-service model and its capability to increase or decrease them elastically. Two significant categories of automation for the cloud regard infrastructure provisioning and configuration management. We call them *infrastructure as code* and *configuration as code*.

Martin Fowler defines **infrastructure as code** as "the approach to defining computing and network infrastructure through source code that can then be treated just like any software system".¹¹

Cloud providers offer convenient APIs to create and provision servers, networks, and storage. By automating those tasks with a tool like Terraform, putting the code in source control, and applying the same test and delivery practices used for application development, we get a more reliable and predictable infrastructure, which is reproducible, more efficient, and less risky. A simple example of an automated task could be creating a new virtual machine with eight CPUs, 64GB of memory, and Ubuntu 22.04 as the operating system.

After provisioning computing resources, we can manage them and automate their configuration. Paraphrasing the previous definition, **configuration as code** is the approach to defining the configuration for computing resources through source code that can be treated just like any software system.

Using tools like Ansible, we can write how a server or a network should be configured. For example, after provisioning the Ubuntu server from the previous paragraph, we can automate the task of installing the Java Runtime Environment (JRE) 17 and opening the ports 8080 and 8443 from the firewall. Configuration as code applies to application configuration as well.

By automating all the infrastructure provisioning and configuration management tasks, we can avoid unstable, unreliable *snowflake servers*. When each server is provisioned, managed, and configured manually, the result is a *snowflake*: a fragile, unique server that cannot be reproduced and is risky to change. Automation helps avoid snowflakes in favor of *phoenix servers*: all tasks acting on those servers are automated, every change can be tracked in source control, reducing

risks, and each setup is reproducible. By taking this concept to its extreme, we achieve the so-called *immutable servers*, also mentioned by the CNCF in its cloud native definition as immutable infrastructure.

NOTE

You might have heard the expression "pets vs. cattle" when comparing traditional snowflake infrastructures (requiring a lot of care and attention, like pets) and immutable infrastructures or containers (characterized by being disposable and replaceable, like cattle). I won't use this expression throughout the book, but it's sometimes used in discussions about this subject, so I'm mentioning it here.

After the initial provisioning and configuration, immutable servers are not changed anymore: they are immutable. If any change is necessary, it's defined as code and delivered. A new server is finally provisioned and configured from the new code while the previous server is destroyed.

For example, you have two options if your current infrastructure consists of Ubuntu 20.04 servers and you want to upgrade to Ubuntu 22.04. The first option is to define the upgrade via code and run the automation scripts to perform the operation on the existing machines (phoenix servers). The second option is automating the provisioning of new machines running Ubuntu 22.04 and starting using those rather than performing the upgrade on the existing machines (immutable servers).

In the next section, I'll talk about automation for building and deploying applications.

1.4.2 Continuous delivery

Continuous delivery is "a software development discipline where you build software in such a way that the software can be released to production at any time".¹²

With continuous delivery, teams implement features in short cycles, ensuring that the software can be released at any time reliably. Such a discipline is key to "make high-impact changes frequently and predictably with minimal toil" as per the cloud native definition by the CNCF.

Continuous integration (CI) is a foundational practice in continuous delivery. Developers commit their changes to the mainline (the main branch) continuously (at least once a day). At each commit, the software is automatically compiled, tested, and packaged as executable artifacts (for example, as JAR files or container images). The idea is to get fast feedback about the software status after each new change. If an error is detected, it should be immediately fixed to ensure the mainline keeps being a stable foundation for further developments.

Continuous delivery (CD) builds on CI and focuses on keeping the mainline always healthy and in a releasable state. After an executable artifact is produced as part of the integration with the mainline, the software is deployed to a production-like environment. It goes through additional

tests to assess its *releasability*, such as user acceptance tests, performance tests, security tests, compliance tests, and any other test that might increase the confidence that the software can be released. If the mainline is always in a releasable state, releasing a new version of the software becomes a business decision instead of a technical one.

Continuous delivery encourages automating the whole process via a **deployment pipeline** (also called *continuous delivery pipeline*), as described in the foundational book "Continuous Delivery" by Jez Humble and Dave Farley (Addison-Wesley, 2010). A deployment pipeline goes from code commit to a releasable outcome, and it's the only way to production. Throughout the book, we'll build a deployment pipeline to keep the main branch of our applications always in a releasable state. In the end, we're going to use it to deploy applications automatically to a Kubernetes cluster in production.

Sometimes continuous delivery is confused with **continuous deployment**. The former approach makes sure that after every change, the software is in a state in which it can be deployed to production. It's a business decision when actually to do that. With continuous deployment, instead, we add one last step to the deployment pipeline to automatically deploy a new release in production after every change.

Continuous delivery is not about tools. It's a discipline that involves cultural and structural changes in your organization. Setting up an automated pipeline to test and deliver your applications doesn't mean that you're doing continuous delivery. Similarly, using a CI server for automating builds doesn't mean doing continuous integration.¹³ That leads us to the next topic, which is also commonly mistaken for being about tools.

SIDE BAR

Continuous delivery vs. CI/CD

Since continuous integration is a foundational practice of continuous delivery, the combination is often referred to as CI/CD. As a consequence, deployment pipelines are often called CI/CD pipelines. I have some reservations about the term because continuous integration is not the only practice included in the continuous delivery discipline. For example, test-driven development (TDD), automated configuration management, acceptance testing, and continuous learning are equally important.

The CI/CD term has never been used by Jez Humble and Dave Farley in their "Continuous Delivery" book nor any other book they authored. Also, it can be confusing. Does CD stand for continuous delivery or continuous deployment? In this book, I will refer to the holistic approach of delivering "better software faster"¹⁴ as continuous delivery rather than CI/CD.

1.4.3 DevOps

DevOps is another of those buzzwords that are pretty popular these days but too often misunderstood. When shifting to cloud native, it's an important concept to grasp.

The origins of DevOps are peculiar. If you're interested in knowing more, I suggest watching the Ken Mugrage talks available on YouTube.¹⁵ One of the curious aspects is that the creators of this concept didn't provide a definition initially. The result was several people used their interpretation and, of course, we ended up using DevOps to mean different things.

Among all the definitions of DevOps, I find the one proposed by Ken Mugrage (Principal Technologist at ThoughtWorks) to be particularly informative and interesting. He highlights what I believe is the true meaning of DevOps.¹⁶

A culture where people, regardless of title or background, work together to imagine, develop, deploy and operate a system.

So, DevOps is a culture, and it's all about working together towards a common goal. Developers, testers, operators, security experts, and other people, regardless of title or background, work together to bring ideas to production and produce value.

It means the end of *silos*, no more walls between feature teams, QA teams, and operation teams. DevOps is often considered a natural continuation of Agile, which is an enabler for DevOps with the concept of small teams delivering value to customers frequently. A concise way to describe DevOps is by a famous sentence that Werner Vogels, Amazon CTO, pronounced in 2006 when DevOps was not even a thing yet: "You build it, you run it".¹⁷

Having defined what DevOps is, let's briefly mention what it isn't.

- **DevOps doesn't mean NoOps.** It's a common mistake to think that developers take care of operations and that the operator's role disappears. It's a collaboration. A team will include both roles, contributing to the overall team skills to bring a product from the original idea to production.
- **DevOps is not a tool.** Tools like Docker, Ansible, Kubernetes, Prometheus are usually referred to as DevOps tools, but that's wrong. DevOps is a culture. You don't turn into a DevOps organization by using tools. In other words, DevOps is not a product, but tools are relevant enablers.
- **DevOps is not automation.** Even if automation is an essential part of DevOps, it's not its definition. DevOps is about developers and operators working together from the idea to production while possibly automating some of their processes, like continuous delivery.
- **DevOps is not a role.** If we consider DevOps as a culture, a mindset, it's hard to make sense of a DevOps role. And yet, there is an increasing request for DevOps engineers. Usually, when recruiters search for DevOps engineers, they look for skills like proficiency with automation tools, scripting, and IT systems.
- **DevOps is not a team.** Organizations not fully understanding the above risk end up

keeping the same silos as before, with one change: adding a DevOps silo to replace the Ops silo or, even worse, as a new one.

The collaboration between developers and operators is paramount when going cloud native. As you may have noticed, designing and building cloud native applications requires that you always keep in mind where you're going to deploy those applications: the cloud. Working together with operators allows developers to design and build higher-quality products.

It's called DevOps, but let's remember that the definition doesn't apply only to developers and operators. Instead, it generically refers to people, regardless of title or background. It means that collaboration also involves other roles like testers and security experts (we probably don't need new terms like DevSecOps, DevTestOps, DevSecTestOps, or DevBizSecTestOps). Together, they are all responsible for the entire product lifecycle and are key to achieving the goals of continuous delivery.

1.5 Is the cloud your best option?

One of the biggest mistakes in our industry is deciding to adopt a technology or approach just because it's new and everyone talks about it. There are endless stories about companies migrating their monoliths to microservices and ending with disastrous failures. I have already explained the properties of the cloud and cloud native applications. Those should provide you with some guidance. If your system doesn't need those properties because it doesn't have the problems they are trying to solve, chances are that "going cloud native" is not the best option for your project.

As technologists, it's easy to get caught up in the latest, trendiest, shiniest technology. The point is to figure out whether a specific technology or approach can solve *your* problems. We turn ideas into software that we deliver to our customers and provide them with some value. That's our end goal. If a technology or approach helps you provide more value to your customers, you should consider it. If it's not worthy, and you decide to go with that anyway, you'll likely end up having higher costs and many problems.

When is moving to the cloud a good idea? Why are companies adopting the cloud native approach? The main goals for going cloud native, illustrated in figure 1.8, are speed, scale, resilience, and cost. If your business vision includes those goals and faces the same problems cloud technologies try to solve, then it's good to consider moving to the cloud and adopting a cloud native approach. Otherwise, it might be better to stay on the ground. For example, if your company is providing services through a monolithic application in its maintenance phase, that will not be further expanded with new features, and that performed well in the last decade, then there might be no good reason to migrate it to the cloud, let alone turn it into a cloud native application.

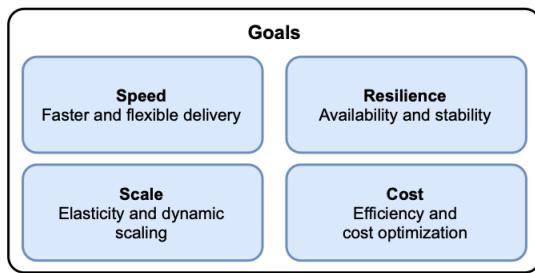


Figure 1.8 Going cloud native has the goals of achieving speed, resilience, scale, and cost optimization.

1.5.1 Speed

Being able to deliver software faster is an important goal for enterprises nowadays. Bringing ideas into production as fast as possible, therefore reducing the time-to-market, is a critical competitive advantage. Going to production with the right idea at the right time might make the difference between success and failure.

Customers expect to get more and more features implemented or bugs fixed, and they want them now. They would not be happy to wait six months for the next release of our software. Their expectations keep increasing, and you need a way to keep up with them. In the end, it's all about providing value to customers and ensuring they are satisfied with the results. Otherwise, your business would not survive the fierce competition out there.

Delivering faster and frequently is not only about competition and customer deadlines. It's also about making the feedback cycle shorter. Frequent and small releases mean that you can get feedback from the customer sooner. A shorter feedback loop, in turn, decreases the risk associated with the new functionality you're releasing. Instead of spending months trying to implement the perfect feature, you can get it out there soon, get feedback from customers, and adjust it to align with their expectations. Also, smaller releases contain fewer changes, thus reducing the number of parts that could fail.

Flexibility is also needed since customers expect continuous evolution of your software. For example, it should be flexible enough to support new types of clients. Nowadays, more and more objects in our daily life have been connected to the Internet. Think about all kinds of mobile and IoT systems. You want to be open to any future extension and client type to provide business services in new ways.

The traditional software development approach does not support this goal. Too often is characterized by massive releases, little flexibility, and extended release-cycles. The cloud native approach, combined with automated tasks, continuous delivery workflows, and DevOps practices, helps businesses go faster and reduce the time to market.

1.5.2 Resilience

Everything changes, and failures happen all the time. Gone are the times when we tried to predict failures and treat them as exceptions. As I mentioned before, changes are not an exception. They are the rule.

Whether there is a failure in the infrastructure or the software, your goal is to guarantee the availability and reliability of your systems. You want to keep providing services to your users, even if only in a degraded operational mode.

Customers want the software to be available 24/7 and upgraded as soon as a new feature is released. Downtime or failures can cause a direct loss of money and customer dissatisfaction. Perhaps they can even affect one's reputation, leading to damage to the organization itself in its future market opportunities.

A resilient system continues to provide its services even when something wrong happens. To guarantee availability, you need to have something in place to face the failures when they arrive, handle them, and ensure that the overall system can still provide its services to its users. Any action required to process the failure and tasks like upgrades should be done with zero downtime. Customers expect that.

No piece of hardware or software is 100% reliable and available, so you need resilience. We want cloud native applications to be resilient in the face of adversity. And cloud technologies enable us to implement strategies for implementing resilient infrastructures. If being always available, safe, and resilient is a requirement for your business, then the cloud native approach is a good option for you. The resilience of a software system, in turn, enables speed: the more stable the system, the more frequently you can safely release new features.

1.5.3 Scale

Elasticity is about being able to scale your software depending on the load. You can scale an elastic system to ensure an adequate service level for all your customers. There might be a higher load than usual, so you need to spin up more instances of your services to support that extra traffic. Or it could be that something terrible happened and some services failed, so you need to be able to spin up new instances to replace them.

The problem is that foreseeing what will happen is hard, if not impossible. It's not enough to build scalable applications. You need them to scale dynamically. Whenever there is a high load, your system should scale out dynamically, quickly, and painlessly. And when the high peak is over, it should scale in again.

If your business needs to adapt quickly and efficiently to new customers or requires the flexibility to support new types of clients (which increase the workload on the servers), the

cloud's very nature can provide you with all the elasticity you need, in combination with cloud native applications that are scalable by definition.

1.5.4 Cost

As a software developer, you might not deal directly with money, but it's your responsibility to take cost into account when designing a solution. The cloud computing model helps optimize IT infrastructure costs with its elasticity and on-demand pay-per-use policy. No more always-on infrastructure: you provision resources when you need them, pay for the actual usage, and then destroy them when you don't need them anymore.

On top of that, adopting the cloud native approach leads to further cost optimizations. Cloud native applications are designed to be scalable to take advantage of the cloud elasticity. They are resilient, so the costs related to downtime and hard failures in production get lower. Being loosely coupled, they enable teams to go faster and speed up the time-to-market, with notable competitive advantages. And the list goes on.

SIDE BAR

The hidden costs of moving to the cloud

Before deciding to migrate to the cloud, it's essential also to consider other types of costs. On the one hand, you can optimize costs as described above. On the other hand, you should consider the cost of migrating and its consequences. Migrating to the cloud requires specific competencies that employees might not have yet. That would mean investing in their education to acquire the necessary skills and perhaps hiring professionals as consultants to help with the migration to the cloud. Depending on the chosen solution, it might be required for an organization to take over some extra responsibilities compared to the previous situation, which in turn requires specific skills (for example, handling security in the cloud). And there are other considerations, like business interruptions during the migration, re-training end-users, and updating documentation and support materials.

1.6 Cloud native topologies

My explanation of cloud native didn't involve specific technologies or architectures. The CNCF mentions some in its definition, like containers and microservices. But as I previously underlined, those are just examples. You don't have to use Docker containers for your applications to be cloud native. Think about serverless or PaaS solutions. Writing functions for the AWS Lambda platform or deploying applications to Heroku doesn't require you to build containers. Still, they are categorized as cloud native.

In this section, I'll describe some common cloud native topologies (figure 1.9). First, I'll introduce the concepts of containers and orchestration, which will be further explored later when

discussing Docker and Kubernetes. Then, I'll present the topic of serverless technologies and functions (FaaS). I will not focus much on the FaaS model in this book, but I will cover the basics of how to build serverless applications using Spring Native and Spring Cloud Function.

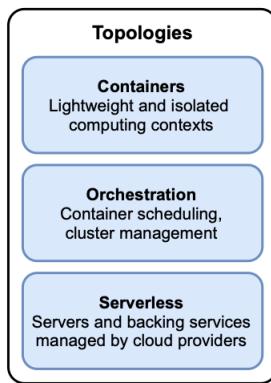


Figure 1.9 The main cloud native computing models are containers (managed by orchestrators) and serverless.

1.6.1 Containers

Imagine you join a team and start working on an application. The first thing you do is follow a guide to set up your local development environment similar to the ones used by your colleagues. You develop a new feature and then test it in a Quality Assurance (QA) environment. Once verified, the application can be deployed in staging for some extra testing and finally to production. The application is built to run in an environment with specific characteristics, so it's essential to have all the mentioned environments as similar as possible. How would you do that? That's where containers enter the scene.

Before containers, you would rely on virtual machines to guarantee the reproducibility of the environment, isolation, and configurability. The virtualization works by leveraging a hypervisor component that abstracts the hardware, making it possible to run multiple operating systems on the same machine in an isolated fashion. The hypervisor would run directly on the machine hardware (type 1) or the host operating system (type 2).

On the other hand, an *OS container* is a lightweight executable package that includes everything needed to run the application inside. Containers share the same kernel as the host: no need to bootstrap full operating systems to add new isolated contexts. On Linux, that is possible by leveraging a couple of features offered by the Linux kernel:

- **namespaces** for partitioning resources among processes so that each process (or group of processes) can only see a subset of the resources available on the machine;
- **cgroups** for controlling and limiting the resource usage for a process (or group of processes).

NOTE

When using virtualization only, the hardware is shared, whereas containers also share the same operating system kernel. In any case, both provide computing environments for running software in isolation, even if the degree of isolation is not the same.

Figure 1.10 shows the difference between virtualization and container technologies.

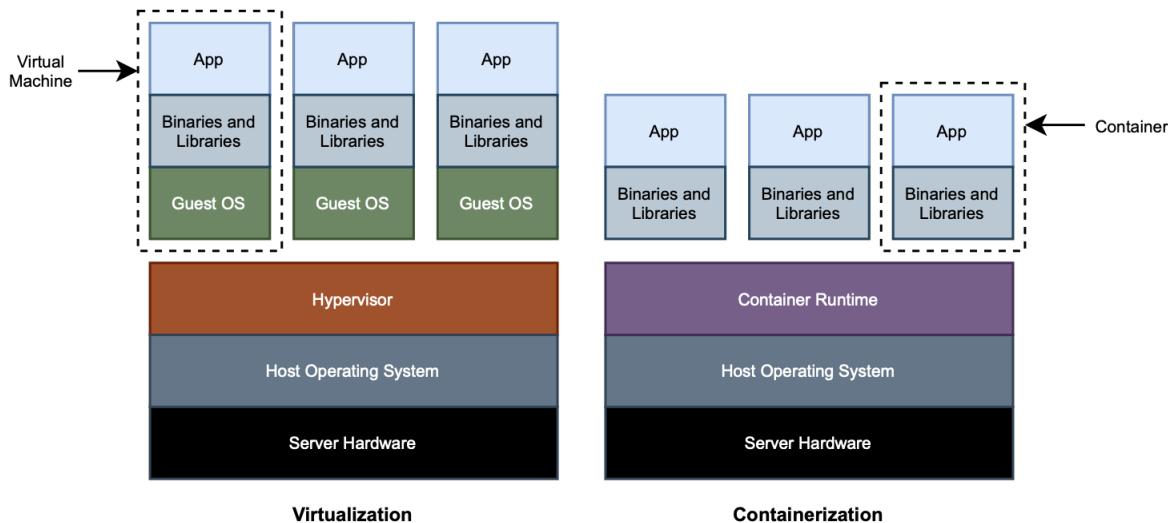


Figure 1.10 Virtualization and container technologies differ in what is shared across isolated contexts. Virtual machines share the hardware only. Containers share the operating system kernel as well. The latter are more lightweight and portable.

Why are containers so popular for cloud native applications? Traditionally, you would have to install and maintain the Java Runtime Environment (JRE) and middleware on a virtual machine to make your application run. Instead, a container can run reliably in almost any computing environment, independent of the application, its dependencies, or middleware. It doesn't matter which kind of application it is, in which language is written, or which libraries it uses. All containers have a similar shape from the outside, just like the containers used for shipping.

Therefore, containers enable agility, portability across different environments, and deployment repeatability. Being so lightweight and less resource-demanding, they are perfect for running in the cloud, where applications are disposable and scaled dynamically and quickly. In comparison, building and destroying virtual machines is much more expensive and time-consuming.

SIDE BAR**Containers! Containers everywhere!**

Container is one of those words that can mean different things depending on the context. Sometimes this ambiguity can generate confusion, so let's see what it means in different contexts.

- **OS.** An OS container is a method for running one or more processes in an environment isolated from the rest of the system. This book will focus on Linux containers, but note that Windows containers also exist.
- **Docker.** A Docker container is an implementation of a Linux container.
- **OCI.** Standardization of the Docker container implementation by the Open Container Initiative (OCI).
- **Spring.** A Spring container is the application context where objects, properties, and other application resources are managed and executed.
- **Servlet.** A Servlet container provides a runtime for web applications leveraging the Java Servlet API. The Catalina component from the Tomcat server is an example of a Servlet container.

Virtualization and containers are not mutually exclusive. In reality, you use them both in a cloud native context, having an infrastructure made up of virtual machines on which you run containers. The IaaS model (Infrastructure as a Service) provides a virtualization layer that you can use to bootstrap new virtual machines. On top of that, you can install a container runtime and run your containers.

An application is usually made up of different containers that can be run on the same machine while developing or performing early testing. But you would quickly reach the point where it gets too complicated to manage many containers, primarily when you start replicating them for scalability and distributing them across different machines. That's when you start relying on the higher level of abstraction provided by the CaaS model (Container as a Service), that offers functionality to deploy and manage containers in clusters of machines. Note that behind the scenes, there is still a virtualization layer.

Containers are involved even when using PaaS platforms like Heroku or Cloud Foundry. You deploy your applications on those platforms by providing just the JAR artifact since they take care of the JRE, the middleware, the OS, and any needed dependencies. Though, behind the scenes, they build a container out of all those components and finally run it. The difference is that it's no longer you being responsible for building a container, but the platform itself does it for you. On the one hand, that is convenient in terms of fewer responsibilities for developers. On the other hand, you give up control over the runtime and middleware and might face a vendor lock-in.

In this book, you'll learn how to use Cloud Native Buildpacks (a CNCF project) to containerize

Spring applications, and you'll use Docker to run them in your local environment.

1.6.2 Orchestration

So you have decided to use containers, great! You can rely on their portability to deploy them to any infrastructure providing a container runtime. You achieve reproducibility, so there are no bad surprises when moving containers from development to staging to production. You can scale them quickly since they're so lightweight and get high availability for your applications. You're ready to adopt them for your next cloud native system. Or are you?

Provisioning and managing containers on a single machine is pretty straightforward. But when you start dealing with tens or hundreds of containers scaled and deployed on several machines, you need something else.

When we move from virtual servers (IaaS model) to container clusters (CaaS model), we also switch our point of view.¹⁸ In IaaS, we focus on single computing nodes, which are virtual servers. In CaaS, the underlying infrastructure is abstracted, focusing on clusters of nodes.

With the new perspective offered by CaaS solutions, the deployment target will not be a machine anymore but rather a cluster of machines. CaaS platforms, such as those based on Kubernetes, provide many features to address all the significant concerns we look for in a cloud native environment, *orchestrating* containers across machines. The two different topologies are shown in figure 1.11.

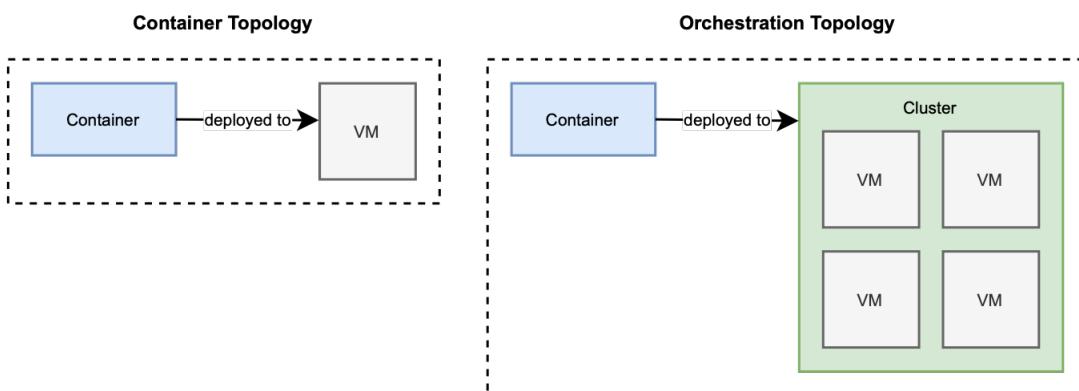


Figure 1.11 The deployment target of containers is a machine, whereas for orchestrators it's a cluster.

Container orchestration helps you automate many different tasks:

- managing clusters, bringing up and down machines when necessary;
- scheduling and deploying containers within a cluster to a machine that meets the container requirements for CPU and memory;
- dynamically scaling containers for high availability and resilience, leveraging health monitoring;
- setting up networks for containers to communicate with each other, defining routing,

- service discovery, and load balancing;
- exposing services to the Internet, establishing ports and networks;
- allocating resources to containers according to specific criteria;
- configuring the applications running within the containers;
- ensuring security and enforcing access control policies.

Instructing orchestration tools is done declaratively, for example, through YAML files. Following the format and language defined by the specific tool, you usually describe the state you'd like to achieve: you would like to have three replicas of your web application container deployed in a cluster, exposing its services to the Internet.

Examples of container orchestrators are Kubernetes (a CNCF project), Docker Swarm, and Apache Mesos. In this book, you'll learn how to use Kubernetes for orchestrating the containers for your Spring applications.

1.6.3 Serverless

After moving from virtual machines to containers, the abstraction layer provided by the cloud service can be further increased: that's where serverless technologies are placed.

Following the serverless computing model, developers focus on implementing the business logic for their applications. Using an orchestrator like Kubernetes would still require considering infrastructure provisioning, capacity planning, and scaling. On the other hand, a serverless platform takes care of setting up the underlying infrastructure needed by the applications to run, including virtual machines, containers, and dynamic scaling.

NOTE

The name **serverless** might be misleading: of course, there is a server. The difference is that it's not you managing it or orchestrating the application deployment on it. It's a platform responsibility now.

Serverless architectures are usually associated with functions, but they comprise two main models often used together.

- **Backend-as-a-Service** (BaaS). This model represents applications relying heavily on third-party services offered by cloud providers, such as databases, authentication services, and message queues. The focus is on reducing development and operational costs related to backing services. Developers can implement frontend applications (such as single-page applications or mobile applications) while offloading most or all of the backend functionality to BaaS vendors. For example, they could use Okta to authenticate users, Google Firebase for persisting data, and Amazon API Gateway to publish and manage REST APIs.
- **Function-as-a-Service** (FaaS). This model represents applications that are stateless, triggered by events, and fully managed by the platform. The focus is on reducing deployment and operations costs related to orchestrating and scaling applications. Developers can implement the business logic for their applications while the platform

takes care of the rest. Serverless applications don't have to be implemented with functions to be categorized as such. There are two main offering types for the FaaS models. One option is to go with vendor-specific FaaS platforms, such as AWS Lambda, Azure Functions, and Google Cloud Functions. Another option is to choose a serverless platform based on open-source projects, and that can run both in a public cloud and on-premises, addressing concerns like vendor lock-in and lack of control. Examples of such projects are Knative and Apache OpenWhisk. Knative provides a serverless runtime environment on top of Kubernetes, as we'll see in chapter 16. It's used as the foundation for enterprise serverless platforms like VMware Tanzu Application Platform, RedHat OpenShift Serverless, and Google Cloud Run.

Serverless applications are typically event-driven and run only when there is an event to handle, such as an HTTP request or a message. The event can be external or produced by another function. For example, a function might be triggered whenever a message is added to a queue, process it, and then exit the execution. When there is nothing to process, the platform shuts down all the resources involved with the function. So you can really pay for the actual usage. In the other cloud native topologies like CaaS or PaaS, there is always a server involved that runs 24/7. Compared to traditional systems, you get the advantage of dynamic scalability to reduce the number of resources provisioned at any given time. Still, there is always something up and running that has a cost. In the serverless model, instead, resources are provisioned only when necessary. If there is nothing to process, everything is shut down. That's what we call *scaling to zero*, one of the main features offered by serverless platforms.

Besides cost optimization, serverless technologies also move some extra responsibility from the application to the platform. That might be an advantage since it allows developers to focus exclusively on the business logic. But it's also essential to consider which degree of control you would like to have and how to deal with vendor lock-in. Each FaaS, and in general serverless, platform has its own features and APIs. Once you start writing functions for a specific platform, you can't move them easily to another, as you would do with containers. With FaaS, you might compromise responsibility and scope with control and portability more than any other approach. That's why Knative became popular quickly: being built on Kubernetes means that you can easily move your serverless workloads between platforms and vendors. In the end, it's a matter of trade-offs.

1.7 Architectures for cloud native applications

We have reached the last step of our journey to define cloud native and introduce the primary characteristics we'll rely on throughout the book. In the previous section, you got familiar with the main cloud native topologies, particularly containers: our computation units. Now, let's look at what is inside and explore some high-level principles about architecting and designing cloud native applications. Figure 1.12 shows the main concepts covered in this section.

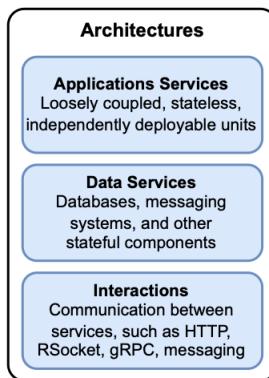


Figure 1.12 Cloud native architectural elements.

1.7.1 From multi-tiered to microservices architectures and beyond

IT infrastructures have always affected the way software applications were architected and designed. In the beginning, we used to have monolithic applications deployed on huge mainframes as single components. When the Internet and PCs became popular, we started designing applications according to the client/server paradigm. A multi-tiered architecture, relying on that paradigm, became widely used for desktop and web applications, decomposing the code into the presentation, business, and data layers.

With the increase in application complexity and the need for agility, new ways of further decomposing the code have been explored, and a new architectural style entered the stage: microservices. In the last few years, this architectural style has become more and more popular, and many companies decided to refactor their applications according to this new style. Microservices are usually compared to monolithic applications, as shown in figure 1.13.

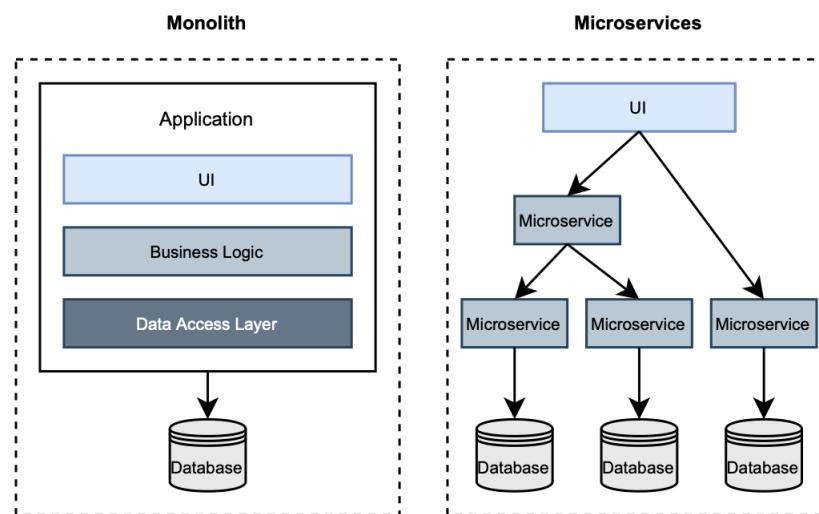


Figure 1.13 Monolith vs. microservices. Monolithic architectures are often multi-tiered. Microservices are made up of different components that can be deployed independently.

The main difference lies in how the application has been decomposed. A monolithic application is associated with using three large layers. Instead, a microservice-based application is associated with many components implementing only one piece of functionality each. Many patterns have been proposed to decompose a monolith into microservices and handle the complexity created by having many components instead of one.

NOTE

This book is not about microservices. Hence, I won't go into details. If you are interested in the topic, you can check out "Building Microservices" by Sam Newman (O'Reilly) and "Microservice Patterns" by Chris Richardson (<https://www.manning.com/books/microservices-patterns>). For a more Spring-oriented analysis, you can find "Spring Microservices" by John Carnell and Hillary Huaylupo Sanchez in the Manning catalog (<https://www.manning.com/books/spring-microservices-in-action-second-edition>). If you are not familiar with microservices, don't worry. That knowledge is not required to follow along with the book.

After many years of fame and failed migrations, there have been intense discussions about the future of this popular architectural style in the developer community. Some engineers started talking about **macroservices** to reduce the number of components and, therefore, the complexity of managing them. Even if their sizes were not originally the point of microservices.¹⁹ The term "macroservices" was suggested sardonically by Cindy Sridharan. Still, since then, it's been adopted in the industry and used by companies like Dropbox and AirBnB to describe their new architectures.²⁰ Others proposed a **citadel** architectural style, consisting of a central monolith surrounded by microservices. Still, others are advocating a return to monolithic applications in the form of modular monoliths.

In the end, I think that the important thing is to choose an architecture that can support us in delivering value to customers and our business. That's why we develop applications in the first place.

Each architectural style has its use cases. There is no such thing as a silver bullet or one-size-fits-all solution. Most of the negative experiences related to microservices were caused by other issues such as bad code modularization or unfitting organization structure. There shouldn't be a battle between monoliths and microservices.

In this book, I'm interested in showing you how to build cloud native applications with Spring and deploy them to Kubernetes as containers. Cloud native applications are distributed systems, just like microservices are. You will find topics usually discussed in the context of microservices but that actually belong to any distributed system, such as routing and service discovery. Cloud native applications are loosely-coupled by definition, and that is also a feature of microservices.

Even if there are some similar aspects, it's essential to understand that cloud native applications

and microservices are not the same. You can definitely use a microservice style for cloud native applications. In fact, it's what many developers do. But it's not a requirement. In this book, I will use an architectural style that we might call *service-based*. Perhaps it's not a catchy name nor fancy, but that's enough for our purpose. We deal with services. They can be of any size, and they can encapsulate logic according to different principles. It doesn't matter. What you want to go after is designing services to fit your development, organizational, and business needs.

1.7.2 Service-based architecture for cloud native applications

Throughout the book, we'll design and build cloud native applications according to a service-based architecture.

Our central unit of work will be a service that can interact with other services in different ways. Using the distinction proposed by Cornelia Davis, we identify two elements of the architecture: services and interactions.²¹

- **Service:** a component that provides any kind of service to another component.
- **Interaction:** the communication of services with each other to accomplish the system's requirements.

Services are quite generic components. They might be anything. We can classify them considering whether they store any kind of state, distinguishing between *application services* (stateless) and *data services* (stateful). Figure 1.14 shows the elements of a cloud native architecture. An application used to manage the inventory for a book library would be an application service. A PostgreSQL database used to store information about the books would be a data service.

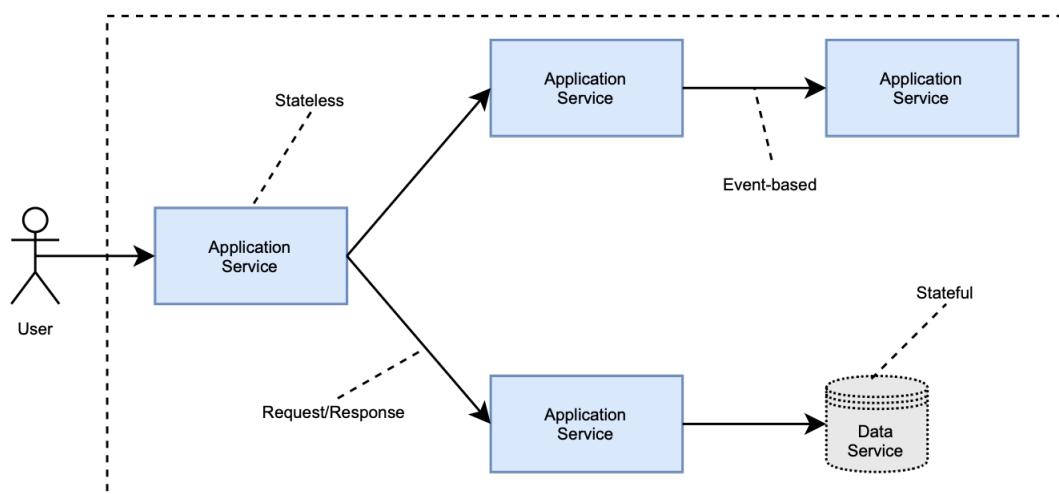


Figure 1.14 Service-based architecture for cloud native applications. The main elements are services (application or data) that interact with each other in different ways.

APPLICATION SERVICES

Application services are stateless and responsible for implementing any kind of logic. They don't have to obey specific rules like those for microservices, as long as they expose all the cloud native properties you learned earlier in this chapter.

It's paramount that you design each service with loose coupling and high cohesion in mind. Services should be as independent as possible. Distributed systems are complex, so you should be extra careful during the design phase. Increasing the number of services leads to increasing the number of problems.

You'll probably develop and maintain most of the application services in your system by yourself, but you can also use some offered by the cloud provider such as authentication or payment services.

DATA SERVICES

Data services are stateful and responsible for storing any kind of state. The *state* is everything that should be preserved when shutting down a service and spinning up a new instance.

They can be relational databases like PostgreSQL, key/value stores like Redis, or message brokers like RabbitMQ. You can manage these services by yourself. It's more challenging than cloud native applications due to the storage required to save the state, but you would gain more control over your own data. Another option is to use the data services offered by a cloud provider, which will be responsible for managing all the concerns related to storage, resilience, scalability, and performance. In the latter case, you can also take advantage of the many data services explicitly built for the cloud, such as Amazon DynamoDB, Azure Cosmos DB, or Google BigQuery.

Cloud native data services are a fascinating topic, but we will mainly deal with applications in this book. Data-related concerns like clustering, replication, consistency, or distributed transactions will not be detailed too much in the book. I would love to, but they deserve their own book to be adequately covered.

INTERACTIONS

Cloud native services communicate with each other to fulfill the requirements of the system. How communication happens will affect the overall properties of the system. For example, choosing a request/response pattern (synchronous HTTP call) over an event-based approach (message streamed through RabbitMQ) will result in different levels of resilience for the application. In this book, we'll use different types of interactions, learn the differences, and when to use each approach.

1.8 Summary

- Cloud native applications are highly distributed systems specifically designed for and live in the cloud.
- The cloud is an IT infrastructure provided as a commodity in terms of computing, storage, and networking resources.
- In the cloud, users pay only for the actual resources they use.
- Cloud platforms deliver their services at different levels of abstraction: infrastructure (IaaS), container (CaaS), platform (PaaS), functions (FaaS), or software (SaaS).
- Cloud native applications are horizontally scalable, loosely coupled, highly-cohesive, resilient to faults, manageable, and observable.
- Cloud native development is supported by automation, continuous delivery, and DevOps.
- Continuous delivery is a holistic engineering practice for delivering high-quality software quickly, reliably, and safely.
- DevOps is a culture enabling collaboration among different roles to deliver business value together.
- Modern businesses go cloud native to produce software that can be delivered fast, scaled dynamically depending on the needs, and always available and resilient to failures while optimizing costs.
- Containers can be used as computational units to design cloud native systems. They are more lightweight than virtual machines and provide portability, immutability, and flexibility. Example: Docker.
- Dedicated platforms offer services to manage containers without directly handling the underlying layers. They provide container orchestration, cluster management, network services, and scheduling. Example: Kubernetes.
- Serverless computing is a model where the platform manages servers and the underlying infrastructure while the developer only focuses on the business logic. The backend functionality is enabled on a pay-per-use basis for cost optimization. Example: Knative.
- A microservices architecture can be used to build cloud native applications, but it's not a requirement.
- To design cloud native applications, we use a service-based style characterized by services and their interactions.
- Cloud native services can be classified into application services (stateless) and data services (stateful).

Cloud native patterns and technologies



This chapter covers

- Understanding development principles for cloud native applications
- Building cloud native applications with Spring Boot
- Containerizing applications with Docker and Buildpacks
- Deploying applications to the cloud with Kubernetes
- Introducing patterns and technologies used in the book

The way we design applications for the cloud is different compared to traditional approaches. The Twelve-Factor methodology, consisting of best practices and development patterns, is a good starting point for building applications that can be considered cloud native. I'll explain the methodology in the first part of the chapter, and I'll expand on it throughout the book.

We'll then build a simple Spring Boot application and run it with Java, Docker, and Kubernetes, as shown in figure 2.1. Throughout the book, I'll go deeper into each of those topics, so don't worry if something is not entirely clear. This chapter aims to provide you with a mental map of the journey we'll undertake from code to production in a cloud environment while familiarizing yourself with patterns and technologies that we'll use in the rest of the book.

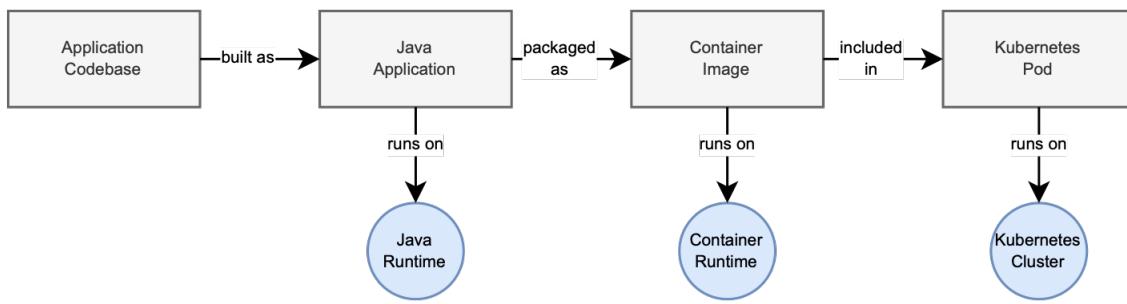


Figure 2.1 The journey of a Spring application from Java to container to Kubernetes.

Finally, I'll introduce you to the cloud native project we'll build chapter after chapter using Spring and Kubernetes. We'll employ all the properties and patterns for cloud native applications mentioned in this first part of the book.

2.1 Cloud native development principles: Twelve Factors and beyond

The engineers working at the Heroku cloud platform have proposed the Twelve-Factor methodology as a collection of development principles to design and build cloud native applications.²² They have distilled their experience into best practices to build web applications with the following characteristics:

- suitable to be deployed on cloud platforms;
- scalable by design;
- portable across systems;
- enabler of continuous deployment and agility.

The goal was to help developers build applications for the cloud, highlighting the significant factors to consider to achieve the best results.

Later, the methodology was revised and expanded by Kevin Hoffman in his book "Beyond the Twelve-Factor App", refreshing the contents of the original factors and adding three extra ones.

²³ From now on, I will refer to this expanded set of principles as the *15-Factor methodology*.

Overall, those fifteen factors will guide you throughout the book since they are a good starting point for developing cloud native applications. If you are building a new application from scratch or migrating a traditional system to the cloud, those principles can help you along the journey. I will elaborate more on them when relevant and illustrate how to apply them to Spring applications, but it's essential to become familiar with them.

Let's dig into each one.

2.1.1 One codebase, one application

The methodology establishes a one-to-one mapping between an application and its codebase. So, one codebase for each application. Any shared code should be tracked in its own codebase as a library that can be included as a dependency or service that can be run in standalone mode, acting as a backing service for other applications. Each codebase can optionally be tracked in its own repository.

A deployment is a running instance of the application. Many deployments are possible in different environments, all sharing the same application artifact. There is no need to re-build the codebase to deploy the application to a specific environment: any aspect that changes between deployments (for example, configuration) should be outside the application codebase.

2.1.2 API first

A cloud native system is usually made up of different services that communicate through APIs. Using an *API first* approach while designing a cloud native application encourages reasoning about fitting it in a distributed system and favors the distribution of the work across different teams. By designing the API first, another team using that application as a backing service could create their solution against that API. By designing the contract up-front, integrating with other systems will be more robust and testable as part of the deployment pipeline. Internally, the API implementation can be changed without affecting other applications (and teams) depending on it.

2.1.3 Dependency management

All application dependencies should be declared explicitly in a manifest and available for the dependency manager to be downloaded from a central repository. When considering Java applications, we are usually very well-equipped to follow this principle using tools like Maven or Gradle. The only implicit dependencies an application can have on the surrounding environment are the language runtime and the dependency manager tool. That means that also private dependencies should be resolved via the dependency manager.

2.1.4 Design, build, release, run

A codebase goes through different stages in its journey from design to deployment in production.

- **Design stage.** Technologies, dependencies, and tools needed by a specific application feature are decided.
- **Build stage.** The codebase is compiled and packaged together with its dependencies as an immutable artifact called *build*. The build artifact must be uniquely identified.
- **Release stage.** The build is combined with a specific configuration for the deployment. Each release is immutable and should be uniquely identifiable, for example, using semantic versioning (such as 3.9.4) or a timestamp (for example, 2011-07-07_17:21). Releases should be stored in a central repository for easy access, like when a rollback to a previous version is required.
- **Run stage.** The application runs in the execution environment from a specific release.

The methodology requires a strict separation of these stages and doesn't allow changes to the code at runtime since it would result in a mismatch with the build stage. The build and the release artifacts should be immutable and labeled with a unique identifier to guarantee reproducibility.

2.1.5 Configuration, credentials, and code

The 15-Factor methodology defines configuration as everything likely to change between deployments. Whenever you need to change the configuration for an application, you should be able to do so without any changes in the code, without building the application again.

The configuration might include resource handles to backing services like a database or a messaging system, credentials to access third-party APIs, and feature flags. Ask yourself if any credential or environment-specific information would be compromised should your codebase suddenly become public. That will tell you whether you have correctly externalized the configuration.

To be compliant with this factor, you can't have the configuration in the code or tracked in the same codebase. The only exception is the default configuration, which can be packaged with the application codebase. You can still use configuration files for any other type of configuration, but you should store them in a separate repository. The methodology recommends storing configuration as environment variables. By doing so, you can have the same application deployed in different environments but with different behaviors depending on the environment's configuration.

2.1.6 Logs

A cloud native application isn't concerned with routing and storage of logs. Applications should log to the standard output, treating logs as events emitted in a sequence ordered by time. Log storage and rotation are not application responsibilities anymore. An external tool (*log aggregator*) will fetch, collect, and make logs available for inspection.

2.1.7 Disposability

In a traditional environment, you would take much care of your applications, ensuring they stay up and running and never terminate. In a cloud environment, you don't care that much: applications are ephemeral. If a failure happens and the application doesn't respond anymore, you terminate it and start a new instance. If you have a high-load peak, you spin up more instances of your applications to sustain the increased workload. We say that an application is disposable if it can be started or stopped at any time.

To handle application instances in such a dynamic way, you should design them to start up quickly whenever you need a new instance and gracefully shut down when you don't need it

anymore. A fast startup enables the elasticity of the system, ensuring robustness and resilience. Without a fast startup, you would have performance and availability issues.

Graceful shutdown is when an application receives a signal to terminate: it stops accepting new requests, completes the ones already in progress, and finally exits. In the case of web processes, that is straightforward. In other cases, such as with worker processes, the job they were responsible for must be returned to the work queue, and only afterward can they exit.

2.1.8 Backing services

Backing services can be defined as external resources an application uses to deliver its functionality. Examples of backing services are databases, message brokers, caching systems, SMTP servers, FTP servers, or RESTful web services. Treating them as attached resources means that you can easily change them without modifying the application code.

Consider how you use databases throughout the software development lifecycle. Chances are that you use a different database depending on the stage: development, testing, or production. If you treat the database as an attached resource, you can use a different service depending on the environment. The attachment is done through resource binding. For example, resource binding would consist of a URL, a username, and a password for a database.

2.1.9 Environment parity

Environment parity is about keeping all environments as similar as possible. In reality, it's likely to find three gaps that this factor tries to address:

- **Time gap.** The period between a code change and its deployment can be quite large. The methodology strives to promote automation and continuous deployment, to reduce the period from the moment a developer writes code to when deployed in production.
- **People gap.** Developers build applications while operators manage their deployment in production. The gap can be resolved by embracing a DevOps culture, having a better collaboration between developers and operators, and realizing the "you build it, you run it" philosophy.
- **Tools gap.** One of the main differences between environments is how backing services are handled. For example, developers might use the H2 database in their local environment, while PostgreSQL is used in production. In general, the same type and version of backing services should be used in all environments.

2.1.10 Administrative processes

Some management tasks are usually needed to support applications. Tasks like database migrations, batch jobs, or maintenance jobs should be treated as one-off processes. The same considerations done for the application processes apply to the administrative ones as well. It means that the code of the administrative tasks should be tracked in revision control, delivered together with the application they support, and executed in the same environment as the application.

It's usually a good idea to frame administrative tasks as small standalone services that run once and then are thrown away, functions configured in a stateless platform to be triggered when certain events happen, or embedded in the application itself with the possibility to activate them by calling a specific endpoint.

2.1.11 Port binding

Applications following the 15-Factor methodology should be self-contained and export their services via port binding. In production, there might be some routing services that translate requests from public endpoints to the internal port-bound services.

An application is self-contained if it doesn't depend on an external server to be available in the execution environment. A Java web application would probably run inside a server container like Tomcat, Jetty, or Undertow. A cloud native application would not require the environment to have a Tomcat server available, but it would manage it itself as any other dependency. Spring Boot, for example, lets you use an embedded server: the application will contain the server rather than depending on one being available in the execution environment. One of the consequences of this approach is that there is always a one-to-one mapping between application and server, unlike the traditional method where multiple applications were deployed to the same server.

The services provided by the application are then exported via port binding. A web application would bind HTTP services to a specific port and potentially become a backing service for another application. That's what usually happens in a cloud native system.

2.1.12 Stateless processes

In the previous chapter, you saw that high scalability is one reason why we move to the cloud. To ensure scalability, we design applications as stateless processes and adopt a *share-nothing architecture*: no state should be shared among different application instances. Ask yourself if any data would be lost, should an instance of your application be destroyed and recreated. If the answer is affirmative, then your application is not stateless.

No matter what, we will always need to save some state, or our applications will be useless in most cases. So, we design applications to be stateless and then limit the state to be handled in

specific stateful services like data stores. In other words, a stateless application delegates the state management and storage to a backing service.

2.1.13 Concurrency

Stateless applications are not enough for scalability. If you need to scale, it means that you need to serve more users. Therefore, your applications should allow concurrent processing to serve many users at the same time.

The methodology defines processes as first-class citizens. Those processes should be horizontally scalable, distributing the workload across many processes on different machines. This concurrent processing is only possible if the applications are stateless. In JVM applications, we handle concurrency through multiple threads, available from thread pools.

Processes can be classified according to their types. For example, you might have web processes that handle HTTP requests and worker processes that execute scheduled jobs in the background.

2.1.14 Telemetry

Observability is one of the properties of cloud native applications. Managing a distributed system in the cloud is complex. The only chance you have to manage such complexity is by ensuring that every system component provides the correct data to monitor the system's behavior remotely. Examples of telemetry data are logs, metrics, traces, health status, and events. Hoffman uses a very catchy image to stress the importance of telemetry: treat your applications like space probes. What kind of telemetry would you need to monitor and control your applications remotely?

2.1.15 Authentication and authorization

Security is one of the essential qualities of a software system, but it often doesn't get the necessary attention. Following a *zero-trust* approach, we must secure any interaction within the system at any architectural and infrastructural level. There is undoubtedly more to security than just authentication and authorization, but that is a good starting point.

With authentication, we can keep track of who is using the application. Knowing that, we can then check for the user permissions to verify whether they are allowed to perform specific actions. A few standards are available to implement identity and access management, including OAuth 2.0 and OpenID Connect we will use in this book.

2.2 Building cloud native applications with Spring

It's time to get more concrete and start talking about technologies. So far, you have become familiar with the cloud native approach and the main development practices to follow. Now, let's have a look at Spring. If you are reading this book, you probably have had some previous experience with Spring, and you'd like to learn how to use it to build cloud native applications.

The Spring ecosystem provides features to handle almost any requirement your applications might have, including those for cloud native applications. It's by far the most used Java framework. It's been around for many years. It's robust and reliable. The community behind Spring is fantastic and willing to move it forward and make it consistently better. Technologies and development practices evolve continuously, and Spring is very good at keeping up with that. Therefore, using Spring for your next cloud native project is an excellent choice.

This section will highlight some interesting features of the Spring landscape. Then, we will start creating a Spring Boot application.

2.2.1 Overview of the Spring landscape

Spring comprises several projects to address many different aspects of software development: web applications, security, data access, integrations, batch processing, configuration, messaging, big data, and many more. The beauty of the Spring platform is that it's designed to be modular, so you can use and combine just the projects you need. It doesn't matter which type of application you need to build. Chances are that Spring can help you with that.

Spring Framework is the core of the Spring platform, the project that started it all. It supports dependency injection, transaction management, data access, messaging, web applications, and more. The framework establishes the "plumbing" of enterprise applications to let you focus on the business logic.

Spring Framework provides an execution context (called *Spring context* or *container*), where beans, properties, and resources are managed throughout the entire application lifecycle. I assume you are already familiar with the framework's core features, so I will not spend too much time on it. In particular, you should be aware of the Spring context's role and be comfortable working with Spring beans, annotation-based configuration, and dependency injection. We will rely on those features, so you should have them sorted out.

Based on the framework, **Spring Boot** makes it possible to build standalone, production-ready applications quickly. By taking an opinionated view of Spring and third-party libraries, Spring Boot comes bundled with a sensible default configuration to let developers get started with minimal up-front work while still providing full customization possibilities.

Throughout the book, you'll have the chance to use several Spring projects to implement patterns

and best practices for cloud native applications, including Spring Boot, Spring Cloud, Spring Data, Spring Security, Spring Session, and Spring Native.

NOTE If you are interested in learning more about the Spring core features, you can find a few books on the subject in the Manning catalog, including "Spring Start Here" by Laurentiu Spilca (<https://www.manning.com/books/spring-start-here>) and "Spring in Action" (6th Edition) by Craig Walls (<https://www.manning.com/books/spring-in-action-sixth-edition>). You can also refer to "Spring Boot: Up & Running" by Mark Heckler (O'Reilly).

2.2.2 Building a Spring Boot application

Imagine you have been hired to build a *Polar Bookshop* application for *Polarsophia*. This organization manages a specialized bookshop and wants to sell its books about the North Pole and the Arctic online. A cloud native approach is being considered.

As a pilot project, your boss has assigned you to demonstrate to your colleagues how to go from implementation to production in the cloud. The web application you are asked to build is Catalog Service, and for now will only have one responsibility: welcoming users to the book catalog. This pilot project will be the foundation for actual products built as cloud native applications, should that be successful and well-received.

Considering the task's goal, you might decide to implement the application as a RESTful service with a single HTTP endpoint responsible for returning a welcome message. Surprisingly enough, you choose to adopt Spring as the primary technology stack for the one service (Catalog Service) composing the application. The architecture of the system is shown in figure 2.2, and you'll be trying your hand at building and deploying the application in the upcoming sections.

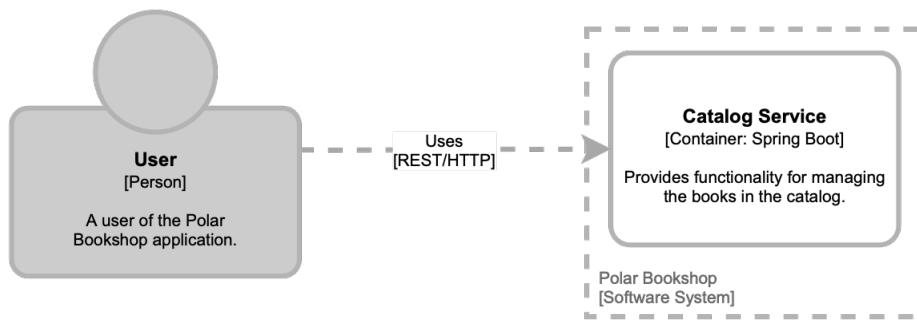


Figure 2.2 Architecture diagram for the Polar Bookshop application, following the C4 model.

In figure 2.2, you can see the notation that I'll use to represent architectural diagrams throughout the book, following the C4 Model created by Simon Brown.²⁴ To describe the architecture for the Polar Bookshop project, I'm relying on three abstractions from the model:

- **Person.** It represents one of the human users of the software system. In our example, it's a customer of the bookshop.
- **System.** It represents the overall application you build to deliver value to its users. In our example, it's the Polar Bookshop system.
- **Container.** It represents a service, either application or data. It's not to be confused with Docker. In our example, it's Catalog Service.

For this task, we'll use the Spring Framework and Spring Boot to do the following:

- declaring the dependencies needed to implement the application;
- bootstrapping the application with Spring Boot;
- implementing a controller to expose an HTTP endpoint for returning a welcome message;
- running and trying the application.

All the examples in this book are based on Java 17, the latest long-term release of Java at the time of writing. Before moving on, follow the instructions in section A.1 of Appendix A to install an OpenJDK 17 distribution. Then, ensure you have an IDE with support for Java, Gradle, and Spring. I'll be using IntelliJ IDEA, but feel free to choose another one, such as Visual Studio Code. Finally, create a free account on GitHub if you don't have one yet (<https://github.com>). We'll use it to store our code and define continuous delivery pipelines.

INITIALIZING THE PROJECT

Throughout the book, we'll build several cloud native applications. I recommend you define a Git repository for each of them and use GitHub for storing them. In the next chapter, I'll talk more about managing codebases. For now, go ahead and create a `catalog-service` Git repository.

Then, you can generate the project from Spring Initializr (<https://start.spring.io>) and store it in the `catalog-service` Git repository you have just created. It's a convenient service you can use from a browser or through its REST API to generate JVM-based projects. It's even integrated into popular IDEs such as IntelliJ IDEA and Visual Studio Code. The initialization parameters for Catalog Service are shown in figure [2.3](#).

The screenshot shows the Spring Initializr web application. At the top, there's a logo for 'spring initializr'. Below it, there are sections for 'Project' (choosing between Maven Project and Gradle Project), 'Language' (choosing Java, Kotlin, or Groovy, with Java selected), and 'Dependencies' (with an 'ADD DEPENDENCIES...' button). The 'Spring Boot' section shows version 2.7.0 selected from options 3.0.0 (SNAPSHOT), 3.0.0 (M3), 2.7.1 (SNAPSHOT), 2.6.9 (SNAPSHOT), and 2.6.8. The 'Project Metadata' section includes fields for Group (com.polarbookshop), Artifact (catalog-service), Name (catalog-service), Description (Provides functionality for managing the books in the catalog), Package name (com.polarbookshop.catalogservice), and Packaging (Jar selected over War). At the bottom, there's a Java version selector with 18, 17 (selected), 11, and 8.

Figure 2.3 The parameters for initializing the Catalog Service project from Spring Initializr.

During the initialization, you can provide a few details about the application you want to build, as shown in table [2.1](#).

Table 2.1 The main parameters you can configure to generate a project from Spring Initializr.

Parameter	Description	Value for Catalog Service
Project	You can decide whether you want to use Gradle or Maven as the build tool for your project. All the examples in this book will use Gradle.	Gradle
Language	Spring supports the three main JVM languages: Java, Kotlin, and Groovy. All the examples in this book will use Java.	Java
Spring Boot	You can choose which version of Spring Boot you want to use. All the examples in this book will use Spring Boot 2.7.0, but any subsequent patch should be fine.	Spring Boot 2.7.0
Group	The group ID for the project, as used in a Maven repository.	com.polarbookshop
Artifact	The artifact ID for the project, as used in a Maven repository.	catalog-service
Name	The project name.	catalog-service
Package name	The base Java package for the project.	com.polarbookshop.catalogservice
Packaging	How to package the project: WAR (for deployment on application servers) or JAR (for standalone applications). Cloud native applications should be packaged as JARs, so all the examples in this book will use that option.	JAR
Java	The Java version you want to use to build the project. All the examples in this book will use Java 17.	17
Dependencies	Which dependencies to include in the project.	Spring Web

The structure of the newly generated project is shown in figure [2.4](#). In the upcoming sections, I'll guide you through it.

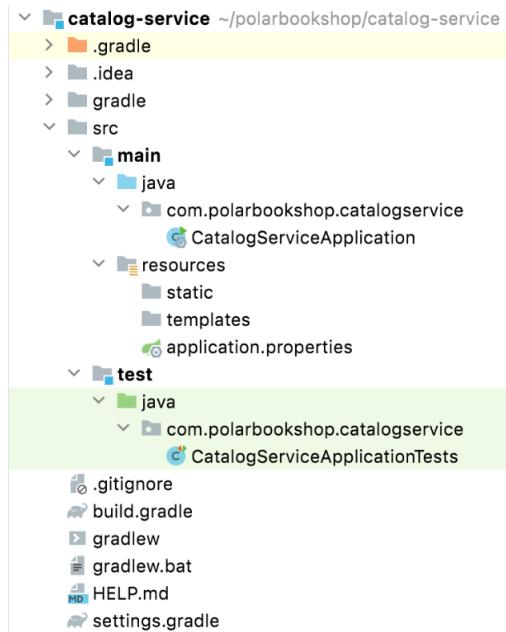


Figure 2.4 The structure of a new Spring Boot project generated from Spring Initializr.

In the code repository accompanying this book (<https://github.com/ThomasVitale/cloud-native-spring-in-action>), you can find begin and end folders for each chapter, so that you can always start with the same setup as me and check the final result. For example, you are currently reading chapter 2, so you will find the related code in Chapter02/02-begin and Chapter02/02-end.

TIP

In the `begin` folder for this chapter, you can find a `curl` command to run in a Terminal window that downloads a zip file containing all the code you need to get started without going through the manual generation through the Spring Initializr website.

SIDE BAR**Gradle or Maven?**

I'm using Gradle in this book, but feel free to use Maven instead. In the code repository accompanying this book, you can find a table mapping Gradle commands to Maven so that you can easily follow along should you use the second option (<https://github.com/ThomasVitale/cloud-native-spring-in-action>). Each project has different needs that might lead you to choose one build tool over another.

My choice of using Gradle is a personal preference and due to two main reasons. Building and testing a Java project with Gradle takes less time than Maven, thanks to its incremental and parallel builds and caching system. Also, I find the Gradle build language (Gradle DSL) to be more readable, expressive, and maintainable than Maven XML. In the Spring ecosystem, you can find projects using Gradle and others using Maven. They're both good choices. I recommend you try them both and pick the tool that makes you more productive.

EXPLORING THE BUILD CONFIGURATION

Start by opening the project you have just initialized in your favorite IDE and looking at the Gradle build configuration for the Catalog Service application, defined in the `build.gradle` file. You can find there all the information you provided to Spring Initializr.

Listing 2.1 The build configuration for the Catalog Service project (build.gradle)

```
plugins {
    id 'org.springframework.boot' version '2.7.0'      ①
    id 'io.spring.dependency-management'
    <version> '1.0.11.RELEASE'          ②
    id 'java'           ③
}

group = 'com.polarbookshop'      ④
version = '0.0.1-SNAPSHOT'       ⑤
sourceCompatibility = '17'        ⑥

repositories {                  ⑦
    mavenCentral()
}

dependencies {                 ⑧
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()          ⑨
}
```

- ① Provides Spring Boot support in Gradle and declares which version to use.
- ② Provides dependency management features for Spring.

- ③ Provides Java support in Gradle, setting up tasks to compile, build, and test the application.
- ④ The group ID for the Catalog Service project.
- ⑤ The version of the application. By default, it's 0.0.1-SNAPSHOT.
- ⑥ The Java version used to build the project.
- ⑦ Artifact repositories where to search for dependencies.
- ⑧ The dependencies used by the application.
- ⑨ Enables testing with the JUnit Platform provided by JUnit 5.

The project contains the following main dependencies:

- **Spring Web** (`org.springframework.boot:spring-boot-starter-web`) provides the necessary libraries for building web applications with Spring MVC and includes Tomcat as the default embedded server.
- **Spring Boot Test** (`org.springframework.boot:spring-boot-starter-test`) provides several libraries and utilities to test applications, including Spring Test, JUnit, AssertJ, and Mockito. It's automatically included in every Spring Boot project.

NOTE

Spring Boot offers convenient starter dependencies that bundle together all the libraries necessary for a specific use case, taking care of choosing versions that are compatible with each other. This feature significantly simplifies your build configuration.

The name of the project is defined in a second file called `settings.gradle`.

```
rootProject.name = 'catalog-service'
```

BOOTSTRAPPING THE APPLICATION

In a previous section, you initialized the Catalog Service project and chose the JAR packaging option. Any Java application packaged as a JAR must have a `public static void main(String[] args)` method to be executed at startup, and Spring Boot is no different. In Catalog Service, you got a `CatalogServiceApplication` class autogenerated during the initialization: that's where the `main()` method is defined and how the Spring Boot application is run.

Listing 2.2 The bootstrap class for Catalog Service (CatalogServiceApplication.java)

```
package com.polarbookshop.catalogservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication ①
public class CatalogServiceApplication {
    public static void main(String[] args) { ②
        SpringApplication.run(CatalogServiceApplication.class, args);
    }
}
```

- ① Defines a Spring configuration class and triggers component scanning and Spring Boot auto-configuration.
- ② The method used to launch the application. It registers the current class to be run during the application bootstrap phase.

The `@SpringBootApplication` annotation is a shortcut to include three different annotations:

- `@Configuration` marks the class as a source of beans definitions.
- `@ComponentScan` enables component scanning to find and register beans in the Spring context automatically.
- `@EnableAutoConfiguration` enables the auto-configuration capabilities offered by Spring Boot.

Spring Boot auto-configuration is triggered depending on several conditions, such as the presence of certain classes in the classpath, the existence of specific beans, or the value of some properties. Since the Catalog Service project depends on `spring-boot-starter-web`, Spring Boot will initialize an embedded Tomcat server instance and apply the minimal configuration required to get a web application up and running in almost zero time.

That was it for the application setup. Let's go ahead and expose an HTTP endpoint from Catalog Service.

IMPLEMENTING THE CONTROLLER

So far, we have looked at the project as generated from Spring Initializr. It's time to implement the business logic for the application. Catalog Service will expose an HTTP endpoint to return a greeting. You can define a handler for it in a controller class. It will be a `GET` endpoint and return a friendly message to the user to welcome them to the book catalog. Figure 2.5 shows the interaction flow.

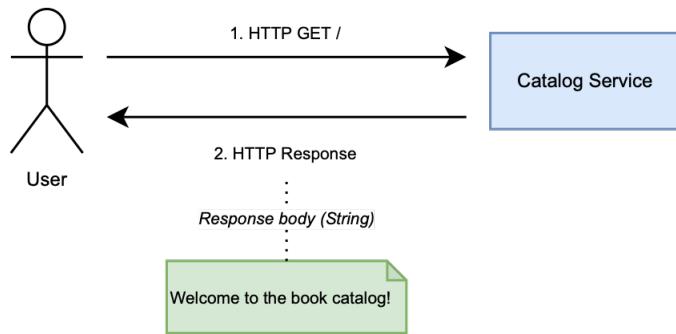


Figure 2.5 Interaction between user and application to get a welcome message from the HTTP endpoint exposed by Catalog Service.

In the Catalog Service project, create a new `HomeController` class and implement a method responsible for handling GET requests to the root endpoint (/).

Listing 2.3 Define an HTTP endpoint to return a welcome message (`HomeController.java`)

```

package com.polarbookshop.catalogservice;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController ①
public class HomeController {

    @GetMapping("/") ②
    public String getGreeting() {
        return "Welcome to the book catalog!";
    }
}
  
```

- ① Identifies a class defining handlers for REST/HTTP endpoints.
- ② Handles GET requests to the root endpoint.

The `@RestController` annotation identifies a class as a controller handling incoming HTTP requests. Using the `@GetMapping` annotation, you can mark the `getGreeting()` method as a handler for GET requests arriving at the root endpoint (/). Any GET request to that endpoint will be processed by this method. In the next chapter, I'll cover how to build RESTful services with Spring in more detail.

TESTING THE APPLICATION

A basic test setup is included when creating a Spring project from Spring Initializr. In the `build.gradle` file, you automatically get the dependencies needed for testing a Spring application. Furthermore, a test class is automatically generated. Let's have a look at how the `CatalogServiceApplicationTests` class might look after initializing the project.

**Listing 2.4 The auto-generated test class verifying the Spring context
(CatalogServiceApplicationTests.java)**

```
package com.polarbookshop.catalogservice;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest ①
class CatalogServiceApplicationTests {

    @Test ②
    void contextLoads() { ③
    }
}
```

- ① Provides a setup for testing Spring Boot applications.
- ② Identifies a test case.
- ③ Empty test used to verify the application context is loaded correctly.

The default test class is identified by the `@SpringBootTest` annotation, which provides many helpful features to test Spring Boot applications. Throughout the book, I'll present them in more detail. For now, it's enough to know that it loads a full Spring application context for the tests to be run in. There is only one test case, which is empty: it's used to verify that the Spring context is loaded correctly.

Open a Terminal window, navigate to the application root folder (`catalog-service`), and run the `test` Gradle task to execute the application's tests.

```
$ ./gradlew test
```

The task should be successful and the test green, meaning that the Spring application can start up without errors. What about the HTTP endpoint? Let's find out.

RUNNING THE APPLICATION

You are done implementing the application, so you can go ahead and run it. There are different ways to do that, and I will show you some of them later. For now, you can use the task provided by the Spring Boot Gradle plugin: `bootRun`.

From the same Terminal window where you launched the tests, run the command:

```
$ ./gradlew bootRun
```

The application should be up and running in a second and ready to accept requests. In figure 2.6, you can see the logs streamed during the startup phase.

```

  _   _ _   _   _ _   _   _ _   _   _ _   _   _ 
  \ \ / / \ \ / / \ \ / / \ \ / / \ \ / / \ \ / / \ 
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
  :: Spring Boot ::          (v2.7.0)

2022-06-04 16:44:45.953  INFO 39119 --- [           main] c.p.c.CatalogServiceApplication      : Starting CatalogServiceApplication using Java 17.0.3
2022-06-04 16:44:45.954  INFO 39119 --- [           main] c.p.c.CatalogServiceApplication      : No active profile set, falling back to 1 default profile: "default"
2022-06-04 16:44:46.260  INFO 39119 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-06-04 16:44:46.263  INFO 39119 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-06-04 16:44:46.263  INFO 39119 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.63]
2022-06-04 16:44:46.293  INFO 39119 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]    : Initializing Spring embedded WebApplicationContext
2022-06-04 16:44:46.293  INFO 39119 --- [           main] w.s.c.WebServerApplicationContext       : Root WebApplicationContext: initialization completed in 317 ms
2022-06-04 16:44:46.397  INFO 39119 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-06-04 16:44:46.403  INFO 39119 --- [           main] c.p.c.CatalogServiceApplication      : Started CatalogServiceApplication in 0.565 seconds (JVM running for 0.794)

```

Figure 2.6 The startup logs from the Catalog Service application.

From the logs in figure [2.6](#), you can notice how the startup phase is composed of two main steps:

- initialization and running of the embedded Tomcat server (which, by default, listens to port 8080 over HTTP);
- initialization and running of the Spring application context.

At this point, you can finally verify if your HTTP endpoint is working as intended. Open a browser window, navigate to `http://localhost:8080/` and be ready to be welcomed to the book catalog for the Polar Bookshop.

```
Welcome to the book catalog!
```

The development part of the Polar Bookshop application is done: you have a Catalog Service application welcoming users to the book catalog. Remember to terminate the `bootRun` process (`Ctrl+C`) to stop the application execution before moving on.

The next step is deploying the application to the cloud. To make it portable on any cloud infrastructure, you should containerize it first. Enter Docker.

2.3 Containerizing applications with Docker

The Catalog Service application is working. Before deploying it to the cloud, you should containerize it. Why? Containers provide isolation from the surrounding environment and are equipped with all the dependencies required by the application to run. In our case, most of the dependencies are managed by Gradle and packaged together with the application (JAR artifact). But the Java runtime is not. Containerizing the application means it will be self-contained and portable across any cloud environment. Without a container, you would have to install the Java runtime on any machine where you want to deploy the application. Using containers, you can manage all applications in a standard way, no matter the language or the framework used to implement them.

The **Open Container Initiative (OCI)**, a Linux Foundation project, defines industry standards for working with containers (<https://opencontainers.org>). In particular, the *OCI Image*

Specification defines how to build container images, the *OCI Runtime Specification* defines how to run those container images, and the *OCI Distribution Specification* defines how to distribute them. The tool we'll use to work with containers is Docker (<https://www.docker.com>), which is compliant with the OCI specifications.

Docker is an open-source platform that "provides the ability to package and run an application in a loosely isolated environment called a container" (<https://docs.docker.com>). *Docker* is also the name of the company behind this technology, which is a founding member of the OCI. The same term is used in several of their commercial products, too. Whenever I write *Docker*, unless specified otherwise, I refer to the open-source platform we'll use to build and run containers.

Before moving on, follow the instructions in section A.2 of Appendix A to install and configure Docker in your development environment.

2.3.1 Introducing Docker: Images and containers

When installing the Docker platform on your machine, you get the Docker Engine package characterized by a client/server architecture. The **Docker server** contains the *Docker daemon*, a background process responsible for creating and managing Docker objects like images, containers, volumes, and networks. The machine where the Docker server runs is called *Docker host*. Each machine where you want to run containers should be a Docker host, so it should have a Docker daemon running. The portability of containers is made possible by the daemon process itself. In the following, I might use *Docker server* and *Docker daemon* interchangeably.

The Docker daemon exposes an API you can use to send instructions, such as for running a container or creating a volume. The **Docker client** talks to the daemon through that API. The client is command-line based and can be used to interact with the Docker daemon either through scripting (for example, Docker Compose) or through the Docker CLI directly.

Besides the client and server components characterizing the Docker Engine, another essential element of the platform is a **container registry**, which has a similar function to a Maven repository. While Maven repositories are used to host and distribute Java libraries, container registries do the same for container images. They follow the OCI Distribution Specification. We distinguish between public and private registries. The Docker company provides a public registry called Docker Hub (<https://hub.docker.com>), configured by default with your local Docker installation and hosting images for many popular open-source projects, like Ubuntu, PostgreSQL, and OpenJDK.

Figure 2.7 shows how the Docker client, Docker server, and container registry interact.

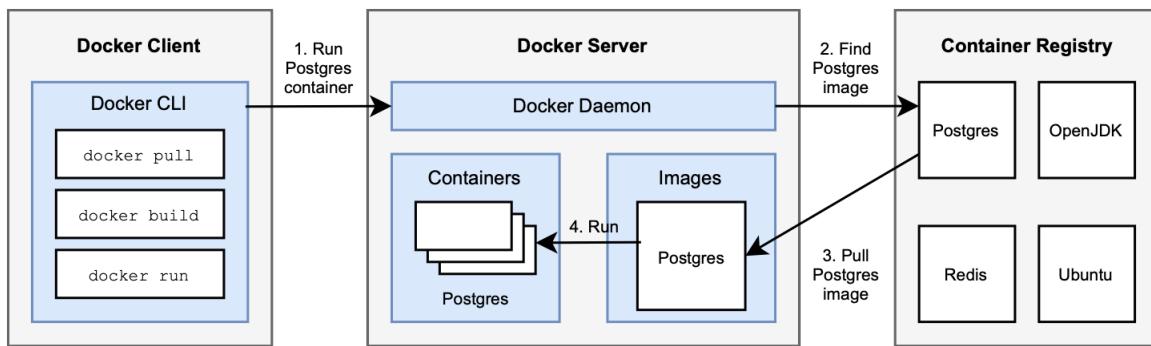


Figure 2.7 The Docker Engine has a client/server architecture and interacts with a registry.

The Docker daemon manages different objects. In this section, I'll focus on images and containers.

A **container image** (or, simply, *image*) is a lightweight executable package that includes everything needed to run the application inside. The *Docker image* format is the most used one for creating container images, later standardized by the Open Container Initiative project (*OCI Image Specification*). OCI images can be created from scratch by defining instructions in a *Dockerfile*, a text-based file containing all the steps to generate the image. Usually, images are created based on another image. For example, you might build an image based on the OpenJDK one, on top of which you can add a Java application. After creation, the image can be pushed to a container registry like Docker Hub. A base name and a tag identify each image, where the tag is usually the version number. For example, the Ubuntu image for version 22.04 is called `ubuntu:22.04`. A colon separates the base name and version.

A **container** is a runnable instance of a container image. You can manage the container lifecycle from the Docker CLI or Docker Compose: you can start, stop, update, and delete containers. Containers are defined by the image on which they are based and the configuration provided at startup time (for example, environment variables to customize the container). By default, containers are isolated from each other and the host machine, but you can make them expose services to the outside world through specific ports with a process called *port forwarding* or *port mapping*. Containers can have any name. If you don't specify one, the Docker server will assign a random one, like `bazinga_schrodinger`. To run OCI images as containers, you need Docker or any other container runtime compatible with the OCI specifications.

When you want to run a new container, you can use the Docker CLI to interact with the Docker daemon, which checks whether the specified image is already present in the local server. If not, it will find the image on a registry, download it, and then use it to run a container. The workflow, again, is shown in Figure 2.7.

SIDE BAR**Docker on macOS and Windows. How does it work?**

In the previous chapter, you learned that containers share the same operating system kernel and rely on Linux features such as namespaces and cgroups. We'll use Docker to run Spring Boot applications in Linux containers, but how is Docker supposed to work on a macOS or Windows machine?

When you install Docker on a Linux operating system, you get the full Docker Engine on your Linux host. On the contrary, if you install either Docker Desktop for Mac or Docker Desktop for Windows, only the Docker client is installed on your macOS/Windows host. Then, under the hood, a lightweight virtual machine is configured with Linux, and the Docker server component is installed on that machine. As a user, you will get almost the same experience as a Linux machine; you would hardly notice any difference. But in reality, whenever you use the Docker CLI to perform operations, you are actually interacting with a Docker server on a different machine (the virtual machine running Linux).

You can verify it yourself by starting Docker and running the `docker version` command. You'll notice that the Docker client is running on a `darwin/amd64` architecture (on macOS) or on `windows/amd64` (on Windows), while the Docker server is running on `linux/amd64`.

```
$ docker version
Client:
  Cloud integration: v1.0.24
  Version:           20.10.14
  API version:      1.41
  Go version:       go1.16.15
  Git commit:       a224086
  Built:            Thu Mar 24 01:49:20 2022
  OS/Arch:          darwin/amd64
  Context:          default
  Experimental:    true

Server:
  Engine:
    Version:         20.10.14
    API version:    1.41 (minimum version 1.12)
    Go version:     go1.16.15
    Git commit:     87a90dc
    Built:          Thu Mar 24 01:45:44 2022
    OS/Arch:        linux/amd64
    Experimental:   false
```

Docker provides support for other architectures other than AMD64. For example, if you use a MacBook with ARM-based chips like the Apple M1, you'll find your Docker client running on a `darwin/arm64` architecture and the Docker server on `linux/arm64`.

2.3.2 Running a Spring application as a container

Let's go back to Catalog Service and see how you can run it as a container. There are different ways to achieve that. Here, you will use the out-of-the-box integration of Spring Boot with Cloud Native Buildpacks (<https://buildpacks.io>), a project initiated by Heroku and Pivotal and now hosted by the CNCF. It provides a high-level abstraction for automatically transforming application source code into container images instead of using a low-level `Dockerfile`.

The Paketo Buildpacks (an implementation of the Cloud Native Buildpacks specification) are fully integrated with the Spring Boot Plugin, both for Gradle and Maven. It means that you can containerize your Spring Boot applications without downloading any additional tool, providing any extra dependency, or writing a `Dockerfile`.

Chapter 6 will describe how the Cloud Native Buildpacks project works and how to configure it for containerizing your Spring Boot applications. For now, I'll give you a little preview of its features.

Open a Terminal window, navigate to the root folder of your Catalog Service project (`catalog-service`), and run the `bootBuildImage` Gradle task. That's all you need to do to package your application as a container image, using Cloud Native Buildpacks under the hood.

```
$ ./gradlew bootBuildImage
```

WARNING At the moment of writing, the Paketo project is working on adding support for ARM64 images. You can follow the feature progress on the [Paketo Buildpacks project on GitHub](#): <https://github.com/paketo-buildpacks/stacks/issues/51>. Until then, you can still use Buildpacks for building containers and run them via Docker Desktop on Apple Silicon computers. However, the build process and the application startup phase will be slower than usual.

The first time you run the task, it will take a minute to download the packages used by Buildpacks to create the container image. The second time, it will take only a few seconds. The resulting image will be named `catalog-service:0.0.1-SNAPSHOT` by default (`<project_name>:<version>`). You can run the following command to get the details of the newly created image.

```
$ docker images catalog-service:0.0.1-SNAPSHOT
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
catalog-service  0.0.1-SNAPSHOT  f0247a113eff  42 years ago  275MB
```

NOTE

You might have noticed from the previous command's output that it looks like the image was created 42 years ago. That's a convention used by Cloud Native Buildpacks to achieve reproducible builds. Subsequent execution of the build command should give the same output if nothing were changed in input. The actual creation timestamp would make that impossible, so Cloud Native Buildpacks uses a conventional timestamp (January 1, 1980).

The last thing left is running the image and verifying if the containerized application is working correctly. Open a Terminal window and run the following command.

```
$ docker run --rm --name catalog-service -p 8080:8080 \
catalog-service:0.0.1-SNAPSHOT
```

WARNING

If you're running the container on an Apple Silicon computer, the previous command might return a message like **WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and no specific platform was requested.** In that case, you need to include this additional argument to the previous command (before the image name) until support for ARM64 is added to Paketo Buildpacks: `--platform linux/amd64`.

You can refer to figure 2.8 for a description of the command.

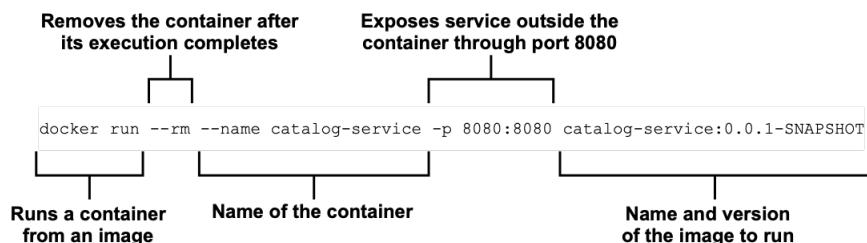


Figure 2.8 The Docker command to start a containerized application from an image.

Open a browser window, navigate to <http://localhost:8080/> and verify that you still get the same greeting you got earlier.

```
Welcome to the book catalog!
```

When you are done, stop the container with `Ctrl+C`.

In chapter 6, you'll learn more about how Docker works, how container images are built from a Spring Boot application, and how to use a container registry. I'll also show you how to use Docker Compose to manage containers instead of the Docker CLI.

2.4 Managing containers with Kubernetes

So far, you have built a web application with Spring Boot (Catalog Service), containerized it with Cloud Native Buildpacks, and run it with Docker. To complete the pilot project for Polar Bookshop, you must perform one last step: deploying the application to a cloud environment. You will use Kubernetes, which has become the de-facto standard for container orchestration. I will cover the subject extensively in later chapters, but I want you to have a first taste of how Kubernetes works and how you can use it to deploy a web application.

Kubernetes (usually shortened as *K8s*) is an open-source system for automating the deployment, scaling, and management of containerized applications (<https://kubernetes.io>). When working with containers in Docker, your deployment target is a machine. In the example from the previous section, it was your computer. In other scenarios, it might be a virtual machine (VM). In any case, it's about deploying containers to a specific machine. What if you need multiple machines? When it comes to deploying containers without downtime, scaling them by leveraging the cloud elasticity, or connecting them across different hosts, then you need something more than a container engine.

The deployment target moves from a specific machine to a cluster of machines. Kubernetes, among other things, manages a cluster of machines for you. I covered this distinction in the previous chapter in the context of topologies. Figure 2.9 reminds you of the different deployment targets in a container topology and an orchestration topology.

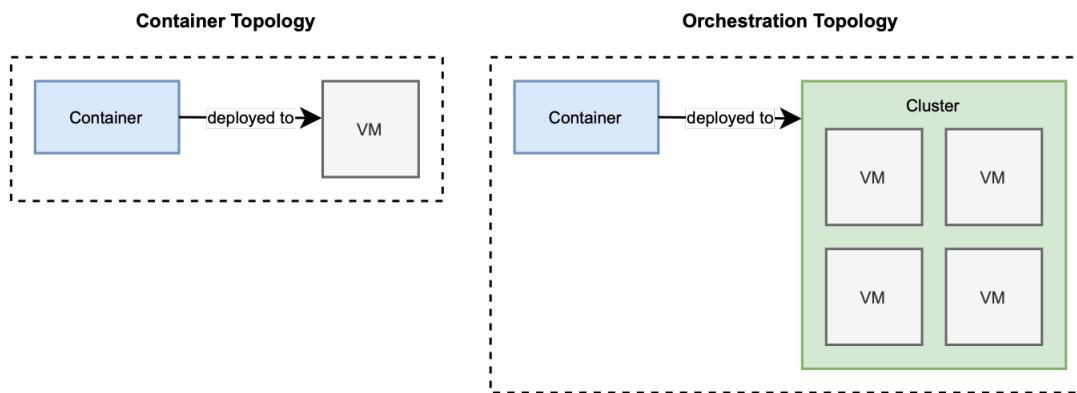


Figure 2.9 The deployment target of containers is a machine, whereas for orchestrators it's a cluster.

Before moving on, follow the instructions in section A.3 of Appendix A to install `minikube` and set up a Kubernetes cluster in your local development environment. Once you are done with the installation procedure, you can start a local Kubernetes cluster with the following command:

```
$ minikube start
```

2.4.1 Introducing Kubernetes: Deployments, Pods, and Services

Kubernetes is an open-source container orchestrator hosted by the CNCF. In just a few years, it has become the most used solution for container orchestration, and all the major cloud providers have offerings for Kubernetes as a service. Kubernetes can run on a desktop, on a data center on-premises, in the cloud, and even on an IoT device.

When using a container topology, you need a machine with a container runtime. With Kubernetes, you switch to an orchestration topology, meaning that you need a cluster. A Kubernetes **cluster** is a set of worker machines (*nodes*) that run containerized applications. Every cluster has at least one worker node. With `minikube`, you can easily create a single-node cluster on your local machine. In production, you'll use a cluster managed by a cloud provider.

A Kubernetes cluster comprises machines called **worker nodes** on which your containerized applications are deployed. They provide capacity such as CPU, memory, network, and storage so that the containers can run and connect to a network.

The **control plane** is the container orchestration layer that manages the worker nodes. It exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. It offers all the essential elements implementing the typical features of an orchestrator, like cluster management, scheduling, and health monitoring.

NOTE

In the context of container orchestration, scheduling means matching a container instance with a node where to run it. The matching happens based on a set of criteria, including the availability of enough computational resources to run the container on the node.

You can interact with Kubernetes through a CLI client, `kubectl`, which communicates with the control plane to perform some operations on the worker nodes. A client doesn't interact with the worker nodes directly. Figure 2.10 shows the high-level components of the Kubernetes architecture.

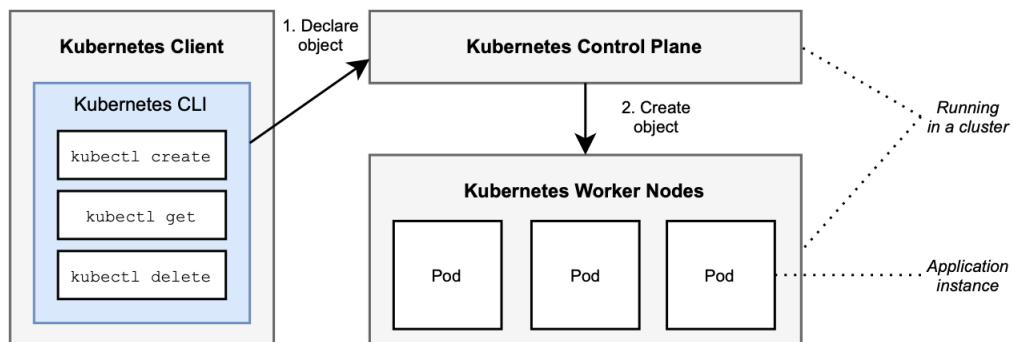


Figure 2.10 Kubernetes main components are the API, the control plane, and the worker nodes.

Kubernetes can manage many different *objects*, either built-in or custom. In this section, you will use Pods, Deployments, and Services.

- **Pod.** The smallest deployable unit. It can include one or more containers. Usually, a Pod contains only one of your applications. It might also include extra containers supporting the primary application (for example, providing additional functionality like logging or administrative tasks to run during the initialization step). Kubernetes manages Pods rather than containers directly.
- **Deployment.** It informs Kubernetes about the desired deployment state for your application. For each instance, it creates a Pod and keeps it healthy.
- **Service.** A Deployment (a set of Pods) can be exposed to other nodes in the cluster or outside by defining a Service that also takes care of balancing the load between Pod instances.

NOTE

Throughout the book, I will spell Kubernetes resources with capital letters to distinguish them from when the same terms are used with different meanings. For example, I use service when referring to an application, whereas I write Service when I mean the Kubernetes object.

When you want to run a new application, you can define a **resource manifest**, a file that describes the desired state for the application, for example, that it should be replicated five times and exposed to the outside world through port 8080. Resource manifests are usually written using YAML. Then, you use the `kubectl` client to ask the control plane to create the resources described by the manifest. In the end, the control plane processes the request using its internal components and finally creates the resources in the worker nodes. The control plane would still rely on a container registry to fetch the image defined in the resource manifest. The workflow, again, is shown in Figure 2.10.

2.4.2 Running a Spring application on Kubernetes

Let's go back to the Polar Bookshop project. In the previous section, you containerized the Catalog Service application. Now, it's time to deploy it to a cluster using Kubernetes. You already have a cluster up and running in your local environment. What you need is a resource manifest.

The standard way to interact with Kubernetes is through declarative instructions that you can define in a YAML or JSON file. I will show you how to write resource manifests in chapter 7. Until then, you will use the Kubernetes CLI as you did with Docker earlier.

First, you need to tell Kubernetes to deploy Catalog Service from a container image. You built one previously (`catalog-service:0.0.1-SNAPSHOT`). By default, `minikube` uses the Docker Hub registry to pull images, and it doesn't have access to your local ones. Therefore, it will not find the image you built for the Catalog Service application, but don't worry: you can manually import it into your local cluster.

Open a Terminal window, and run the following command.

```
$ minikube image load catalog-service:0.0.1-SNAPSHOT
```

The deployment unit will be a Pod, but you will not manage Pods directly. Instead, you want to let Kubernetes handle that. Pods are application instances, and as such, they are ephemeral. To achieve the cloud native goals, you want the platform to take care of instantiating Pods so that if one goes down, it can be replaced by another one. What you need is a *Deployment* resource that will make Kubernetes create application instances as *Pod* resources.

From a Terminal window, run the following command.

```
$ kubectl create deployment catalog-service \
--image=catalog-service:0.0.1-SNAPSHOT
```

You can refer to figure [2.11](#) for a description of the command.

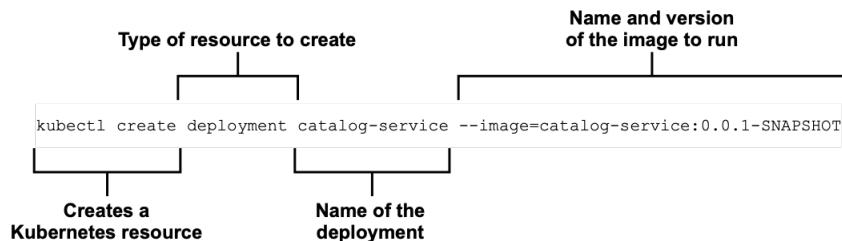


Figure 2.11 The Kubernetes command to create a Deployment from a container image. Kubernetes will take care of creating Pods for the application.

You can verify the creation of the Deployment object as follows.

```
$ kubectl get deployment
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
catalog-service   1/1      1           1          7s
```

Behind the scenes, Kubernetes created a Pod for the application defined in the Deployment resource. You can verify the creation of the Pod object as follows.

```
$ kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
catalog-service-5b9c996675-nzbhd   1/1     Running   0          21s
```

TIP You can check the application logs by running `kubectl logs deployment/catalog-service`.

By default, applications running in Kubernetes are not accessible. Let's fix that. First, you can expose Catalog Service to the cluster through a Service resource by running the following command.

```
$ kubectl expose deployment catalog-service \
--name=catalog-service \
--port=8080
```

You can refer to figure 2.12 for a description of the command.

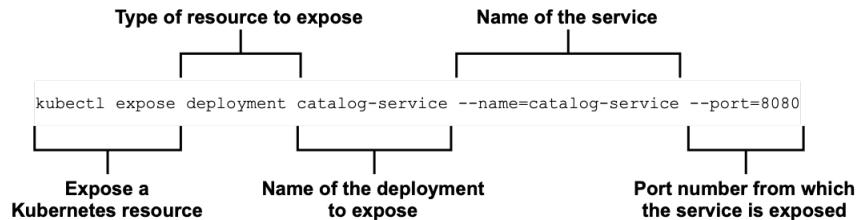


Figure 2.12 The Kubernetes command to expose a Deployment as a Service. The Catalog Service application will be exposed to the cluster network through port 8080.

The Service object exposes the application to other components inside the cluster. You can verify it's been created correctly with the following command.

```
$ kubectl get service catalog-service
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
catalog-service   ClusterIP   10.96.141.159   <none>        8080/TCP   7s
```

You can then forward the traffic from a local port on your computer (for example, 8000) to the port exposed by the Service inside the cluster (8080). Remember the port mapping in Docker? It works in a similar way. The output of the command will tell you if the port forwarding is configured correctly.

```
$ kubectl port-forward service/catalog-service 8000:8080
Forwarding from 127.0.0.1:8000 -> 8080
Forwarding from [::1]:8000 -> 8080
```

You can refer to figure 2.13 for a description of the command.

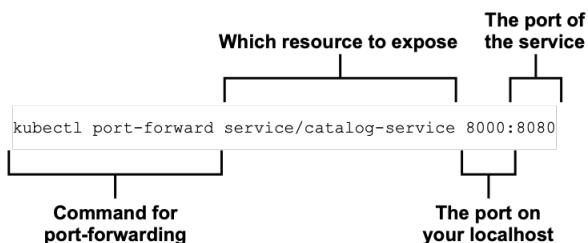


Figure 2.13 The Kubernetes command for forwarding a port from your localhost to a Service inside the cluster. The Catalog Service application will be exposed to your localhost through port 8000.

Now, whenever you access port 8000 on your localhost, you will be forwarded to the Service inside the Kubernetes cluster responsible for exposing the Catalog Service application. Open a browser window, navigate to <http://localhost:8000/> (make sure you use 8000 and not 8080) and verify that you still get the same greeting as before.

Welcome to the book catalog!

Well done! You started with a Java application implemented using Spring Boot. Then, you containerized it with Cloud Native Buildpacks and ran it on Docker. Finally, you used Kubernetes to get the application deployed to a cluster. Sure, it was a local cluster, but it could have been a remote one in the cloud. The beauty of this process is that it works in the same way independently of the environment. You could use the same exact approach to deploy Catalog Service to a cluster in a public cloud infrastructure. Isn't it great?

In chapter 7, you will work more with Kubernetes. For now, terminate the port-forwarding process with `Ctrl+C`, delete the Service with `kubectl delete service catalog-service`, and the Deployment with `kubectl delete deployment catalog-service`. Finally, you can stop the Kubernetes cluster with `minikube stop`.

2.5 Polar Bookshop - A cloud native application

My goal for this book is to provide real-world code examples as much as possible. Now that you have explored some of the key concepts and you have tried your hand at building, containerizing, and deploying Spring applications, let's take on a slightly more complicated project: an online bookshop. I'll guide you through developing a cloud native system until its deployment to an actual Kubernetes cluster on a public cloud.

For each concept covered in the following chapters, I'll show you how to apply it to a real-world cloud native scenario for a complete hands-on learning experience. Remember that all the code used for the project is available on the GitHub repository accompanying the book.

This section will define the requirements for the cloud native project we'll build and describe its architecture. Then, I'll go through the main technologies and patterns we'll use to implement it.

At the end of the book, you'll have built a complete cloud native system based on Spring applications, containerized it, and deployed it with Kubernetes in a public cloud.

2.5.1 Understanding the requirements of the system

Polar Bookshop is a specialized bookshop with the mission to spread knowledge and information about the North Pole and the Arctic, where it's located: history, geography, animals, and so on. The organization managing the library, *Polarsophia*, has decided to start selling its books online to spread them worldwide, but that is only the beginning. The project is very ambitious, and the vision includes a suite of software products to fulfill the Polarsophia mission. After a successful pilot project, the organization decided to embark on a cloud native journey.

Throughout the book, you'll build the core part of a system with endless possibilities in terms of functionality and integrations. The management plans to deliver new features in short iterations,

reduce the time to market, and get early feedback from users. The goal is to bring the bookshop close to everyone, everywhere, so the application should be highly scalable. With a worldwide audience and highly scaled, such a system needs to be highly available, so resilience is essential. Polarsophia is a small organization, and they need to optimize costs, especially those regarding infrastructure. They can't afford to build their own data center, so they have decided to rent IT hardware from a third party.

By now, you can probably recognize some of the reasons why companies move to the cloud. That's what will do for the Polar Bookshop application. Of course, it will be a cloud native application.

Books will be available on sale through the application. When a customer purchases a book, they can check the status of their order. Two categories of people will use the Polar Bookshop application:

- customers can browse books in the catalog, buy one, and check their orders;
- employees can manage books, update the existing ones, and add new items to the catalog.

Figure 2.14 describes the architecture of the Polar Bookshop cloud native system. As you can see, it's made up of several services. Some will implement the business logic of the system to provide the functionality explained above. Other services will implement shared concerns like centralized configuration. For better clarity, the diagram doesn't show the services responsible for the security and observability concerns. You'll familiarize yourself with them later in the book.

In the following chapters, I'll guide you through the diagram in figure 2.14 in more detail, adding more information on specific services and adopting different points of view to visualize the system in its deployment stage. Now, let's move on and go through the patterns and technologies used in the project.

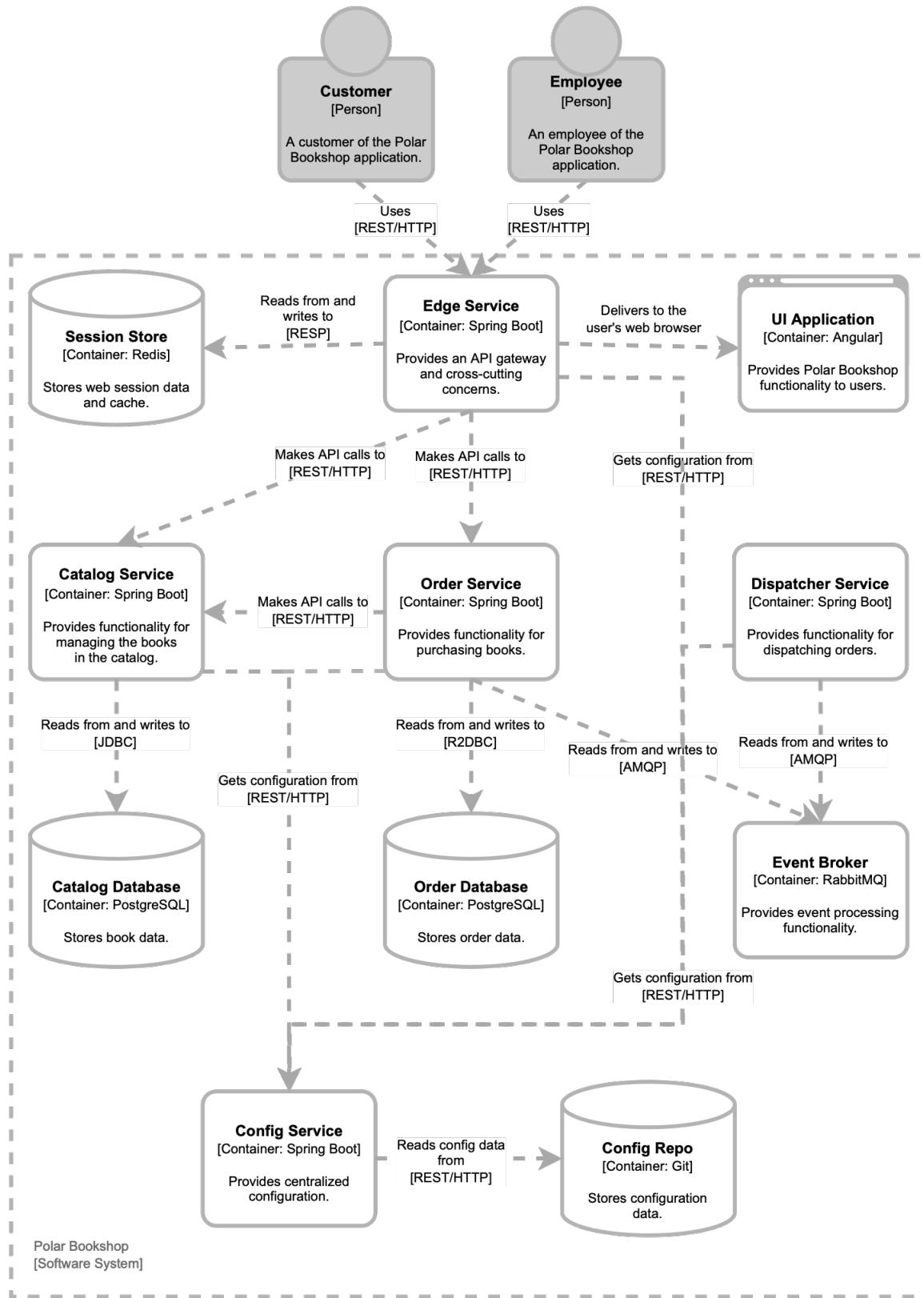


Figure 2.14 Architecture of Polar Bookshop. The cloud native system comprises application and data services with different responsibilities. For clarity, security and observability services are not shown.

2.5.2 Exploring patterns and technologies used in the project

I'll show you how to apply the covered technology or pattern to the Polar Bookshop project whenever I introduce a new topic in the book. In this section, I'll provide you with an overview of the main concerns we are going to address and which technologies and patterns we'll use to accomplish them.

WEB AND INTERACTIONS

Polar Bookshop comprises several services that will have to communicate with each other to provide their functionality. You'll build RESTful services interacting synchronously over HTTP, both in a blocking way (using traditional servlets) and non-blocking (using reactive programming). **Spring MVC** and **Spring WebFlux** (on top of Project Reactor) will be your main tools to accomplish such a result.

When building cloud native applications, you should remember to design loosely coupled services and consider how to keep data consistent in a distributed system context. Synchronous communications can generate problems when more services are involved in accomplishing a function. That's why event-driven programming is getting very popular in the cloud: it allows you to overcome the issues of synchronous communications.

I'll show you how to use events and messaging systems to decouple services and ensure data consistency. You'll use **Spring Cloud Stream** to implement data streams between services, and **Spring Cloud Function** to define message handlers as functions. The latter approach could naturally evolve in serverless applications deployed to platforms like Azure Functions, AWS Lambda, or Knative.

DATA

Data are a crucial part of software systems. In Polar Bookshop, you'll use the relational database **PostgreSQL** to store the data processed by the application permanently. I'll show you how to integrate an application with a data source using **Spring Data JDBC** (imperative) and **Spring Data R2DBC** (reactive). You'll then see how to evolve a data source and manage schema migrations with **Flyway**.

Cloud native applications should be stateless, but the state needs to be stored somewhere. In Polar Bookshop, you'll use **Redis** to externalize the session storage to a data store and keep applications stateless and scalable. Spring Session makes it easy to implement clustered user sessions. In particular, I'll show you how to use **Spring Session Data Redis** to integrate the application session management with Redis.

Besides persisted data and session data, you'll also deal with messages to implement event-driven programming patterns. **Spring AMQP** and **RabbitMQ** will be the technologies you'll use to do so.

Locally, you'll run these data services in Docker containers. In production, you'll rely on the managed services offered by a cloud provider like DigitalOcean or Azure, which take care of critical concerns like high availability, clustering, storage, and data replication.

CONFIGURATION

Throughout the book, I'll show you how to configure the services in Polar Bookshop in different ways. I'll start by exploring the options offered by **Spring Boot** properties and profiles, and when to use them. You'll then learn how to apply external configuration by using environment variables when running a Spring Application as a JAR and as a container. Then, you'll see how to centralize the configuration management through a configuration server with **Spring Cloud Config**. Finally, I'll teach you how to use ConfigMaps and Secrets in **Kubernetes**.

ROUTING

Polar Bookshop, being a distributed system, will need some routing configuration. **Kubernetes** has a built-in service discovery feature to help you decouple services from their physical addresses and hostnames. Cloud native applications are scalable, so any interaction between them should take that into account: which instance should you call? Once again, Kubernetes offers you a load balancing feature natively, so you don't need to implement anything in your application.

With **Spring Cloud Gateway**, I'll guide you through implementing a service that will act as an API gateway to shield the outside from any internal API change. It will also be an edge service that you'll use to address cross-cutting concerns like security and resilience in one place. Such a service will be the entry-point to Polar Bookshop and will have to be highly available, well-performant, and fault-tolerant.

OBSERVABILITY

The services in the Polar Bookshop system should be observable to be defined as cloud native. I'll show you how to use **Spring Boot Actuator** to set up health and info endpoints and expose metrics with Micrometer to be fetched and processed by **Prometheus**. You'll then use **Grafana** to visualize the most critical metrics in informative dashboards.

Requests can be handled by more than one service, so you'll need a distributed tracing functionality to follow the request flow from one service to another. You'll set that up with **OpenTelemetry**. Then, **Grafana Tempo** will fetch, process, and visualize the traces to give you a complete picture of how the system accomplishes its functions.

Finally, we need a logging strategy in place. We should handle logs as event streams, so you'll make your Spring applications stream log events to the standard output without considering how they are processed or stored. **Fluent Bit** will take care of collecting logs from all services, **Loki** will store and process them, and **Grafana** will let you browse them.

RESILIENCE

Cloud native applications should be resilient. For the Polar Bookshop project, I'll show you various techniques to make an application resilient using **Project Reactor**, **Spring Cloud Circuit Breaker**, and **Resilience4J** to implement circuit breakers, retries, timeouts, and other patterns.

SECURITY

Security is a vast subject that I won't be able to cover deeply in this book. Still, I recommend exploring the topic since it's one of the most critical software concerns nowadays. It's a pervasive concern that should be addressed continuously from the very beginning of the project and never let go.

For Polar Bookshop, I'll show you how to add authentication and authorization functionality to cloud native applications. You'll see how to secure communications between services, and between users and applications. **OAuth 2.1** and **OpenID Connect** will be the standards you rely on to implement such functionality. **Spring Security** supports those standards and integrates seamlessly with external services to provide authentication and authorization. You'll use **Keycloak** for identity and access control management.

Also, I'll introduce the concepts of secrets management and encryption. I won't be able to go too deep into those topics, but I'll show you how to manage secrets to configure Spring Boot applications.

TESTING

Automated testing is paramount for the success of a cloud native application. A few levels of automated tests will cover the Polar Bookshop application. I'll show you how to use **JUnit5** to write unit tests. **Spring Boot** adds many convenient utilities to improve integration testing, and you will use them to ensure the quality of your services. You'll write tests for the various features used in Polar Bookshop, including REST endpoints, messaging streams, data integrations, and security.

Keeping parity across environments is essential to ensure the quality of our applications. That is especially true when it comes to backing services. In production, you will use services such as PostgreSQL and Redis. During testing, you should use similar services rather than mocks or test-specific tools like the H2 in-memory database. The **Testcontainers** framework will help you use real services as containers in your automated tests.

BUILD AND DEPLOYMENT

Polar Bookshop's primary services are using Spring. You'll see how to package a Spring application, run it as a JAR file, containerize it with **Cloud Native Buildpacks**, run it with **Docker**, and finally, you'll deploy containers with **Kubernetes**. You'll also see how to compile Spring applications to native images using **Spring Native** and **GraalVM**, and use them in serverless architectures taking advantage of their instant startup time, instant peak performance, reduced memory consumption, and reduced image size. Then, you'll deploy them on a serverless platform built on top of Kubernetes with **Knative**.

I'll show you how to automate the build stage by setting up a deployment pipeline with **GitHub Actions**. The pipeline will build the application at every commit, run the tests, and package it ready for deployment. Such automation will support a continuous delivery culture to bring value to customers quickly and reliably. In the end, you'll also automate the deployment of Polar Bookshop to the production Kubernetes cluster using **GitOps** practices and **ArgoCD**.

UI

This book is focused on backend technologies, so I'll not teach you any frontend subject. Of course, your application will need a frontend for the users to interact with it. For Polar Bookshop, you'll rely on a client application using the Angular framework. I'll not show you the UI application code in this book because it's out of scope, but I've included it in the code repository accompanying the book.

2.6 Summary

- The 15-Factor methodology identifies development principles to build applications that offer maximum portability across execution environments, are suitable to be deployed on cloud platforms, can be scaled, guarantee environment parity between development and production, and enable continuous delivery.
- Spring is a suite of projects which provide all the most common functionality to build modern applications in Java.
- Spring Framework provides an application context in which beans and properties are managed throughout the entire lifecycle.
- Spring Boot lays the foundation for cloud native development by speeding up building a production-ready application, including embedded servers, auto-configuration, monitoring, and containerization features.
- Container images are lightweight executable packages that include everything needed to run the applications inside.
- Docker is an OCI-compliant platform for building and running containers.
- A Spring Boot application can be packaged as a container image with Cloud Native Buildpacks, a CNCF project that specifies how to convert application source code into production-ready container images.
- When dealing with several containers, usually the case in a cloud native system, you need a way to manage such a complex system. Kubernetes provides functionality to orchestrate, schedule, and manage containers.
- Kubernetes Pods are the minimum deployment units.
- Kubernetes Deployments describe how to create application instances as Pods, starting from a container image.
- Kubernetes Services allow you to expose application endpoints outside the cluster.

3

Getting started with cloud native development

This chapter covers

- Bootstrapping a cloud native project
- Working with embedded servers and Tomcat
- Building a RESTful application with Spring MVC
- Testing a RESTful application with Spring Test
- Automating build and tests with GitHub Actions

The cloud native landscape is so broad that getting started can be overwhelming. In Part 1, you got a theoretical introduction to cloud native applications and the processes supporting them, and you had a first hands-on experience building a minimal Spring Boot application and deploying it to Kubernetes as a container. All of that will help you understand better and place correctly in the overall cloud native picture the topics I'll be covering in the rest of the book.

The cloud opened up endless possibilities in what we can achieve with many different types of applications. In this chapter, I chose to start with one of the most common types: a web application exposing its functionality over HTTP through a REST API. I'll guide you through the development process you'll be following in all the subsequent chapters, addressing the significant differences between traditional and cloud native web applications, consolidating some necessary aspects of Spring Boot and Spring MVC, and highlighting essential testing and production considerations. I'll also explain some of the guidelines recommended by the 15-Factor methodology, including dependency management, concurrency, and API first.

Along the way, you'll implement the Catalog Service application you initialized in the previous chapter. It will be responsible for managing the catalog of books in the Polar Bookshop system.

NOTE

The source code for the examples in this chapter is available in the Chapter03/03-begin and Chapter03/03-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

3.1 Bootstrapping a cloud native project

Starting a new development project is always exciting. The 15-Factor methodology contains some practical guidelines when bootstrapping a cloud native application.

- **One codebase, one application.** Cloud native applications should consist of a single codebase tracked in a version control system.
- **Dependency management.** Cloud native applications should use a tool for managing dependencies explicitly and shouldn't rely on implicit dependencies from the environment where they are deployed.

In this section, I'll provide a few more details about those two principles and how to apply them to Catalog Service, the first cloud native application in the Polar Bookshop system you'll be developing throughout the book.

3.1.1 One codebase, one application

A cloud native application should consist of a single codebase tracked in a version control system like Git. Each codebase must produce immutable artifacts, called *builds*, that can be deployed to multiple environments. Figure 3.1 shows the relationship between codebase, build and deployments.

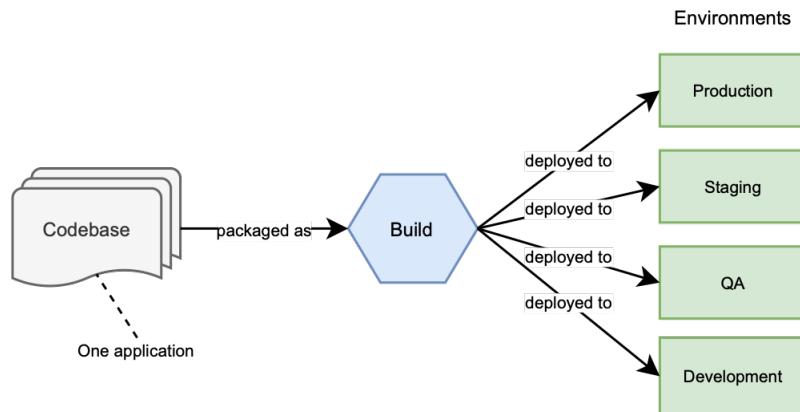


Figure 3.1 Each application has its codebase, from which immutable builds are produced and deployed to any environment without changes to the code.

As you'll see in the next chapter, anything environment-specific like configuration must be outside of the application codebase. In case of code needed by more than one application, you

should either turn it into an independent service itself or into a library that you can import into the project as a dependency. You should carefully evaluate the latter option to prevent the system from becoming a *distributed monolith*.

NOTE

Thinking about how the code is organized into codebases and repositories can help you focus more on the system architecture and identify those parts that might actually stand on their own as independent services. If done correctly, the codebase organization can favor modularity and loose coupling.

According to the 15-Factor methodology, each codebase should be mapped to an application, but nothing is said about repositories. You can decide to have each codebase tracked in a separate repository or the same one. Both options are used in the cloud native business. Throughout the book, you'll build several applications for which I recommend you track each codebase in its own Git repository because it'll improve maintainability and deployability.

In the previous chapter, you initialized the first application in the Polar Bookshop system, Catalog Service, and placed it in a `catalog-service` Git repository. I recommend you use GitHub for storing your repositories because later, we're going to use GitHub Actions as the workflow engine to define deployment pipelines in support of continuous delivery.

3.1.2 Dependency management with Gradle and Maven

How you manage dependencies for your applications is relevant because it affects their reliability and portability. In the Java ecosystem, the two most used tools for dependency management are Gradle and Maven. Both of them provide the functionality to declare dependencies in a manifest and download them from a central repository. The goal of listing all the dependencies your project needs is to ensure that you do not depend on any implicit library leaking from the surrounding environment.

NOTE

Apart from dependency management, those tools offer additional features to build, test, and configure a Java project, becoming fundamental for application development. All the examples in the book will be using Gradle, but feel free to use Maven instead.

Even though you have a dependency manifest in place, you still need to provide the dependency manager itself. Both Gradle and Maven offer a feature to run the tool from a *wrapper script* named `gradlew` or `mvnw` that you can include in your codebase. For example, rather than running a Gradle command like `gradle build` (that assumes you have Gradle installed on your machine), you run `./gradlew build`. The script invokes the specific version of the build tool defined in the project. If not present yet, the wrapper script will download it first and then run the command. Using the wrapper, you can ensure that all team members and automated tools

building the project use the same Gradle or Maven version. When generating a new project from Spring Initializr, you also get a wrapper script ready to use, so you don't need to download or configure anything.

NOTE

No matter what, you usually have at least one external dependency: the runtime. In our case, that's the Java Runtime Environment (JRE). If you package your application as a container image, the Java runtime would be included in the image itself, granting you more control over it. On the other hand, the final application artifact would depend on the container runtime required to run the image. You'll learn more about the containerization process in chapter 6.

Now, on to the code. The Polar Bookshop system needs a Catalog Service application responsible for managing the books available in the catalog. In the previous chapter, we have already initialized the project. The architecture of the system is shown again in figure 3.2.

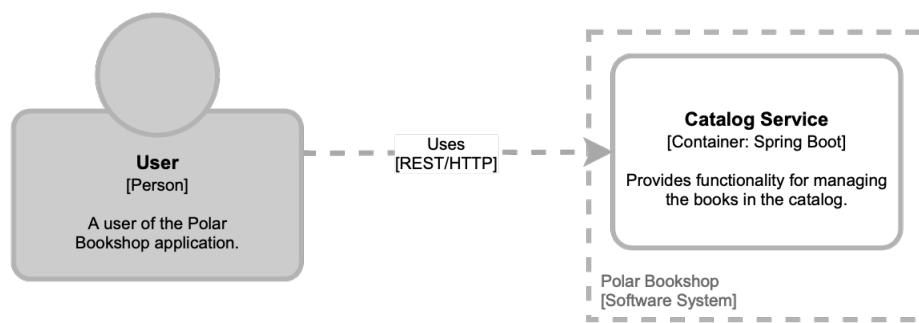


Figure 3.2 The architecture of the Polar Bookshop system, currently comprising of only one application service.

All the dependencies required by the application are listed in the autogenerated `build.gradle` file (`catalog-service/build.gradle`).

```

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

```

- **Spring Web** (`org.springframework.boot:spring-boot-starter-web`) provides the necessary libraries for building web applications with Spring MVC and includes Tomcat as the default embedded server.
- **Spring Boot Test** (`org.springframework.boot:spring-boot-starter-test`) provides several libraries and utilities to test applications, including Spring Test, JUnit, AssertJ, and Mockito. It's automatically included in every Spring Boot project.

One of the great features of Spring Boot is the way it handles dependency management. Starter dependencies like `spring-boot-starter-web` relieve you from managing many more dependencies and verifying that the particular versions you import are compatible with each

other. It's yet another Spring Boot feature to get you started in a simple and productive way.

In the next section, you'll learn more about how the server embedded in Spring Boot works and how to configure it.

3.2 Working with embedded servers

With Spring Boot, you can build different types of applications (e.g., web, event-driven, serverless, batch, task) characterized by various use cases and patterns. In a cloud native context, they all share some common aspects:

- they are entirely self-contained, with no external dependency other than the runtime;
- they are packaged as standard, executable artifacts.

Consider a web application. Traditionally, you would package it as a WAR or EAR file (archive formats used for packaging Java applications) and deploy it to a web server like Tomcat or an application server like Wildfly. The external dependency on a server would limit the portability and evolution of the application itself and increase the maintenance costs.

In this section, you'll see how to solve those problems in a cloud native web application with Spring Boot, Spring MVC, and an embedded server, but similar principles apply to other types of applications as well. You'll learn the differences between traditional and cloud native applications, how an embedded server like Tomcat works, and how you can configure it. I'll also elaborate on a few guidelines from the 15-Factor methodology regarding servers, port binding, and concurrency.

- **Port binding.** Unlike traditional applications depending on an external server available in the execution environment, cloud native applications are self-contained and export their services by binding to a port that can be configured depending on the environment.
- **Concurrency.** In JVM applications, we handle concurrency through multiple threads available as thread pools. When the concurrency limit is hit, we favor horizontal over vertical scaling. Instead of adding more computational resources to your application, we prefer deploying more instances and distributing the workload among them.

Following those principles, we'll continue working on Catalog Service to ensure it's self-contained and packaged as an executable JAR.

SIDE BAR**Servers! Servers everywhere!**

So far, I have used the terms **application servers** and **web servers**. Later on, I will also mention **Servlet containers**. What's the difference?

- **Web server.** A server that handles HTTP requests coming from a client and replies with HTTP responses. Example: Apache HTTPD.
- **Servlet container.** A component, part of a web server, that provides an execution context for web applications using the Java Servlet API (like Spring MVC apps). Example: Tomcat (Catalina).
- **Application server.** A server that provides a complete execution environment (like Jakarta EE) for different types of applications and supports several protocols. Example: Wildfly.

3.2.1 Executable JARs and embedded servers

One of the differences between a traditional approach and a cloud native one is how you package and deploy applications. Traditionally, we used to have application servers or standalone web servers. They were expensive to set up and maintain in production, so they were used to deploy several applications packaged as EAR or WAR artifacts for the sake of efficiency. Such a scenario would create coupling among the applications. If any of them wanted to change something at the server level, the change would have to be coordinated with other teams and applied to all the applications, limiting agility and application evolution. Besides that, the deployment of an application would depend on a server being available on the machine, limiting the application portability across different environments.

When going cloud native, things are different. Cloud native applications should be self-contained and not dependent on a server being available in the execution environment. The solution is to bring the necessary server capabilities into the application itself. Spring Boot offers a built-in embedded server functionality that helps you remove the external dependency and make the application standalone. Spring Boot comes bundled with a pre-configured Tomcat server, but it's possible to replace it with Undertow, Jetty, or Netty.

After solving the server dependency problem, we need to change how we package the application accordingly. In the JVM ecosystem, cloud native applications are packaged as JAR artifacts. Since they are self-contained, they can run as standalone Java applications with no external dependency apart from the JVM. Spring Boot is flexible enough to allow both JAR and WAR types of packaging. Still, for cloud native applications, you want to use self-contained JARs, also called *fat-JARs* or *uber-JARs* since they contain the application itself, the dependencies, and the embedded server.

Figure [3.3](#) compares the traditional and the cloud native ways of packaging and running web

applications.

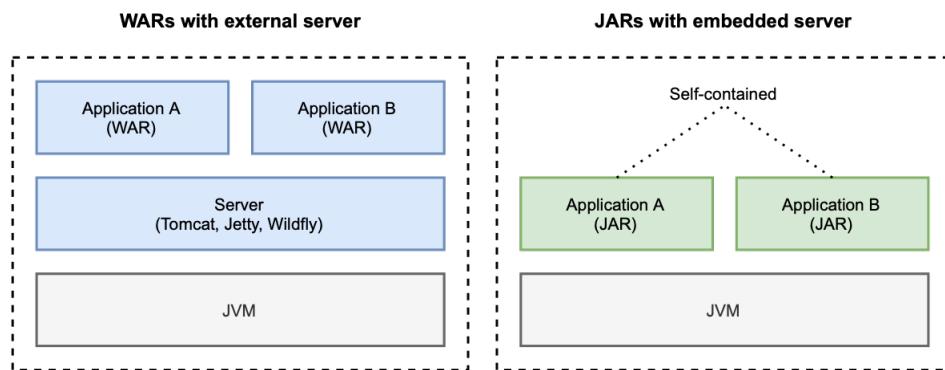


Figure 3.3 Traditionally, applications are packaged as WARs and require a server to be available in the execution environment to run. Cloud native applications are packaged as JARs, are self-contained, and use an embedded server.

The embedded servers used for cloud native applications usually comprise a web server component and an execution context to make a Java web application interact with the web server. For example, Tomcat contains a web server component (Coyote) and an execution context based on the Java Servlet API, usually called Servlet container (Catalina). In the following, I will use *web server* and *Servlet container* interchangeably. On the other hand, application servers are not recommended for cloud native applications.

In the previous chapter, when generating the Catalog Service project, we chose the JAR packaging option. We then ran the application using the `bootRun` Gradle task. That is a convenient way to build the project and run it as a standalone application during development. Now that you know more about the embedded server and JAR packaging, I'll show you another way.

First, let's package the application as a JAR file. Open a Terminal window, navigate to the root folder of the Catalog Service project (`catalog-service`), and run the following command.

```
$ ./gradlew bootJar
```

The `bootJar` Gradle task compiles the code and packages the application as a JAR file. By default, the JAR is generated in the `build/libs` folder. You should get an executable JAR file named `catalog-service-0.0.1-SNAPSHOT.jar`. Once you get the JAR artifact, you can go ahead and run it like any standard Java application.

```
$ java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar
```

NOTE

Another practical Gradle task is `build`, which combines the operations of the `bootJar` and `test` tasks.

Since the project contains the `spring-boot-starter-web` dependency, Spring Boot automatically configures an embedded Tomcat server. By looking at the logs in figure 3.4, you can see how one of the first execution steps is initializing a Tomcat server instance embedded in the application itself.

```


. --- C--- \ \ \ \
\ \ \ \ \ | | | | \ \ \ \
W ---| | | | | | | | | | | | | | | |
' | | | | | | | | | | | | | | | | | | |
=====|_|====|=====|_|=/_/_/_/
:: Spring Boot ::          (v2.7.0)

2022-06-05 08:05:42.391 INFO 46814 --- [           main] c.p.c.CatalogServiceApplication      : Starting CatalogServiceApplication using Java 17.0.3
2022-06-05 08:05:42.392 INFO 46814 --- [           main] c.p.c.CatalogServiceApplication      : No active profile set, falling back to 1 default profile: "default"
2022-06-05 08:05:42.664 INFO 46814 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer: Tomcat initialized with port(s): 9001 (http)
2022-06-05 08:05:42.666 INFO 46814 --- [           main] o.apache.catalina.core.StandardService: Starting service [Tomcat]
2022-06-05 08:05:42.667 INFO 46814 --- [           main] org.apache.catalina.core.StandardEngine: Starting Servlet engine: [Apache Tomcat/9.0.63]
2022-06-05 08:05:42.693 INFO 46814 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]: Initializing Spring embedded WebApplicationContext
2022-06-05 08:05:42.693 INFO 46814 --- [           main] w.s.c.WebServerApplicationContext      : Root WebApplicationContext: initialization completed in 282 ms
2022-06-05 08:05:42.803 INFO 46814 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer: Tomcat started on port(s): 9001 (http) with context path ''
2022-06-05 08:05:42.807 INFO 46814 --- [           main] c.p.c.CatalogServiceApplication      : Started CatalogServiceApplication in 0.529 seconds (JVM running for 0.735)

```

Figure 3.4 The startup logs from the Catalog Service application.

In the next section, you'll learn more about how the embedded server works in Spring Boot. Before moving on, you can stop the application with `Ctrl+C`.

3.2.2 Understanding the thread-per-request model

Let's consider the request/response pattern commonly used in web applications to establish synchronous interactions over HTTP. A client sends an HTTP request to a server that performs some computation and then replies with an HTTP response.

In web applications running in a Servlet container like Tomcat, requests are processed based on a model called *thread-per-request*. For each request, the application allocates a thread dedicated exclusively to handling that specific request and will not be used for anything else until a response is returned to the client. When the request handling involves intensive operations like I/O, the thread will block until they are completed. For example, if a database read is required, the thread will wait until data is returned from the database. That's why we say that this type of processing is **synchronous** and **blocking**.

Tomcat is initialized with a thread pool used to manage all incoming HTTP requests. New requests will be queued when all threads are in use, waiting for a thread to get free. In other words, the number of threads in Tomcat defines an upper limit to how many requests are supported concurrently, something very useful to remember when debugging performance issues. If the thread concurrency limit is hit continuously, you can always tune the thread pool configuration to accept more workload. With traditional applications, we would keep adding more computational resources to a specific instance. For cloud native applications, we would rather rely on horizontal scaling and deploying more replicas.

NOTE

In some applications having to respond to high demands, the thread-per-request model might not be ideal because it doesn't use the available computational resources in the most efficient way due to the blocking. In chapter 8, I'll introduce an asynchronous, non-blocking alternative with Spring WebFlux and Project Reactor, adopting the reactive programming paradigm.

Spring MVC is the library included in the Spring Framework to implement web applications, either full MVC or REST-based. Either way, the functionality is based on a server like Tomcat that provides a Servlet container compliant with the Java Servlet API. Figure 3.5 shows how the REST-based request/response interaction works in a Spring web application.

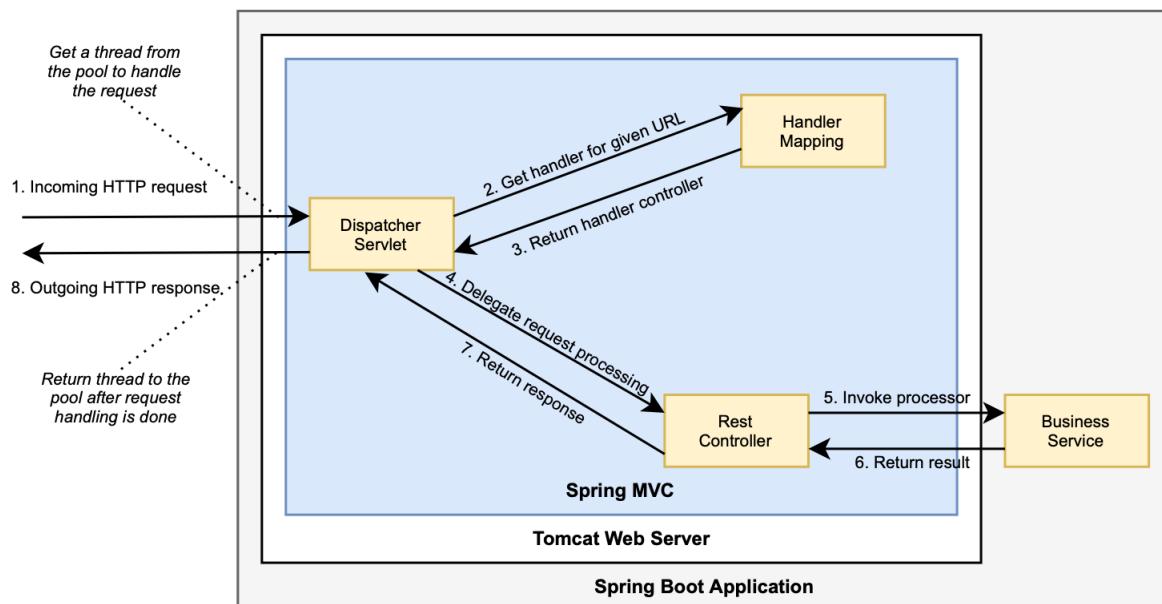


Figure 3.5 The DispatcherServlet is the entry-point to the Servlet container. It delegates the actual HTTP request processing to a Controller identified by the HandlerMapping as the one responsible for a given endpoint.

The DispatcherServlet provides a central entry point for request processing. When a client sends a new HTTP request for a specific URL pattern, the DispatcherServlet asks the HandlerMapping component, the controller responsible for that endpoint, and finally delegates the given controller the actual processing of the request. The controller processes the request, possibly by calling some other services, and then returns a response to the DispatcherServlet that finally replies to the client with an HTTP response.

Notice how the Tomcat server is embedded into the Spring Boot application. Spring MVC relies on the web server to accomplish its functionality. The same is true for any web server implementing the Servlet API, but since we are explicitly using Tomcat, let's go ahead and explore some options to configure it.

3.2.3 Configuring the embedded Tomcat

Tomcat is the default server that comes pre-configured with any Spring Boot web application. Sometimes, the default configuration might be enough, but for applications in production, it's likely you need to customize its behavior according to specific requirements.

NOTE

In a traditional Spring application, you would configure a server like Tomcat in dedicated files such as `server.xml` and `context.xml`. With Spring Boot, you can configure an embedded web server in two ways: through properties or in a `WebServerFactoryCustomizer` bean.

This section will show you how to configure Tomcat through properties. You'll learn more about configuring applications in the next chapter. For now, it's enough to know that you can define properties in an `application.properties` or `application.yml` file located in the `src/main/resources` folder of your project. You're free to choose which format to use: `.properties` files rely on key-value pairs while `.yml` files use the YAML format. In this book, I define properties using YAML. Spring Initializr generates an empty `application.properties` file by default, so remember to change its extension from `.properties` to `.yml` before moving on.

Let's go ahead and configure the embedded server for the Catalog Service application (`catalog-service`). All the configuration properties will go in the `application.yml` file.

HTTP PORT

By default, the embedded server is listening on port 8080. That is fine as long as you are working with only one application. When you run more Spring applications during development, usually the case with cloud native systems, you want to specify a different port number for each application using the `server.port` property.

Listing 3.1 Configure the web server port (application.yml)

```
server:
  port: 9001
```

CONNECTION TIMEOUT

The `server.tomcat.connection-timeout` property defines a limit for how much time Tomcat should wait between accepting a TCP connection from a client and actually receiving the HTTP request. It helps prevent DoS (Denial-of-Service) attacks where a connection is established, Tomcat reserves a thread to handle the request, but the request will never come. The same timeout is used to limit the time spent reading the HTTP request body when there is one.

The default value is 20s (20 seconds), which is probably too much for a standard cloud native application. In the context of highly distributed systems in the cloud, we probably don't want to wait more than a couple of seconds and risk a cascading failure due to a Tomcat instance hanging for too long. Something like 2s would be better. You can also use the `server.tomcat.keep-alive-timeout` property to configure how long to keep a connection open while waiting for new HTTP requests.

Listing 3.2 Configure timeouts for Tomcat (application.yml)

```
server:
  port: 9001
  tomcat:
    connection-timeout: 2s
    keep-alive-timeout: 15s
```

THREAD POOL

Tomcat has a pool of threads that process requests following the thread-per-request model. The number of available threads will determine how many requests can be handled simultaneously. You can configure the maximum number of request processing threads through the `server.tomcat.threads.max` property. You can also define the minimum number of threads that should always be kept running (`server.tomcat.threads.min-spare`), which is also how many threads are created at startup.

Determining the best configuration for a thread pool is complicated, and there's no magic formula to compute it. Resource analysis, monitoring, and many trials are usually necessary to find a suitable configuration. The default thread pool can grow up to 200 threads and has 10 worker threads always running, which are good starting values in production. In your local environment, you might want to lower those values to optimize resource consumption since it increases linearly with the number of threads.

Listing 3.3 Configure the Tomcat thread pool (application.yml)

```
server:
  port: 9001
  tomcat:
    connection-timeout: 2s
    keep-alive-timeout: 15s
    threads:
      max: 50
      min-spare: 5
```

So far, you saw that cloud native applications with Spring Boot are packaged as JAR files and rely on an embedded server to remove extra dependencies on the execution environment and enable agility. You learned how the thread-per-request works, familiarized with the request processing flow with Tomcat and Spring MVC, and configured Tomcat. In the next section, we'll move on to the business logic of Catalog Service and the implementation of a REST API with Spring MVC.

3.3 Building a RESTful application with Spring MVC

If you're building cloud native applications, chances are that you are working on a distributed system composed of several services interacting with each other to accomplish the overall functionality of the product, such as microservices. Your application might be consumed by a service developed by another team in your organization. Or you might be exposing its functionality to third parties. Either way, there's an essential aspect for any inter-service communication: the API.

The 15-Factor methodology promotes the **API first** pattern. It encourages you to establish the service interface first and work on the implementation later. The API represents a public contract between your application and its consumers, and it's in your best interest to define it as the first thing.

Suppose you agree on a contract and define the API first. In that case, other teams can start working on their solutions and develop against your API to implement the integration with your application. If you don't, there will be a bottleneck, forcing other teams to wait until you're done with your application. Furthermore, discussing the API upfront enables productive discussions with the stakeholders that can lead to clarifying the scope of the application and even define user stories to implement.

In the cloud, any application could be a backing service for another one. Adopting an API first mentality will help you evolve your applications and adapt them to future requirements.

This section will guide you by defining a contract for Catalog Service as a REST API, the most used service interface model for cloud native applications. You'll use Spring MVC to implement the REST API, validate it, and test it. I'll also go through some considerations for evolving an API for future requirements, a common issue in highly distributed systems like cloud native applications.

3.3.1 REST API first, business logic later

Designing the API first assumes you have already defined the requirements, so let's start with that. Catalog Service will be responsible for supporting the following use cases:

- view the list of books in the catalog;
- search books by their International Standard Book Number (ISBN);
- add a new book to the catalog;
- edit information for an existing book;
- remove a book from the catalog.

In other words, we can say the application should provide an API to perform CRUD operations on books. The format will follow the REST style applied to HTTP. There are several ways to

design an API to fulfill those use cases. In the following, we'll use the one described in table 3.1.

Table 3.1 Specifications for the REST API that will be exposed by Catalog Service.

Endpoint	HTTP Method	Req. Body	Status	Res. body	Description
/books	GET		200	Book[]	Get all the books in the catalog.
/books	POST	Book	201	Book	Add a new book to the catalog.
			422		A book with the same ISBN already exists.
/books/{isbn}	GET		200	Book	Get the book with the given ISBN.
			404		No book with the given ISBN exists.
/books/{isbn}	PUT	Book	200	Book	Update the book with the given ISBN.
			201	Book	Create a book with the given ISBN.
/books/{isbn}	DELETE		204		Delete the book with the given ISBN.

SIDE BAR

Documenting APIs

Documenting APIs is an essential task when following the API-first approach. In the Spring ecosystem, there are two main options.

- Spring provides a Spring REST Docs project (<https://spring.io/projects/spring-restdocs>) that helps you document REST APIs via test-driven development (TDD), resulting in high-quality and maintainable documentation. The resulting documentation is aimed at people, relying on formats like AsciiDocs or Markdown. If you'd like to obtain an OpenAPI representation as well, you can check the `restdocs-api-spec` community-driven project to add OpenAPI support to Spring REST Docs (<https://github.com/ePages-de/restdocs-api-spec>).
- The `springdoc-openapi` community-driven project helps automate the generation of API documentation according to the OpenAPI 3 format (<https://springdoc.org>).

The contract is established through the REST API, so let's move on and look at the business logic. The solution is centered around three concepts:

- **Entity.** An entity represents the noun in a domain, such as "book".
- **Service.** A service defines the use cases for the domain. For example, "adding a book to the catalog".
- **Repository.** A repository is an abstraction to let the domain layer access data independently from its source.

Let's start with the domain entity.

DEFINING THE DOMAIN ENTITY

The REST API previously defined should make it possible to operate on books. That's the *domain entity*. In the Catalog Service project, create a new `com.polarbookshop.catalogservice.domain` package for the business logic and create a Book Java record for representing the domain entity.

Listing 3.4 The Book record defines a domain entity for the application (Book.java)

```
package com.polarbookshop.catalogservice.domain;

public record Book ( ❶
    String isbn, ❷
    String title,
    String author,
    Double price
){}
```

- ❶ The domain model is implemented as a record, an immutable object.
- ❷ Uniquely identifies a book.

IMPLEMENTING THE USE CASES

The use cases enumerated by the application requirements can be implemented in a `@Service` class. In the `com.polarbookshop.catalogservice.domain` package, create a `BookService` class, as shown in the following listing. The service relies on some classes that you'll create in a minute.

Listing 3.5 Implement the use cases for the application (BookService.java)

```

package com.polarbookshop.catalogservice.domain;

import org.springframework.stereotype.Service;

@Service ①
public class BookService {
    private final BookRepository bookRepository;

    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository; ②
    }

    public Iterable<Book> viewBookList() {
        return bookRepository.findAll();
    }

    public Book viewBookDetails(String isbn) {
        return bookRepository.findByIsbn(isbn) ③
            .orElseThrow(() -> new BookNotFoundException(isbn));
    }

    public Book addBookToCatalog(Book book) {
        if (bookRepository.existsByIsbn(book.isbn())) { ④
            throw new BookAlreadyExistsException(book.isbn());
        }
        return bookRepository.save(book);
    }

    public void removeBookFromCatalog(String isbn) {
        bookRepository.deleteByIsbn(isbn);
    }

    public Book editBookDetails(String isbn, Book book) {
        return bookRepository.findByIsbn(isbn)
            .map(existingBook -> {
                var bookToUpdate = new Book( ⑤
                    existingBook.isbn(),
                    book.title(),
                    book.author(),
                    book.price());
                return bookRepository.save(bookToUpdate);
            })
            .orElseGet(() -> addBookToCatalog(book)); ⑥
    }
}

```

- ① Stereotype annotation that marks a class to be a service managed by Spring.
- ② BookRepository is provided through constructor autowiring.
- ③ When trying to view a book that doesn't exist, a dedicated exception is thrown.
- ④ When adding the same book to the catalog multiple times, a dedicated exception is thrown.
- ⑤ When editing the book, all the Book fields can be updated except the ISBN code because it's the entity identifier.
- ⑥ When changing the details for a book not in the catalog yet, create a new book.

NOTE

The Spring Framework provides two flavors of dependency injection: constructor-based and setter-based. We'll use constructor-based dependency injection in any production code as advocated by the Spring team because it ensures that the required dependencies are always returned fully initialized and never `null`. Furthermore, it encourages building immutable objects and improves their testability. For more information, refer to the Spring Framework documentation (<https://spring.io/projects/spring-framework>).

USING THE REPOSITORY ABSTRACTION FOR DATA ACCESS

The `BookService` class relies on a `BookRepository` object to retrieve and save books. The domain layer should be unaware of how data are persisted, so `BookRepository` should be an interface to decouple the abstraction from the actual implementation. Create a `BookRepository` interface in the `com.polarbookshop.catalogservice.domain` package to define the abstraction for accessing book data.

Listing 3.6 The abstraction used by the domain layer to access data (`BookRepository.java`)

```
package com.polarbookshop.catalogservice.domain;

import java.util.Optional;

public interface BookRepository {
    Iterable<Book> findAll();
    Optional<Book> findByIsbn(String isbn);
    boolean existsByIsbn(String isbn);
    Book save(Book book);
    void deleteByIsbn(String isbn);
}
```

While the repository interface belongs to the domain, its implementation is part of the persistence layer. We'll add a data persistence layer using a relational database in chapter 5. For now, it's enough to add a simple in-memory map to retrieve and save books. You can define the implementation in a `InMemoryBookRepository` class, located in a new `com.polarbookshop.catalogservice.persistence` package.

Listing 3.7 In-memory implementation of the BookRepository interface (InMemoryBookRepository.java)

```

package com.polarbookshop.catalogservice.persistence;

import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;
import com.polarbookshop.catalogservice.domain.Book;
import com.polarbookshop.catalogservice.domain.BookRepository;
import org.springframework.stereotype.Repository;

@Repository ①
public class InMemoryBookRepository implements BookRepository {
    private static final Map<String, Book> books = ②
        new ConcurrentHashMap<>();

    @Override
    public Iterable<Book> findAll() {
        return books.values();
    }

    @Override
    public Optional<Book> findByIsbn(String isbn) {
        return existsByIsbn(isbn) ? Optional.of(books.get(isbn)) :
            Optional.empty();
    }

    @Override
    public boolean existsByIsbn(String isbn) {
        return books.get(isbn) != null;
    }

    @Override
    public Book save(Book book) {
        books.put(book.isbn(), book);
        return book;
    }

    @Override
    public void deleteByIsbn(String isbn) {
        books.remove(isbn);
    }
}

```

- ① Stereotype annotation that marks a class to be a repository managed by Spring.
- ② In-memory map to store books for testing purposes.

USING EXCEPTIONS TO SIGNAL ERRORS IN THE DOMAIN

Let's complete the business logic for Catalog Service by implementing the two exceptions we used in listing [3.5](#).

`BookAlreadyExistsException` is a runtime exception thrown when we try to add to the catalog a book already there. It prevents having duplicated entries in the catalog.

**Listing 3.8 Exception thrown when adding a book that already exists
(BookAlreadyExistsException.java)**

```
package com.polarbookshop.catalogservice.domain;

public class BookAlreadyExistsException extends RuntimeException {
    public BookAlreadyExistsException(String isbn) {
        super("A book with ISBN " + isbn + " already exists.");
    }
}
```

BookNotFoundException is a runtime exception thrown when trying to fetch a book that is not in the catalog.

Listing 3.9 Exception thrown when a book cannot be found (BookNotFoundException.java)

```
package com.polarbookshop.catalogservice.domain;

public class BookNotFoundException extends RuntimeException {
    public BookNotFoundException(String isbn) {
        super("The book with ISBN " + isbn + " was not found.");
    }
}
```

That completes the business logic for the Catalog Service. It's relatively simple, but it's recommended not to be influenced by how data are persisted or exchanged with clients. The business logic should be independent of anything else, including the API. If you're interested in this topic, I suggest exploring the concepts of *Domain-Driven Design* and *Hexagonal Architecture*.

3.3.2 Implementing a REST API with Spring MVC

After implementing the business logic, we can expose the use cases through a REST API. Spring MVC provides `@RestController` classes to define methods that handle incoming HTTP requests for specific HTTP methods and resource endpoints.

As you saw in the previous section, the `DispatcherServlet` component will invoke the right controller for each request. Figure 3.6 shows the scenario where a client sends an HTTP GET request to view the details of a specific book.

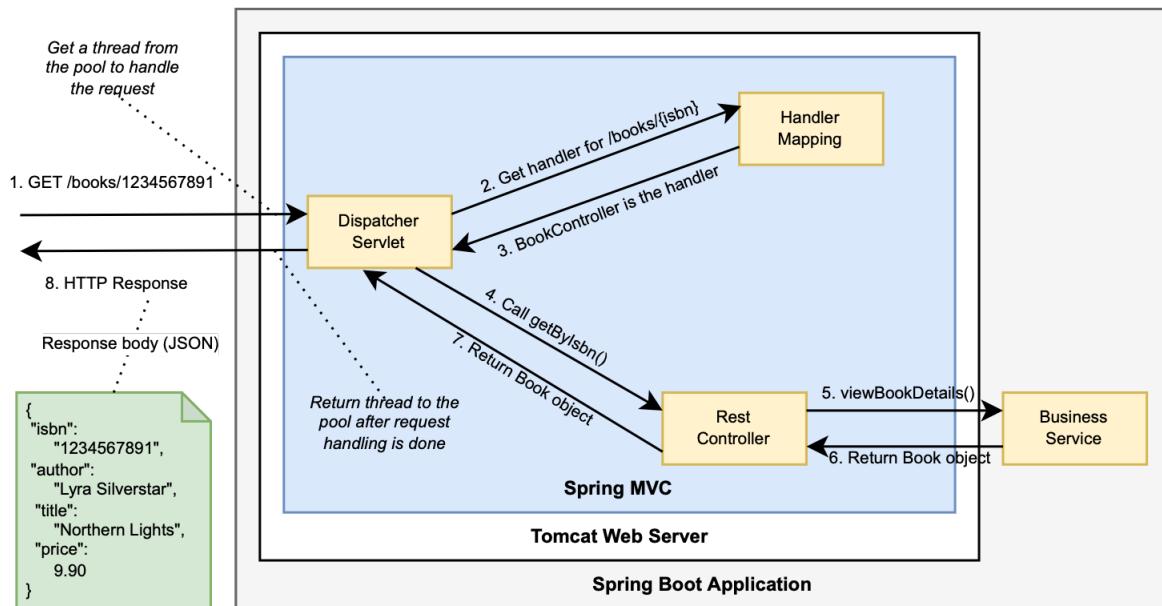


Figure 3.6 The processing flow for an HTTP GET requests arriving at the "/books/<isbn>" endpoint.

We want to implement a method handler for each use case defined in the application requirements since we want to make all of them available to clients. Create a package for the web layer (`com.polarbookshop.catalogservice.web`) and add a `BookController` class responsible for handling HTTP requests sent to the `/books` base endpoint.

Listing 3.10 Define handlers for REST endpoints (BookController.java)

```

package com.polarbookshop.catalogservice.web;

import com.polarbookshop.catalogservice.domain.Book;
import com.polarbookshop.catalogservice.domain.BookService;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

@RestController ①
@RequestMapping("books") ②
public class BookController {
    private final BookService bookService;

    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping ③
    public Iterable<Book> get() {
        return bookService.viewBookList();
    }

    @GetMapping("{isbn}") ④
    public Book getByIsbn(@PathVariable String isbn) { ⑤
        return bookService.viewBookDetails(isbn);
    }

    @PostMapping ⑥
    @ResponseStatus(HttpStatus.CREATED) ⑦
    public Book post(@RequestBody Book book) { ⑧
        return bookService.addBookToCatalog(book);
    }

    @DeleteMapping("{isbn}") ⑨
    @ResponseStatus(HttpStatus.NO_CONTENT) ⑩
    public void delete(@PathVariable String isbn) {
        bookService.removeBookFromCatalog(isbn);
    }

    @PutMapping("{isbn}") ⑪
    public Book put(@PathVariable String isbn, @RequestBody Book book) {
        return bookService.editBookDetails(isbn, book);
    }
}

```

- ① Stereotype annotation marking a class as a Spring component and a source of handlers for REST endpoints.
- ② Identifies the root path mapping URI for which the class provides handlers ("books").
- ③ Maps HTTP GET requests onto the specific handler method.
- ④ A URI template variable appended to the root path mapping URI ("books/{isbn}").
- ⑤ @PathVariable binds a method parameter to a URI template variable ({isbn}).
- ⑥ Maps HTTP POST requests onto the specific handler method.
- ⑦ Returns a 201 status if the book is created successfully.
- ⑧ @RequestBody binds a method parameter to the body of a web request.

- ⑨ Maps HTTP DELETE requests onto the specific handler method.
- ⑩ Returns a 204 status if the book is deleted successfully.
- ⑪ Maps HTTP PUT requests onto the specific handler method.

Go ahead and run the application (`./gradlew bootRun`). When verifying HTTP interactions with an application, you can either use a command-line tool like `curl` or software with a graphical user interface like Insomnia. I'm going to use a convenient command-line tool called **HTTPie** (<https://httpie.org>). You can find information about how to install it in section A.4 of Appendix A.

Open a Terminal window and perform an HTTP POST request to add a book to the catalog.

```
$ http POST :9001/books author="Lyra Silverstar" \
    title="Northern Lights" isbn="1234567891" price=9.90
```

The result should be an HTTP response with the 201 code, meaning that the book has been created successfully. Let's double-check by submitting an HTTP GET request to fetch the book with the ISBN code we used on creation.

```
$ http :9001/books/1234567891

HTTP/1.1 200
Content-Type: application/json

{
  "author": "Lyra Silverstar",
  "isbn": "1234567891",
  "price": 9.9,
  "title": "Northern Lights"
}
```

When you're done trying the application, stop its execution with `Ctrl+C`.

SIDE BAR**About content negotiation**

All handler methods in `BookController` work on `Book` Java objects. And yet, when you perform a request, you get back a JSON object. How is that possible? Spring MVC relies on an `HttpMessageConverter` bean to convert the returned object into a specific representation the client supports. The decision about the content type is driven by a process called content negotiation, during which the client and the server agree on a representation that both can understand.

The client can inform the server about which content types it supports through the `Accept` header in the HTTP request. By default, Spring Boot configures a set of `HttpMessageConverter` beans to return objects represented as JSON, and the `HTTPPie` tool is configured to accept any content type by default. The result is that both client and server support the JSON content type, so they agree on using that to communicate.

The application we have implemented so far is still not complete. For example, nothing prevents you from posting a new book with an ISBN in the wrong format or without specifying a title. We need to validate the input.

3.3.3 Data validation and error handling

As a general rule, before saving any data, you should always validate the content, both for data consistency and security reasons. A book with no title will be of no use in our application, and it would probably make it fail.

For the `Book` class, we might think of using these validation constraints:

- the ISBN must be defined and in the correct format (ISBN-10 or ISBN-13);
- the title must be defined;
- the author must be defined;
- the price must be defined and greater than zero.

Java Bean Validation is a popular specification for expressing constraints and validation rules on Java objects via annotations. Spring Boot provides a convenient starter dependency containing the Java Bean Validation API and its implementation. Add the new dependency in the `build.gradle` file for your Catalog Service project. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 3.11 Add dependency for Spring Boot Validation starter in Catalog Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-validation'
}
```

You can now use the Java Bean Validation API to define validation constraints as annotations directly on the Book record fields.

Listing 3.12 Validation constraints defined for each field (Book.java)

```
package com.polarbookshop.catalogservice.domain;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Positive;

public record Book (

    @NotBlank(message = "The book ISBN must be defined.")
    @Pattern( ①
        regexp = "^([0-9]{10}|[0-9]{13})$",
        message = "The ISBN format must be valid."
    )
    String isbn,

    @NotBlank( ②
        message = "The book title must be defined."
    )
    String title,

    @NotBlank(message = "The book author must be defined.")
    String author,

    @NotNull(message = "The book price must be defined.")
    @Positive( ③
        message = "The book price must be greater than zero."
    )
    Double price
) {}
```

- ① The annotated element must match the specified regular expression (standard ISBN format).
- ② The annotated element must not be null and must contain at least one non-whitespace character.
- ③ The annotated element must not be null and greater than zero.

NOTE

Books are uniquely identified according to their ISBN (International Standard Book Number). ISBNs used to be composed of 10 digits, but they now consist of 13. For simplicity, we'll limit ourselves to checking for their length and whether all the elements are digits by using a regular expression.

The annotations from the Java Bean Validation API define the constraints, but they are not enforced yet. We can instruct Spring to validate the `Book` object in the `BookController` class by using the `@Valid` annotation whenever a `@RequestBody` is specified as a method argument. In this way, whenever we create or update a book, Spring will run the validation and throw an error if any constraint is violated. We can update the `post()` and `put()` methods in the `BookController` class as follows.

Listing 3.13 Validate books passed in a request body (BookController.java)

```
...
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Book post(@Valid @RequestBody Book book) {
    return bookService.addBookToCatalog(book);
}

@PutMapping("{isbn}")
public Book put(@PathVariable String isbn, @Valid @RequestBody Book book)
{
    return bookService.editBookDetails(isbn, book);
}
...
```

Spring lets you handle error messages in different ways. When building an API, it's good to consider which types of errors it can throw since they are just as important as the domain data. When it's a REST API, you want to ensure that the HTTP response uses a status code that best fits the purpose and includes a meaningful message to help the client identify the problem.

When the validation constraints we have just defined are not met, a `MethodArgumentNotValidException` is thrown. What if we try to fetch a book that doesn't exist? The business logic we previously implemented throws dedicated exceptions (`BookAlreadyExistsException` and `BookNotFoundException`). All those exceptions should be handled in the REST API context to return the error codes defined in the original specification.

To handle errors for a REST API, we can use the standard Java Exceptions and rely on a `@RestControllerAdvice` class to define what to do when a given exception is thrown. It's a centralized approach that allows decoupling the exception handling from the code throwing the exception. In the `com.polarbookshop.catalogservice.web` package, create a `BookControllerAdvice` class as follows.

**Listing 3.14 The advice class defining how to handle exceptions
(BookControllerAdvice.java)**

```
package com.polarbookshop.catalogservice.web;

import java.util.HashMap;
import java.util.Map;
import com.polarbookshop.catalogservice.domain.BookAlreadyExistsException;
import com.polarbookshop.catalogservice.domain.BookNotFoundException;
import org.springframework.http.HttpStatus;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice ①
public class BookControllerAdvice {

    @ExceptionHandler(BookNotFoundException.class) ②
    @ResponseStatus(HttpStatus.NOT_FOUND)
    String bookNotFoundHandler(BookNotFoundException ex) {
        return ex.getMessage(); ③
    }

    @ExceptionHandler(BookAlreadyExistsException.class)
    @ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY) ④
    String bookAlreadyExistsHandler(BookAlreadyExistsException ex) {
        return ex.getMessage();
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public Map<String, String> handleValidationExceptions(
        MethodArgumentNotValidException ex ⑤
    ) {
        var errors = new HashMap<String, String>();
        ex.getBindingResult().getAllErrors().forEach(error -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage); ⑥
        });
        return errors;
    }
}
```

- ① Marks the class as a centralized exception handler.
- ② Defines the exception for which the handler must be executed.
- ③ The message that will be included in the HTTP response body.
- ④ Define the status code for the HTTP response created when the exception is thrown.
- ⑤ Handles the exception thrown when the Book validation fails.
- ⑥ Collects meaningful error messages about which Book fields were invalid instead of returning an empty message.

The mapping provided in the `@RestControllerAdvice` class makes it possible to obtain an HTTP response with status 422 (unprocessable entity) when we try to create a book that already

exists in the catalog, a response with status 404 (not found) when we try to read a book that doesn't exist, and a response with status 400 (bad request) when one or more fields in a `Book` object are invalid. Each response will contain a meaningful message we defined as part of the validation constraint or custom exception.

Build and rerun the application (`./gradlew bootRun`): if you now try to create a book without a title and with a wrongly formatted ISBN, the request will fail.

```
$ http POST :9001/books author="Jon Snow" title="" isbn=123ABC456Z \
  price=9.90
```

The result will be an error message with a "400 Bad Request" status, meaning that the server couldn't process the HTTP request because it was incorrect. The response body contains a detailed message about which part of the request was incorrect and how to fix it, just as we defined in listing [3.12](#).

```
HTTP/1.1 400
Content-Type: application/json

{
  "isbn": "The ISBN format must be valid.",
  "title": "The book title must be defined."
}
```

When you're done trying the application, stop its execution with `Ctrl+C`.

That concludes the implementation of the REST API to expose the functionality related to the management of books provided by Catalog Service. Next, I'll discuss a few aspects of evolving APIs to adapt to new requirements.

3.3.4 Evolving APIs for future requirements

In a distributed system, we need a plan to evolve APIs not to break the functionality of other applications. It is a challenging task because we want independent applications, but they probably exist to provide services to some other applications, so we get somewhat limited in the number of changes we make independently from the clients.

The best approach is to make backward-compatible changes to the API. For example, we can add an optional field to the `Book` object without affecting the clients of the Catalog Service application.

Sometimes, breaking changes are necessary. One thing you could do is **API versioning**. Should you decide to make a breaking change to the REST API for the Catalog Service application, you might introduce a versioning system for the endpoints. The version might be part of the endpoint itself, like `/v2/books`. Or it might be specified as an HTTP header. This system helps prevent existing clients breaking, but they will have to update their interface to match the new API version sooner or later, meaning that coordination is needed.

A different approach focuses on making the REST API client as much resilient to API changes as possible. The solution is to use the **hypermedia** aspect of the REST architecture, as described by Dr. Roy Fielding.

REST APIs can return the object requested and information about *where* to go next, *links* to perform related operations. The beauty of this feature is that the links are only shown when it makes sense to follow them, providing information about *when* to go.

HATEOAS (Hypermedia as the Engine of Application State) is the highest level of REST API maturity, according to Richardson's Maturity Model. Spring provides the Spring HATEOAS project to add *hypermedia* support to a REST API. I will not use it in this book, but I encourage you to check the project's online documentation at <https://spring.io/projects/spring-hateoas>.

These considerations conclude the discussion around building a RESTful application with Spring. In the next section, you'll see how to write automated tests to verify the behavior of the application.

3.4 Testing a RESTful application with Spring

Automated tests are paramount to producing high-quality software. One of the goals for adopting a cloud native approach is speed. It's impossible to move fast if the code is not adequately tested in an automated fashion, let alone implementing a continuous delivery process.

As a developer, you usually implement a feature, deliver it, and then move on to a new one, possibly refactoring the existing code. Refactoring code is risky since you might break some existing functionality. Automated tests reduce the risk and encourage refactoring, knowing that there's a test failing should you break something. You would probably also like to reduce the feedback cycle and see if you made some mistakes as soon as possible. That leads you to design tests in a way that maximizes their usefulness and efficiency. You shouldn't aim to reach maximum test coverage but rather write meaningful tests. For example, writing tests for standard getters and setters doesn't make sense.

An essential practice of continuous delivery is **test-driven development** (TDD), which helps achieve the goal of delivering software quickly, reliably, and safely. The idea is to drive software development by writing tests before implementing the production code. I recommend adopting TDD in real-world scenarios. However, it's not very suitable when teaching new technologies and frameworks in a book, so I'm not going to follow its principles.

Automated tests assert that any new feature works as intended, and you haven't broken any existing functionality. It means that automated tests work as *regression tests*. You should write tests to protect your colleagues and yourself from making mistakes. What and how in-depth to test is driven by the risk associated with a specific piece of code. Writing tests is also a learning

experience to improve your skills, especially if you're beginning your software development journey.

One way of classifying software tests is provided by the **Agile Testing Quadrants** model originally introduced by Brian Marick, and later described and expanded by Lisa Crispin and Janet Gregory in their books "Agile Testing", "More Agile Testing", and "Agile Testing Condensed". Their model was also embraced by Jez Humble and Dave Farley in their "Continuous Delivery" book. The *Quadrants* classify software tests based on whether they are technology or business-facing, and whether they support development teams or are used to critique the product. Figure 3.7 shows some examples of testing types we'll mention throughout the book, based on the model presented in the "Agile Testing Condensed" book.

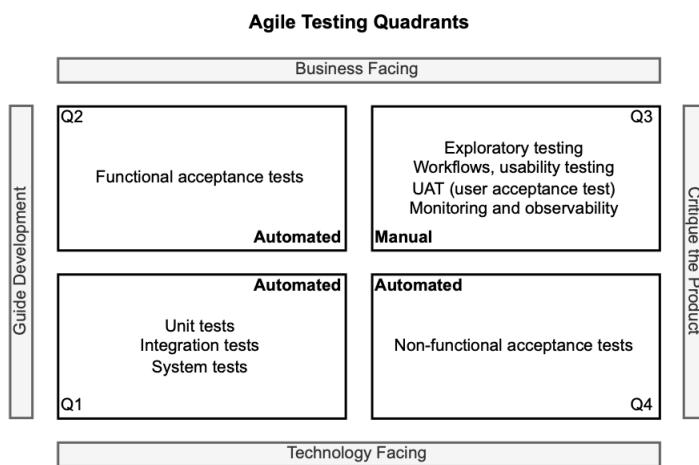


Figure 3.7 The Agile Testing Quadrants are a model helpful to plan a software testing strategy.

Following continuous delivery practices, we should aim at achieving fully automated tests in three out of four quadrants, as shown in figure 3.7. Throughout the book, we'll mainly focus on the bottom-left quadrant. In this section, we'll work with unit tests and integration tests (sometimes called *component tests*). We write **unit tests** to verify the behavior of single application components in isolation while **integration tests** to assert the overall functioning of different parts of an application interacting with each other.

In a Gradle or Maven project, test classes are usually placed in a `src/test/java` folder. In Spring, unit tests aren't required to load the Spring application context and don't rely on any Spring library. On the other hand, integration tests need a Spring application context to run. This section will show you how to test a RESTful application like Catalog Service with unit and integration tests.

3.4.1 Unit tests with JUnit 5

Unit tests are not aware of Spring and don't rely on any Spring library. They are intended to test the behavior of single components as isolated units. Any dependency at the edge of the unit is mocked to keep the test shielded from any external component.

Writing unit tests for Spring applications is not different from writing them for any other Java application. Therefore, I won't go into detail. By default, any Spring project created from Spring Initializr contains the `spring-boot-starter-test` dependency that imports testing libraries like JUnit 5, Mockito, and AssertJ into the project. So, we're all set for writing unit tests.

The business logic of an application is usually a sensible area to cover by unit tests. If we consider the Catalog Service application, a good candidate for unit testing might be the validation logic for the `Book` class. The validation constraints are defined using the Java Validation API annotations. We are interested in testing that they are applied correctly to the `Book` class. We can check that in a `BookValidationTests` class, as shown in the following listing.

Listing 3.15 Unit tests to verify book validation constraints (BookValidationTests.java)

```
package com.polarbookshop.catalogservice.domain;

import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;

class BookValidationTests {
    private static Validator validator;

    @BeforeAll ①
    static void setUp() {
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }

    @Test ②
    void whenAllFieldsCorrectThenValidationSucceeds() {
        var book = ③
            new Book("1234567890", "Title", "Author", 9.90);
        Set<ConstraintViolation<Book>> violations = validator.validate(book);
        assertThat(violations).isEmpty(); ④
    }

    @Test
    void whenIsbnDefinedButIncorrectThenValidationFails() {
        var book = ⑤
            new Book("a234567890", "Title", "Author", 9.90);
        Set<ConstraintViolation<Book>> violations = validator.validate(book);
        assertThat(violations).hasSize(1);
        assertThat(violations.iterator().next().getMessage())
            .isEqualTo("The ISBN format must be valid."); ⑥
    }
}
```

- ① Identifies a block of code executed before all tests in the class.
- ② Identifies a test case.
- ③ Creates a book with a valid ISBN.
- ④ Asserts that there is no validation error.
- ⑤ Creates a book with a non-valid ISBN code.
- ⑥ Asserts that the violated validation constraint is about the incorrect ISBN.

Then, run the tests with the following command.

```
$ ./gradlew test --tests BookValidationTests
```

3.4.2 Integration tests with @SpringBootTest

Integration tests cover the interaction among software components, and in Spring, they require an application context to be defined. The `spring-boot-starter-test` dependency also imports the test utilities from Spring Framework and Spring Boot.

Spring Boot offers a powerful `@SpringBootTest` annotation you can use on a test class to bootstrap an application context automatically when running tests. The configuration used to create the context can be customized if needed. Otherwise, the class annotated with `@SpringBootApplication` will become the configuration source for component scanning and properties, including the usual auto-configuration provided by Spring Boot.

When working with web applications, you can run tests on a mock web environment or a running server. You can configure that by defining a value for the `webEnvironment` attribute the `@SpringBootTest` annotation provides, as shown in table 3.2.

Table 3.2 A Spring Boot integration test can be initialized with a mock web environment or a running server.

Web Environment Option	Description
MOCK	Creates a web application context with a mock Servlet container. Default option.
RANDOM_PORT	Creates a web application context with a Servlet container listening on a random port.
DEFINED_PORT	Creates a web application context with a Servlet container listening on the port defined through the <code>server.port</code> property.
NONE	Creates an application context without a Servlet container.

When using a mock web environment, you can rely on the `MockMvc` object to send HTTP requests to the application and check their results. For environments with a running server, the `TestRestTemplate` utility lets you perform REST calls to the application running on an actual server. By inspecting the HTTP response, you can verify if the API works as intended.

Recent versions of Spring Framework and Spring Boot have extended the features for testing web applications. You can now use the `WebTestClient` class to test REST APIs both on mock environments and running servers. Compared to `MockMvc` and `TestRestTemplate`, `WebTestClient` provides a modern and fluent API and additional features. Furthermore, you can use it for both imperative (e.g., Catalog Service) and reactive applications, optimizing learning and productivity.

Since `WebTestClient` is part of the Spring WebFlux project, you need to add a new dependency in your Catalog Service project (`build.gradle`). Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 3.16 Add test dependency for Spring Reactive Web in Catalog Service (build.gradle)

```
dependencies {
    ...
    testImplementation 'org.springframework.boot:spring-boot-starter-webflux'
}
```

Chapter 8 will cover Spring WebFlux and reactive applications. For now, we're only interested in the `WebTestClient` object for testing the API exposed by Catalog Service. In the previous chapter, you saw that Spring Initializr generated an empty `CatalogServiceApplicationTests` class. Let's populate it with integration tests. For the setup, we'll use the `@SpringBootTest` annotation configured to provide a full Spring application context, including a running server exposing its services through a random port (because it doesn't matter which one).

Listing 3.17 Integration tests for Catalog Service (CatalogServiceApplicationTests.java)

```
package com.polarbookshop.catalogservice;

import com.polarbookshop.catalogservice.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;
import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest( ❶
    webEnvironment = SpringApplication.WebEnvironment.RANDOM_PORT
)
class CatalogServiceApplicationTests {

    @Autowired
    private WebTestClient webTestClient; ❷

    @Test
    void whenPostRequestThenBookCreated() {
        var expectedBook = new Book("1231231231", "Title", "Author", 9.90);

        webTestClient
            .post() ❸
            .uri("/books") ❹
            .bodyValue(expectedBook) ❺
            .exchange() ❻
            .expectStatus().isCreated() ❻
            .expectBody(Book.class).value(actualBook -> {
                assertThat(actualBook).isNotNull(); ❽
                assertThat(actualBook.isbn())
                    .isEqualTo(expectedBook.isbn()); ❾
            });
    }
}
```

- ❶ Loads a full Spring web application context and a Servlet container listening on a random port.
- ❷ Utility to perform REST calls for testing.
- ❸ Sends an HTTP POST request.
- ❹ Sends the request to the "/books" endpoint.
- ❺ Sets the request body to the `expectedBook`.
- ❻ Checks if the response status is `CREATED`.
- ❽ Checks if the actual book is not null.
- ❾ Checks if the ISBN of the actual book is equal to the ISBN of the expected book.

- ⑤ Adds the book in the request body.
- ⑥ Sends the request.
- ⑦ Verifies that the HTTP response has status "201 Created".
- ⑧ Verifies that the HTTP response has a non-null body.
- ⑨ Verifies that the created object is as expected.

NOTE

You might be wondering why I haven't used constructor-based dependency injection in listing 3.17, considering that I previously stated that's the recommended option. Using field-based dependency injection in production code has been deprecated and strongly discouraged, but it's still acceptable to autowire dependencies in a test class. In every other scenario, I recommend sticking with constructor-based dependency injection for the reasons I explained earlier. For more information, you can refer to the Spring Framework official documentation (<https://spring.io/projects/spring-framework>).

Then, run the tests with the following command.

```
$ ./gradlew test --tests CatalogServiceApplicationTests
```

Depending on the application size, loading a full application context with auto-configuration for all integration tests might be too much. Spring Boot has a convenient feature (enabled by default) to cache the context so that it's re-used in all test classes annotated with `@SpringBootTest` and the same configuration. Sometimes, that's not enough.

Test execution time matters, so Spring Boot is fully equipped to run integration tests loading only the parts of the application needed. Let's see how that works.

3.4.3 Testing REST controllers with `@WebMvcTest`

Some integration tests might not need a fully initialized application context. For example, there's no need to load the web components when testing the data persistence layer. If you're testing the web components, you don't need to load the data persistence layer.

Spring Boot offers the possibility of using contexts initialized only with a subgroup of components (beans), targeting a specific application slice. **Slice tests** don't use the `@SpringBootTest` annotation, but one among a set of annotations dedicated to particular parts of an application: Web MVC, Web Flux, REST Client, JDBC, JPA, Mongo, Redis, JSON, and others. Each of those annotations initializes an application context, filtering out all the beans outside the slice.

We can test that Spring MVC controllers work as intended by using the `@WebMvcTest`

annotation. `@WebMvcTest` loads a Spring application context in a mock web environment (no running server), configures the Spring MVC infrastructure, and includes only the beans used by the MVC layer, like `@RestController` and `@RestControllerAdvice`. It's a good idea to limit the context to the beans used by the specific controller under test. We can do so by providing the controller class as an argument to the `@WebMvcTest` annotation, as shown in the following listing.

Listing 3.18 Integration tests for the web MVC slice (BookControllerMvcTests.java)

```
package com.polarbookshop.catalogservice.web;

import com.polarbookshop.catalogservice.domain.BookNotFoundException;
import com.polarbookshop.catalogservice.domain.BookService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;
import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request
➥.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result
➥.MockMvcResultMatchers.status;

@WebMvcTest(BookController.class) ①
class BookControllerMvcTests {

    @Autowired
    private MockMvc mockMvc; ②

    @MockBean ③
    private BookService bookService;

    @Test
    void whenGetBookNotExistingThenShouldReturn404() throws Exception {
        String isbn = "73737313940";
        given(bookService.viewBookDetails(isbn))
            .willThrow(BookNotFoundException.class); ④
        mockMvc
            .perform(get("/books/" + isbn)) ⑤
            .andExpect(status().isNotFound()); ⑥
    }
}
```

- ① Identifies a test class that focuses on Spring MVC components, explicitly targeting `BookController`.
- ② Utility class to test the web layer in a mock environment.
- ③ Adds a mock of `BookService` to the Spring application context.
- ④ Defines the expected behavior for the `BookService` mock bean.
- ⑤ `MockMvc` is used to perform an HTTP GET request and verify the result.
- ⑥ Expects the response to have a "404 Not Found" status.

WARNING If you use IntelliJ IDEA, you might get a warning that `MockMvc` cannot be autowired. Don't worry. It's a false positive. You can get rid of the warning by annotating the field with
`@SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")`

Then, run the tests with the following command.

```
$ ./gradlew test --tests BookControllerMvcTests
```

`MockMvc` is a utility class that lets you test web endpoints without loading a server like Tomcat. Such a test is naturally lighter than the one we wrote in the previous section, where an embedded server was needed to run the test.

Slice tests run against an application context containing only the parts of the configuration requested by that application slice. In the case of collaborating beans outside the slice, for example, the `BookService` class, we use mocks.

Mocks created with the `@MockBean` annotation are different from a standard mock (for example, one created with Mockito) since the class is not only mocked, but the mock is also included in the application context. So, whenever the context is asked to autowire that bean, it automatically injects the mock rather than the actual implementation.

3.4.4 Testing the JSON serialization with `@JsonTest`

The `Book` objects returned by the methods in `BookController` are parsed into JSON objects. By default, if you don't change the configuration, Spring Boot automatically configures the Jackson library to parse Java objects into JSON (*serialization*) and vice-versa (*deserialization*).

Using the `@JsonTest` annotation, you can test JSON serialization and deserialization for your domain objects. `@JsonTest` loads a Spring application context and auto-configures the JSON mappers for the specific library in use (by default, it's Jackson). Furthermore, it configures the `JacksonTester` utility you can use to check that the JSON mapping works as expected, relying on the `JsonPath` and `JSONAssert` libraries.

NOTE `JsonPath` provides expressions you can use to navigate a JSON object and extract data from it. Let's consider the `Book` object. If I wanted to get the `isbn` field from its JSON representation, I could use the following `JsonPath` expression: `@.isbn`. For more information on the `JsonPath` library, you can refer to the project documentation: <https://github.com/json-path/JsonPath>.

Listing 3.19 shows an example of both serialization and deserialization test.

Listing 3.19 Integration tests for the JSON slice (BookJsonTests.java)

```

package com.polarbookshop.catalogservice.web;

import com.polarbookshop.catalogservice.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;
import static org.assertj.core.api.Assertions.assertThat;

@JsonTest      ①
class BookJsonTests {

    @Autowired
    private JacksonTester<Book> json;      ②

    @Test
    void testSerialize() throws Exception {
        var book = new Book("1234567890", "Title", "Author", 9.90);
        var jsonContent = json.write(book);      ③
        assertThat(jsonContent).extractingJsonPathStringValue("$.isbn")
            .isEqualTo(book.isbn());
        assertThat(jsonContent).extractingJsonPathStringValue("$.title")
            .isEqualTo(book.title());
        assertThat(jsonContent).extractingJsonPathStringValue("$.author")
            .isEqualTo(book.author());
        assertThat(jsonContent).extractingJsonPathNumberValue("$.price")
            .isEqualTo(book.price());
    }

    @Test
    void testDeserialize() throws Exception {
        var content = """      ④
        {
            "isbn": "1234567890",
            "title": "Title",
            "author": "Author",
            "price": 9.90
        }
        """;
        assertThat(json.parse(content))      ⑤
            .usingRecursiveComparison()
            .isEqualTo(new Book("1234567890", "Title", "Author", 9.90));
    }
}

```

- ① Identifies a test class that focuses on JSON serialization.
- ② Utility class to assert JSON serialization and deserialization.
- ③ Verifying the parsing from Java to JSON, using the JsonPath format to navigate the JSON object.
- ④ Defines a JSON object using the Java text block feature.
- ⑤ Verifies the parsing from JSON to Java.

WARNING If you use IntelliJ IDEA, you might get a warning that `JacksonTester` cannot be autowired. Don't worry. It's a false positive. You can get rid of the warning by annotating the field with `@SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")`.

Then, run the tests with the following command.

```
$ ./gradlew test --tests BookJsonTests
```

In the code repository accompanying the book, you can find more examples of unit and integration tests for the Catalog Service project. After automating the tests for your application, it's time to automate its execution whenever a new feature or bug fix is delivered. The following section will introduce the key pattern of continuous delivery: the deployment pipeline.

3.5 Deployment pipeline: Build and test

As introduced in chapter 1, continuous delivery is a holistic approach for quickly, reliably, and safely delivering high-quality software. The primary pattern for adopting such an approach is the deployment pipeline, which goes from code commit to releasable software. It should be automated as much as possible and represent the only path to production.

Based on the concepts described by Jez Humble and Dave Farley in their "Continuous Delivery" book and by Dave Farley in his "Continuous Delivery Pipelines" book, we can identify a few key stages in a deployment pipeline.

- **Commit stage.** After a developer commits new code to the mainline, this stage goes through build, unit tests, integration tests, static code analysis, and packaging. At the end of this stage, an executable application artifact is published to an artifact repository. That is a *release candidate*. For example, it can be a JAR artifact published to a Maven repository or a container image published to a container registry. This stage supports the continuous integration practice. It's supposed to be fast, possibly under five minutes, to provide developers with fast feedback about their changes and allow them to move on to the next task.
- **Acceptance stage.** The publication of a new release candidate to the artifact repository triggers this stage, which consists of deploying the application to production-like environments and running additional tests to increase the confidence about its releasability. The tests running in the acceptance stage are usually slow, but we should strive to keep the whole deployment pipeline execution under one hour. Examples of tests included in this stage are functional acceptance tests and non-functional acceptance tests, such as performance tests, security tests, and compliance tests. If necessary, this stage can also include manual tasks like exploratory and usability tests. At the end of this stage, the release candidate is ready to be deployed to production at any time. If we are still not confident about it, this stage is missing some tests.
- **Production stage.** After a release candidate has gone through the commit and acceptance

stages, we are confident enough to deploy it to production. This stage is triggered manually or automatically, depending on whether it's been decided to adopt a continuous deployment practice. The new release candidate is deployed to a production environment using the same deployment scripts employed (and tested) in the acceptance stage. Optionally, some final automated tests can be run to verify the deployment was successful.

This section will guide you through bootstrapping a deployment pipeline for Catalog Service and defining the first steps in the commit stage. Then, I'll show you how to automate those steps using GitHub Actions.

3.5.1 Understanding the commit stage of the deployment pipeline

Continuous integration is a foundational practice of continuous delivery. When adopted successfully, developers work in small steps and commit to the mainline (the main branch) multiple times a day. After each code commit, the commit stage of the deployment pipeline takes care of building and testing the application with the new changes.

This stage should be fast because a developer would wait until it completes successfully before moving on to the next task. That's a critical point. If the commit stage fails, the developer responsible for it should immediately deliver a fix or revert their changes so not to leave the mainline in a broken state, and prevent all other developers from integrating their code.

Let's start designing a deployment pipeline for a cloud native application like Catalog Service. For now, we'll focus on the first few steps composing the commit stage (figure 3.8).

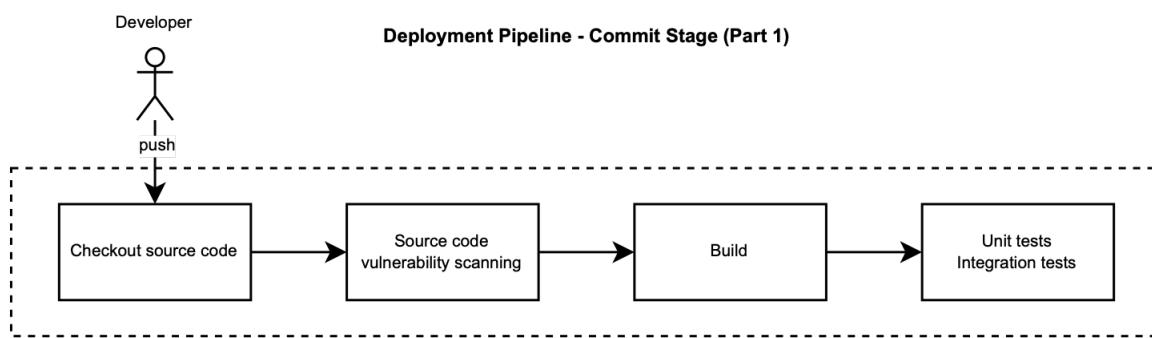


Figure 3.8 The first part of the commit stage in a deployment pipeline.

After a developer pushes new code to the mainline, the commit stage starts by checking out the source code from the repository. The starting point is always a commit to the main branch. Following the continuous integration practices, we aim to work in small steps and integrate our changes with the main branch multiple times a day (*continuously*).

Next, the pipeline can perform several types of static code analysis. For this example, we'll focus on vulnerability scanning. In a real-world project, you probably want to include additional steps, such as running static code analysis to identify security issues and check compliance with

specific coding standards (*code linting*).

Finally, the pipeline builds the application and runs automated tests. In the commit stage, we include technically-focused tests that don't require deploying the entire application to be executed. They are unit tests and often integration tests. If the integration tests take too long, it's better to move them to the acceptance stage to keep the commit stage fast.

The vulnerability scanner we'll use in the Polar Bookshop project is `grype` (<https://github.com/anchore/grype>), a powerful open-source tool increasingly used in the cloud native world. For example, it's part of the security supply chain solution provided by the VMware Tanzu Application Platform. You can find instructions on how to install it in section A.4 of Appendix A.

Let's see how it works. Open a Terminal window, navigate to the root folder of your Catalog Service project (`catalog-service`), and build the application with `./gradlew build`. Then, use `grype` to scan your Java codebase for vulnerabilities. The tool will download a list of known vulnerabilities (a *vulnerability database*) and scan your project against them. The scanning happens locally on your machine, which means none of your files or artifacts is sent to an external service. That makes it a good fit for more regulated environments or air-gapped scenarios.

```
$ grype .
Vulnerability DB      [updated]
Indexed .
Cataloged packages    [35 packages]
Scanned image          [0 vulnerabilities]

No vulnerabilities found
```

NOTE Remember that security is not a static property of a system. At the moment of writing, the dependencies used by Catalog Service have no known vulnerability. It doesn't mean that will be true forever. You should scan your projects continuously and apply security patches as soon as they are released to fix newly-discovered vulnerabilities.

Chapters 6 and 7 will cover the remaining steps in the commit stage. For now, let's see how to automate a deployment pipeline with GitHub Actions. That's the subject of the following section.

3.5.2 Implementing the commit stage with GitHub Actions

When it comes to automating deployment pipelines, there are many solutions from which you can choose. In this book, I'm going to use GitHub Actions (<https://github.com/features/actions>). It's a managed solution, it provides all the features we need for our project, and it's conveniently already configured for all GitHub repositories. I'm introducing this topic so early in the book so that you can use the deployment pipeline to verify your changes while working on the project throughout the book.

NOTE

In the cloud native ecosystem, Tekton (<https://tekton.dev>) is a popular choice for defining deployment pipelines and other software workflows. It's an open-source and Kubernetes-native solution hosted at the Continuous Delivery Foundation (<https://cd.foundation>). It runs directly on a cluster and lets you declare pipelines and tasks as Kubernetes custom resources.

GitHub Actions is a platform built into GitHub that lets you automate software workflows directly from your code repositories. A *workflow* is an automated process. We'll use it to model the commit stage of our deployment pipeline. Each workflow is listening to specific *events* that trigger its execution.

Workflows should be defined in a `.github/workflows` folder in a GitHub repository root and described following the YAML format provided by GitHub Actions. In your Catalog Service project (`catalog-service`), create a `commit-stage.yml` file under a new `.github/workflows` folder. This workflow will be triggered whenever new code is pushed to the repository.

Listing 3.20 Define workflow name and trigger (commit-stage.yml).

```
name: Commit Stage ①
on: push ②
```

- ① The name of the workflow.
- ② The workflow is triggered when new code is pushed to the repository.

Each *workflow* is organized into *jobs* that run in parallel. For now, we'll define one single job to collect the steps described previously in figure 3.8. Each job is executed on a *runner* instance, a server provided by GitHub. You can choose between Ubuntu, Windows, and macOS. For Catalog Service, we'll run everything on Ubuntu runners provided by GitHub. We also want to be specific regarding which *permissions* each job should have. The "Build and Test" job will need read access to the Git repository and write access to the security events when submitting a vulnerability report to GitHub.

Listing 3.21 Configure a job for building and testing the application (commit-stage.yml).

```
name: Commit Stage
on: push

jobs:
  build: ①
    name: Build and Test ②
    runs-on: ubuntu-22.04 ③
    permissions: ④
      contents: read ⑤
      security-events: write ⑥
```

- ① The job unique identifier.
- ② A human-friendly name for the job.
- ③ The type of machine where to run the job.
- ④ The permissions granted to the job.
- ⑤ Permission to check out the current Git repository.
- ⑥ Permission to submit security events to GitHub.

Each job is composed of *steps*, which are executed sequentially. A step could be either a shell command or an *action*. Actions are custom applications used to perform complex tasks in a more structured and reproducible way. For example, you could have actions for packaging an application into an executable, running tests, creating a container image, or pushing it to a container registry. The GitHub organization provides a basic set of actions, but there's also a marketplace with many more actions developed by the community.

WARNING When using actions from the GitHub marketplace, you should handle them like any other third-party application and manage the security risks accordingly. Prefer using trusted actions provided by GitHub or verified organizations over other third-party options.

Let's complete this first part of the commit stage by describing what steps the "Build and Test" job should run. The final result is as follows.

Listing 3.22 Implement the steps for building and testing the application (commit-stage.yml).

```

name: Commit Stage
on: push

jobs:
  build:
    name: Build and Test
    runs-on: ubuntu-22.04
    permissions:
      contents: read
      security-events: write
    steps:
      - name: Checkout source code
        uses: actions/checkout@v3 ①
      - name: Set up JDK
        uses: actions/setup-java@v3 ②
        with:
          distribution: temurin
          java-version: 17
          cache: gradle
      - name: Code vulnerability scanning
        uses: anchore/scan-action@v3 ④
        id: scan ⑤
        with:
          path: "${{ github.workspace }}" ⑥
          fail-build: false ⑦
          severity-cutoff: high ⑧
          acs-report-enable: true ⑨
      - name: Upload vulnerability report
        uses: github/codeql-action/upload-sarif@v2 ⑩
        if: success() || failure() ⑪
        with:
          sarif_file: ${{ steps.scan.outputs.sarif }} ⑫
      - name: Build, unit tests and integration tests
        run:
          | chmod +x gradlew ⑬
          | ./gradlew build ⑭

```

- ① Checks out the current Git repository ("catalog-service").
- ② Installs and configures a Java runtime.
- ③ Defines which version, distribution, and cache type to use.
- ④ Scans the codebase for vulnerabilities using grype.
- ⑤ Assigns an identifier to the current step so that it can be referenced from subsequent steps.
- ⑥ The path to the checked-out repository.
- ⑦ Whether to fail the build in case of security vulnerabilities.
- ⑧ The minimum security category to be considered as an error (low, medium, high, critical).
- ⑨ Whether to enable the generation of a report after the scan is completed.
- ⑩ Uploads the security vulnerability report to GitHub (SARIF format).
- ⑪ Uploads the report even if the previous step fails.

- ⑫ Fetches the report from the output of the previous step.
- ⑬ Ensures the Gradle wrapper is executable, solving Windows incompatibilities.
- ⑭ Runs the Gradle "build" task, which compiles the codebase and runs unit and integration tests.

WARNING The action uploading the vulnerability report requires the GitHub repository to be public. It works for private repositories only if you have an enterprise subscription. If you prefer keeping your repository private, you need to skip the "Upload vulnerability report" step. Throughout the book, I'll assume all the repositories we create on GitHub for the Polar Bookshop project are public.

After completing the declaration of the initial commit stage for the deployment pipeline, commit your changes and push them to the remote GitHub repository. The newly-created workflow will be immediately triggered. You can see the execution results on your GitHub repository page in the "Actions" tab. Figure 3.9 shows an example of results after running the workflow in listing 3.22. By keeping the result of the commit stage green, you can be quite sure that you haven't broken anything or introduced new regressions (assuming that you have proper tests in place).

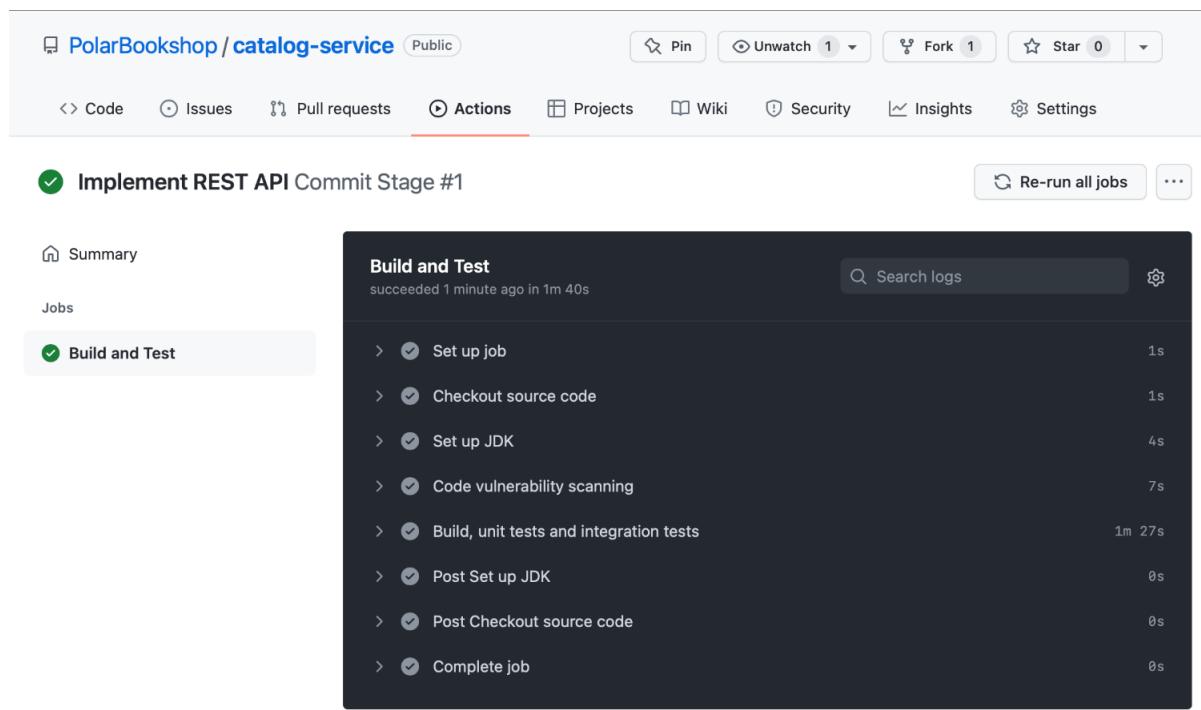


Figure 3.9 The commit stage workflow is executed after pushing new changes to the remote repository.

The step for running the vulnerability scan is based on an action provided by Anchore, the company behind `grype`. In listing 3.22, we are not failing the workflow if serious vulnerabilities are discovered. However, you can find the scanning results in the "Security" section of your

catalog-service GitHub repository. At the moment of writing, there is no high or critical vulnerability found in the Catalog Service project. Things might be different in the future. If that's the case, consider using the latest available security patches for the affected dependencies. For the sake of the example and because I didn't want to disrupt your learning journey, I decided not to make the build fail on found vulnerabilities. In a real-world scenario, I recommend you configure and tune `grype` carefully and according to your company policies regarding supply chain security, and make the workflow fail if the result is not compliant (setting the `fail-build` property to `true`). For more information, refer to the official project documentation (<https://github.com/anchore/grype>).

After scanning the Java project for vulnerabilities, we also added a step to fetch the security report generated by `grype` and upload it to GitHub, independently of whether the build succeeds. If any security vulnerability is found, you can see the results in the "Security" tab of your GitHub repository page (figure [3.10](#)).

Latest scan	Branch	Workflow	Duration	Result
now	main	Commit Stage	1s	115 alerts

Filters	Q is:open branch:main	Tool	Branch	Rule	Severity	Sort
<input type="checkbox"/> 115 Open	<input checked="" type="checkbox"/> 0 Closed					
<input type="checkbox"/> GHSA-p6xc-xr62-6r2g	High vulnerability for log4j-api package	(Library) .../lib/log4j-api-2.14.1.jar:1	Detected now by Anchore Container Vulnerability Report (T0)	<input type="checkbox"/> Error	main	
<input type="checkbox"/> GHSA-m72m-mhq2-9p6c	High vulnerability for jsoup package	(Library) .../lib/jsoup-1.12.1.jar:1	Detected now by Anchore Container Vulnerability Report (T0)	<input type="checkbox"/> Error	main	
<input type="checkbox"/> GHSA-jfh8-c2jp-5v3q	Critical vulnerability for log4j-api package	(Library) .../lib/log4j-api-2.14.1.jar:1	Detected now by Anchore Container Vulnerability Report (T0)	<input type="checkbox"/> Error	main	
<input type="checkbox"/> GHSA-gwprp-pvrq-jmwv	Medium vulnerability for commons-io package	(Library) .../lib/commons-io-2.6.jar:1	Detected now by Anchore Container Vulnerability Report (T0)	<input type="checkbox"/> Error	main	

Figure 3.10 The security vulnerability report generated by `grype` and published to GitHub.

NOTE

At the moment of writing, `grype` doesn't find any vulnerability in the codebase provided with the book. To show you an example of a vulnerability report in figure [3.10](#), I let `grype` scan a different version of the project, purposely full of known vulnerabilities.

That's it for this chapter. Next, I'll cover one of the main cloud native development practices: externalized configuration.

3.6 Summary

- Each cloud native application should be tracked in its own codebase, and all the dependencies should be declared in a manifest using tools like Gradle or Maven.
- Cloud native applications don't depend on servers being injected into the environment. Instead, they use an embedded server and are self-contained.
- Tomcat is the default embedded server for Spring Boot applications and can be configured through properties to customize the ports it's listening on, connections, timeouts, and threads.
- The request/response interaction provided by a Servlet container like Tomcat is both synchronous and blocking. Each thread handles an HTTP request until a response is returned.
- The "API first" principle recommends designing the API before implementing the business logic to establish a contract. In this way, other teams can develop their services to consume your application based on the contract itself without waiting for the application to be finished.
- In Spring MVC, REST APIs are implemented in `@RestController` classes.
- Each REST controller method handles an incoming request with a specific method (GET, POST, PUT, DELETE) and endpoint (e.g., `/books`).
- Controller methods can declare endpoints and operations they handle through the annotations `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@RequestMapping`.
- Methods of a `@RestController` class can validate the HTTP request body before the processing happens by applying the `@Valid` annotation.
- The validation constraints for a given Java object are defined using the annotations from the Java Bean Validation API on the fields. For example, `@NotBlank`, `@Pattern`, `@Positive`.
- Java Exceptions thrown during the processing of an HTTP request can be mapped to an HTTP status code and body in a centralized `@RestControllerAdvice` class, decoupling the exception handling for the REST API from the code throwing the exception.
- Unit tests are not aware of the Spring configuration but can be written as standard Java tests using familiar tools like JUnit, Mockito, and AssertJ.
- Integration tests need a Spring application context to run. A full application context, including an optional embedded server, can be initialized for testing using the `@SpringBootTest` annotation.
- When tests are focused only on a "slice" of the application and only need a part of the configuration, Spring Boot provides several annotations for more targeted integration tests. When using those annotations, a Spring application context is initialized, but only the components and configuration parts used by the specific functionality slice are loaded.
 - `@WebMvcTest` is for testing Spring MVC components.
 - `@JsonTest` is for testing JSON serialization and deserialization.
 - GitHub Actions is a tool provided by GitHub to declare pipelines (or workflows) to automate tasks. It can be used to build a deployment pipeline.

Externalized configuration management



This chapter covers

- Configuring Spring with properties and profiles
- Applying external configuration with Spring Boot
- Implementing a configuration server with Spring Cloud Config Server
- Configuring applications with Spring Cloud Config Client

In the previous chapter, we built a RESTful application for managing a catalog of books. As part of the implementation, we defined some data to configure certain aspects of the application (in an `application.yml` file), for example, the Tomcat thread pool or connection timeout. The next step might be to deploy the application to different environments: first in a test environment, then staging, and finally in production. What if you needed a different Tomcat configuration for each of these environments? How would you achieve that?

Traditional applications were usually packaged as a bundle, including the source code and a series of configuration files containing data for different environments and selected through a flag at runtime. The implication was that you had to make a new application build every time you needed to update configuration data for a specific environment. A variant to this process was to create a different build for each environment, meaning that you had no guarantee whether what you ran in a staging environment would have worked in the same way in production because they would be different artifacts.

Configuration is defined as everything likely to change between deployments (as per the 15-Factor methodology), like credentials, resource handlers, and URLs to backing services. Depending on where it is deployed, the same application will have different needs and will most

likely require a different configuration. A key aspect of cloud native applications is that the application artifact will stay immutable across environments. No matter which environment you deploy it to, the application build will not be changed.

Each release you deploy is a combination of build and configuration. The same build can be deployed to different environments with different configuration data, as shown in figure 4.1.

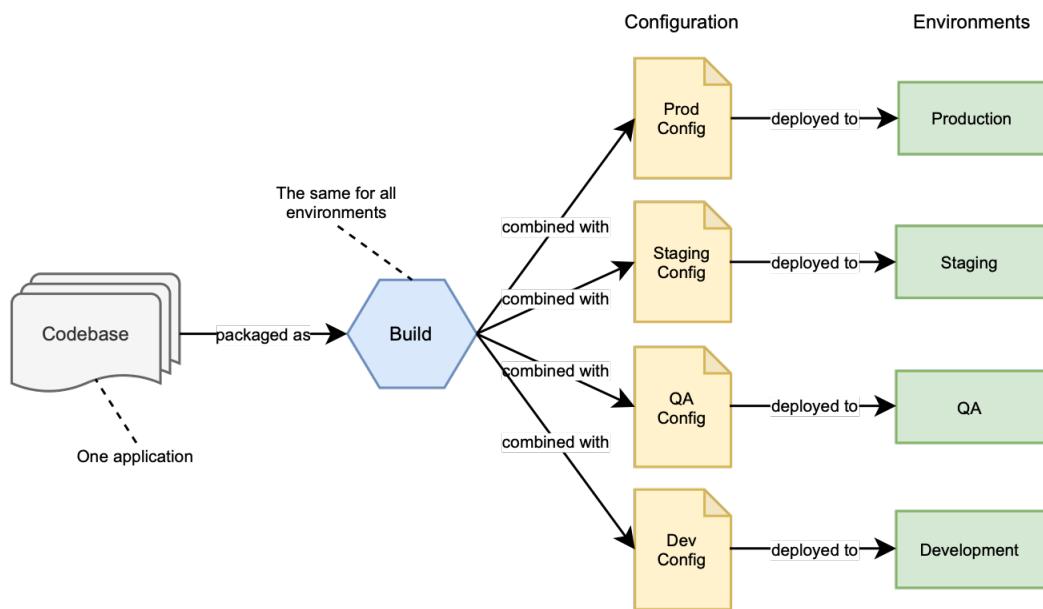


Figure 4.1 Each release you deploy is a combination of build and configuration, which is different for each environment.

Anything that might need to change across deployments should be configurable. For example, you probably want to change credentials to access backing services, feature flags, resource handles to databases, or URLs to external APIs, all depending on the environment to which you’re deploying the application. Cloud native applications favor externalized configuration so that you can replace it without having to rebuild your code. Regarding credentials, it’s even more critical not to store them with your application code. There have been countless data breaches because of companies including credentials in a repository that went public by accident. Make sure you won’t be one of them.

In Spring, configuration data are abstracted as properties, key/value pairs defined in different sources, such as property files, JVM system properties, and system environment variables. This chapter covers various aspects of configuring Spring applications that are relevant in a cloud native context. I’ll present the main concepts about how Spring handles configuration, including properties and profiles, and how to apply externalized configuration with Spring Boot. Then, I’ll show you how to set up a configuration server with Spring Cloud Config Server, using a Git repository as the backend for storing your configuration data. Finally, you’ll learn how to use the config server to configure a Spring Boot application by relying on Spring Cloud Config Client.

By the end of this chapter, you'll be able to configure your cloud native Spring applications in different ways, depending on your needs and the type of configuration data. Table 4.1 summarizes the three main strategies covered in this chapter to define configuration data for cloud native applications. Chapter 13 will further extend the subjects covered here, including secrets management and how to use ConfigMaps and Secrets in Kubernetes.

NOTE

The source code for the examples in this chapter is available in the Chapter04/04-begin and Chapter04/04-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

Table 4.1 Cloud native applications can be configured according to different strategies. You will likely use them all, depending on the type of configuration data and application requirements.

Configuration Strategy	Characteristics
Property files packaged with the application	<ul style="list-style-type: none"> Can act as specifications of what configuration data the application supports. Useful to define sensible default values, mainly oriented to the development environment.
Environment variables	<ul style="list-style-type: none"> Environment variables are supported by any operating system, so they are great for portability. Most programming languages allow you access to the environment variables. In Java, you do that with the <code>System.getenv()</code> method. In Spring, you can also rely on the <code>Environment</code> abstraction. Useful to define configuration data depending on the infrastructure/platform where the application is deployed, such as active profiles, hostnames, service names, and port numbers.
Configuration service	<ul style="list-style-type: none"> Provides configuration data persistence, auditing, and accountability. Allows secrets management by using encryption or dedicated secret vaults. Useful to define configuration data specific to the application, such as connection pools, credentials, feature flags, thread pools, and URLs to third-party services.

4.1 Configuration in Spring: properties and profiles

The term *configuration* can have different meanings depending on the context. When discussing the Spring Framework's core features and its `ApplicationContext`, configuration refers to which beans (i.e., Java objects registered in Spring) have been defined to be managed by the Spring container and injected where needed. For example, you can define beans in an XML file (XML configuration), in a `@Configuration` class (Java configuration), or by relying on annotations like `@Component` (annotation-driven configuration).

In this book, unless specified otherwise, whenever I mention *configuration*, I don't mean the previous concept, but rather everything likely to change between deployments, just as defined by the 15-Factors methodology.

Spring provides you with a convenient `Environment` abstraction that gives you access to any configuration data, no matter its source. The two key aspects of a Spring application environment are properties and profiles. You have already worked with properties in the previous chapter. Profiles are a tool to label logical groups of beans or configuration data loaded at runtime only if a given profile is enabled. Figure 4.2 shows the main aspects of a Spring application environment.

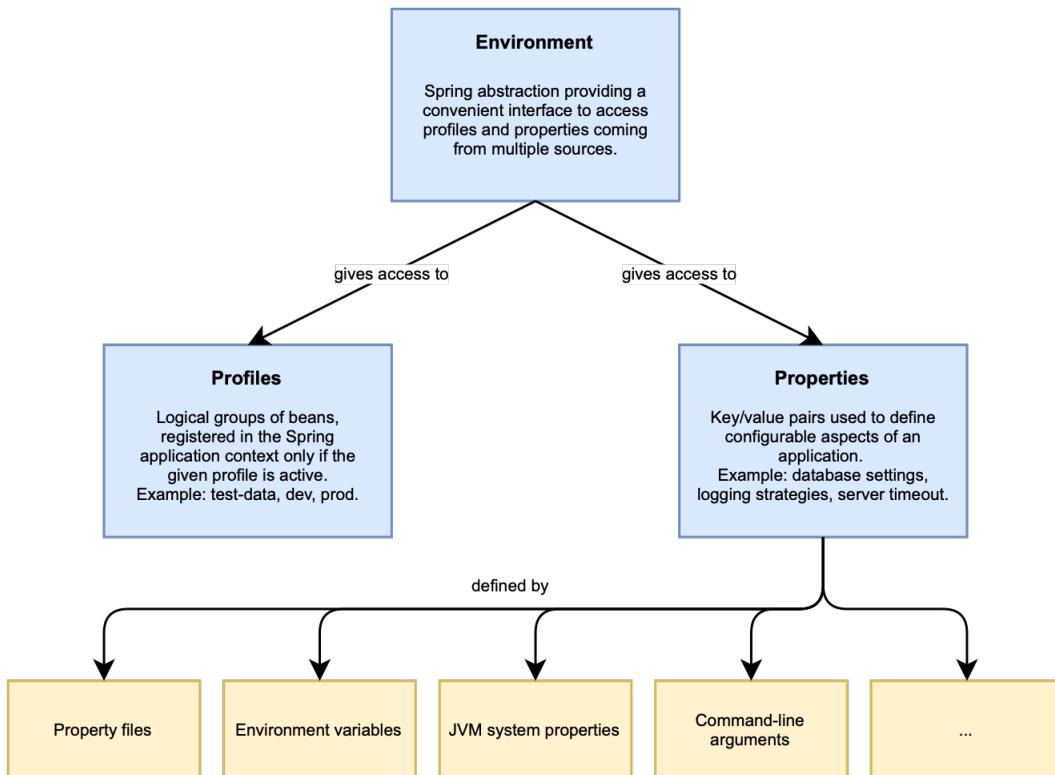


Figure 4.2 The `Environment` interface gives access to the two key aspects of any Spring application configuration: properties and profiles.

This section will cover the fundamental aspects of properties and profiles for cloud native

applications, including how to define custom properties and when to use profiles.

4.1.1 Properties: Key/value pairs for configuration

Properties are key/value pairs supported in Java as first-class citizens with `java.util.Properties`. They play an essential role in many applications to store configuration parameters outside of the compiled Java code. And Spring Boot automatically loads them from different sources. When the same property is defined in multiple sources, there are rules to decide which one takes precedence. For example, if you specify a value for the `server.port` property both in a property file and a command-line argument, the latter takes precedence over the former. The following is a prioritized list of some of the most common property sources, starting from the one with the highest priority. For the complete list, you can refer to the Spring Boot documentation (<https://spring.io/projects/spring-boot>).

1. `@TestPropertySource` annotations on test classes.
2. Command-line arguments.
3. JVM System properties from `System.getProperties()`.
4. OS environment variables from `System.getenv()`.
5. Configuration data files.
6. `@PropertySource` annotations on `@Configuration` classes.
7. Default properties from `SpringApplication.setDefaultProperties`.

Configuration data files can be further prioritized, starting from the one with the highest priority.

1. Profile-specific application properties from `application-{profile}.properties` and `application-{profile}.yml` packaged outside your JAR.
2. Application properties from `application.properties` and `application.yml` packaged outside your JAR.
3. Profile-specific application properties from `application-{profile}.properties` and `application-{profile}.yml` packaged inside your JAR.
4. Application properties from `application.properties` and `application.yml` packaged inside your JAR.

The beauty of property handling in Spring is that you don't need to know the specific source to get a value: the `Environment` abstraction lets you access any property defined in any source through a unified interface. And if the same property is defined in multiple sources, it returns the one with the highest priority. You can even add your own custom sources and assign a priority to them.

NOTE

Spring Framework has built-in support for properties defined according to the Properties format. On top of that, Spring Boot adds support for defining properties using the YAML format. YAML is a superset of JSON that provides more flexibility than the simpler Properties format. The official website (<https://yaml.org>) describes YAML as "a human-friendly data serialization standard for all programming languages". Feel free to choose either of them in your applications. All the examples in the book will be using YAML.

USING APPLICATION PROPERTIES

There are a few ways to access properties from a Java class, as shown in figure 4.3. The most generic approach is based on the `Environment` interface that you can autowire wherever you need access to an application property. For example, you can use it to access the value for the `server.port` property as follows.

```
@Autowired
private Environment environment;

public String getServerPort() {
    return environment.getProperty("server.port");
}
```

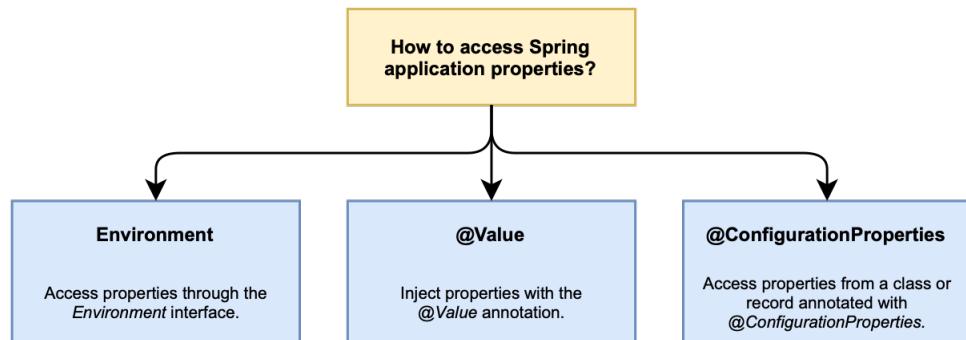


Figure 4.3 You can access Spring properties in different ways.

Properties can also be injected without explicitly invoking the `Environment` object. Just like you use the `@Autowired` annotation for injecting Spring beans, you can apply the `@Value` annotation to inject property values.

```
@Value("${server.port}")
private String serverPort;

public String getServerPort() {
    return serverPort;
}
```

You can configure the application using properties without hard-coding values inside the code, one of our goals. But when using the `Environment` object or the `@value` annotation, you still

have a hard-coded value that can become difficult to manage: the property key. A more robust and maintainable option, which is also the one recommended by the Spring team, is using special beans marked with the `@ConfigurationProperties` annotation to hold configuration data. You'll explore this feature in the next section while learning how to define custom properties.

DEFINING CUSTOM PROPERTIES

Spring Boot comes bundled with tons of properties to configure any aspect of your application, depending on which starter dependency you import into your project. Sooner or later, though, it comes the time when you need to define your own properties.

Let's consider the Catalog Service application we have been working on so far. In chapter 2, we defined an HTTP endpoint returning a welcome message to users. We now have a new requirement to implement: the welcome message should be configurable. It is probably not the most useful functionality, but it will help me show you the different configuration options.

The first thing to do is tell Spring Boot to scan the application context for configuration data beans. We can do so by adding the `@ConfigurationPropertiesScan` annotation to the `CatalogServiceApplication` class in your Catalog Service project (`catalog-service`).

Listing 4.1 Enables scanning of configuration data beans (CatalogServiceApplication.java)

```
package com.polarbookshop.catalogservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties
.ConfigurationPropertiesScan;

@SpringBootApplication
@ConfigurationPropertiesScan ①
public class CatalogServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(CatalogServiceApplication.class, args);
    }
}
```

- ① Loads configuration data beans in the Spring context.

NOTE

Instead of making Spring scan the application context searching for configuration data beans, you can directly specify which ones Spring should consider by using the `@EnableConfigurationProperties` annotation.

Then, define a new `com.polarbookshop.catalogservice.config` package, and create a `PolarProperties` class annotated with `@ConfigurationProperties` to mark it as a holder of configuration data. The `@ConfigurationProperties` annotation takes a `prefix` argument, combined with a field name to produce the final property key. Spring Boot will try to map all properties with that prefix to fields in the class. In this case, there's only one property mapped to

the bean: `polar.greeting`. Optionally, you can add a description for each property using JavaDoc comments that can be converted into metadata, as I'll show you in a minute.

Listing 4.2 Define custom properties in a Spring bean (PolarProperties.java)

```
package com.polarbookshop.catalogservice.config;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "polar") ①
public class PolarProperties {
    /**
     * A message to welcome users.
     */
    private String greeting; ②

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}
```

- ① Marks the class as a source for configuration properties starting with the prefix "polar".
- ② Field for the custom "polar.greeting" (prefix + field name) property parsed as String.

Optionally, you can add a new dependency on the Spring Boot Configuration Processor in your `build.gradle` file. That will automatically generate metadata for the new properties and store them in `META-INF/spring-configuration-metadata.json` when building the project. The IDE can pick them up, show you the description message for each property, and help you with auto-completion and type checks. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 4.3 Add dependency for the Spring Boot Configuration Processor (build.gradle)

```
configurations { ①
    compileOnly {
        extendsFrom annotationProcessor
    }
}

dependencies {
    ...
    annotationProcessor
    'org.springframework.boot:spring-boot-configuration-processor'
}
```

- ① Configures Gradle to use the annotation processor when building the project.

Then, you can trigger the metadata generation by building your project (`./gradlew clean build`). At this point, go ahead and define a default value for the `polar.greeting` property in

the `application.yml` file. While inserting the new property, your IDE should provide you with an autocompletion option and type check, as shown in figure 4.4.

Listing 4.4 Define a value for our custom property in Catalog Service (application.yml)

```
polar:
  greeting: Welcome to the local book catalog!
```

In listing 4.2, the `greeting` field will be mapped to the `polar.greeting` property for which you have just defined a value in `application.yml`.

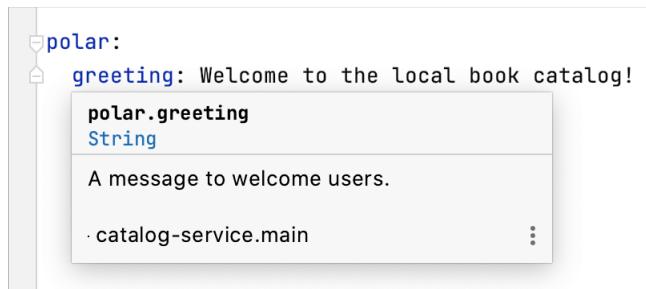


Figure 4.4 Using the Spring Boot annotation processor, the JavaDoc comment from your custom property bean is converted into metadata used by your IDE to provide useful information, autocompletion, and type check.

USING CUSTOM PROPERTIES

Classes or records annotated with `@ConfigurationProperties` are standard Spring beans, so you can inject them wherever you need them. Spring Boot initializes all the configuration beans at startup and populates them with the data provided through any of the supported configuration data sources. In the case of Catalog Service, the data will be populated from the `application.yml` file.

The new requirement for Catalog Service is to make the welcome message returned by the root endpoint configurable via the `polar.greeting` property. Open the `HomeController` class and update the handler method to get the message from the custom property instead of using a fixed value.

Listing 4.5 Using custom properties from a configuration properties bean (HomeController.java)

```
package com.polarbookshop.catalogservice;

import com.polarbookshop.catalogservice.config.PolarProperties;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HomeController {
    private final PolarProperties polarProperties; ①

    public HomeController(PolarProperties polarProperties) {
        this.polarProperties = polarProperties;
    }

    @GetMapping("/")
    public String getGreeting() {
        return polarProperties.getGreeting(); ②
    }
}
```

- ① Bean to access the custom properties injected via constructor autowiring.
- ② Uses the welcome message from the configuration data bean.

You can now build and run the application to verify it works as intended (`./gradlew bootRun`). Then, open a Terminal window, and send a GET request to the root endpoint exposed by Catalog Service. The result should be the message you configured in `application.yml` for the `polar.greeting` property.

```
$ http :9001/
Welcome to the local book catalog!
```

NOTE Property files packaged with your application code are useful for defining sensible default values for your configuration data. They can also act as specifications for what configuration properties your application supports.

The following section will cover the other key aspect modeled by the Spring Environment abstraction, profiles, and how to use them for cloud native applications. Before moving on, you can stop the application with `Ctrl+C`.

4.1.2 Profiles: Feature flags and configuration groups

Sometimes, you might want to load a bean into the Spring context only under specific conditions. For example, you might want to define a bean responsible for generating test data only when working locally or testing the application. Profiles are logical groups of beans loaded into the Spring context only if the specified profile is active. You can activate zero, one, or more profiles at a time. All beans not assigned to a profile will always be activated. Beans assigned to the `default` profile are only activated when no other profile is active. Spring Boot extends this concept to property files, allowing to define groups of configuration data that are loaded only if a specific profile is active.

This section presents Spring profiles in the context of two different use cases: feature flags and configuration groups.

USING PROFILES AS FEATURE FLAGS

The first use case for profiles is for loading groups of beans only if the specified profile is active. The deployment environment shouldn't influence the reasoning behind the grouping too much. A common mistake is using profiles like `dev` or `prod` to load beans conditionally. If you do that, the application will be coupled with the environment, which is usually not what we want for a cloud native application.

Consider the case where you deploy applications to three different environments (development, test, production) and define three profiles accordingly to load certain beans conditionally (`dev`, `test`, `prod`). At some point, you decide to add a staging environment, where you also want to enable the beans marked with the `prod` profile. What do you do? You have two options. Either you activate the `prod` profile also in the staging environment (which doesn't make much sense), or you update the source code to add a `staging` profile and assign it to the beans marked as `prod` (which prevents your application from being immutable and deployable to any environment without any change to the source code).

Instead, I recommend using profiles as feature flags when associated with groups of beans to be loaded conditionally. Consider which functionality a profile provides and name it accordingly rather than thinking about where it will be enabled. However, there might be cases where a bean handling infrastructural concerns is required in specific platforms. For example, you might have certain beans that should only be loaded when the application is deployed to a Kubernetes environment (no matter if staging or production). In that case, you could define a `kubernetes` profile.

In chapter 3, we built the Catalog Service application to manage books. Whenever you run it locally, there are no books in the catalog yet, and you need to add some explicitly if you want to work with the application. A better option would be to let the application generate some test data

at startup, but only when needed (for example, in a development or test environment). Loading test data can be modeled as a feature that you enable/disable through configuration. You can define a `testdata` profile to toggle the test data loading. In that way, you keep profiles independent from the deployment environment and aim to use them as feature flags with no constraint on the deployment environment whatsoever. Let's do that.

First, add a new `com.polarbookshop.catalogservice.demo` package to your Catalog Service project and create a `BookDataLoader` class. You can instruct Spring to load this class only when the `testdata` profile is active by applying the `@Profile` annotation. Then, you can use the `BookRepository` we implemented in chapter 3 to save the data. Finally, the `@EventListener(ApplicationReadyEvent.class)` annotation will trigger the test data generation after the application has completed the startup phase.

Listing 4.6 Load book test data when the `testdata` profile is active (`BookDataLoader.java`)

```
package com.polarbookshop.catalogservice.demo;

import com.polarbookshop.catalogservice.domain.Book;
import com.polarbookshop.catalogservice.domain.BookRepository;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
@Profile("testdata") ①
public class BookDataLoader {
    private final BookRepository bookRepository;

    public BookDataLoader(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @EventListener(ApplicationReadyEvent.class) ②
    public void loadBookTestData() {
        var book1 = new Book("1234567891", "Northern Lights",
            "Lyra Silverstar", 9.90);
        var book2 = new Book("1234567892", "Polar Journey",
            "Iorek Polarson", 12.90);
        bookRepository.save(book1);
        bookRepository.save(book2);
    }
}
```

- ① Assigns the class to the "testdata" profile: it will be registered only when the "testdata" profile is active.
- ② The test data generation is triggered when an `ApplicationReadyEvent` is sent, that is when the application startup phase is completed.

In your development environment, you can use the `spring.profiles.active` property to set the `testdata` profile as active. You could set it in the `application.yml` file for the Catalog Service project, but it's not ideal to have the test data feature enabled by default. What if you forgot to overwrite it in production? A better option is configuring it specifically for the local

development environment when running the `bootRun` task. You can achieve that by adding the following to the `build.gradle` file.

Listing 4.7 Define the active profiles for the development environment (build.gradle)

```
bootRun {
    systemProperty 'spring.profiles.active', 'testdata'
}
```

Let's verify if it works. Build and run the application (`./gradlew bootRun`). First of all, you can see a message in the application logs listing all the active profiles (in this case, it's just `testdata`, but there can be more), as shown in figure 4.5.

```

:: Spring Boot ::      (v2.7.0)

2022-06-05 15:39:01.091 INFO 49597 --- [           main] c.p.c.CatalogServiceApplication        : Starting CatalogServiceApplication using Java 17.0.3
2022-06-05 15:39:01.092 INFO 49597 --- [           main] c.p.c.CatalogServiceApplication        : The following 1 profile is active: "testdata"
2022-06-05 15:39:01.358 INFO 49597 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9001 (http)
2022-06-05 15:39:01.361 INFO 49597 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-06-05 15:39:01.361 INFO 49597 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.63]
2022-06-05 15:39:01.387 INFO 49597 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[]       : Initializing Spring embedded WebApplicationContext
2022-06-05 15:39:01.387 INFO 49597 --- [           main] w.s.c.WebServerApplicationContext        : Root WebApplicationContext: initialization completed in 276 ms
2022-06-05 15:39:01.496 INFO 49597 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9001 (http) with context path ''
2022-06-05 15:39:01.501 INFO 49597 --- [           main] c.p.c.CatalogServiceApplication        : Started CatalogServiceApplication in 0.529 seconds (JVM running for 0.731)

```

Figure 4.5 The logs for Catalog Service when the "testdata" profile is active

Then, send a request to the application to fetch all the books in the catalog. It should return the test data we created in listing 4.6.

```
$ http :9001/books
```

When you're done, stop the application with `Ctrl+C`.

NOTE

Instead of using profiles as feature flags, a more scalable and structured approach is defining custom properties to configure functionality and relying on annotations such as `@ConditionalOnProperty` and `@ConditionalOnCloudPlatform` to control when certain beans should be loaded into the Spring application context. That's one of the foundation of Spring Boot auto-configuration. For example, you could define a `polar.testdata.enabled` custom property and use the `@ConditionalOnProperty(name = "polar.testdata.enabled", havingValue = "true")` annotation on the `BookDataLoader` class.

Next, I'll show you how to use profiles to group configuration data.

USING PROFILES AS CONFIGURATION GROUPS

The Spring Framework's profile functionality allows you to register some beans only if a given profile is active. Likewise, Spring Boot lets you define configuration data to be loaded only when a specific profile is active. A common way to do that is inside a property file named with the profile as a suffix. Considering the Catalog Service again, you could create a new `application-dev.yml` file and define a value for the `polar.greeting` property used by Spring Boot only if the `dev` profile is active. Profile-specific property files take precedence over the non-specific ones, so the value defined in `application-dev.yml` would take precedence over the one in `application.yml`.

In this context, profiles are used to group configuration data and can be mapped to the deployment environments without facing the same issues affecting the use case analyzed previously. That is true as long as you don't package the profile-specific property files with the application. The 15-Factors methodology recommends not batching configuration values into groups named after environments and bundled with the application source code. The reason is that it wouldn't scale: as a project grows, new environments might be created for different stages. Developers might create their own custom environments to try out new functionality. You can quickly end up with way too many configuration groups, implemented like Spring profiles and requiring new builds. Instead, you want to keep them outside the application, for example, in a dedicated repository served by a configuration server, as you'll see later in this chapter. The only exception is for defaults and development-oriented configuration.

The following section will cover how Spring Boot addresses externalized configuration and teach you how to use command-line arguments, JVM system properties, and environment variables to provide configuration data from the outside while using the same application build.

4.2 Externalized configuration: one build, multiple configurations

Property files bundled with the application source code are useful for defining some sensible defaults. Still, when you need to provide different values depending on the environment, you need something else. Externalized configuration allows you to configure your application depending on where it's deployed while consistently using the same immutable build for your application code. The critical aspect is that you won't change your application anymore after you build and package it. If any configuration change is needed (for example, different credentials or database handles), it's done from the outside.

The 15-Factor methodology promotes storing configuration in the environment, and Spring Boot provides several ways to achieve that. You can use one of the higher priority property sources to override the default values depending on where the application is deployed. In this section, you'll

see how to use command-line arguments, JVM properties, and environment variables for configuring a cloud native application without rebuilding it. Figure 4.6 illustrates how the precedence rules work to override a Spring property.

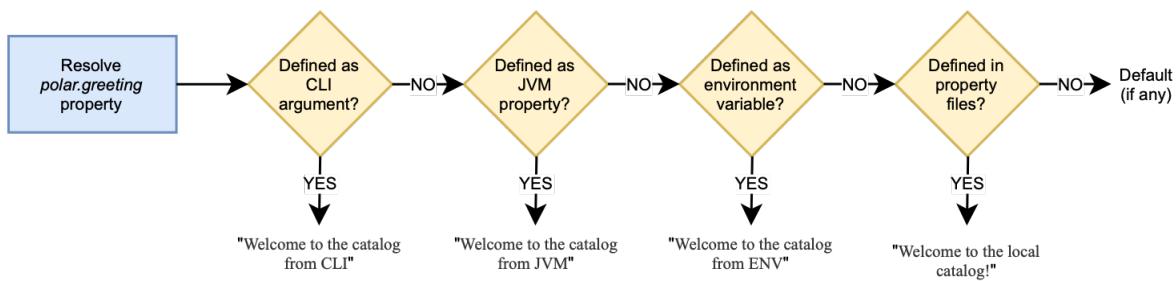


Figure 4.6 Spring Boot evaluates all property sources according to a precedence list. In the end, each property will have the value defined from the source with the highest precedence.

Let's consider the Catalog Service application. First, you need to package the application as a JAR artifact. You can do so from a Terminal window, navigating to the project root folder and running the following command.

```
$ ./gradlew bootJar
```

We're not relying on Gradle to run the application this time because I want to demonstrate how to change the application configuration while using the same, immutable JAR artifact (that is, without the need to rebuild the application). You can run it as a standard Java application.

```
$ java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar
```

You haven't overridden any property yet, so the root endpoint will return the `polar.greeting` value defined in the `application.yml` file.

```
$ http :9001/
Welcome to the local book catalog!
```

In the following sections, you'll see how to provide a different value for the `polar.greeting` property. Remember to terminate the Java process (`ctrl+C`) before moving on to new examples.

4.2.1 Configuring an application through command-line arguments

By default, Spring Boot converts any command-line argument to a property key/value pair and includes it in the `Environment` object. In a production application, that is the property source with the highest precedence. Using the same JAR you built earlier, you can specify a command-line argument to customize the application configuration.

```
$ java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar \
--polar.greeting="Welcome to the catalog from CLI"
```

The command-line argument has the same name as the Spring property, prefixed with the

familiar -- for CLI arguments. This time, the application will use the message defined as a command-line argument since it takes precedence over property files.

```
$ http :9001/
Welcome to the catalog from CLI
```

4.2.2 Configuring an application through JVM system properties

JVM system properties can override Spring properties similarly to command-line arguments but with a lower priority. It's all part of externalizing configuration, so you don't need to build a new JAR artifact: you can still use the one packaged earlier. Terminate the Java process from the previous example (`Ctrl+C`) and run the following command.

```
$ java -Dpolar.greeting="Welcome to the catalog from JVM" \
-jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar
```

The JVM system property has the same name as the Spring property, prefixed by the usual `-D` for JVM arguments. This time, the application will use the message defined as a JVM system property since it takes precedence over property files.

```
$ http :9001/
Welcome to the catalog from JVM
```

What if you specify both a JVM system property and a CLI argument? Then, the precedence rules will make Spring use the value specified as a command-line argument, since it takes precedence over JVM properties. Once again, terminate the previous Java process (`Ctrl+C`) and run the following command.

```
$ java -Dpolar.greeting="Welcome to the catalog from JVM" \
-jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar \
--polar.greeting="Welcome to the catalog from CLI"
```

The result, as you can imagine, will be:

```
$ http :9001/
Welcome to the catalog from CLI
```

Both CLI arguments and JVM properties let you externalize the configuration and keep the application build immutable. However, they require a different command to run the application, which might result in errors at deployment time. A better approach is using environment variables, as recommended by the 15-Factor methodology. Before moving to the next section, terminate the current Java process (`Ctrl+C`).

4.2.3 Configuring an application through environment variables

Environment variables defined in the operating system are commonly used for externalized configuration, and they are the recommended option according to the 15-Factor methodology. One of the advantages of environment variables is that any operating system supports them, making them portable across any environment. Furthermore, most programming languages provide features to access environment variables. For example, in Java, you can do that by calling the `System.getenv()` method.

In Spring, you are not required to read environment variables from the surrounding system explicitly. Spring automatically reads them during the startup phase and adds them to the Spring `Environment` object, making them accessible, just like any other property. For example, if you run a Spring application in an environment where the `MY_ENV_VAR` variable is defined, you can access its value either from the `Environment` interface or using the `@Value` annotation.

On top of that, Spring Boot extends the Spring Framework functionality by allowing you to use environment variables to override Spring properties automatically. For command-line arguments and JVM system properties, you used the same naming convention as the Spring property. However, environment variables have some naming constraints dictated by the operating system. For example, on Linux, the common syntax consists of having all capital letters and words separated by an underscore.

You can turn a Spring property key into an environment variable by making all letters uppercase and replacing any dot or dash with an underscore. Spring Boot will map it correctly to the internal syntax. For example, a `POLAR_GREETING` environment variable is recognized as the `polar.greeting` property. This feature is called *relaxed binding*.

In the Catalog Service application, you can override the `polar.greeting` property with the following command.

```
$ POLAR_GREETING="Welcome to the catalog from ENV" \
  java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar
```

TIP

On Windows, you can achieve the same result by running
`$env:POLAR_GREETING="Welcome to the catalog from ENV"; java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar.` **from a PowerShell console.**

During the startup phase of Catalog Service, Spring Boot will read the variables defined in the surrounding environment, recognize that `POLAR_GREETING` can be mapped to the `polar.greeting` property and store its value in the Spring `Environment`, overriding the one defined in `application.yml`.

The result will be:

```
$ http :9001/
Welcome to the catalog from ENV
```

After testing the application, stop the process with `Ctrl+C`. If you ran the application from the Windows PowerShell, remember to unset the environment variable with `Remove-Item Env:\Polar_Greeting`.

When you use environment variables for storing configuration data, you don't have to change the command to run your application (as you did for CLI arguments and JVM properties). Spring will automatically read environment variables from the context where it's deployed. This approach is less error-prone and fragile than using CLI arguments or JVM system properties.

NOTE

You can use environment variables to define values for your configuration data depending on the infrastructure or platform where the application is deployed, such as profiles, port numbers, IP addresses, and URLs.

Environment variables work seamlessly on virtual machines, OCI containers, and Kubernetes clusters. However, they might not be enough. In the next section, I'll go through some of the issues affecting environment variables and how Spring Cloud Config can help address them.

4.3 Centralized configuration management with Spring Cloud Config Server

With environment variables, you can externalize your application's configuration and follow the 15-Factor methodology. However, there are some issues they cannot handle:

- Configuration data is as important as the application code, so it should be handled with the same care and attention, starting from its persistence. Where should you store configuration data?
- Environment variables don't provide granular access control features. How can you control access to configuration data?
- Configuration data will evolve and require changes, just like application code. How should you keep track of the revisions to configuration data? How should you audit the configuration used in a release?
- After changing your configuration data, how can you make your application read it at runtime without requiring a full restart?
- When the number of application instances increases, it can get challenging to handle configuration in a distributed fashion for each instance. How can you overcome such challenges?
- Neither Spring Boot properties nor environment variables support configuration encryption, so you can't safely store passwords. How should you manage secrets?

The Spring ecosystem offers many options to address those issues. We can categorize them into

three groups.

- **Configuration services.** The Spring Cloud project provides modules you can consider to run your own configuration services and use them to configure your Spring Boot applications.
 - Spring Cloud Alibaba provides a configuration service using Alibaba Nacos as the data store.
 - Spring Cloud Config provides a configuration service backed by a pluggable data source, such as a Git repository, a data store, or HashiCorp Vault.
 - Spring Cloud Consul provides a configuration service using Hashicorp Consul as the data store.
 - Spring Cloud Vault provides a configuration service using Hashicorp Vault as the data store.
 - Spring Cloud Zookeeper provides a configuration service using Apache Zookeeper as the data store.
- **Cloud vendor services.** If you run your applications on a platform provided by a cloud vendor, you might consider using one of their configuration services. Spring Cloud provides integration with the main cloud vendor configuration services you can use to configure your Spring Boot applications.
 - Spring Cloud AWS provides integration with AWS Parameter Store and AWS Secrets Manager.
 - Spring Cloud Azure provides integration with Azure Key Vault.
 - Spring Cloud GCP provides integration with GCP Secret Manager.
- **Cloud platform services.** When running your applications on a Kubernetes platform, you can seamlessly use ConfigMaps and Secrets to configure Spring Boot.

This section will show you how to set up a centralized configuration server implemented with Spring Cloud Config, responsible for delivering configuration data stored in a Git repository to all applications. Chapter 14 will then cover more advanced configuration topics, including secrets management and Kubernetes features like ConfigMaps and Secrets. Many of the features and patterns you'll use with Spring Cloud Config are easily applicable to the other solutions involving configuration services and cloud vendor services.

NOTE

Choosing an option for a configuration service depends on your infrastructure and requirements. For example, suppose you're running your workloads on Azure already and need a GUI to manage configuration data. In that case, it might make sense to use Azure Key Vault rather than running a configuration service yourself. If you want to version-control your configuration data with Git, Spring Cloud Config or Kubernetes ConfigMaps/Secrets would be a better choice. You might even compromise and use a managed Spring Cloud Config service offered by some vendors like Azure or VMWare Tanzu.

The idea of centralized configuration is built around two main components:

- a data store for configuration data, providing persistence, versioning, and possibly access control;

- a server sitting on top of it to manage configuration data and serve it to multiple applications.

Imagine having many applications deployed in different environments. A configuration server would manage configuration data for all of them from a centralized place. The configuration data might be stored in different ways. For example, you could use a dedicated Git repository for storing non-sensitive data and HashiCorp Vault to store your secrets. No matter how data is stored, a configuration server would deliver it to different applications through a unified interface. Figure 4.7 shows how centralized configuration works.

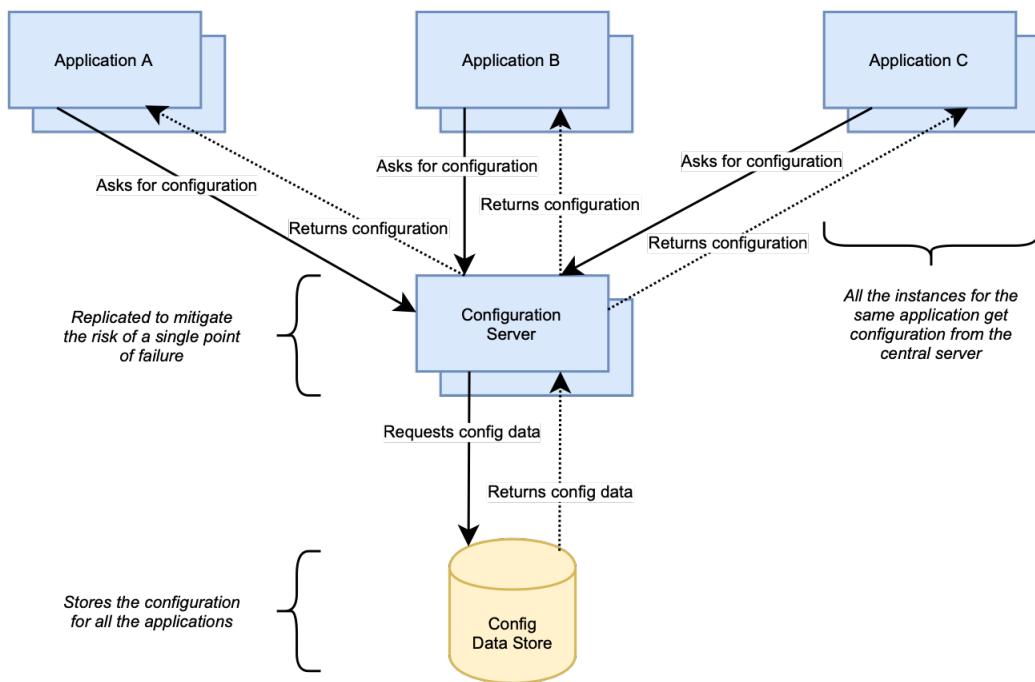


Figure 4.7 A centralized configuration server manages external properties for many applications across all environments.

It's clear from figure 4.7 that the configuration server becomes a backing service for all the applications, which means it's at risk of being a single point of failure. If it's suddenly unavailable, all the applications will probably fail to start up. This risk can be easily mitigated by scaling the config server, as you would with other applications requiring high availability. When using a configuration server, it's fundamental to deploy at least two replicas.

NOTE

You can use a centralized configuration server for your configuration data not depending on the specific infrastructure or deployment platform, such as credentials, features flags, URLs to third-party services, thread pools, and timeouts.

We'll use Spring Cloud Config Server to set up a centralized configuration server for the Polar Bookshop system. The project also provides a client library (Spring Cloud Config Client) that

you can use to integrate Spring Boot applications with the config server.

Let's start by defining a repository for storing configuration data.

4.3.1 Using Git to store your configuration data

The config server will be responsible for providing configuration data to Spring applications. Before setting that up, we need a way to store and keep track of that data. Spring Cloud Config Server integrates with many different backend solutions to store configuration data. One of the most common choices is a Git repository.

First of all, create a new `config-repo` Git repository (for the final result, you can refer to [Chapter04/04-end/config-repo](#)). It can be local or remote. For this example, I recommend initializing a remote one on GitHub, just like you did for the application repositories. I'm using `main` as the default branch name.

Inside the config repo, you can store properties directly in the Spring format as `*.properties` or `*.yml` files. Continuing with the Catalog Service example from before, why not define an external property for the welcome message?

Navigate to the `config-repo` folder and create a `catalog-service.yml` file. Then, define a value for the `polar.greeting` property used by the Catalog Service.

Listing 4.8 Define a new welcome message for when the config server is used (catalog-service.yml)

```
polar:
  greeting: "Welcome to the catalog from the config server"
```

Next, create a `catalog-service-prod.yml` file and define a different value for the `polar.greeting` property to be used only when the `prod` profile is active.

Listing 4.9 Define a new welcome message for when the "prod" profile is active (catalog-service-prod.yml)

```
polar:
  greeting: "Welcome to the production catalog from the config server"
```

Finally, remember to commit and push your changes to the remote repository.

How does Spring Cloud Config resolve the correct configuration data for each application? How should you organize the repository to host properties for multiple applications? The library relies on three parameters to identify which property file to use to configure a specific application:

- `{application}`: the name of the application as defined by the `spring.application.name` property.

- `{profile}`: one of the active profiles defined by the `spring.profiles.active` property.
- `{label}`: a discriminator defined by the specific configuration data repository. In the case of Git, it can be a tag, a branch name, or a commit id. It's useful to identify a versioned set of config files.

Depending on your needs, you can organize the folder structure using different combinations, for example:

```
/{{application}}/application-{profile}.yml
/{{application}}/application.yml
/{{application}}-{profile}.yml
/{{application}}.yml
/{{application}}-{profile}.yml
/{{application}}.yml
```

For each application, you can either use property files named after the application itself and placed in the root folder (e.g. `/catalog-service.yml` or `/catalog-service-prod.yml`) or use the default naming and put them in a subfolder named after the application (e.g. `/catalog-service/application.yml` or `/catalog-service/application-prod.yml`).

You can also put `application.yml` or `application-{profile}.yml` files in the root folder to define default values for all applications. They can be used as a fallback whenever there is no more specific property source. Spring Cloud Config Server will always return the properties from the most specific path, using the application name, active profiles, and Git labels.

The *label* concept is particularly interesting when using Git as the backend for the config server. For example, you could create long-lived branches of your config repo for different environments or short-lived branches while testing specific features. Spring Cloud Config Server can use the label information to return the correct configuration data from the right Git branch, tag, or commit id.

Now that you have a Git repository in place for your configuration data, it's time to set up a config server to manage them.

4.3.2 Setting up a configuration server

Spring Cloud Config Server is a project that lets you set up a configuration server with minimal effort. It's a standard Spring Boot application with specific properties enabling the configuration server functionality and a reference to the Git repository as the configuration data backend. We'll rely on this server for the Polar Bookshop system to provide configuration to the Catalog Service application. Figure 4.8 illustrates the architecture of the solution.

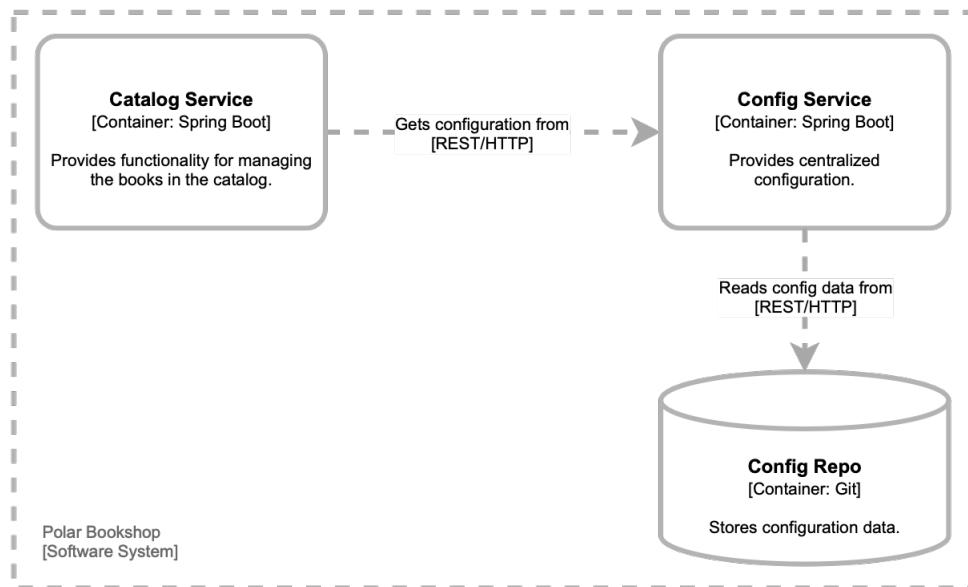


Figure 4.8 A centralized configuration server, backed by a Git repository, provides configuration to the Catalog Service application.

Now, on to the code.

BOOTSTRAPPING THE PROJECT

The Polar Bookshop system needs a Config Service application to provide centralized configuration. You can initialize the project from Spring Initializr (<https://start.spring.io/>) and store the result in a new config-service Git repository. The parameters for the initialization are shown in figure 4.9.

The screenshot shows the Spring Initializr interface with the following settings:

- Project:** Maven Project (selected)
- Language:** Java (selected)
- Spring Boot:** 2.7.0 (selected)
- Dependencies:** Config Server (selected)
- Project Metadata:**
 - Group: com.polarbookshop
 - Artifact: config-service
 - Name: config-service
 - Description: Provides functionality for centralizing the application configuration.
 - Package name: com.polarbookshop.configservice
 - Packaging: Jar (selected)
 - Java version: 17 (selected)

Figure 4.9 The parameters for initializing the Config Service project from Spring Initializr.

TIP

In the `begin` folder for this chapter, you can find a `curl` command to run in a Terminal window that downloads a zip file containing all the code you need to get started without going through the manual generation through the Spring Initializr website.

In the generated `build.gradle` file, you can see how the management of Spring Cloud dependencies works differently compared to Spring Boot. All Spring Cloud projects follow an independent release train that relies on a BOM (Bill of Material) to manage all the dependencies. Spring Cloud release trains are named after a year (e.g., 2021.0.3) rather than adopting a semantic version strategy (for example, the Spring Boot version is 2.7.0).

Listing 4.10 The Gradle configuration for Config Service (build.gradle)

```
plugins {
    id 'org.springframework.boot' version '2.7.0'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.polarbookshop'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2021.0.3") ①
}

dependencies {
    implementation 'org.springframework.cloud:spring-cloud-config-server'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:
    ↵spring-cloud-dependencies:${springCloudVersion}" ②
    }
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- ① Defines the Spring Cloud version to use.

- ② BOM (Bill of Material) for Spring Cloud dependency management.

- **Spring Cloud Config Server** (`org.springframework.cloud:spring-cloud-config-server`) provides libraries and utilities to build a configuration server on top of Spring Web.
- **Spring Boot Test** (`org.springframework.boot:spring-boot-starter-test`) provides several libraries and utilities to test applications, including Spring Test, JUnit, AssertJ, and Mockito. It's automatically included in every Spring Boot project.

ENABLING THE CONFIGURATION SERVER

Turning the project initialized earlier into a functioning configuration server doesn't require too many steps. The only thing you need to do in Java is adding the `@EnableConfigServer` annotation on a configuration class, for example, `ConfigServiceApplication`.

Listing 4.11 Enable configuration server in Spring Boot (`ConfigServiceApplication.java`)

```
package com.polarbookshop.configservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer ①
public class ConfigServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServiceApplication.class, args);
    }
}
```

- ① Activates the configuration server implementation in the Spring Boot application.

That was it on the Java side.

CONFIGURING THE CONFIGURATION SERVER

The next step is configuring the behavior of the config server. Yes, that's right. Even a configuration server needs configuration! First of all, Spring Cloud Config Server runs on an embedded Tomcat server, so you can configure connection timeout and thread pools as you did for Catalog Service.

You previously initialized a Git repository to host configuration data. You should now instruct Spring Cloud Config Server on where to find it. You can do so in an `application.yml` file located in the `src/main/resources` path of your Config Service project (after renaming the autogenerated `application.properties` file).

The `spring.cloud.config.server.git.uri` property should point to where you defined the

config repo. If you followed along, it is on GitHub, and the default branch is called `main`. You can configure which branch the config server should consider by default setting the `spring.cloud.config.server.git.default-label` property. Remember that when using Git repositories, the label concept is an abstraction over Git branches, tags, or commit ids.

Listing 4.12 Configure integration between config server and config repo (application.yml)

```
server:
  port: 8888 ①
  tomcat:
    connection-timeout: 2s
    keep-alive-timeout: 15s
    threads:
      max: 50
      min-spare: 5

spring:
  application:
    name: config-service ②
  cloud:
    config:
      server:
        git:
          uri: <your-config-repo-github-url> ③
          default-label: main ④
```

- ① The port that the Config Service application will listen on.
- ② A name for the current application.
- ③ The URL to the remote Git repository to use as the configuration data backend. For example, <https://github.com/PolarBookshop/config-repo>.
- ④ By default, the server will return configuration data from the "main" branch.

WARNING The configuration I used for Config Service assumes the config repo is publicly available on GitHub. When you use private repositories (which is often true for real-world applications), you need to specify how to authenticate with the code repository provider using additional configuration properties. For more information, refer to the official Spring Cloud Config documentation (<https://spring.io/projects/spring-cloud-config>). Further considerations about handling credentials will be done in chapter 14.

4.3.3 Making the configuration server resilient

The Config Service might become a single point of failure in your system. If all the applications rely on it to fetch configuration data, you need to ensure it's highly available. The first step towards that goal is deploying multiple instances of Config Service in a production environment. If one of them stops working for some reason, another replica can provide the required configuration. In chapter 7, you'll learn more about scaling applications and how to make that work in Kubernetes.

However, scaling the Config Service is not enough. Since it's using a remote Git repository as the configuration data backend, you need to make that interaction more resilient. First, you can define a timeout to prevent the config server from waiting too much time to establish a connection with the remote repository. You can do so with the `spring.cloud.config.server.git.timeout` property.

Spring Cloud Config is implemented to clone the remote repository locally upon the first request for configuration data. I recommend using the `spring.cloud.config.server.git.clone-on-start` property so that the repo clone happens at startup. Even though it makes the startup phase a bit slower, it makes your deployment fail faster if there's any issue communicating with the remote repository, rather than waiting for the first request to find out something is wrong. Also, it makes the first request from a client faster.

The local copy of the repository improves the config server fault tolerance because it ensures it can return configuration data to the client applications even if the communication with the remote repository is temporarily failing (for example, if GitHub is down or there's a problem with the network). However, if the config server hasn't cloned the repository locally yet, there's no fallback in place. That's why it's better to fail fast at startup and investigate the issue right away.

When a local copy of the repository is created successfully, there's a chance the local repo gets changed independently from the remote repo. You can ensure your config server always uses the same data defined in the remote repository by setting the `spring.cloud.config.server.git.force-pull` property so that a fresh copy is always pulled and any local change discarded. By default, the local repository is cloned in a folder with a random name. If needed, you can control where it gets cloned through the `spring.cloud.config.server.git.basedir` property. For Config Service, we'll rely on the default behavior.

You can update the `application.yml` file for the Config Service application as follows and make it more resilient to faults affecting the interaction with the code repository service (in this case, GitHub).

Listing 4.13 Make Config Service more resilient (application.yml)

```
spring:
  application:
    name: config-service
  cloud:
    config:
      server:
        git:
          uri: <your-config-repo-github-url>
          default-label: main
          timeout: 5 ①
          clone-on-start: true ②
          force-pull: true ③
```

- ① Time limit to establish a connection with the remote repository.
- ② Clones the remote repository locally on startup.
- ③ Forces pulling the remote repository and discarding any local change.

In the next section, we'll verify that Config Service is working correctly.

4.3.4 Understanding the configuration server REST API

Spring Cloud Config Server works seamlessly with Spring Boot applications, providing properties in their native format through a REST API. You can try it out fairly easily. Build and run Config Service (`./gradlew bootRun`), open a Terminal window, and make an HTTP GET request to `/catalog-service/default`.

```
$ http :8888/catalog-service/default
```

The result is the configuration returned when no Spring profile is active. You can try fetching the configuration for the scenario where the `prod` profile is active as follows.

```
$ http :8888/catalog-service/prod
```

As shown in figure 4.10, the result is the configuration defined for the Catalog Service application in `catalog-service.yml` and `catalog-service-prod.yml`, where the latter takes precedence over the former because the `prod` profile is specified.



Figure 4.10 The configuration server exposes a REST API to fetch configuration data in terms of the application name, profile, and label. This image shows the result for the `"/catalog-service/prod"` endpoint.

When you are done testing the application, stop its execution with `Ctrl+C`.

Spring Cloud Config Server exposes properties through a series of endpoints, using different combinations of the {application}, {profile}, and {label} parameters.

```
/{{application}}/{{profile}}[/{{label}}]
/{{application}}-{{profile}}.yml
/{{label}}/{{application}}-{{profile}}.yml
/{{application}}-{{profile}}.properties
/{{label}}/{{application}}-{{profile}}.properties
```

You won't need to call these endpoints from your application when using Spring Cloud Config Client (that does that for you), but it's useful to know how the server exposes configuration data. A configuration server built with Spring Cloud Config Server exposes a standard REST API that any application over a network can access. You can use the same server for applications built with other languages and frameworks and use the REST API directly.

In chapter 14, I'll address more aspects of how to handle configuration. For example, Spring Cloud Config has a few features to encrypt properties containing secrets before storing them in a Git repository. Also, multiple backend solutions are allowed as configuration data repositories, meaning that you could save all the non-sensitive properties in Git and use HashiCorp Vault to store secrets. Furthermore, the REST API itself should be protected, and I'll talk about that as well. I will address all those critical aspects from a security point of view, necessary before deploying to production.

For now, let's complete the solution and update Catalog Service to integrate with the Config Service application.

4.4 Using a configuration server with Spring Cloud Config Client

The Config Service application built in the previous section is a server that exposes configuration through a REST API. In general, applications would explicitly interact with this API, but you can use Spring Cloud Config Client for Spring applications.

This section will teach you how to use Spring Cloud Config Client and integrate Catalog Service with the config server. You'll see how to make the interaction more robust and refresh the client's configuration when new changes are pushed to the config repo.

4.4.1 Setting up a configuration client

The first thing to integrate a Spring Boot application with a config server is adding a new dependency to Spring Cloud Config Client. Update the `build.gradle` file for the Catalog Service project (`catalog-service`) as follows. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 4.14 Add dependency for Spring Cloud Config Client in Catalog Service (build.gradle)

```
ext {
    set('springCloudVersion', "2021.0.3")
}

dependencies {
    ...
    implementation 'org.springframework.cloud:spring-cloud-starter-config'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:
spring-cloud-dependencies:${springCloudVersion}"
    }
}
```

Catalog Service needs to know the URL to contact the Config Service. First, use the `spring.config.import` property to inform Spring Boot you want to import configuration data from a config server. You can do that by passing `configserver:` as the property value. You probably don't want the config server running in your local environment when working with client applications like Catalog Service. If that's the case, you can make the interaction optional through the `optional:` prefix (`optional:configserver:`). If the config server is not running when you start Catalog Service, the application will log a warning, but it won't stop working. Be careful when you do this, and make sure you don't make it optional in production, or you'll risk using the wrong configuration.

Regarding the Config Service URL, you have two options. Either you add it to the `spring.config.import` property (`optional:configserver:http://localhost:8888`) or rely on the more specific `spring.cloud.config.uri` property. We'll use the second option to simplify changing only the URL value when deploying the application in different environments.

Since the config server uses the application name to return the correct configuration data, you also need to set the `spring.application.name` property to `catalog-service`. Remember the `{application}` parameter? That's where the `spring.application.name` value is used.

Open the `application.yml` file for your Catalog Service project and apply the following configuration.

Listing 4.15 Instruct Catalog Service to import configuration from Config Service (application.yml)

```
spring:
  application:
    name: catalog-service ①
  config:
    import: "optional:configserver:" ②
  cloud:
    config:
      uri: http://localhost:8888 ③
```

- ① The name of the application, used by the config server to filter the configuration.
- ② Imports configuration data from a config server when available.
- ③ The URL of the configuration server.

Let's go ahead and verify it works correctly. The Catalog Service application contains a value for the `polar.greeting` property that is "Welcome to the local book catalog!". When using a configuration server, the centralized properties take precedence over the local ones, so the value you defined in the `config-repo` will be used instead.

First, run Config Service (`./gradlew bootRun`). Then, package Catalog Service as a JAR artifact (`./gradlew bootJar`) and run it as follows.

```
$ java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar
```

Then, in another Terminal window, send a GET request to the root endpoint.

```
$ http :9001/
Welcome to the catalog from the config server!
```

As expected, the welcome message returned by the application is the one defined in the `config-repo`, specifically in the `catalog-service.yml` file. You can also try running the application with the `prod` profile enabled and see how the value from `catalog-service-prod.yml` will be returned instead.

Stop Catalog Service with `Ctrl+C`, then start the application again with `prod` as an active profile.

```
$ java -jar build/libs/catalog-service-0.0.1-SNAPSHOT.jar \
--spring.profiles.active=prod
```

The expected result is now the message defined in the `catalog-service-prod.yml` file of the `config-repo`.

```
$ http :9001/
Welcome to the production catalog from the config server
```

Once again, stop the previous application execution with `Ctrl+C`.

The following section will cover how to make the interaction between application and config server more fault-tolerant.

4.4.2 Making the configuration client resilient

When the integration with the config server is not optional, the application fails to start up if it cannot contact a config server. If it's up and running, you could still experience issues due to the distributed nature of the interaction. Therefore it's a good idea to define some timeouts to make the application fail faster. You can use the `spring.cloud.config.request-connect-timeout` to control the time limit for establishing a connection with the config server. The `spring.cloud.config.request-read-timeout` lets you limit the time spent reading configuration data from the server.

Open the `application.yml` for your Catalog Service project and apply the following configuration to make the interaction with Config Service more resilient. Once again, there's no universal rule for setting up timeouts. Depending on your architecture and infrastructure characteristics, you might need to adjust them.

Listing 4.16 Make Spring Cloud Config Client more resilient (application.yml)

```
spring:
  application:
    name: catalog-service
  config:
    import: "optional:configserver:"
  cloud:
    config:
      uri: http://localhost:8888
      request-connect-timeout: 5000 ①
      request-read-timeout: 5000 ②
```

- ① Timeout on waiting to connect to the config server (ms).
- ② Timeout on waiting to read configuration data from the config server (ms).

Even if Config Service is replicated, there's still a chance it's temporarily unavailable when a client application like Catalog Service starts up. In that scenario, you can leverage the *retry* pattern and configure the application to try again connecting with the config server before giving up and failing. The retry implementation for Spring Cloud Config Client is based on Spring Retry, so you need to add a new dependency to the `build.gradle` file for your Catalog Service project. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 4.17 Add dependency for Spring Retry in Catalog Service (build.gradle)

```
dependencies {
  ...
  implementation 'org.springframework.retry:spring-retry'
}
```

In chapter 8, I'll explain the retry pattern in detail. For now, I'll show you how to configure Catalog Service so that it retries connecting to Config Service a few times before failing (`spring.cloud.config.retry.max-attempts`). Each connection attempt is delayed according

to an exponential backoff strategy, computed as the current delay multiplied by the value of the `spring.cloud.config.retry.multiplier` property. The initial delay is configured by `spring.cloud.config.retry.initial-interval` and each delay cannot exceed the value of `spring.cloud.config.retry.max-interval`.

You can add the retry configuration to the `application.yml` file in the Catalog Service project.

Listing 4.18 Apply retry pattern to Spring Cloud Config Client (application.yml)

```
spring:
  application:
    name: catalog-service
  config:
    import: "optional:configserver:"
  cloud:
    config:
      uri: http://localhost:8888
      request-connect-timeout: 5000
      request-read-timeout: 5000
      fail-fast: true ①
      retry:
        max-attempts: 6 ②
        initial-interval: 1000 ③
        max-interval: 2000 ④
        multiplier: 1.1 ⑤
```

- ① Makes a failure to connect to the config server fatal.
- ② Maximum number of attempts.
- ③ Initial retry interval for backoff (ms).
- ④ Maximum retry interval for backoff (ms)
- ⑤ Multiplier to compute the next interval.

The retry behavior is enabled only when the `spring.cloud.config.fail-fast` property is set to `true`. You probably don't want to retry in your local environment if the config server is down, especially considering you made it an optional backing service. Feel free to test the application behavior when retrying the connection with the config server down, but remember to set the `fail-fast` property back to `false` if you want to keep it optional in your local environment. In production, you can set it to `true` using one of the strategies covered in this chapter.

When you're done testing the applications, stop both of them with `Ctrl+C`.

You are now ready to use the Config Service to configure any application you want. However, there's still an aspect I haven't covered yet. How can we change the configuration at runtime?

4.4.3 Refreshing configuration at runtime

What happens when new changes are pushed to the Git repository backing the Config Service? With a standard Spring Boot application, you would have to restart it when changing a property (either in a property file or an environment variable). Spring Cloud Config gives you additional options. Whenever a new change is pushed to the configuration repository, you can signal all the applications integrated with the config server. Upon receiving the signal, they would reload the parts affected by the configuration change at runtime.

Spring Cloud Config offers different ways for refreshing configuration in the client applications. In this section, I'll show you a simple refresh option consisting of sending a special POST request to a running Catalog Service instance to trigger the reloading of the configuration data changed (*hot reload*). Figure 4.11 shows how it works.

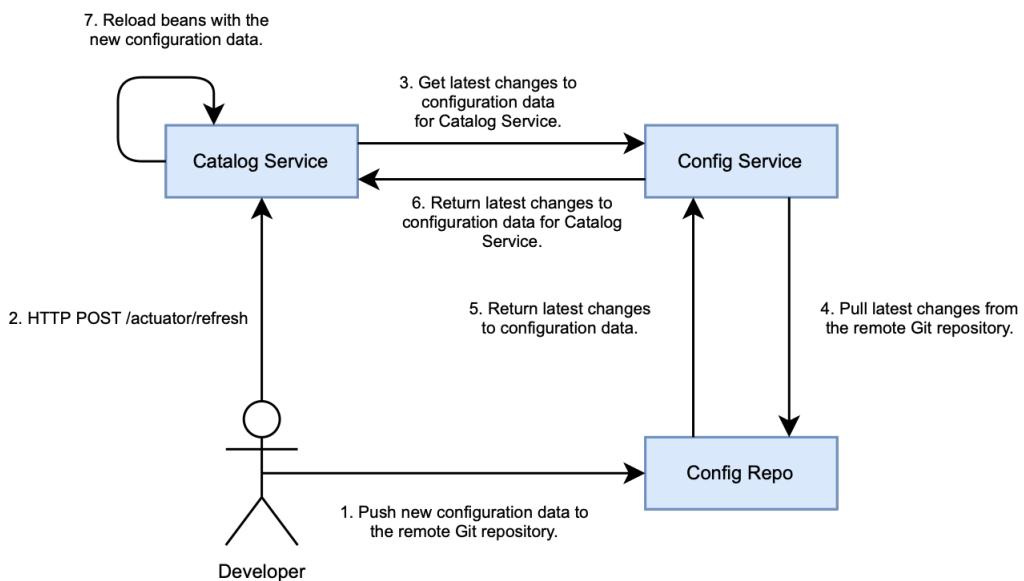


Figure 4.11 After changing the configuration in the Git repository backing the Config Service, a signal is sent to Catalog Service to refresh the parts of the app using the configuration.

This functionality is one of those *administrative processes* described by the 15-Factors methodology introduced in chapter 2. In this case, the strategy adopted for managing such a process was to embed it in the applications itself with the possibility to activate it by calling a specific HTTP endpoint.

NOTE

In production, you probably want a more automated and efficient way of refreshing configuration without explicitly triggering each application instance. When a remote Git repository backs your config server, you can configure a webhook that notifies the config server automatically whenever new changes are pushed to the repository. In turn, the config server can notify all client applications through a message broker like RabbitMQ, using Spring Cloud Bus. Chapter 14 will cover more scenarios for refreshing configuration in production.

ENABLING CONFIGURATION REFRESH

After committing and pushing the new configuration changes to the remote Git repository, you can send a POST request to a client application through a specific endpoint that will trigger a RefreshScopeRefreshedEvent inside the application context. You can rely on the Spring Boot Actuator project to expose the refresh endpoint by adding a new dependency in the build.gradle file for the Catalog Service project. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 4.19 Add dependency for Spring Boot Actuator in Catalog Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
}
```

The Spring Boot Actuator library configures an /actuator/refresh endpoint that triggers a refresh event. By default, the endpoint is not exposed, so you have to enable it explicitly in the application.yml file of your Catalog Service project (`catalog-service`).

Listing 4.20 Configure Spring Boot Actuator to expose the "refresh" HTTP endpoint (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: refresh ①
```

- ① Exposes the "/actuator/refresh" endpoint through HTTP.

NOTE

I will discuss Spring Boot Actuator extensively in chapter 13, so don't worry if you don't fully get the previous configuration. For now, it's enough to know that Spring Boot Actuator provides many useful endpoints for monitoring and managing applications in production.

The refresh event alone, RefreshScopeRefreshedEvent, will have no effect if there is no

component listening. You can use the `@RefreshScope` annotation on any bean you'd like to be reloaded whenever a refresh is triggered. Here's the nice part: since you defined your custom properties through a `@ConfigurationProperties` bean, it is already listening to `RefreshScopeRefreshedEvent` by default, so you don't need to make any change to your code. When a refresh is triggered, the `PolarProperties` bean will be reloaded with the latest configuration available. Let's see if it works.

CHANGING CONFIGURATION AT RUNTIME

First, make sure you have both the Config Service and Catalog Service up and running (`./gradlew bootRun`). Then, open the `config-repo` hosting the configuration data and change the value of the `polar.greeting` property in `config-repo/catalog-service.yml`.

Listing 4.21 Update the welcome message value in the config repo (catalog-service.yml)

```
polar:
  greeting: "Welcome to the catalog from a fresh config server"
```

Next, commit and push the changes. Config Service will now return the new property value. You can check that by running the `http :8888/catalog-service/default` command. However, no signal has been sent yet to Catalog Service. If you try running the `http :9001/` command, you will still get the old "Welcome to the catalog from the config server" message. Let's trigger a refresh.

Go ahead and send a POST request to the Catalog Service application at the `/actuator/refresh` endpoint.

```
$ http POST :9001/actuator/refresh
```

The request will trigger a `RefreshScopeRefreshedEvent`. The `PolarProperties` bean, because it's annotated with `@ConfigurationProperties`, will react to the event and read the new configuration data. Let's verify it.

```
$ http :9001/
Welcome to the catalog from a fresh config server
```

Finally, stop the execution of both applications with `Ctrl+C`.

Good job! You have just updated the configuration of an application at runtime, without restarting it, without rebuilding the application, and ensuring the traceability of changes. It's perfect for the cloud. In chapter 14, you'll learn more advanced techniques for managing configuration that you'll need in production, including secrets management, ConfigMaps, and Secrets.

4.5 Summary

- The Spring Environment abstraction provides a unified interface to access properties and profiles.
- Properties are key/value pairs used to store configuration.
- Profiles are logical groups of beans registered only when a specific profile is active.
- Spring Boot collects properties from different sources according to precedence rules. From the highest to the lowest precedence, properties can be defined in command-line arguments, JVM system variables, OS environment variables, profile-specific property files, and generic property files.
- Spring beans can access properties from the `Environment` object, by injecting the value with the `@Value` annotation, or from a bean mapped to a set of properties with the `@ConfigurationProperties` annotation.
- The active profiles can be defined with the `spring.profiles.active` property.
- The `@Profile` annotation marks beans or configuration classes to be considered only when the specified profile is active.
- Properties, as managed in Spring Boot, provide externalized configuration as defined by the 15-Factor methodology, but that's not enough.
- A configuration server handles aspects like secret encryption, configuration traceability, versioning, and context refreshing at runtime with no restart.
- A configuration server can be set up with the Spring Cloud Config Server library.
- The configuration itself can be stored according to different strategies, such as in a dedicated Git repository.
- The config server uses application name, active profiles, and Git-specific labels to identify which configuration should be provided to which application.
- A Spring Boot application can be configured through a config server using the Spring Cloud Config Client library.
- `@ConfigurationProperties` beans are configured to listen to `RefreshScopeRefreshedEvent`.
- `RefreshScopeRefreshedEvent` can be triggered after a new change is pushed to the configuration repository so that the client application reloads the context using the latest configuration data.
- Spring Boot Actuator defines an `/actuator/refresh` endpoint that you can use to trigger the event manually.



Persisting and managing data in the cloud

This chapter covers

- Understanding databases in a cloud native system
- Implementing data persistence with Spring Data JDBC
- Testing data persistence with Spring Boot and Testcontainers
- Managing databases in production with Flyway

In chapter 1, I distinguished between application services and data services in a cloud native system. So far, we have worked with application services that should be stateless to play well in a cloud environment. However, most applications are useless without storing any state or data somewhere. For example, the Catalog Service application we built in chapter 3 has no persistent storage mechanism, so you can't really use it to manage a catalog of books. Once you shut it down, all the books you added to the catalog will be gone. As a consequence of being stateful, you can't even scale the application horizontally.

The *state* is everything that should be preserved when shutting down a service and spinning up a new instance. Data services are the stateful components of a system. For example, they can be data stores like PostgreSQL, Cassandra, and Redis, or messaging systems like RabbitMQ and Apache Kafka.

This chapter will introduce databases for cloud native systems and the main aspects of persisting data in the cloud. You'll rely on Docker to run PostgreSQL in your local environment, but in production, you're going to replace it with a managed service offered by a cloud platform. Then, you'll add a data persistence layer to Catalog Service using Spring Data JDBC. Finally, I'll cover some common concerns about managing and evolving databases in production with Flyway.

NOTE

The source code for the examples in this chapter is available in the Chapter05/05-begin, Chapter05/05-intermediate, and Chapter05/05-end folders, containing the initial, intermediate, and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

5.1 Databases for cloud native systems

Data can be stored in many ways. Traditionally, you would probably use a single massive database server to save as much as possible since getting a new one would be expensive and time-consuming. Depending on the organizational processes, a task like that could take from a few days to several months. Not in the cloud.

The cloud offers elastic, self-service, and on-demand provisioning features, a strong motivator for migrating your data services there. For each cloud native application you design, you should consider the most suitable storage type for the data it will produce. Then, your cloud platform should let you provision it through an API or a graphical user interface. A task that used to be very time-consuming will now require only a few minutes. For example, deploying an instance of a PostgreSQL database server on Azure is as simple as running the `az postgres server create` command.

Cloud native applications are designed to be stateless because of the nature of the cloud itself. It's a dynamic infrastructure where the computing nodes can spread across different clusters, geographical regions, and clouds. The problem of applications storing state is evident. How could the state survive such a distributed and dynamic environment? That's why we want to keep applications stateless.

Yet, we need to achieve statefulness in the cloud. This section will present the challenges of data services and persistence management in the cloud and describe your options depending on whether you want to manage a data service yourself or rely on one of the offerings from a cloud provider. Then, I'll guide you through the setup of a PostgreSQL database instance as a container for your local environment.

5.1.1 Data services in the cloud

Data services are the components of a cloud native architecture designed to be stateful. By designing applications to be stateless, you can limit cloud storage challenges to those few components.

Traditionally, storage aspects were handled by operations engineers and database administrators. But when dealing with the cloud and DevOps practices, developers are allowed to pick the data service that best fits the application's requirements and deploy it following the same approach

used for cloud native applications. Specialists like database administrators are consulted to make the most of the technology developers have picked, addressing aspects like performance, security, and efficiency. However, the goal is to provide storage and data services on-demand, just like you would do for cloud native applications, and configure them in a self-service manner.

The difference between application and data services can also be visualized in terms of the three basic building blocks of cloud infrastructures: compute, storage, and network. As shown in figure 5.1, application services use computing and networking resources because they are stateless. On the other hand, data services are stateful and require storage to persist the state.

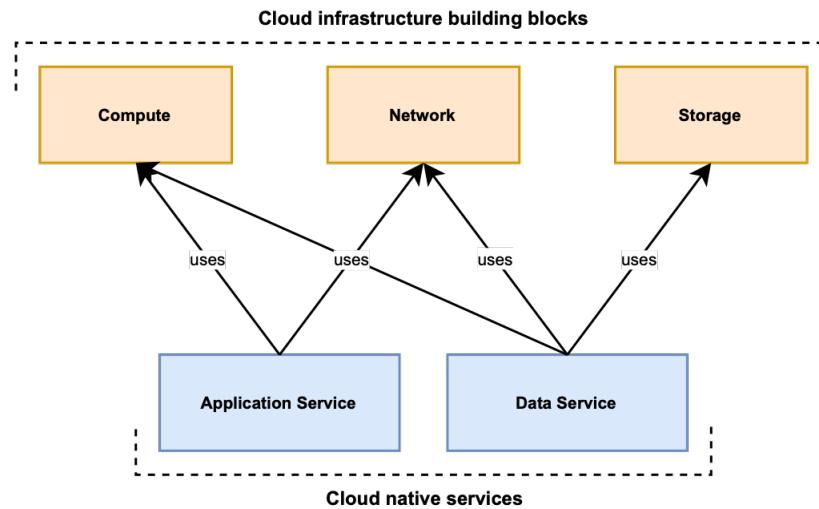


Figure 5.1 Application services (stateless) only use computing and networking resources in a cloud infrastructure. Data services (stateful) also need storage.

Let's see what the challenges of data services in a cloud environment are and the main categories from which you can choose the most suitable solution for your applications.

CHALLENGES WITH DATA SERVICES

Data services in a cloud native system are most likely off-the-shelf components like databases and message brokers. There are a few properties you should consider before deciding which technology to use and ensuring you choose the most suitable one.

- **Scalability.** Cloud native applications can scale in and out dynamically. Data services are no different: they should scale to adapt to increasing or decreasing workloads. The new challenge is scaling while ensuring safe access to the data storage. The amount of data flying through a system in the cloud is larger than ever. There can be sudden increments, so data services should support the likelihood of increasing workloads and be resilient.
- **Resilience.** Similar to cloud native applications, data services should be resilient to failures. The new aspect here is that also the data persisted using a specific storage technology should be resilient. One of the key strategies for having resilient data and preventing data loss is duplication. Replicating data across different clusters and geographical zones makes it even more resilient, but it comes at a cost. Data services like

relational databases allow replication while ensuring data consistency. Others, like some non-relational databases, provide a high level of resilience but can't always guarantee data consistency (what is referred to as *eventual consistency*).

- **Performance.** The way data is duplicated can affect performance, which is also limited by the I/O access latency of the specific storage technology and network latency. Where the storage is located compared to the data services relying on it becomes important, a concern that we haven't encountered while working with cloud native applications.
- **Compliance.** With data services more than cloud native applications, you might face challenges in terms of compliance. Persisted data is usually critical for businesses and often contains information protected by specific laws, regulations, or customer agreements regarding how it's managed. For example, when dealing with personal and sensitive information, it's vital to manage data in accordance with privacy laws. In Europe, that would mean following the General Data Protection Regulation (GDPR). In California, there is the California Consumer Privacy Act (CCPA). In specific domains, further laws apply. For example, health data in the US should be handled in compliance with the Health Insurance Portability and Accountability Act (HIPAA). Both the cloud native storage and cloud provider should comply with whatever law or agreement you are required to respect. Because of this challenge, some organizations dealing with very sensitive data like healthcare providers and banks prefer to use a type of cloud native storage on their premises to have more control over data management and ensure compliance with applicable regulations.

CATEGORIES OF DATA SERVICES

Data services can be categorized depending on who is responsible for them: the cloud provider or you. Cloud providers have multiple offerings for data services, addressing all the main challenges of cloud native storage for you.

You can find industry-standard services like PostgreSQL, Redis, and MariaDB. Some cloud providers even offer enhancements on top of them, optimized for scalability, availability, performance, and security. For example, should you need a relational database, you could use Amazon Relational Database Service (RDS), Azure Database, or Google Cloud SQL.

Cloud providers also offer new types of data services specifically built for the cloud and exposing their own unique APIs. For instance, Google BigQuery is a serverless data warehouse solution with a particular focus on high scalability. Another example is the extremely fast, non-relational database Cosmos DB offered by Azure.

The other option is to manage data services by yourself, increasing complexity yet giving you more control over the solution. You can choose to use a more traditional setup based on virtual machines, or you can decide to use containers and the lessons learned with managing cloud native applications. Using containers would allow you to manage all services in your system through a unified interface such as Kubernetes, handling both compute and storage resources, and reducing costs. Figure 5.2 illustrates the categorization of data services for the cloud.

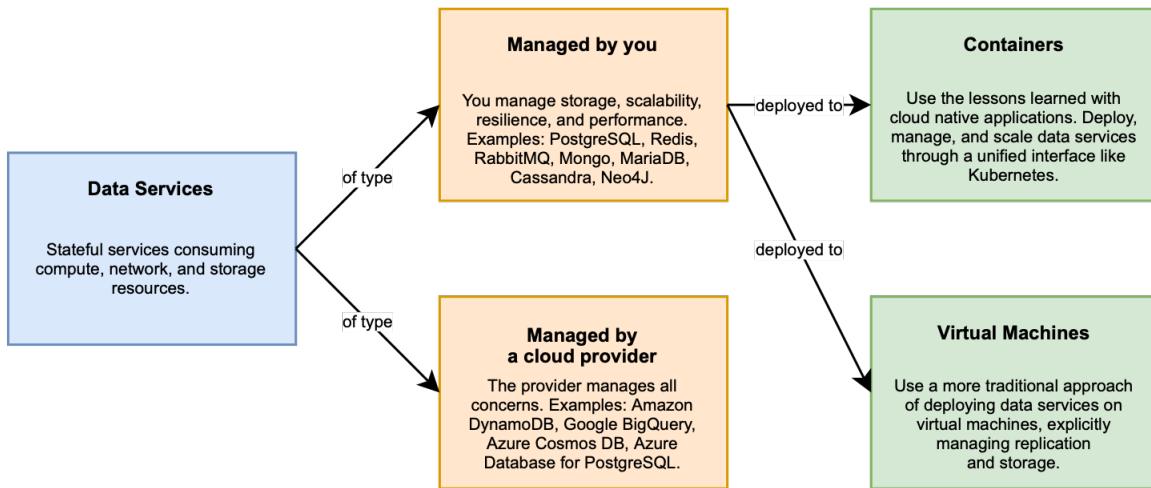


Figure 5.2 Data services can be managed by you (as containers or on virtual machines) or by the cloud provider. In the first case, you can use more traditional services, whereas in the second case, you can also access multiple services built specifically for the cloud by the provider.

NOTE

When choosing to run and manage a data service by yourself (virtual machine or container on Kubernetes), another essential decision is what type of storage to use. Local persistence storage? Remote persistence storage? The "cloud native storage" subject is fascinating, but it's out of scope for this book. If you'd like to learn more, I recommend checking the dedicated section on the CNCF Landscape (<https://landscape.cncf.io>).

The following section will focus on relational databases and guide you through setting up a PostgreSQL container for your local environment.

5.1.2 Running PostgreSQL as a container

For the Catalog Service application, we'll use a relational database, PostgreSQL, to store the data about the books in the catalog. PostgreSQL is a popular open-source database with strong reliability, robustness, and performance (<https://www.postgresql.org>), supporting both relational and non-relational data. Most cloud providers offer PostgreSQL as a managed service, freeing you from dealing with aspects like high availability, resilience, and persistent storage on your own. Examples are Azure Database for PostgreSQL, Amazon RDS for PostgreSQL, Google Cloud SQL for PostgreSQL, Alibaba Cloud ApsaraDB RDS for PostgreSQL, and DigitalOcean PostgreSQL.

Later in the book, we'll deploy the Polar Bookshop system to a Kubernetes cluster managed by a cloud provider, and I'll show you how to use their offerings for managed PostgreSQL. You want to ensure environment parity, as recommended by the 15-Factor methodology. That's why you will use PostgreSQL in development as well. Docker makes running databases locally easier than ever, so I'll show you how to run PostgreSQL as a container on your local machine.

In chapter 2, you gave Docker a first try with the Catalog Service application. Running PostgreSQL as a container is not different. Make sure your Docker Engine is up and running, open a Terminal window, and execute the following command.

```
$ docker run -d \
--name polar-postgres \ ①
-e POSTGRES_USER=user \ ②
-e POSTGRES_PASSWORD=password \ ③
-e POSTGRES_DB=polardb_catalog \ ④
-p 5432:5432 \ ⑤
postgres:14.3 ⑥
```

- ① The name of the container
- ② Defines the username for the admin user.
- ③ Defines the password for the admin user.
- ④ Defines the name of the database to be created.
- ⑤ Exposes the database to port 5432 on your machine.
- ⑥ The PostgreSQL container image pulled by Docker Hub.

Compared to how you ran the Catalog Service container, you can notice a few new elements. First, the Docker image from which you run a container (`postgres:14.3`) is not created by you, but it's pulled from the Docker Hub container registry (configured by default when you install Docker).

The second new thing is passing environment variables as arguments to the container. PostgreSQL accepts a few environment variables used during the container creation to configure a database.

NOTE

In this book, I won't cover how to configure storage in Docker (volumes). It means that all data saved in your local PostgreSQL container will be lost once you remove the container. It might seem counter-intuitive given the topic of this chapter, but any storage-related concern will be handled by the cloud provider in production, so you won't have to deal with it yourself. If you need to add persistence storage to your local containers, though, you can read how to use volumes in the official Docker documentation (<https://docs.docker.com>).

In the next section, you'll see how to add data persistence to a Spring Boot application using Spring Data JDBC and PostgreSQL.

NOTE

If you need to, you can stop the container with `docker stop polar-postgres` and start it again with `docker start polar-postgres`. If you want to start over, you can remove the container with `docker rm -fv polar-postgres` and create it again with the previous `docker run` command.

5.2 Data persistence with Spring Data JDBC

Spring supports a wide variety of data persistence technologies through the Spring Data project, which contains specific modules dedicated to relational (JDBC, JPA, R2DBC) and non-relational databases (Cassandra, Redis, Neo4J, MongoDB, and so on). Spring Data provides common abstractions and patterns, making it straightforward to navigate the different modules. This section focuses on relational databases, but the key points of the interaction between an application using Spring Data and a database (shown in figure 5.3) apply to all of them.

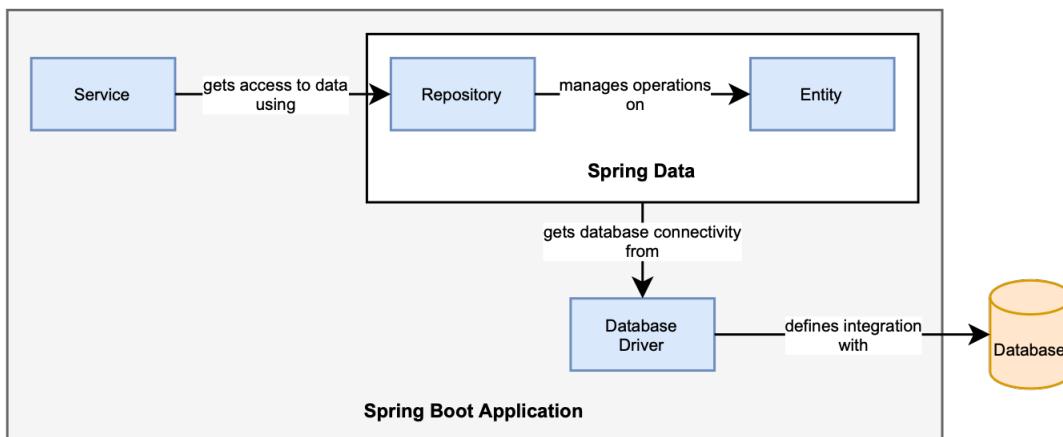


Figure 5.3 A driver configures the connection between the application and the database. Entities represent domain objects and can be stored and retrieved through a repository.

The main elements in the interaction shown in figure 5.3 are database drivers, entities, and repositories.

- **Database drivers.** Components that provide integration with a specific database (through *connection factories*). For relational databases, you can use a JDBC driver (Java Database Connectivity API) in imperative/blocking applications or an R2DBC driver in reactive/non-blocking applications. For non-relational databases, each vendor has its own dedicated solution.
- **Entities.** Domain objects that are persisted in a database. They must contain a field to uniquely identify each instance (*primary key*) and can use dedicated annotations to configure the mapping between Java objects and database entries.
- **Repositories.** The abstractions used for data storage and retrieval. Spring Data provides basic implementations, which are further extended by each module to provide features specific to the database in use.

This section will show you how to use Spring Data JDBC to add data persistence to a Spring Boot application like Catalog Service. You'll configure a connection pool to interact with a PostgreSQL database through the JDBC driver, define entities to be persisted, use repositories to access data, and work with transactions. Figure 5.4 shows how the Polar Bookshop architecture will look by the end of this chapter.

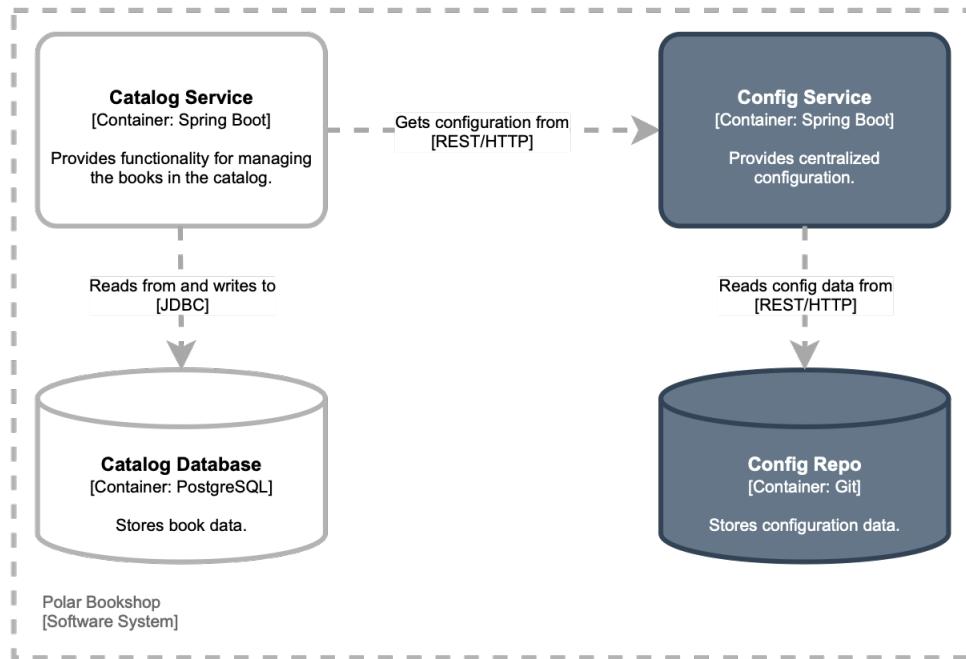


Figure 5.4 The Catalog Service application uses a PostgreSQL database to persist books data.

SIDE BAR**Spring Data JDBC or Spring Data JPA?**

Spring Data offers two main options for integrating applications with a relational database over the JDBC driver: Spring Data JDBC and Spring Data JPA. How to choose between the two? As always, the answer is that it depends on your requirements and specific context.

Spring Data JPA (<https://spring.io/projects/spring-data-jpa>) is the most-used module in the Spring Data project. It's based on the Java Persistence API (JPA), a standard specification included in Jakarta EE (previously known as Java EE). Hibernate is the most popular implementation. It's a robust and battle-tested Object/Relational Mapping (ORM) framework to manage data persistence in Java applications. Hibernate provides many useful features, but it's also a complex framework. Suppose you're not aware of aspects like persistence context, lazy loading, dirty checking, or sessions. In that case, you might incur issues that will be hard to debug without a sound familiarity with JPA and Hibernate. Once you know the framework better, you'll appreciate how much Spring Data JPA simplifies things and boosts your productivity. To learn more about JPA and Hibernate, you can check out "High-Performance Java Persistence and SQL" by Vlad Mihalcea (<https://vladmihalcea.com>) and "Java Persistence with Spring Data and Hibernate" by Catalin Tudose (<https://www.manning.com/books/java-persistence-with-spring-data-and-hibernate>).

Spring Data JDBC (<https://spring.io/projects/spring-data-jdbc>) is a more recent addition to the Spring Data family. It integrates with relational databases following the Domain-Driven Design (DDD) concepts like aggregates, aggregate roots, and repositories. It's lightweight, simpler, and an excellent choice for microservices where domains are usually defined as bounded contexts (another DDD concept). It gives developers more control over SQL queries and allows using immutable entities. Being a simpler alternative to Spring Data JPA, it's not a drop-in replacement for every scenario since it doesn't provide all the features offered by JPA. I recommend learning both, considering the requirements, and then deciding which module suits the specific scenario better.

I chose to cover Spring Data JDBC because it's a good fit for cloud native applications and for its simplicity. Thanks to the Spring Data common abstractions and patterns, you can easily convert the project from Spring Data JDBC to Spring Data JPA. In the following sections, I'll point out the main differences between the two to give you enough information in case you want to try implementing the same requirements using Spring Data JPA. In the code repository accompanying the book, you'll also find a JPA version of Catalog Service you can use as a reference (Chapter05/05-end/catalog-service-jpa).

5.2.1 Connecting to a database with JDBC

Let's start implementing the data persistence layer for the Catalog Service application. As a minimum, you need to import the Spring Data module for the specific database you want to use and, if required, also the database driver. Since Spring Data JDBC supports different relational databases, you need to explicitly declare a dependency on the specific database driver you want to use.

You can add the two new dependencies to the `build.gradle` file for the Catalog Service project (`catalog-service`). Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 5.1 Add dependency for Spring Data JDBC in Catalog Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-data-jdbc'
    runtimeOnly 'org.postgresql:postgresql'
}
```

- **Spring Data JDBC** (`org.springframework.boot:spring-boot-starter-data-jdbc`) provides the necessary libraries to persist data in relational databases using Spring Data and JDBC.
- **PostgreSQL** (`org.postgresql:postgresql`) provides a JDBC driver that allows the application to connect to a PostgreSQL database.

The PostgreSQL database is a backing service to the Catalog Service application. As such, it should be handled as an attached resource according to the 15-factor methodology. The attachment is done through resource binding, which in the case of PostgreSQL, consists of:

- a URL to define which driver to use, where to find the database server, and to which database connect the application;
- username and password to establish a connection with the specified database.

Thanks to Spring Boot, you can provide those values as configuration properties. By doing so, you can easily replace the attached database by changing the values for the resource binding.

Open the `application.yml` file for the Catalog Service project and add the properties for configuring the connection with PostgreSQL. Those values are the ones you defined earlier as environment variables when creating the PostgreSQL container.

Listing 5.2 Configure the connection to a database using JDBC (application.yml)

```
spring:
  datasource: ①
    username: user
    password: password
    url: jdbc:postgresql://localhost:5432/polardb_catalog
```

- ① The credentials for a user with privileges to access the given database and a JDBC URL to identify with which database you want to establish a connection.

Opening and closing database connections are relatively expensive operations, so you don't want to do that every time your application accesses data. The solution is *connection pooling*: the application establishes several connections with the database and reuses them, rather than creating new ones at each data access operation. That is a considerable performance optimization.

Spring Boot uses HikariCP for connection pooling, and you can configure it from the `application.yml` file. You want to configure at least a connection timeout (`spring.datasource.hikari.connection-timeout`) and a maximum number of connections in the pool (`spring.datasource.hikari.maximum-pool-size`), because they both affect application resilience and performance. Like for the Tomcat thread pool, multiple factors influence which values to use. As a starting point, you can refer to the HikariCP analysis of pool sizing (<https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing>).

Listing 5.3 Configure the connection pool for interactions with the database (application.yml)

```
spring:
  datasource:
    username: user
    password: password
    url: jdbc:postgresql://localhost:5432/polardb_catalog
    hikari:
      connection-timeout: 2000  ①
      maximum-pool-size: 5  ②
```

- ① The maximum time (ms) waiting to get a connection from the pool.
- ② The maximum number of connections HikariCP will keep in the pool.

Now that you connected a Spring Boot application to a PostgreSQL database, you can go ahead and define what data to persist.

5.2.2 Defining persistent entities with Spring Data

In Catalog Service, you already have a `Book` record representing a domain entity for the application. Depending on the business domain and its complexity, you might want to distinguish the domain entity from the persistent entity, making the domain layer utterly independent of the persistence layer. If you'd like to explore how to model that scenario, I recommend referring to domain-driven design and hexagonal architecture principles.

In this case, the business domain is quite simple, so we'll update the `Book` record to be also a persistent entity.

MAKING A DOMAIN CLASS PERSISTENT

Spring Data JDBC encourages working with immutable entities. Using Java records to model entities is an excellent choice since they're immutable by design and expose an all-args constructor the framework can use to populate objects.

A persistent entity must have a field acting as the identifier for the object, which will translate to the primary key in the database. You can mark a field as an identifier with the `@Id` annotation (from the `org.springframework.data.annotation` package). The database is responsible for generating a unique identifier for each created object automatically.

NOTE

Books are uniquely identified by an ISBN, which we can call a natural key (or business key) for the domain entity. We could decide to use it also as the primary key or introduce a technical key (or surrogate key). There are pros and cons to both approaches. I chose to use a technical key to make it easier to manage and decouple domain concerns from persistence implementation details.

That is enough to create and persist a `Book` in the database. It's also fine when a single user updates an existing `Book` object in isolation. What happens if the same entity is updated by multiple users concurrently? Spring Data JDBC supports *optimistic locking* to address that concern. Users can read data concurrently. When attempting an update operation, the application checks if there has been any change since the last read. If there was, the operation is not performed, and an exception is thrown. The check is based on a numeric field that starts counting from 0 and automatically increases at every update operation. You can mark such a field with the `@Version` annotation (from the `org.springframework.data.annotation` package).

When the `@Id` field is `null` and the `@Version` field is 0, Spring Data JDBC assumes it's a new object. Consequently, it relies on the database to generate an identifier when inserting the new row in the table. When values are provided, it expects to find the object in the database already and update it.

Let's go ahead and add two new fields to the `Book` record for the identifier and the version number. Since both fields are populated and handled by Spring Data JDBC under the hood, using the all-args constructor might be too verbose in situations like when generating test data. For convenience, let's add a static factory method to the `Book` record for building an object by passing only the business fields.

Listing 5.4 Define identifier and version for Book objects (Book.java)

```
package com.polarbookshop.catalogservice.domain;

public record Book (
    @Id ①
    Long id,
    @NotBlank(message = "The book ISBN must be defined.")
    @Pattern(
        regexp = "^(\\d{10}|\\d{13})$",
        message = "The ISBN format must be valid."
    )
    String isbn,
    @NotBlank(message = "The book title must be defined.")
    String title,
    @NotBlank(message = "The book author must be defined.")
    String author,
    @NotNull(message = "The book price must be defined.")
    @Positive(message = "The book price must be greater than zero.")
    Double price,
    @Version ②
    int version
) {
    public static Book of(
        String isbn, String title, String author, Double price
    ) {
        return new Book(
            null, isbn, title, author, price, 0 ③
        );
    }
}
```

- ① Identifies the field as the primary key for the entity.
- ② The entity version number. Used for optimistic locking.
- ③ An entity is considered new when the id is null and the version is 0.

NOTE

Spring Data JPA works with mutating objects, so you can't use Java records. JPA entity classes must be marked with the `@Entity` annotation and expose a no-args constructor. JPA identifiers are annotated with `@Id` and `@Version` from the `javax.persistence` package instead of `org.springframework.data.annotation`.

After adding the new fields, we need to update a few classes using the `Book` constructor, which now requires passing values for `id` and `version`.

The `BookService` class contains the logic for updating books. Open it and change the `editBookDetails()` method to ensure the book identifiers and versions are correctly passed along when calling the data layer.

Listing 5.5 Include the existing identifier and version when updating a book (BookService.java)

```
package com.polarbookshop.catalogservice.domain;

@Service
public class BookService {

    ...

    public Book editBookDetails(String isbn, Book book) {
        return bookRepository.findByIsbn(isbn)
            .map(existingBook -> {
                var bookToUpdate = new Book(
                    existingBook.id(),           ①
                    existingBook.isbn(),
                    book.title(),
                    book.author(),
                    book.price(),
                    existingBook.version());     ②
                return bookRepository.save(bookToUpdate);
            })
            .orElseGet(() -> addBookToCatalog(book));
    }
}
```

- ① Uses the identifier of the existing book.
- ② Uses the version of the existing book. It will be increased automatically if the update operation succeeds.

In BookDataLoader, we can use the new static factory method for building Book objects. The framework will take care of handling the id and version fields.

Listing 5.6 Use the static factory method when creating a book (BookDataLoader.java)

```
package com.polarbookshop.catalogservice.demo;

@Component
@Profile("testdata")
public class BookDataLoader {

    ...

    @EventListener(ApplicationReadyEvent.class)
    public void loadBookTestData() {
        var book1 = Book.of("1234567891", "Northern Lights",
            "Lyra Silverstar", 9.90);           ①
        var book2 = Book.of("1234567892", "Polar Journey",
            "Iorek Polarson", 12.90);          ①
        bookRepository.save(book1);
        bookRepository.save(book2);
    }
}
```

- ① The framework takes care of assigning a value for the identifier and the version under the hood.

I'll leave it to you to update the autotests similarly. You can also extend the tests in

`BookJsonTests` to verify the serialization and deserialization of the new fields. As a reference, you can check `chapter05/05-intermediate/catalog-service` from the code repository accompanying this book.

As a persistent entity, the `Book` record will be automatically mapped to relational resources. Class and field names are transformed into lowercase, and the camel case is translated into words joined by underscores. The `Book` record will result in the `book` table, the `title` field will result in the `title` column, the `price` field will result in the `price` column, and so on. Figure 5.5 shows the mapping between Java object and relational table.

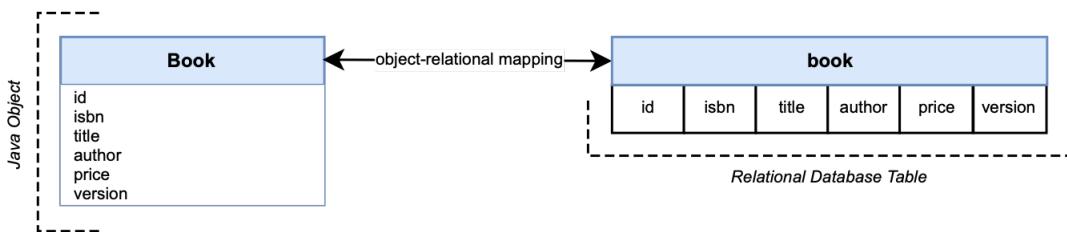


Figure 5.5 Java classes marked as persistent entities are automatically mapped by Spring Data JDBC to relational resources in the database.

CREATING THE DATABASE SCHEMA

The database should have a table defined as shown in figure 5.5 for the mapping to work, or else it will fail. Spring Data offers a feature to initialize a data source at startup time. By default, you can use a `schema.sql` file to create schemas and a `data.sql` file to insert data in the newly created tables. Such files should be placed in the `src/main/resources` folder.

That is a convenient feature, useful for demos and experiments. However, it's too limited for using it in production. As you'll see later in the chapter, it's better to create and evolve relational resources with a more sophisticated tool like Flyway or Liquibase, letting you version-control your database.

NOTE

Hibernate, the foundation for Spring Data JPA, offers an interesting feature for automatically generating schemas from the entities defined in Java. Once again, it's convenient for demos and experiments. However, please think twice before using it in production.

In your Catalog Service project, add a new `schema.sql` file in the `src/main/resources` folder. Then, write the SQL instruction to create the `book` table that will be mapped to the `Book` record in Java.

Listing 5.7 Define the SQL instruction to create the "book" table (schema.sql)

```
DROP TABLE IF EXISTS book;      ①
CREATE TABLE book (
    id                  BIGSERIAL PRIMARY KEY NOT NULL,  ②
    author             varchar(255) NOT NULL,
    isbn               varchar(255) UNIQUE NOT NULL,   ③
    price              float8 NOT NULL,
    title              varchar(255) NOT NULL,          ④
    version            integer NOT NULL             ⑤
);
```

- ① Drops the "book" table if it already exists.
- ② The primary key for the table. The database will generate it as a sequence of numbers ("bigserial" type).
- ③ The "unique" constraint ensures that the same ISBN is assigned only to one book.
- ④ The "not null" constraint ensures the related column is assigned a value.
- ⑤ The entity version number. Stored as an integer.

By default, Spring Data loads the `schema.sql` file only when using an embedded, in-memory database. Since we're using PostgreSQL, we need to enable the functionality explicitly. In the `application.yml` file for your Catalog Service project, add the following configuration to initialize the database schema from the `schema.sql` file.

Listing 5.8 Initialize the database schema from an SQL script (application.yml)

```
spring:
  sql:
    init:
      mode: always
```

At startup, Spring Data will read the file and execute the SQL instruction in the PostgreSQL database to create a new `book` table and make it possible to start inserting data.

In the next section, you'll make it possible to capture audit events related to the persistent entities and keep track of when each row has been inserted into the table and modified the last time.

5.2.3 Enabling and configuring JDBC auditing

When persisting data, it's useful to know the creation date for each row in a table and the date of when it's been updated last. After securing an application with authentication and authorization, you can even register who created each entity and recently updated it. All of that is called *database auditing*.

With Spring Data JDBC, you can enable auditing for all the persistent entities using the `@EnableJdbcAuditing` annotation on a configuration class. In the `com.polarbookshop.catalogservice.config` package, add a `DataConfig` class to gather

JDBC-related configuration.

Listing 5.9 Enable JDBC auditing via annotation configuration (DataConfig.java)

```
package com.polarbookshop.catalogservice.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.jdbc.repository.config.EnableJdbcAuditing;

@Configuration ①
@EnableJdbcAuditing ②
public class DataConfig {}
```

- ① Indicates a class as a source of Spring configuration.
- ② Enables auditing for persistent entities.

NOTE

In Spring Data JPA, you would use the `@EnableJpaAuditing` annotation to enable JPA auditing, and you would annotate the entity class with `@EntityListeners(AuditingEntityListener.class)` to make it listen to audit events, which doesn't happen automatically like in Spring Data JDBC.

When this feature is enabled, audit events are generated whenever data is created, updated, or deleted. Spring Data provides convenient annotations (table 5.1) we can use on dedicated fields to capture the information from such events (*audit metadata*) and store it in the database as part of the entity.

Table 5.1 When database auditing is enabled, these annotations can be used on entity fields to capture audit metadata.

Annotation	What it does on an entity field
<code>@CreatedBy</code>	Identifies the field representing the user that created the entity. It's defined on creation and never changed.
<code>@CreatedDate</code>	Identifies the field representing when the entity has been created. It's defined on creation and never changed.
<code>@LastModifiedBy</code>	Identifies the field representing the user that recently modified the entity. It's updated at every create/update operation.
<code>@LastModifiedDate</code>	Identifies the field representing when the entity has been recently modified. It's updated at every create/update operation.

In Catalog Service, we can add the new `createdDate` and `lastModifiedDate` fields to the Book record. In chapter 12, after introducing Spring Security, we'll extend this object to capture also who created or updated the entity.

Open the Book record, add the two new fields, and update the static factory method accordingly. They can be `null` when instantiating a new object because they will be populated by Spring Data under the hood.

Listing 5.10 Define fields to capture audit metadata in a persistent entity (Book.java)

```
package com.polarbookshop.catalogservice.domain;

public record Book (
    @Id
    Long id,
    ...
    @CreatedDate ①
    Instant createdDate,
    @LastModifiedDate ②
    Instant lastModifiedDate,
    @Version
    int version
) {
    public static Book of(
        String isbn, String title, String author, Double price
    ) {
        return new Book(null, isbn, title, author, price, null, null, 0);
    }
}
```

- ① When the entity has been created.
- ② When the entity has been modified the last time.

After extending the `Book` record, the `BookService` class needs to be updated one more time. Open it and change the `editBookDetails()` method to ensure the audit metadata is correctly passed along when calling the data layer.

Listing 5.11 Include the existing audit metadata when updating a book (BookService.java)

```
package com.polarbookshop.catalogservice.domain;

@Service
public class BookService {

    ...
    public Book editBookDetails(String isbn, Book book) {
        return bookRepository.findByIsbn(isbn)
            .map(existingBook -> {
                var bookToUpdate = new Book(
                    existingBook.id(),
                    existingBook.isbn(),
                    book.title(),
                    book.author(),
                    book.price(),
                    existingBook.createdDate(), ①
                    existingBook.lastModifiedDate(), ②
                    existingBook.version());
                return bookRepository.save(bookToUpdate);
            })
            .orElseGet(() -> addBookToCatalog(book));
    }
}
```

- ① Uses the creation date of the existing book record.
- ② Uses the last modified date of the existing book record. It will be updated automatically by Spring Data if the operation succeeds.

Next, let's update the `schema.sql` file to add the columns for the new fields to the `book` table.

Listing 5.12 Add columns for audit metadata to the "book" table (`schema.sql`)

```
DROP TABLE IF EXISTS book;
CREATE TABLE book (
    id          BIGSERIAL PRIMARY KEY NOT NULL,
    author      varchar(255) NOT NULL,
    isbn        varchar(255) UNIQUE NOT NULL,
    price       float8 NOT NULL,
    title       varchar(255) NOT NULL,
    created_date timestamp NOT NULL,   ①
    last_modified_date timestamp NOT NULL, ②
    version     integer NOT NULL
);
```

- ① When the entity has been created. Stored as a timestamp.
- ② When the entity has been modified the last time. Stored as a timestamp.

I'll leave it to you to update the autotests accordingly, when necessary. You can also extend the tests in `BookJsonTests` to verify the serialization and deserialization of the new fields. As a reference, you can check `Chapter05/05-intermediate/catalog-service` from the code repository accompanying this book.

So far, you have got everything in place to map your Java objects to relational objects in a database, including audit metadata. You still need a way to access data from the database, though. That's the subject of the next section.

5.2.4 Data repositories with Spring Data

The *repository* pattern provides an abstraction to access data independently of their source. An example of a repository is the `BookRepository` interface used by `BookService`. The domain layer, which contains the business logic, doesn't need to know where the data comes from as long as it gets access. In chapter 3, we added an implementation for the repository interface to store data in memory. Now that we're building the persistence layer, we need a different implementation to access data from PostgreSQL.

The good news is that we can use Spring Data repositories, a technical solution that provides access to data from a data store independently of the specific persistence technology used. It's one of the most valuable features of Spring Data since we can use the same repository abstraction in any persistence scenario, both relational and non-relational.

USING A DATA REPOSITORY

When using Spring Data repositories, your responsibility is limited to defining an interface. At startup time, Spring Data will generate an implementation for your interface on the fly. In the Catalog Service project (`catalog-service`), go ahead and delete the `InMemoryBookRepository` class.

Let's now see how to refactor the `BookRepository` interface from the Catalog Service project. First, it should extend one of the available `Repository` interfaces provided by Spring Data. Most Spring Data modules add `Repository` implementations specific to the data source supported. The Catalog Service application requires standard CRUD operations on `Book` objects, so you can make the `BookRepository` interface extend from `CrudRepository`.

`CrudRepository` provides methods to perform CRUD operations on books, including `save()` and `findAll()`, so you can remove their explicit declaration from your interface. The default methods defined by `CrudRepository` for `Book` objects are based on their `@Id`-annotated field. Since the application needs to access books based on the ISBN, we must explicitly declare those operations.

Listing 5.13 Repository interface for accessing books (BookRepository.java)

```
package com.polarbookshop.catalogservice.domain;

import java.util.Optional;
import org.springframework.data.jdbc.repository.query.Modifying;
import org.springframework.data.jdbc.repository.query.Query;
import org.springframework.data.repository.CrudRepository;

public interface BookRepository
    extends CrudRepository<Book, Long> { ①

    Optional<Book> findByIsbn(String isbn); ②
    boolean existsByIsbn(String isbn);

    @Modifying ③
    @Query("delete from Book where isbn = :isbn") ④
    void deleteByIsbn(String isbn);
}
```

- ① Extends a repository providing CRUD operations, specifies the type of managed entity (`Book`) and its primary key type (`Long`).
- ② Methods implemented by Spring Data at runtime.
- ③ Identifies an operation that will modify the database state.
- ④ Declares the query that Spring Data will use to implement the method.

At startup time, Spring Data will provide an implementation for `BookRepository` with all the most common CRUD operations and the methods you declare in the interface. There are two main options for defining custom queries in Spring Data.

- Using the `@Query` annotation to provide an SQL-like statement that will be executed by the method.
- Defining query methods following a specific naming convention, as described in the official documentation (<https://spring.io/projects/spring-data>). Generally, you can build a method name by combining multiple pieces, as described in table 5.2. At the moment of writing, Spring Data JDBC supports this option only for read operations. On the other hand, Spring Data JPA provides full support for it.

Table 5.2 You can add custom queries to a repository and have Spring Data generate an implementation for you by following a specific naming convention comprising these building blocks.

Repository method building block	Examples
Action	<code>find, exists, delete, count</code>
Limit	<code>One, All, First10</code>
By	-
Property expressions	<code>findByIsbn, findByTitleAndAuthor, findByAuthorOrPrice</code>
Comparison	<code>findByTitleContaining, findByIsbnEndingWith, findByPriceLessThan</code>
Ordering operator	<code>orderByTitleAsc, orderByTitleDesc</code>

Using some of the methods provided by the `CrudRepository` interface and inherited by `BookRepository`, we can improve the `BookDataLoader` class so that we start from an empty database during development and create books with one single command.

Listing 5.14 Use the new methods provided by Spring Data for delete and save operations (`BookDataLoader.java`)

```
package com.polarbookshop.catalogservice.demo;

@Component
@Profile("testdata")
public class BookDataLoader {
    private final BookRepository bookRepository;

    public BookDataLoader(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @EventListener(ApplicationReadyEvent.class)
    public void loadBookTestData() {
        bookRepository.deleteAll();      ①
        var book1 = Book.of("1234567891", "Northern Lights",
            "Lyra Silverstar", 9.90);
        var book2 = Book.of("1234567892", "Polar Journey",
            "Iorek Polarson", 12.90);
        bookRepository.saveAll(List.of(book1, book2));  ②
    }
}
```

- ① Deletes all existing books, if any, to start from an empty database.
- ② Saves multiple objects at once.

DEFINING TRANSACTIONAL CONTEXTS

The repositories provided by Spring Data come configured with transactional contexts for all the operations. For example, all methods in `CrudRepository` are transactional. That means you can safely call the `saveAll()` method, knowing that it will be executed in a transaction.

When you add your own query methods as you did for `BookRepository`, it's up to you to define which ones should be part of a transaction. You can rely on the declarative transaction management provided by the Spring Framework and use the `@Transactional` annotation (from the `org.springframework.transaction.annotation` package) on classes or methods to ensure they are executed as part of a single *unit of work*.

Among the custom methods you defined in `BookRepository`, `deleteByIsbn()` is a good candidate for being transactional since it modifies the database state. You can ensure it runs in a transaction by applying the `@Transactional` annotation.

Listing 5.15 Define transactional operations (`BookRepository.java`)

```
package com.polarbookshop.catalogservice.domain;

import java.util.Optional;
import org.springframework.data.jdbc.repository.query.Modifying;
import org.springframework.data.jdbc.repository.query.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;

public interface BookRepository extends CrudRepository<Book, Long> {

    Optional<Book> findByIsbn(String isbn);
    boolean existsByIsbn(String isbn);

    @Modifying
    @Transactional ①
    @Query("delete from Book where isbn = :isbn")
    void deleteByIsbn(String isbn);
}
```

- ① Identifies the method for being executed in a transaction.

NOTE	For more information about the declarative transaction management provided by Spring Framework, you can refer to the official documentation (https://spring.io/projects/spring-framework).
-------------	--

Great job! You successfully added data persistence capabilities to the Catalog Service application. Let's verify if it works correctly. First of all, make sure the PostgreSQL container is still running. If not, run it as described at the beginning of this chapter. Then, start the application (`./gradlew bootRun`), send HTTP requests to each REST endpoint, and ensure it works as expected. When you're done, remove the database container (`docker rm -fv polar-postgres`) and stop the application (`Ctrl+C`).

TIP

In the repository accompanying the book, you can find useful commands to query the PostgreSQL database directly and verify schemas and data generated by the application (Chapter05/05-intermediate/catalog-service/README.md).

Manual verification of data persistence is fine, but automated is better. That's what the next section is all about.

5.3 Testing data persistence with Spring and Testcontainers

In the previous sections, we added data persistence functionality to an application by developing against a PostgreSQL database in a container, the same technology used in production. That was a good step towards the environment parity recommended by the 15-Factor methodology. Keeping all environments as similar as possible improves the quality of the project.

Data sources are one of the primary causes of differences across environments. It's common practice to use an in-memory database while developing locally, something like H2 or HSQL. And that affects the predictability and robustness of your applications. Even if all relational databases speak the SQL language and Spring Data JDBC provides generic abstractions, each vendor has its own dialect and unique features that make it essential to use the same database in production that you use in development and testing. Otherwise, you could be prevented from catching errors that might happen only in production.

'What about testing?' you might ask. That is an excellent question. Another reason for using in-memory databases is to make integration tests more accessible. However, integration tests are also supposed to test the integration with your application's external services. Using something like H2 makes those tests less reliable. Each commit should be a candidate for release when adopting a continuous delivery approach. Suppose the autotests run by the deployment pipeline are not using the same backing services used in production. In that case, you need to do extra manual testing before deploying the application safely in production because you can't be sure it will work correctly. Therefore, it's essential to reduce the gap between environments.

Docker makes it easier to set up and develop applications with an actual database locally, as you experienced with PostgreSQL. In a similar way, Testcontainers (a Java library for testing) makes it easy to use backing services as containers in the context of integration tests.

This section will show you how to write slice tests for the data persistence layer using the `@DataJdbcTest` annotation and include a database in integration tests using the `@SpringBootTest` annotation. In both cases, you'll rely on Testcontainers to run the autotests against an actual PostgreSQL database.

5.3.1 Configuring Testcontainers for PostgreSQL

Testcontainers (<https://testcontainers.org>) is a Java library for testing, supports JUnit, and provides lightweight, throwable containers such as databases, message brokers, and web servers. It's perfect for implementing integration tests with the actual backing services used in production. The result is more reliable and stable tests, which lead to higher-quality applications and favor continuous delivery practices.

You can configure a lightweight PostgreSQL container with Testcontainers and use it in your autotests involving a data persistence layer. Let's see how it works.

First, you need to add a dependency to the Testcontainers module for PostgreSQL in the `build.gradle` file of your Catalog Service project. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 5.16 Add dependency for Testcontainers in Catalog Service (build.gradle)

```
ext {
    ...
    set('testcontainersVersion', "1.17.2")      ①
}

dependencies {
    ...
    testImplementation 'org.testcontainers:postgresql' ②
}

dependencyManagement {
    imports {
        ...
        mavenBom "org.testcontainers:
testcontainers-bom:${testcontainersVersion}" ③
    }
}
```

- ① Defines the Testcontainers version to use.
- ② Provides container management features for PostgreSQL databases.
- ③ BOM (Bill of Material) for Testcontainers dependency management.

When running tests, we want the application to use a PostgreSQL instance provided by Testcontainers rather than the one we configured earlier via the `spring.datasource.url` property. We can overwrite that value in a new `application-integration.yml` file created under `src/test/resources`. Any property defined in this file will take precedence over the main one when the `integration` profile is enabled. In this case, we'll overwrite the value for `spring.datasource.url` following a format defined by Testcontainers.

Create a new `application-integration.yml` file in `src/test/resources` and add the following configuration.

Listing 5.17 Use a PostgreSQL data source provided by Testcontainers (application-integration.yml)

```
spring:
  datasource:
    url: jdbc:tc:postgresql:14.3:/ ①
```

- ① Identifies the PostgreSQL module in Testcontainers. "14.3" is the version of PostgreSQL to use.

That's all we need to configure Testcontainers. When the `integration` profile is enabled, Spring Boot will use the PostgreSQL container instantiated by Testcontainers. We're now ready to write autotests to verify the data persistence layer.

5.3.2 Testing data persistence with `@DataJdbcTest` and Testcontainers

As you might recall from chapter 3, Spring Boot allows you to run integration tests by loading only the Spring components used by a specific application slice (*slice tests*). In Catalog Service, we created tests for the MVC and the JSON slices. Now, I'll show you how to write tests for the data slice.

Create a `BookRepositoryJdbcTests` class and mark it with the `@DataJdbcTest` annotation. That will trigger Spring Boot to include all Spring Data JDBC entities and repositories in the application context. It will also autoconfigure `JdbcAggregateTemplate`, a lower-level object we can use to set up the context for each test case instead of using the repository (the object under testing).

Listing 5.18 Integration tests for the Data JDBC slice (BookRepositoryJdbcTests.java)

```

package com.polarbookshop.catalogservice.domain;

import java.util.Optional;
import com.polarbookshop.catalogservice.config.DataConfig;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.DataJdbcTest;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.context.annotation.Import;
import org.springframework.data.jdbc.core.JdbcAggregateTemplate;
import org.springframework.test.context.ActiveProfiles;
import static org.assertj.core.api.Assertions.assertThat;

@DataJdbcTest ①
@Import(DataConfig.class) ②
@AutoConfigureTestDatabase(③
    replace = AutoConfigureTestDatabase.Replace.NONE
)
@ActiveProfiles("integration") ④
class BookRepositoryJdbcTests {

    @Autowired
    private BookRepository bookRepository;

    @Autowired
    private JdbcAggregateTemplate jdbcAggregateTemplate; ⑤

    @Test
    void findBookByIsbnWhenExisting() {
        var bookIsbn = "1234561237";
        var book = Book.of(bookIsbn, "Title", "Author", 12.90);
        jdbcAggregateTemplate.insert(book); ⑥

        Optional<Book> actualBook = bookRepository.findByIsbn(bookIsbn);

        assertThat(actualBook).isPresent();
        assertThat(actualBook.get().isbn()).isEqualTo(book.isbn());
    }
}

```

- ① Identifies a test class that focuses on Spring Data JDBC components.
- ② Imports the data configuration. Needed to enable auditing.
- ③ Disables the default behavior relying on an embedded test database since we want to use Testcontainers.
- ④ Enables the "integration" profile to load configuration from "application-integration.yml".
- ⑤ A lower-level object to interact with the database.
- ⑥ JdbcAggregateTemplate is used to prepare the data targeted by the test.

The `@DataJdbcTest` annotation encapsulates handy features. For example, it makes each test method run in a transaction and rolls it back at its end, keeping the database clean. After running the test method in listing 5.18, the database will not contain the book created in `findBookByIsbnWhenExisting()` because the transaction is rolled back at the end of the method execution.

Let's verify if the Testcontainers configuration works. First, make sure the Docker Engine is running in your local environment. Then, open a Terminal window, navigate to the root folder of your Catalog Service project, and run this command to ensure the tests are successful. Under the hood, Testcontainers creates a PostgreSQL container before the test execution and removes it at the end.

```
$ ./gradlew test --tests BookRepositoryJdbcTests
```

In the code repository accompanying the book, you can find more examples of unit and integration tests for the Catalog Service project. The following section will cover how to run full integration tests with Testcontainers.

5.3.3 Integration tests with `@SpringBootTest` and Testcontainers

In the Catalog Service application, we already have a `CatalogServiceApplicationTests` class annotated with `@SpringBootTest` and containing full integration tests. The Testcontainers configuration we defined earlier applies to all autotests for which the `integration` profile is enabled, so we need to add the profile configuration to the `CatalogServiceApplicationTests` class.

Listing 5.19 Enable "integration" profile for the integration tests (CatalogServiceApplicationTests.java)

```
package com.polarbookshop.catalogservice;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("integration") ①
class CatalogServiceApplicationTests {
    ...
}
```

- ① Enables the "integration" profile to load configuration from "application-integration.yml".

Open a Terminal window, navigate to the root folder of your Catalog Service project, and run this command to ensure the tests are successful. Under the hood, Testcontainers creates a PostgreSQL container before the test execution and removes it at the end.

```
$ ./gradlew test --tests CatalogServiceApplicationTests
```

Good job! You have added data persistence to a Spring Boot application and written tests while ensuring environment parity. Let's move on and complete this chapter by discussing how to manage schemas and data in production.

5.4 Managing databases in production with Flyway

It's good practice registering any database change just like you do for your application source code through version control. You need a deterministic and automated way to infer the database state, whether specific changes have already been applied, how to recreate a database from scratch, and how to migrate it in a controlled, repeatable, and reliable way. The continuous delivery approach encourages automating as much as possible, including database management.

In the Java ecosystem, the two most used tools to track, version, and deploy database changes are Flyway (<https://flywaydb.org>) and Liquibase (<https://liquibase.org>). Both of them are fully integrated with Spring Boot. This section will show you how to use Flyway.

5.4.1 Understanding Flyway: Version control for your database

Flyway is a tool that provides version control for your database. It offers a single source of truth for the version of your database state and keeps track of any change incrementally. It automates changes and lets you reproduce or roll back the state of a database. Flyway is highly reliable, safe to use in cluster environments, and supports several relational databases, including the cloud ones like Amazon RDS, Azure Database, and Google Cloud SQL.

NOTE

In this section, I will introduce some features offered by Flyway, but I recommend you check the official documentation to discover all the powerful possibilities provided by this tool (<https://flywaydb.org>).

At its core, Flyway manages database changes. Any database change is called a *migration*. Migrations can be either *versioned* or *repeatable*. Versioned migrations are identified by a unique version number and applied in order exactly once. For each *regular* versioned migration, you can also provide an optional *undo* migration to revert its effects (in case something goes wrong). They can be used to create/alter/drop relational objects like schemas, tables, columns, and sequences or correct data. On the other hand, repeatable migrations are applied every time their checksum changes. They can be used for creating/updating views, procedures, and packages.

Both migrations can be defined in standard SQL scripts (useful for DDL changes) or Java classes (useful for DML changes, like data migrations). Flyway keeps track of which migrations have already been applied through a `flyway_schema_history` table automatically created in the database the first time it runs. You can picture migrations as commits in a Git repository and the *schema history table* as the repository log containing the list of all the commits applied over time (figure 5.6).

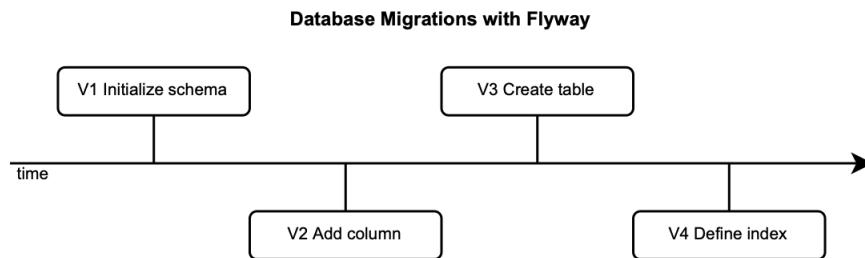


Figure 5.6 Flyway migrations represent database changes that can be pictured as commits in a Git repository.

NOTE

A prerequisite for using Flyway is that both the database to manage and a user with the correct access privileges exist. Once you have a database and a user, Flyway can manage database changes for you. You shouldn't use Flyway to manage users.

You can use Flyway in standalone mode or embedded in a Java application. Spring Boot provides auto-configuration for it, making it very convenient to include Flyway in your applications. When integrated with Spring Boot, Flyway will search for SQL migrations in the `src/main/resources/db/migration` folder and Java migrations in `src/main/java/db/migration`.

This functionality is one of those administrative processes described by the 15-Factors methodology introduced in chapter 2. In this case, the strategy adopted for managing such a process was to embed it in the application itself. By default, it's activated during the application startup phase. Let's see how to implement it for Catalog Service.

Open your Catalog Service project (`catalog-service`) and add a dependency to Flyway in the `build.gradle` file. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 5.20 Add dependency for Flyway in Catalog Service (build.gradle)

```

dependencies {
    ...
    implementation 'org.flywaydb:flyway-core'
}
    
```

In the next section, you'll learn how to create your first migration to initialize the database schema.

5.4.2 Initializing a database schema with Flyway

The first database change to apply is usually for initializing the schema. So far, we've been relying on the built-in data source initialization capabilities offered by Spring Boot and providing a `schema.sql` file with the SQL statements to run. Now, we can initialize the schema using an SQL Flyway migration.

First, delete the `schema.sql` file and remove the `spring.sql.init.mode` property from the `application.yml` file in your Catalog Service project.

Next, create a `src/main/resources/db/migration` folder. It's the path where Flyway will look for SQL migrations by default. Inside the folder, create a `v1__Initial_schema.sql` file that will contain the SQL statement for initializing the database schema required by the Catalog Service application. Ensure you type two underscores after the version number.

Flyway expects SQL migration files to comply with a specific naming pattern. Regular versioned migrations should follow this structure:

- *Prefix.* v for versioned migrations
- *Version.* Version number using dots or underscores to separate it into multiple parts (e.g., 2.0.1)
- *Separator.* Two underscores: __
- *Description.* Words separated by underscores
- *Suffix.* .sql

In the `v1__Initial_schema.sql` migration script, you can include the SQL instructions to create a `book` table that Spring Boot JDBC will map to the `Book` persistent entity.

Listing 5.21 Flyway migration script for schema initialization (V1__Initial_schema.sql)

```
CREATE TABLE book (
    id          BIGSERIAL PRIMARY KEY NOT NULL,      ②
    author      varchar(255) NOT NULL,
    isbn        varchar(255) UNIQUE NOT NULL,       ③
    price       float8 NOT NULL,
    title       varchar(255) NOT NULL,
    created_date timestamp NOT NULL,
    last_modified_date timestamp NOT NULL,
    version     integer NOT NULL
);
```

- ① Definition of the book table.
- ② Declares the "id" field as the primary key.
- ③ Constrains the "isbn" field to be unique.

When you let Flyway manage changes to the database schema, you gain all the benefits of version control. You can now start a new PostgreSQL container following the instructions

provided earlier in the chapter (if you still have the previous one running, remove it with `docker rm -fv polar-postgres`), run the application (`./gradlew bootRun`) and verify if everything works correctly.

NOTE

In the repository accompanying the book, you can find useful commands to query the PostgreSQL database directly and verify schemas and data generated by Flyway (Chapter05/05-end/catalog-service/README.md).

Your autotests will also use Flyway. Go ahead and run them; they should all succeed. Once you’re done, push your changes to your remote Git repository and check the commit stage results from GitHub Actions. They should be successful as well. Finally, stop the application execution (`Ctrl+C`) and the PostgreSQL container (`docker rm -fv polar-postgres`).

In the final section, you’ll learn how to use Flyway migrations to evolve a database.

5.4.3 Evolving a database with Flyway

Imagine you completed the Catalog Service application and deployed it in production. Employees of the bookshop have started adding books to the catalog and gathered feedback about the application. The result is a new requirement for the catalog functionality: it should provide information about the book publisher. How can you do that?

Since the application is already in production and some data have already been created, you can use Flyway to apply a new database change altering the `book` table to add a new `publisher` column. Create a new `v2__Add_publisher_column.sql` file in the `src/main/resources/db/migration` folder of your Catalog Service project, and add the SQL instruction to add the new column.

Listing 5.22 Flyway migration script for updating a table schema (V2__Add_publisher_column.sql)

```
ALTER TABLE book
ADD COLUMN publisher varchar(255);
```

Then, update the Book Java record accordingly. The change should take into account that, in production, there are already books saved in the database without the publisher information, so it must be an optional field, or else the existing data will become invalid. Update also the static factory method accordingly.

Listing 5.23 Add a new optional field to an existing data entity (Book.java)

```
package com.polarbookshop.catalogservice.domain;

public record Book (
    @Id
    Long id,
    ...
    String publisher, ①
    @CreatedDate
    Instant createdDate,
    @LastModifiedDate
    Instant lastModifiedDate,
    @Version
    int version
) {
    public static Book of(
        String isbn, String title, String author, Double price, String publisher
    ) {
        return new Book(
            null, isbn, title, author, price, publisher, null, null, 0
        );
    }
}
```

- ① A new, optional field.

NOTE

After this change, you must update the classes that call the static factory method and the `Book()` constructor to include a value for the `publisher` field. You can use either `null` (since it's optional) or a `String` value like `Polarsophia`. Check the source code ([Chapter05/05-end/catalog-service](#)) to see the final result. In the end, check that both the autotests and the application run correctly.

When this new version of Catalog Service is deployed to production, Flyway will skip the `v1__Initial_schema.sql` migration because it's already been applied but will execute the change described in `v2__Add_publisher_column.sql`. At this point, bookshop employees can start including the publisher name when adding new books to the catalog while all the existing data will still be valid.

What if you need to make the `publisher` field mandatory? You can do that in a third version of Catalog Service, using an SQL migration to enforce the `publisher` column to be `NON NULL` and implementing a Java migration that adds a publisher to all the existing books in the database which don't have one already.

This two-step approach is very common to ensure backward compatibility during an upgrade. As you'll learn in a later chapter, there are usually multiple instances of the same application

running. Deploying a new version is generally done through a *rolling upgrade* procedure consisting of updating one (or a few) instance(s) at a time to ensure zero downtime. There will be both old and new versions of the application running during the upgrade, so it's paramount that the old instances can still run correctly even after applying the database changes introduced in the latest version.

5.5 Summary

- The state is everything that should be preserved when shutting down a service and spinning up a new instance.
- Data services are the stateful components of a cloud native architecture, requiring storage technologies to persist the state.
- Using data services in the cloud is challenging because it's a dynamic environment.
- Some aspects to consider when choosing a data service are scalability, resilience, performance, and compliance with specific regulations and laws.
- You can use data services offered and managed by your cloud provider or manage your own, either relying on virtual machines or containers.
- Spring Data provides common abstractions and patterns for accessing data, making it straightforward to navigate the different modules dedicated to relational and non-relational databases.
- The main elements in Spring Data are database drivers, entities, and repositories.
- Spring Data JDBC is a framework that supports integrating Spring applications with relational databases relying on a JDBC driver.
- Entities represent domain objects and can be managed by Spring Data JDBC as immutable objects. They must have the field hosting the primary key annotated with `@Id`.
- Spring Data lets you capture audit metadata whenever an entity is created or updated. You can enable this feature with `@EnableJdbcAuditing`.
- Data repositories grant access to entities from the database. You need to define an interface, and then Spring Data will generate the implementation for you.
- Depending on your requirements, you can extend one of the available `Repository` interfaces provided by Spring Data, such as `CrudRepository`.
- In Spring Data JDBC, all mutating custom operations (create/update/delete) should run in transactions.
- Use the `@Transactional` annotation to run operations in a single unit of work.
- You can run integration tests for the Spring Data JDBC slice using the `@DataJdbcTest` annotation.
- Environment parity is essential for the quality and reliability of your tests and deployment pipeline.
- You can test the integration between your application and backing services defined as containers using the `Testcontainers` library. It lets you use lightweight, throwable containers in your integration tests.
- Database schemas are critical for applications. In production, you should use a tool like Flyway, which provides version control for your database.
- Flyway should manage any database change to ensure reproducibility, traceability, and reliability.

Containerizing Spring Boot

This chapter covers

- Working with container images on Docker
- Packaging Spring Boot applications as container images
- Managing Spring Boot containers with Docker Compose
- Automating image build and push with GitHub Actions

So far, we have developed a Catalog Service application that exposes a REST API and persists data through a PostgreSQL database running inside a container. We’re getting closer to deploying the first components of the Polar Bookshop system to a Kubernetes cluster. Before doing that, you need to learn how to package Spring Boot applications as container images and manage their lifecycle.

This chapter will teach you the essential characteristics of container images and how to build one. We’ll use Docker to work with containers, but you can do the same with any other container runtime compatible with the Open Container Initiative (OCI) standards (<https://opencontainers.org>). In the following, whenever I refer to a *container image* or *Docker image*, I mean an image compatible with the OCI Image Specification.

Along the way, I’ll share with you several considerations regarding building container images for production, such as security and performance. We’ll explore two possibilities: Dockerfiles and Cloud Native Buildpacks.

When we start working with more than one container, the Docker CLI is not very efficient. Instead, we’ll begin to use Docker Compose for managing several containers and their lifecycle.

Finally, we’ll continue the work on the deployment pipeline started in chapter 3. I’ll show you

how to add new steps to the commit stage for packaging and publishing container images automatically to GitHub Container Registry.

NOTE

The source code for the examples in this chapter is available in the Chapter06/06-begin and Chapter06/06-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

6.1 Working with container images on Docker

In chapter 2, I introduced the main components of the Docker platform. The Docker Engine has a client/server architecture. The Docker CLI is the client you use to interact with the Docker server. The latter is responsible for managing all Docker resources (for example, images, containers, and networks) through the Docker daemon. The server can also interact with container registries to upload and download images. For your convenience, figure 6.1 shows the interaction flow among those components again.

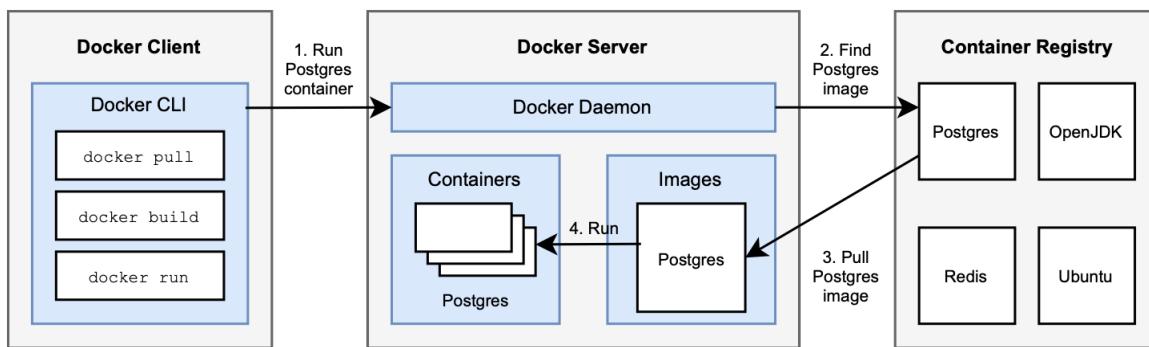


Figure 6.1 The Docker Engine has a client/server architecture and interacts with a container registry.

This section will continue from where we left off in chapter 2 and elaborate more on container images, lightweight executable packages that include everything needed to run the application inside. You'll learn the main characteristics of a container image, how to create one, and finally publish it to a container registry. Before moving on, make sure the Docker Engine on your computer is up and running by executing the `docker version` command from a Terminal window.

6.1.1 Understanding container images

Container images are the product of executing an ordered sequence of instructions, each resulting in a *layer*. Each image is made up of several layers. Each layer represents a modification produced by the corresponding instruction. The final artifact, an image, can be run as a container.

Images can be created from scratch or starting from a base image. The latter is the most common

approach. For example, you can start from an Ubuntu image and apply a series of modifications on top of it. The sequence of instructions would be as follows:

1. Use Ubuntu as the base image.
2. Install the Java Runtime Environment.
3. Run the `java --version` command.

Each of these instructions will generate a layer, producing the final container image shown in figure 6.2.

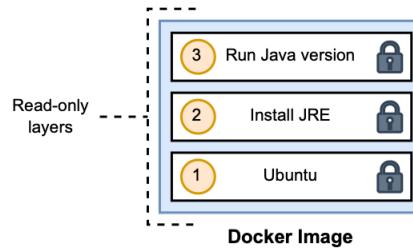


Figure 6.2 Container images are composed of an ordered sequence of read-only layers. The first one represents the base image; the others represent modifications applied on top of it.

All layers in a container image are read-only. Once they are applied, you can't modify them anymore. If you need to change something, you can do so by applying a new layer on top of it (by executing a new instruction). Changes applied to the upper layers will not affect the lower ones. This approach is called *copy-on-write*: a copy of the original item is created in the upper layer, and changes are applied to the copy rather than the original item.

When an image is *run* as a container, one last layer is automatically applied on top of all the existing ones: the *container layer*. It is the only writable layer used to store data created during the execution of the container itself. At runtime, this layer might be used to generate files required by the application to run or maybe for storing temporary data. Even though it's writable, remember that it's volatile: once you delete your container, everything stored in that layer is gone. Figure 6.3 compares the layers in a running container and those in the corresponding image.

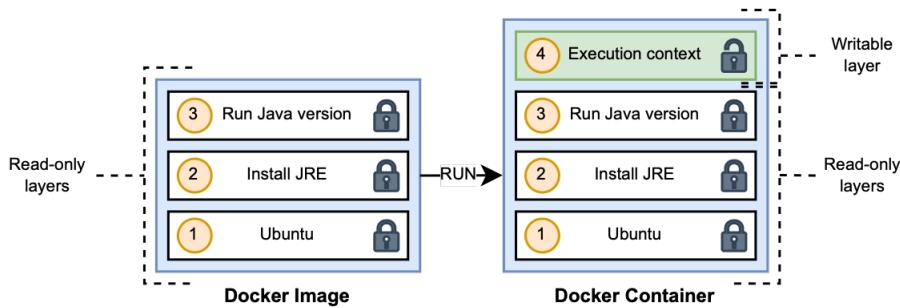


Figure 6.3 Running containers have an extra layer on top of the image layers. That is the only writable layer, but remember that it's volatile.

NOTE

The fact that all the layers in a container image are read-only has some security implications. You should never store secrets or sensitive information in the lower layers because they will always be accessible, even if the upper layers delete them. For example, you shouldn't package passwords or encryption keys within a container image.

So far, you have learned how container images are composed, but you haven't yet seen how to create one. That's coming up next.

6.1.2 Creating images with Dockerfiles

Following the OCI format, you can define a container image by listing the sequence of instructions in a particular file called *Dockerfile*. It's a script that acts as a recipe with all the steps to *build* the desired image.

In a Dockerfile, each instruction starts with a command from the Docker-specific syntax. Then, you can pass familiar shell commands as arguments to the instructions, depending on which Linux distribution you're using as your base image. The format is as follows:

INSTRUCTION arguments

NOTE

Docker supports Linux containers on machines with AMD64 and ARM64 architectures. It also supports working with Windows containers (runnable only on Windows systems), but we will exclusively use Linux containers in this book.

Let's put that into practice by defining a Dockerfile to build the container image mentioned in the previous section, comprising of the following main instructions:

1. Use Ubuntu as the base image.
2. Install the Java Runtime Environment.
3. Run the `java --version` command.

Create a `my-java-image` folder and an empty file inside named `Dockerfile`, with no extension (`Chapter06/06-end/my-java-image`). You might call it differently, but in this case, let's go with the default convention.

Listing 6.1 Dockerfile with the instructions to build a container image (Dockerfile)

```
FROM ubuntu:22.04      ①
RUN apt-get update && apt-get install -y default-jre    ②
ENTRYPOINT ["java", "--version"]    ③
```

- ① Bases the new image on the official image for Ubuntu, version 22.04.
- ② Installs the JRE using familiar bash commands.
- ③ Define the execution entry point for the running container.

By default, Docker is configured to use Docker Hub to find and download images. That's from where the `ubuntu:22.04` image comes. Docker Hub is a registry that you can use for free (within specific rate limits) and is automatically configured when installing Docker.

The `java --version` command is the *entry point* of the executing container. If you don't specify any entry point, the container will not run as an executable. Unlike virtual machines, containers are meant to run tasks, not operating systems. Indeed, when running an Ubuntu container with `docker run ubuntu`, the container will exit right away because no task has been defined as the entry point, only the operating system.

The most common instructions defined in a Dockerfile are listed in table [6.1](#).

Table 6.1 The most common instructions used in a Dockerfile for building container images.

Instruction	Description	Example
FROM	Defines the base image for the subsequent instructions. It must be the first instruction in a Dockerfile.	<code>FROM ubuntu:22.04</code>
LABEL	Adds metadata to the image, following a key/value format. Multiple <code>LABEL</code> instructions can be defined.	<code>LABEL version="1.2.1"</code>
ARG	Defines a variable that users can pass at build-time. Multiple <code>ARG</code> instructions can be defined.	<code>ARG JAR_FILE</code>
RUN	Executes the commands passed as arguments in a new layer on top of the existing ones. Multiple <code>RUN</code> instructions can be defined.	<code>RUN apt-get update && apt-get install -y default-jre</code>
COPY	Copies files or directories from the host file system to the one inside the container.	<code>COPY app-0.0.1-SNAPSHOT.jar app.jar</code>
USER	Defines the user to run all the subsequent instructions and the image itself (as a container).	<code>USER sheldon</code>
ENTRYPOINT	Defines the program to execute when the image is run as a container. Only the last <code>ENTRYPOINT</code> instruction in a Dockerfile is considered.	<code>ENTRYPOINT ["/bin/bash"]</code>
CMD	Specifies defaults for an executing container. If the <code>ENTRYPOINT</code> instruction is defined, they are passed as arguments. If not, it should also contain an executable. Only the last <code>CMD</code> instruction in a Dockerfile is considered.	<code>CMD ["sleep", "10"]</code>

Once you have the specifications for creating a container image declared in a Dockerfile, you can use the `docker build` command to run all the instructions one by one, producing a new layer for each of them. The whole process from Dockerfile to image to container is described in figure [6.4](#). Notice how the first instruction in a Dockerfile produces the lowest layer of the image.

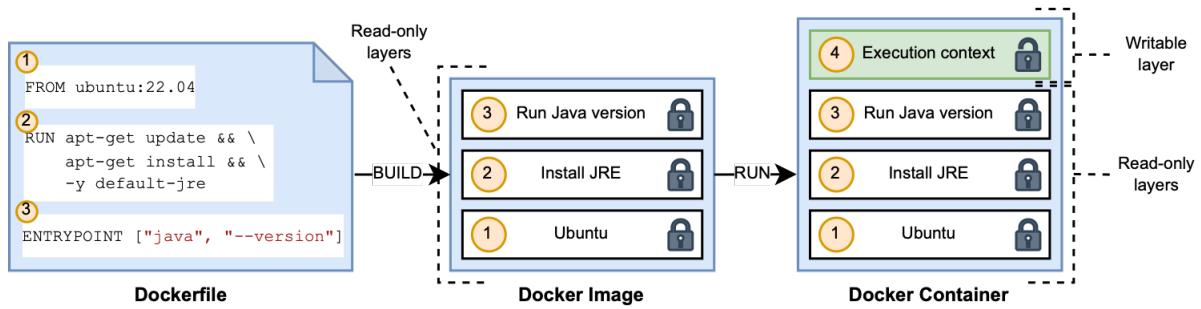


Figure 6.4 Images are built starting from Dockerfiles. Each instruction in a Dockerfile results in an ordered sequence of layers in the image.

Now, open a Terminal window, navigate to the `my-java-image` folder where your Dockerfile is located, and run the following command (don't forget the final dot).

```
$ docker build -t my-java-image:1.0.0 .
```

The command syntax is explained in figure [6.5](#).

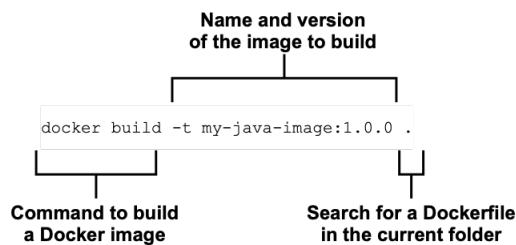


Figure 6.5 The Docker CLI command to build a new image with a given name and version.

When it's done, you can get some details about your newly created image using the `docker images` command.

```
$ docker images my-java-image
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
my-java-image   1.0.0   96d1f58857aa   6 seconds ago   549MB
```

The layered approach makes the image build very performant. Each image layer is a delta from the previous one, and Docker caches all of them. If you make changes to only one of them and build the image again, only that layer and the subsequent ones are recreated. For that reason, it's recommended to order layers based on their likelihood to change, optimizing the image build process. If you run a container from a new version of an image stored in a registry, only the new layers will be downloaded, improving performance at runtime.

Therefore, it's good practice to have the instructions that change more often towards the end of the Dockerfile to take advantage of the layer caching and improve the overall build and run performance.

A container image can be run with the `docker run` command, which starts a container and executes the process described in the Dockerfile as the entry point.

```
$ docker run --rm my-java-image:1.0.0
openjdk 11.0.15 2022-04-19
OpenJDK Runtime Environment (build 11.0.15+10-Ubuntu-0ubuntu0.22.04.1)
OpenJDK 64-Bit Server VM (build 11.0.15+10-Ubuntu-0ubuntu0.22.04.1, mixed mode)
```

After completing the execution, the container will stop. Since you used the `--rm` argument, the container gets removed automatically after the execution ends.

NOTE

Running the previous command, you could see that the default OpenJDK in Ubuntu 22.04 is Java 11 instead of the 17 version we are working with throughout the book.

Let's now see how to publish the image to a container registry.

6.1.3 Publishing images on GitHub Container Registry

So far, you have learned how to define, build, and run a container image. In this section, I'll complete the picture by expanding on container registries.

A container registry is to images what a Maven repository is to Java libraries. Many cloud providers offer their own registry solutions with extra services like image scanning for vulnerabilities and certified images. By default, a Docker installation is configured to use the container registry provided by the Docker company (Docker Hub), hosting images for many popular open-source projects, like PostgreSQL, RabbitMQ, and Redis. We'll keep using it to pull images for third parties, as you did for Ubuntu in the previous section.

How about publishing your own images? You can certainly use Docker Hub or one of the registries offered by cloud providers like Azure Container Registry. Considering the specific project we are working on throughout the book, I chose to rely on the GitHub Container Registry (<https://docs.github.com/en/packages>).

- It is available for use with all personal GitHub accounts, and it's free for public repositories. You can also use it with private repositories, but with some limitations.
- It allows you to access public container images anonymously without rate-limiting, even with a free account.
- It is fully integrated into the GitHub ecosystem and makes it possible to navigate from images to related source code seamlessly.
- It lets you generate multiple tokens to access the registry even with a free account. It's recommended to issue a different access token for each use case. GitHub lets you do that via the Personal Access Token (PAT) feature without limitations on the number of tokens. Furthermore, if you access GitHub Container Registry from GitHub Actions, you don't even need to configure a PAT, but you get a token auto-configured by GitHub

out-of-the-box and provided securely to the automated pipeline without further configuration.

Publishing images to the GitHub Container Registry requires you to be authenticated, and for that, you need a Personal Access Token. Go to your GitHub account, navigate to *Settings > Developer Settings > Personal access tokens*, and choose "Generate new token". Input a meaningful name, and assign it the `write:packages` scope to give the token permissions to publish images to the container registry (figure 6.6). Finally, generate the token and copy its value. GitHub will show you the token value only once. Make sure you save it since you'll need it soon.

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

local-dev-environment

What's this token for?

Expiration *

30 days The token will expire on Tue, Mar 8 2022

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input checked="" type="checkbox"/> read:packages	Download packages from GitHub Package Registry

Figure 6.6 A Personal Access Token (PAT) granting write access to the GitHub Container Registry.

Next, open a Terminal window and authenticate with GitHub Container Registry (make sure your Docker Engine is running). When asked, insert username (your GitHub username) and password (your GitHub token).

```
$ docker login ghcr.io
```

If you followed along, you should have your custom `my-java-image` Docker image on your machine. If not, make sure you performed the operations described in the previous section.

Container images follow common naming conventions which are adopted by OCI-compliant container registries: `<container_registry>/<namespace>/<name>[:<tag>]`.

1. **Container registry.** The hostname for the container registry where the image is stored. When using Docker Hub, the hostname is `docker.io` and is usually omitted. The Docker Engine will implicitly prepend the image name with `docker.io` when you don't specify one. When using GitHub Container Registry, the hostname is `ghcr.io` and must be explicit.
2. **Namespace.** When using Docker Hub or GitHub Container Registry, it will be your Docker/GitHub username written all in lowercase. In other registries, it might be the path to the repository.
3. **Name and tag.** The image name represents the repository (or *package*) that contains all the versions of your image. It's optionally followed by a tag for selecting a specific version. If not defined, the `latest` tag will be used by default.

Official images like `ubuntu` or `postgresql` can be downloaded by specifying the name only, which is implicitly converted to fully qualified names like `docker.io/library/ubuntu` or `docker.io/library/postgres`.

When uploading your images to GitHub Container Registry, you are required to use fully qualified names, according to the `ghcr.io/<your_github_username>/<image_name>` format. For example, my GitHub username is `ThomasVitale`, and all my personal images are named like `ghcr.io/thomasvitale/<image_name>` (notice how the username is converted to lowercase).

Since you previously built the image with the name `my-java-image:1.0.0`, you have to assign it a fully qualified name before publishing it to a container registry (that is, you need to *tag* the image). You can do so with the `docker tag` command.

```
$ docker tag my-java-image:1.0.0 \
  ghcr.io/<your_github_username>/my-java-image:1.0.0
```

Then, you can finally *push* it to GitHub Container Registry.

```
$ docker push ghcr.io/<your_github_username>/my-java-image:1.0.0
```

Go to your GitHub account, navigate to your profile page, and enter the *Packages* section. You should see a new `my-java-image` entry. If you click on it, you'll find the `ghcr.io/<your_github_username>/my-java-image:1.0.0` image you have just published (figure 6.7). By default, the repository hosting your new image will be private.

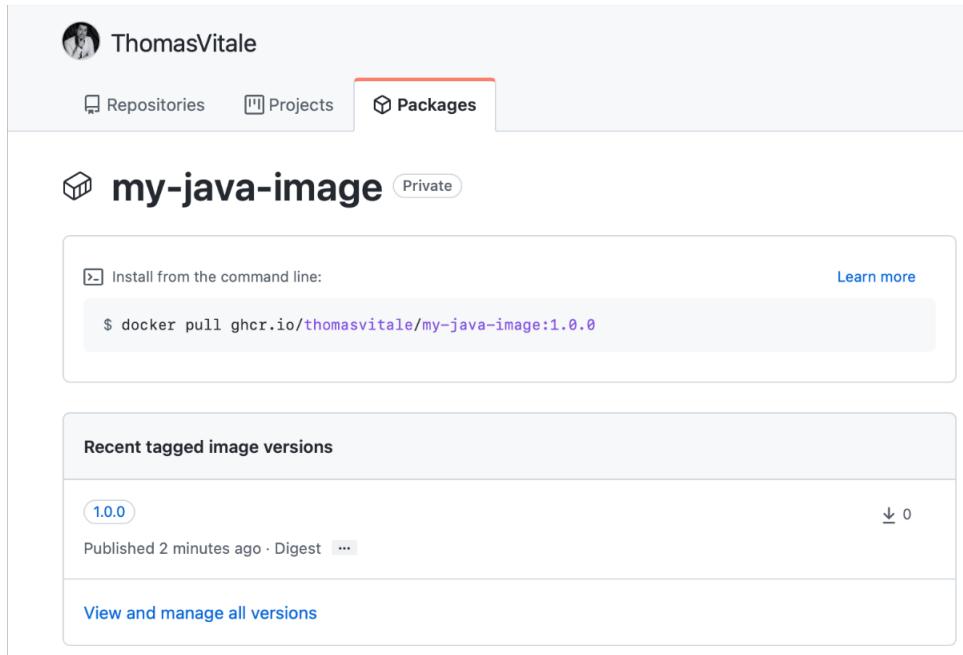


Figure 6.7 GitHub Container Registry is a public registry that you can use to publish your container images. You can see your images in the "Packages" section of your GitHub profile.

TIP

From the same page, you can also delete the published image or the entire image repository (called package in GitHub) by going to the "Package settings".

That concludes the section. Now that you know the main features of container images, how to create them, and how to publish them, let's dive deeper into packaging Spring Boot applications as images.

6.2 Packaging Spring Boot applications as container images

In the previous chapters, we built the Catalog Service application, featuring a REST API and a database integration. In this section, as an intermediate step before deploying it to Kubernetes, we'll build an image to run Catalog Service as a container on Docker.

First, I'll review some aspects you should consider when packaging a Spring Boot application as a container image. Then, I'll show you how to do that with a Dockerfile and with Cloud Native Buildpacks.

6.2.1 Preparing Spring Boot for containerization

Packaging a Spring Boot application as a container image means that the application will run in an isolated context, including computational resources and network. Two main questions may arise from this isolation:

- How can you reach the application through the network?
- How can you make it interact with other containers?

EXPOSING APPLICATION SERVICES THROUGH PORT FORWARDING

In chapter 2, when you ran Catalog Service as a container, you mapped the port 8080 on which the application exposed its services to the port 8080 on your local machine. After doing that, you could use the application by visiting <http://localhost:8080>. What you did there is called *port forwarding* or *port mapping* or *port publishing* and is used to make your containerized application accessible from the outside world.

By default, containers join an isolated network inside the Docker host. If you want to access any container from your local network, you must explicitly configure the port mapping. For example, when you ran the Catalog Service application, you specified the mapping as an argument to the `docker run` command: `-p 8080:8080` (where the first is the external port, the second is the container port). Figure 6.8 illustrates how it works.

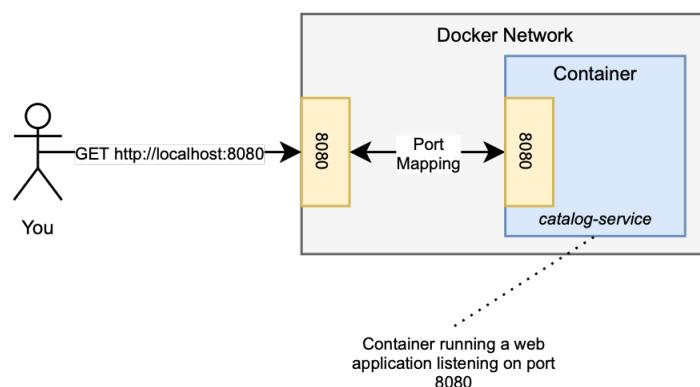


Figure 6.8 Port mapping lets you access the services exposed by a containerized application by forwarding the traffic from the container network to the outside world.

USING DOCKER BUILT-IN DNS SERVER FOR SERVICE DISCOVERY

Thanks to port forwarding, in the previous chapter, the Catalog Service application could access the PostgreSQL database server through the URL `jdbc:postgresql://localhost:5432`, even if it was running inside a container. The interaction is shown in figure 6.9.

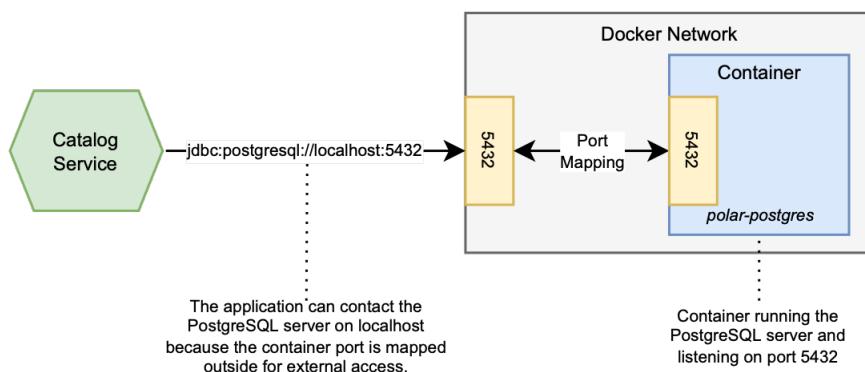


Figure 6.9 The Catalog Service application can interact with the PostgreSQL container thanks to the port mapping, making the database accessible from the outside world.

When running Catalog Service as a container, you will not be able to do that anymore since `localhost` would represent the inside of your container and not your local machine. How can you solve this problem?

Docker has a built-in DNS server that you can rely upon for letting containers in the same network find each other using the container name rather than a hostname or an IP address.

For example, Catalog Service will be able to call the PostgreSQL server through the URL `jdbc:postgresql://polar-postgres:5432`, where `polar-postgres` is the container name. Figure 6.10 shows how it works. Later in the chapter, you'll see how to achieve this result in code.

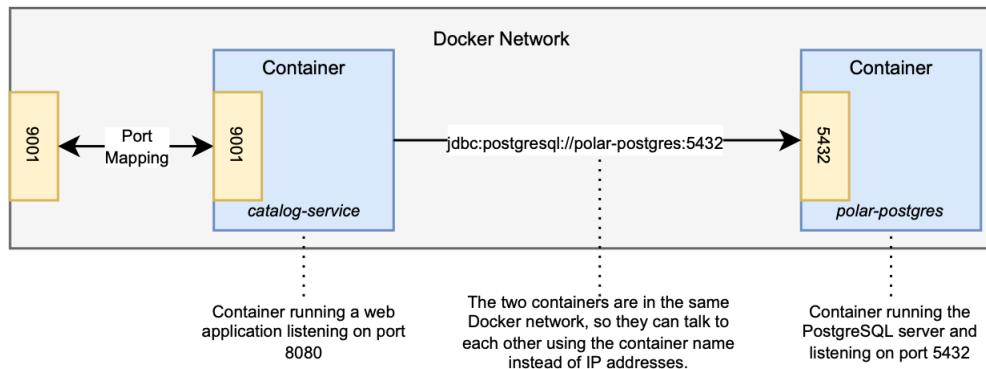


Figure 6.10 The Catalog Service container can directly interact with the PostgreSQL container because they are both on the same Docker network.

So, before moving on, let's create a network inside which Catalog Service and PostgreSQL can talk to each other using the container name instead of an IP address or a hostname. You can run this command from any Terminal window.

```
$ docker network create catalog-network
```

Next, verify the network has been successfully created.

```
$ docker network ls
NETWORK ID      NAME        DRIVER      SCOPE
178c7a048fa9  catalog-network  bridge      local
...
```

You can then start a PostgreSQL container, specifying that it should be part of the `catalog-network` you just created. Using the `--net` argument, the container will join the specified network and rely on the Docker built-in DNS server.

```
$ docker run -d \
--name polar-postgres \
--net catalog-network \
-e POSTGRES_USER=user \
-e POSTGRES_PASSWORD=password \
-e POSTGRES_DB=polaradb_catalog \
-p 5432:5432 \
postgres:14.3
```

If the command fails, you might have the PostgreSQL container from chapter 5 still running. Remove it with `docker rm -fv polar-postgres` and run the previous command again.

6.2.2 Containerizing Spring Boot with Dockerfiles

Cloud native applications are self-contained. Spring Boot lets you package your applications as standalone JARs, including everything they need to run except the runtime environment. That makes the containerization very straightforward since all you need in a container image besides the JAR artifact is an operating system and a JRE. This section will show how to containerize the Catalog Service application using a Dockerfile.

First, you need to identify on which image you want to base yours. You could choose an Ubuntu image as you did before and then explicitly install the JRE, or you could directly choose a base image providing a JRE already, which is more convenient. All the major OpenJDK distributions also have a related image available on Docker Hub. Feel free to choose the one you prefer. In the following, I'll be using Eclipse Temurin 17, the same OpenJDK distribution I've been using locally so far. Then, you need to copy the JAR file of Catalog Service into the image itself. Finally, declare that the entry point for the container execution is the command to run the application on the JRE.

Open your Catalog Service project (`catalog-service`), and create an empty file called `Dockerfile` (no extension) in the root folder. That file will contain the recipe for containerizing your application.

Listing 6.2 Dockerfile for describing the Catalog Service image (Dockerfile)

```
FROM eclipse-temurin:17    ①
WORKDIR workspace          ②
ARG JAR_FILE=build/libs/*.jar ③
COPY ${JAR_FILE} catalog-service.jar ④
ENTRYPOINT ["java", "-jar", "catalog-service.jar"] ⑤
```

- ① Ubuntu base image with the Eclipse Temurin distribution of the JRE pre-installed.
- ② Changes the current working directory to "workspace".
- ③ Build argument specifying the location of the application JAR file in your project.
- ④ Copies the application JAR file from the local machine into the image.
- ⑤ Sets the container entry point to run the application.

This Dockerfile declares a `JAR_FILE` argument that can be specified when creating the image with the `docker build` command. Before moving on, you need to build the JAR artifact for the Catalog Service application. Open a Terminal window and navigate to the Catalog Service project root folder. First, build the JAR artifact.

```
$ ./gradlew clean bootJar
```

By default, the Dockerfile script will copy the application JAR file from the location path used by Gradle: `build/libs/`. So, if you're using Gradle, you can build the container image by running this command.

```
$ docker build -t catalog-service .
```

If you're using Maven, you can specify the location used by Maven as a build argument with the following command (don't forget the final dot).

```
$ docker build --build-arg JAR_FILE=target/*.jar -t catalog-service .
```

In either case, you'll end up having your Catalog Service application packaged as a container image. Since we haven't specified any version, the image will be tagged as `latest` automatically. Let's verify if it works.

Remember the two aspects I covered in the previous section: port forwarding and using the Docker built-in DNS server. You can handle them by adding two arguments to the `docker run` command:

- `-p 9001:9001` will map port 9001 inside the container (where the Catalog Service is exposing its services) to the 9001 port on your localhost;
- `--net catalog-network` will connect the Catalog Service container to the `catalog-network` you previously created so that it can contact the PostgreSQL container.

That is still not enough. In the previous chapter, we set the `spring.datasource.url` property for Catalog Service to `jdbc:postgresql://localhost:5432/polardb_catalog`. Since it points to `localhost`, it will not work from within a container. You already know how to configure a Spring Boot application from the outside without having to recompile it, right? An environment variable will do. We need to overwrite the `spring.datasource.url` property and specify the same URL, replacing `localhost` with the PostgreSQL container name:

`polar-postgres`. Using another environment variable, we can also enable the `testdata` Spring profile to trigger the creation of test data in the catalog.

```
$ docker run -d \
  --name catalog-service \
  --net catalog-network \
  -p 9001:9001 \
  -e SPRING_DATASOURCE_URL=
  ↪jdbc:postgresql://polar-postgres:5432/polardb_catalog \
  -e SPRING_PROFILES_ACTIVE=testdata \
  catalog-service
```

That's quite a long command, isn't it? You won't use the Docker CLI for long, though, I promise. Later in the chapter, I will introduce Docker Compose.

Open a Terminal window, call the application, and verify that it works correctly as it did in chapter 5.

```
$ http :9001/books
```

When you're done, remember to delete both containers.

```
$ docker rm -f catalog-service polar-postgres
```

The approach you've just followed is perfectly fine for experimenting with Docker in your development environment and understanding how images work. But you need to consider several aspects before achieving a production-grade image. That's the topic of the next section.

6.2.3 Building container images for production

Getting started with Dockerfiles might not be that difficult at the beginning. But building production-grade images can be challenging. In this section, you'll see how to improve the image you built in the previous section.

You'll use the layered-JAR feature provided by Spring Boot to build more efficient images. Then, you'll consider essential security aspects related to container images. Finally, I'll go through some considerations to choosing between Dockerfiles and Cloud Native Buildpacks to containerize applications.

PERFORMANCE

When building container images, you should consider performance at build time and run time. The layered architecture characterizing OCI images enables caching and reusing the unchanged layers when building an image. Container registries store images by layers so that when you pull a new version, only the changed layers are downloaded. That is quite an advantage in a cloud environment, considering the time and the bandwidth saved for all your application instances.

In the previous section, you copied the Catalog Service standalone JAR file into a layer in the

image. As a result, whenever you change something in your application, the whole layer must be rebuilt. Consider the scenario where you just add a new REST endpoint to your application. Even if all the Spring libraries and dependencies are unchanged and the only difference is in your own code, you must rebuild the whole layer since everything is together. We can do better. And Spring Boot can help us.

Putting uber-JARs inside a container image has never been efficient. A JAR artifact is a compressed archive containing all dependencies, classes, and resources used by the application. All those files are organized in folders and subfolders within the JAR. We could expand standard JAR artifacts and put each folder on a different container image level. That's what we've been doing for many years. Starting with version 2.3, Spring Boot made that even more efficient by introducing a new way of packaging applications as JAR artifacts: the layered-JAR mode. And since Spring Boot 2.4, that's the default mode, so you don't need any extra configuration to use the new functionality.

Applications packaged using the **layered-JAR mode** are made up of layers, similar to how container images work. This new feature is excellent for building more efficient images. When using the new JAR packaging, we can expand the JAR artifact and then create a different image layer for each JAR layer. The goal is to have your own classes (changing more frequently) on a separate layer than the project dependencies (changing less frequently).

By default, Spring Boot applications are packaged as JAR artifacts made up of the following layers, starting from the lowest:

- `dependencies` for all the main dependencies added to the project;
- `spring-boot-loader` for the classes used by the Spring Boot loader component;
- `snapshot-dependencies` for all the snapshot dependencies;
- `application` for your application classes and resources.

If you consider the previous scenario where you just add a new REST endpoint to an existing application, only the `application` layer must be built when you containerize it. Moreover, when you upgrade the application in production, only that new layer must be downloaded to the nodes where the container is running, making the upgrade faster and cheaper (especially on cloud platforms billing per bandwidth used).

Let's now update the previous Dockerfile for containerizing Catalog Service more efficiently using the layered-JAR mode. Using this new strategy means doing some preparation work to copy the JAR file into the image and expand it into the four layers described previously. In the end, we don't want to keep the original JAR file inside the image, or else our optimization plan would not work. Docker provides a solution for that: *multi-stage builds*. We'll divide the work into two stages. The first stage is only needed to extract the layers from the JAR file. The second

stage is where we place each JAR layer into a separate image layer. In the end, the result of the first stage is discarded (including the original JAR file), while the second one will produce the final container image.

Listing 6.3 More efficient Dockerfile for describing the Catalog Service image (Dockerfile)

```
FROM eclipse-temurin:17 AS builder ①
WORKDIR workspace
ARG JAR_FILE=build/libs/*.jar ②
COPY ${JAR_FILE} catalog-service.jar ③
RUN java -Djarmode=layertools -jar
catalog-service.jar extract ④

FROM eclipse-temurin:17 ⑤
WORKDIR workspace
COPY --from=builder workspace/dependencies/ ./ ⑥
COPY --from=builder workspace/spring-boot-loader/ ./
COPY --from=builder workspace/snapshot-dependencies/ ./
COPY --from=builder workspace/application/ ./
ENTRYPOINT ["java",
"org.springframework.boot.loader.JarLauncher"] ⑦
```

- ① OpenJDK base image for the first stage.
- ② Build argument specifying the location of the application JAR file in your project.
- ③ Copies the application JAR file from the local machine into the image inside the "workspace" folder.
- ④ Extracts the layers from the archive applying the layered-JAR mode.
- ⑤ OpenJDK base image for the second stage.
- ⑥ Copies each JAR layer from the first stage to the second stage inside the "workspace" folder.
- ⑦ Uses the Spring Boot Launcher to start the application from the layers rather than an uber-JAR.

NOTE

What if you want to change the configuration for the layers in the JAR file? As always, Spring Boot provides sensible defaults, but you can customize it and adapt it to your needs. Perhaps, your project has internal shared dependencies you might want to place in a separate layer since they change more often than third-party dependencies. You can do that through the Spring Boot Gradle or Maven plugin. For more information, refer to the Spring Boot documentation at <https://spring.io/projects/spring-boot>.

The process for building and running the container is the same as before, but now the image is more efficient and optimized at build and execution time. However, it's still not ready for production. What about security? That's the topic of the next section.

SECURITY

Security is a critical aspect that is too often underestimated when getting started with Docker and containerization. You should be aware that containers run using the root user by default, potentially letting them get root access to the Docker host. You can mitigate the risk by creating a non-privileged user and using it to run the entry-point process defined in the Dockerfile, following the principle of least privilege.

Consider the Dockerfile you wrote for Catalog Service. You can improve it by adding new steps for creating a new non-root user that will run the application, therefore reducing the risks of exposing root access to the Docker host.

Listing 6.4 More secure Dockerfile for describing the Catalog Service image (Dockerfile)

```
FROM eclipse-temurin:17 AS builder
WORKDIR workspace
ARG JAR_FILE=build/libs/*.jar
COPY ${JAR_FILE} catalog-service.jar
RUN java -Djarmode=layertools -jar catalog-service.jar extract

FROM eclipse-temurin:17
RUN useradd spring ①
USER spring ②
WORKDIR workspace
COPY --from=builder workspace/dependencies/ ../
COPY --from=builder workspace/spring-boot-loader/ ../
COPY --from=builder workspace/snapshot-dependencies/ ../
COPY --from=builder workspace/application/ ../
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

- ① Creates a "spring" user.
- ② Configures "spring" as the current user.

As previously noted, you should never store secrets like passwords or keys in a container image. Even if they are removed in an upper layer, they would remain intact in the original layer and easily accessible.

Finally, it's also critical to use up-to-date base images and libraries in your Dockerfile. Scanning your container images for vulnerabilities is a best practice that should be embraced and automated in your deployment pipeline. In chapter 3, you learned how to use `grype` to scan codebases for vulnerabilities. Now, we'll also use it to scan a container image.

Using the updated Dockerfile, build a new container image for Catalog Service. Open a Terminal window, navigate to the Catalog Service root folder, and run this command (don't forget the final dot).

```
$ docker build -t catalog-service .
```

Next, use `grype` to check if the newly-created image contains any vulnerability.

```
$ grype catalog-service
```

Have you found any high-severity vulnerability? Discussing supply chain security and related risk management is out of scope for this book. Still, I want to show you how to perform and automate vulnerability scanning for application artifacts, but I'll leave it up to you to follow up on the scan results. I can't stress enough how important it is to define a security strategy in your organization and automate as much as possible its compliance verification throughout the whole value stream.

In this section, I mentioned a few essential aspects to consider when building production-grade container images, but there's even more to cover. Is there another way? The following section introduces a different option for building container images.

DOCKERFILES OR BUILDPACKS

Dockerfiles are very powerful and give you complete fine-grained control over the result. However, they require extra care and maintenance, and can lead to several challenges in your value stream.

As a developer, you might not want to deal with all the previous performance and security concerns, and you might prefer to focus on the application code instead. After all, one reason to move to the cloud is to deliver value faster to customers. Adding the Dockerfile additional step and considering all those issues might not be for you.

As an operator, it might be challenging to control and secure the supply chain within the organization when container images are built from a Dockerfile. It's pretty common to invest some time in scripting the "perfect" Dockerfile and copying that around in multiple repositories for different applications. But then it's hard to keep all teams on the same page, verifying the adherence to the approved Dockerfile, synchronizing any change throughout the organization, and understanding who is responsible for what.

Cloud Native Buildpacks provide a different approach, focusing on consistency, security, performance, and governance. As a developer, you get a tool to automatically build a production-ready OCI image from your application source code without writing any Dockerfile. As an operator, you get a tool to define, control, and secure application artifacts within the entire organization.

Ultimately, the decision between using a Dockerfile or a tool like Buildpacks depends on your organization and requirements. Both ways are valid and used in production. In general, my recommendation is to use Buildpacks unless it makes sense not to.

NOTE Another option for packaging Java applications as container images without writing a Dockerfile is using Jib, a Gradle and Maven plugin developed by Google (<https://github.com/GoogleContainerTools/jib>).

In the next section and the rest of the book, we'll use Cloud Native Buildpacks rather than Dockerfiles. It was important for me to show you how a Dockerfile works because it makes understanding the container image features and layers better. Furthermore, I wanted to show you how to write a basic Dockerfile for containerizing a Spring Boot application to highlight what is needed and illustrate the execution of the application JAR from within the container. Finally, it's going to be less troublesome for you to debug containers when something goes wrong, even when automatically generated by Buildpacks because you know how to build images from scratch. If you'd like to learn more about Dockerfiles for Spring Boot applications, I recommend you check the official documentation (<https://spring.io/projects/spring-boot>).

6.2.4 Containerizing Spring Boot with Cloud Native Buildpacks

Cloud Native Buildpacks (<https://buildpacks.io>) is a project hosted by the CNCF to "transform your application source code into images that can run on any cloud". When introducing containers in chapter 1, I underlined how PaaS platforms like Heroku and Cloud Foundry are actually using containers behind the scenes, converting your application source code into containers before running them. Buildpacks is precisely the tool they use to accomplish that.

It's been developed and advanced based on Heroku and Pivotal's many years of experience running cloud native applications as containers on their PaaS platforms. It's a mature project, and since Spring Boot 2.3, it's integrated natively in the Spring Boot Plugin for both Gradle and Maven, so you're not required to install the dedicated Buildpacks CLI (`pack`).

Some of its features are:

- it auto-detects the type of application and packages it without requiring a Dockerfile;
- it supports multiple languages and platforms;
- it's highly performant through caching and layering;
- it guarantees reproducible builds;
- it relies on best practices in terms of security;
- it produces production-grade images;
- it supports building native images using GraalVM.

NOTE If you'd like to know more about Cloud Native Buildpacks, I recommend watching this screencast with Emily Casey, member of the Buildpacks core team: <https://www.youtube.com/watch?v=HaXe7KYKSS4>.

The generation process is orchestrated by a *builder* image containing the complete information

on how to containerize your application. Such information is provided as a sequence of *buildpacks*, each dedicated to a specific aspect of the application (e.g., operating system, OpenJDK, JVM configuration). The Spring Boot Plugin adopts the Paketo Buildpacks builder, an implementation of the Cloud Native Buildpacks specification that provides support for many types of applications, including Java and Spring Boot ones (<https://paketo.io>).

The Paketo builder component relies on a series of default buildpacks to do the actual build operation. This structure is highly modular and customizable. You can add new buildpacks to the sequence (for example, to add a monitoring agent to the application), replace existing ones (for example, to replace the default Bellsoft Liberica OpenJDK with the Microsoft OpenJDK), or even use a different builder image entirely.

NOTE

The Cloud Native Buildpacks project manages a registry to discover and analyze buildpacks you can use to containerize your applications, including all the buildpacks from the Paketo implementation (<https://registry.buildpacks.io>).

The Buildpacks integration provided by the Spring Boot Plugin can be configured in the `build.gradle` file located in your Catalog Service project. Let's configure the image name and define which Java version to use through an environment variable.

Listing 6.5 Configuration for containerizing Catalog Service (build.gradle)

```
bootBuildImage { ①
    imageName = "${project.name}" ②
    environment = [ "BP_JVM_VERSION" : "17.*" ] ③
}
```

- ① The Spring Boot Plugin task to build an OCI image using Buildpacks.
- ② The name for the OCI image to build. The name is the same defined in the Gradle configuration for the project. We rely on the implicit "latest" tag when working locally rather than a version number.
- ③ The JVM version to be installed in the image. It uses the latest Java 17 version.

Go ahead and build the image by running the following command.

```
$ ./gradlew bootBuildImage
```

WARNING At the moment of writing, the Paketo project is working on adding support for ARM64 images. You can follow the feature progress on the Paketo Buildpacks project on GitHub: <https://github.com/paketo-buildpacks/stacks/issues/51>. Until then, you can still use Buildpacks for building containers and run them via Docker Desktop on Apple Silicon computers. However, the build process and the application startup phase will be slower than usual.

The first time you run the task, it will take a minute to download the packages used by Buildpacks to create the container image. The second time, it will take only a few seconds. If you look closely at the output of the command, you can see all the steps performed by Buildpacks to generate the image. These steps include adding a JRE and using the layered JAR built by Spring Boot. The plugin accepts more properties to customize its behavior, for example, providing your own builder component instead of Paketo's. Check the official documentation for the complete list of configuration options (<https://spring.io/projects/spring-boot>).

Let's try running Catalog Service as a container once again, but this time we'll use the image generated by Buildpacks. Remember to start the PostgreSQL container first following the instructions from the previous section.

```
$ docker run -d \
  --name catalog-service \
  --net catalog-network \
  -p 9001:9001 \
  -e SPRING_DATASOURCE_URL=
  ➔ jdbc:postgresql://polar-postgres:5432/polardb_catalog \
  -e SPRING_PROFILES_ACTIVE=testdata \
  catalog-service
```

WARNING If you're running the container on an Apple Silicon computer, the previous command might return a message like WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and no specific platform was requested. In that case, you need to include this additional argument to the previous command (before the image name) until support for ARM64 is added to Paketo Buildpacks: --platform linux/amd64.

Open a browser window, call the application on <http://localhost:9001/books>, and verify that it works correctly. When you're done, remember to delete both PostgreSQL and Catalog Service containers.

```
$ docker rm -f catalog-service polar-postgres
```

Finally, you can remove the network you used to make Catalog Service communicate with

PostgreSQL. You won't need it anymore after introducing Docker Compose in the next section.

```
$ docker network rm catalog-network
```

Since Spring Boot 2.4, you can also configure the Spring Boot plugin to publish the image directly to a container registry. First, you need to add configuration for authenticating with the specific container registry.

Listing 6.6 Configuration for containerizing Catalog Service (build.gradle)

```
bootBuildImage {
    imageName = "${project.name}"
    environment = [ "BP_JVM_VERSION" : "17.*" ]

    docker { ①
        publishRegistry { ②
            username = project.findProperty("registryUsername")
            password = project.findProperty("registryToken")
            url = project.findProperty("registryUrl")
        }
    }
}
```

- ① Section to configure the connection with a container registry.
- ② Section to configure authentication to the publishing container registry. The values are passed as Gradle properties.

The details on how to authenticate with the container registry are externalized as Gradle properties both for flexibility (you can publish the image to different registries without changing the Gradle build) and for security (the token, in particular, should never be included in version control).

Remember a golden rule of credentials: you should never give your password away. Never! If you need to delegate some service to access a resource on your behalf, you should rely on an access token. The Spring Boot plugin lets you use a password to authenticate with the registry, but you should use a token instead. In a previous section, you generated a Personal Access Token in GitHub specifically crafted to let you push images to the GitHub Container Registry from your local environment. If you don't know its value anymore, feel free to generate a new one following the procedure I explained earlier in the chapter.

Finally, you can build and publish the image running the `bootBuildImage` task. With the `--imageName` argument, you can define a fully qualified image name as container registries require. With the `--publishImage` argument, you can instruct the Spring Boot plugin to push the image to the container registry directly. Also, remember to pass values for the container registry Gradle properties.

```
$ ./gradlew bootBuildImage \
--imageName ghcr.io/<your_github_username>/catalog-service \
--publishImage \
-PregistryUrl=ghcr.io \
-PregistryUsername=<your_github_username> \
-PregistryToken=<your_github_token>
```

Once the command completes successfully, go to your GitHub account, navigate to your profile page, and enter the *Packages* section. You should see a new `catalog-service` entry (by default, packages hosting container images are private), similar to what happened with the `my-java-image` you published earlier. If you click on the `catalog-service` entry, you'll find the `ghcr.io/<your_github_username>/catalog-service:latest` image you have just published (figure 6.11).

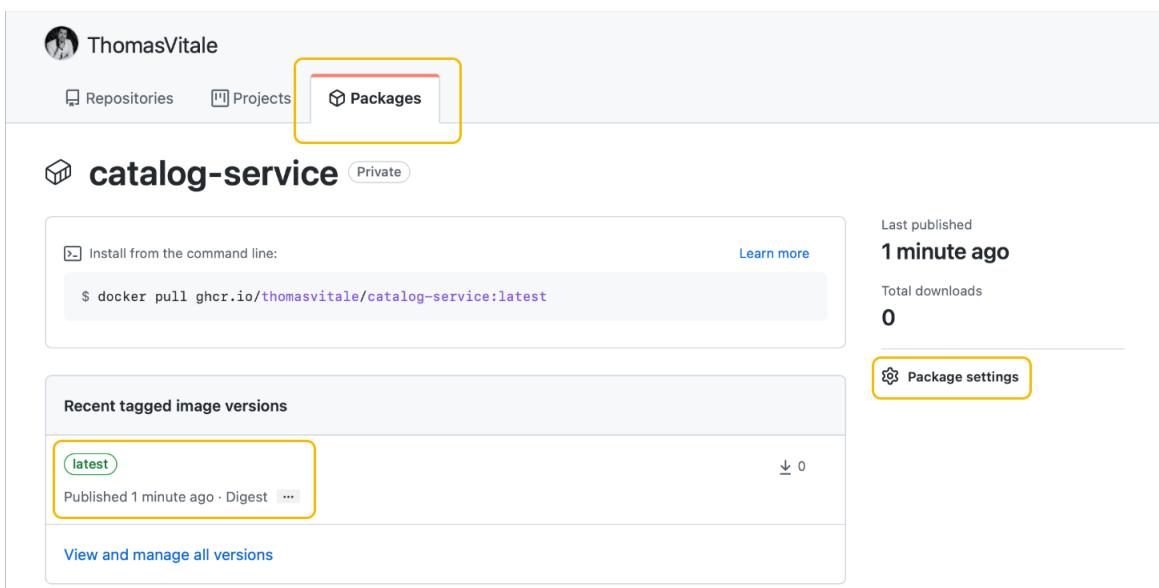


Figure 6.11 Images published to the GitHub Container Registry are organized as "packages".

However, the `catalog-service` package is not linked to your `catalog-service` source code repository yet. Later, I'll show you how to automate building and publishing your images with GitHub Actions, which makes it possible to publish images in the context of the source code repository from which they are built.

For now, let's remove the `catalog-service` package that got created when we published the image so that it won't cause any conflict once we start using GitHub Actions to publish images. From the `catalog-service` package page (figure 6.11), select "Package settings" from the sidebar menu, scroll to the bottom of the settings page, and choose "Delete this package" (figure 6.12).

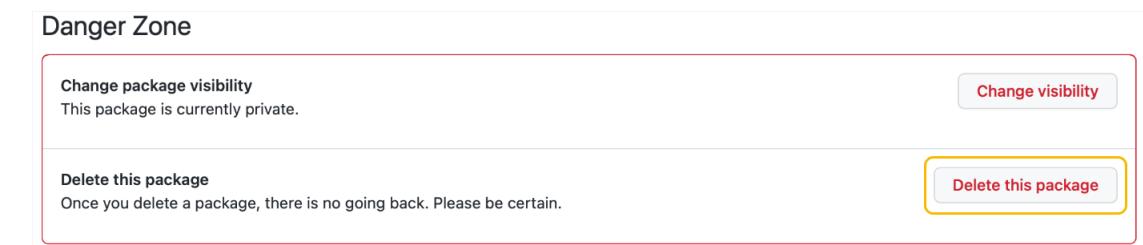


Figure 6.12 Delete the "catalog-service" package create manually.

NOTE So far, we've been using the implicit `latest` tag for naming container images. That's not recommended for production scenarios. In chapter 15, you'll see how to handle versions when releasing your applications. Until then, we'll rely on the implicit `latest` tag.

6.3 Managing Spring Boot containers with Docker Compose

Cloud Native Buildpacks lets you containerize Spring Boot applications quickly and efficiently without writing a Dockerfile yourself. But when it comes to running multiple containers, the Docker CLI can be a bit cumbersome. Writing commands in a Terminal window can be error-prone, hard to read, and challenging to apply version control.

Docker Compose provides you with a better experience over the Docker CLI. Instead of a command line, you work with YAML files that describe which containers you want to run and their characteristics. With Docker Compose, you can define all the applications and services composing your system in one place and manage their lifecycle together.

In this section, you'll configure the execution of the Catalog Service and PostgreSQL containers using Docker Compose. Then, you'll learn how to debug a Spring Boot application running within a container.

If you have installed Docker Desktop for Mac or Docker Desktop for Windows, you already have Docker Compose installed. If you're on Linux, visit the Docker Compose installation page at <https://www.docker.com> and follow the instructions for your distribution. In both cases, you can verify that Docker Compose is correctly installed by running the command `docker-compose --version`.

6.3.1 Using Docker Compose to manage the container lifecycle

The Docker Compose syntax is very intuitive and self-explanatory. Often, it can be mapped one-to-one to Docker CLI arguments. The two root sections of a `docker-compose.yml` file are `version` to specify which syntax of Docker Compose you want to use and `services`, containing the specifications for all the containers you want to run. Other optional root-level sections you may add are `volumes` and `networks`.

NOTE If you don't add any network configuration, Docker Compose will automatically create one for you and make all the containers in the file join it. That means they can interact with each other through their container names, relying on the Docker built-in DNS server.

It's good practice to gather all deployment-related scripts in a separate codebase and, possibly, in a separate repository. Go ahead and create a new `polar-deployment` repository on GitHub. It'll contain all Docker and Kubernetes scripts to run the applications composing the Polar Bookshop system. Inside the repository, create a `docker` folder for hosting the Docker Compose configuration for Polar Bookshop. In the source code accompanying the book, you can refer to `Chapter06/06-end/polar-deployment` for the final result.

In the `polar-deployment/docker` folder, create a `docker-compose.yml` file, and define the services to run as follows.

Listing 6.7 Docker Compose file describing the catalog services (`docker-compose.yml`)

```
version: "3.8"      ①
services:          ②

catalog-service:   ③
  depends_on:
    - polar-postgres ④
  image: "catalog-service" ⑤
  container_name: "catalog-service" ⑥
  ports:
    - 9001:9001
  environment:
    - BPL_JVM_THREAD_COUNT=50 ⑧
    - SPRING_DATASOURCE_URL=
  ↪ jdbc:postgresql://polar-postgres:5432/polardb_catalog
    - SPRING_PROFILES_ACTIVE=testdata ⑩

polar-postgres: ⑪
  image: "postgres:14.3"
  container_name: "polar-postgres"
  ports:
    - 5432:5432
  environment:
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=password
    - POSTGRES_DB=polardb_catalog
```

- ① The Docker Compose syntax version.
- ② The section containing all the containers to run.
- ③ The section describing the catalog-service container.
- ④ Catalog Service should start after the PostgreSQL database.
- ⑤ The image used to run the container.
- ⑥ The name of the container.

- ⑦ The section listing the port mappings.
- ⑧ The section listing the environment variables.
- ⑨ Paketo Buildpacks environment variable to configure the number of threads for memory calculation.
- ⑩ Enables the "testdata" Spring profile.
- ⑪ The section describing the polar-postgres container.

You might have noticed the presence of an additional environment variable for the Catalog Service container. In chapter 15, you'll learn about the Java memory calculator provided by the Paketo Buildpacks and how to configure CPU and memory for Spring Boot applications. For now, it's enough to know that the `BPL_JVM_THREAD_COUNT` environment variable is used to configure the number of threads for which to allocate memory in the JVM stack. The default value for Servlet-based applications is 250. In chapter 3, we used a low value for the Tomcat thread pool. It's good to do the same for the JVM memory configuration to keep the overall container memory usage low locally. You're going to deploy many containers throughout the book (both applications and backing services), and such configuration helps make that possible without overloading your computer.

Docker Compose configures both containers on the same network by default, so you don't need to specify one explicitly as you did previously. Let's see now how to spin them up. Open a Terminal window, navigate to the folder containing the file, and run the following command to start the containers in detached mode.

```
$ docker-compose up -d
```

When the command is done, try calling the Catalog Service application at <http://localhost:9001/books> and verify that it works correctly. Then, keep your containers running and move on to the next section, where you'll debug the Catalog Service application.

6.3.2 Debugging Spring Boot containers

When running a Spring Boot application as standard Java from your IDE, you can specify whether you want to run it in debug mode. If you do, the IDE will attach a debugger to the local Java process running your application. However, when you run it from within a container, your IDE can't do that anymore because the process is not running on the local machine.

Spring Boot applications running in a container can be debugged almost as easily as when running locally. First, you need to instruct the JVM inside the container to listen for debug connections on a specific port. The container image produced by Paketo Buildpacks supports

dedicated environment variables to run the application in debug mode (`BPL_DEBUG_ENABLED` and `BPL_DEBUG_PORT`). Then, you need to expose the debug port outside the container so that your IDE can reach it. Figure 6.13 illustrates how it works.

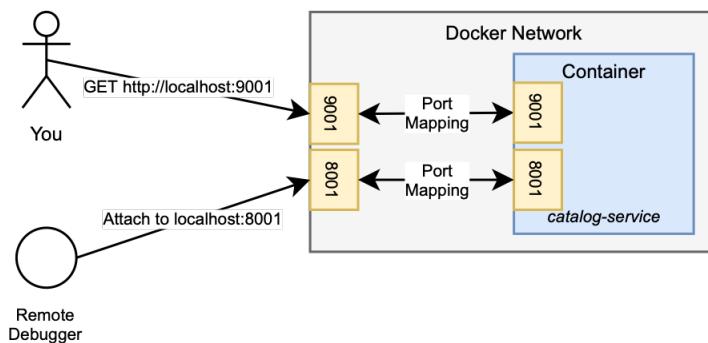


Figure 6.13 From a container, you can expose as many ports as you want. For Catalog Service, expose both the server port and the debug port.

Update your `docker-compose.yml` file to configure the Catalog Service application for debugging.

Listing 6.8 Configure Catalog Service to run in debug mode (docker-compose.yml)

```

version: "3.8"
services:

  catalog-service:
    depends_on:
      - polar-postgres
    image: "catalog-service"
    container_name: "catalog-service"
    ports:
      - 9001:9001
      - 8001:8001 ①
    environment:
      - BPL_JVM_THREAD_COUNT=50
      - BPL_DEBUG_ENABLED=true ②
      - BPL_DEBUG_PORT=8001 ③
      - SPRING_DATASOURCE_URL=
    =>jdbc:postgresql://polar-postgres:5432/polardb_catalog
      - SPRING_PROFILES_ACTIVE=testdata
    ...
  
```

- ① The port where the JVM will listen for debug connections.
- ② Activates the JVM configuration for accepting debug connections (provided by Buildpacks).
- ③ Debug connections are accepted via a socket on port 8001 (provided by Buildpacks).

From a Terminal window, navigate to the folder where the `docker-compose.yml` file is located, and rerun the following command.

```
$ docker-compose up -d
```

You'll notice how Docker Compose is smart enough to know that the PostgreSQL container configuration is unchanged and will do nothing about it. Instead, it will reload the Catalog Service container with the new configuration.

Then, in your IDE of choice, you need to configure a remote debugger and point it to port 8001. Refer to your IDE documentation to find the instructions on how to do that. Figure 6.14 shows how to configure a remote debugger in IntelliJ IDEA.

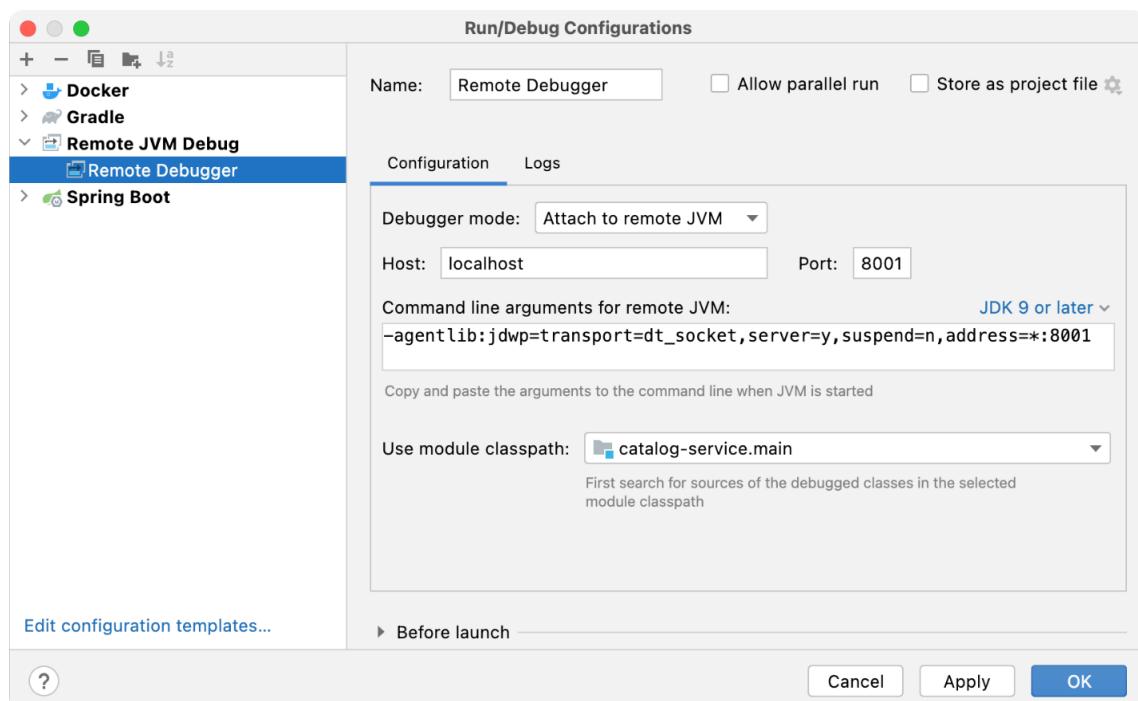


Figure 6.14 Configuration to debug a containerized Java application from IntelliJ IDEA.

Once you run it, you can debug the Catalog Service as though it were running locally. That's it for this section. You can stop and remove both containers with the following command from the same folder where you keep your `docker-compose.yml` file.

```
$ docker-compose down
```

NOTE

In this book, I'm only covering those Docker topics needed in your journey to deploy Spring Boot applications in production with Kubernetes successfully. If you're interested in learning more about Docker images, networks, volumes, security, and architecture, refer to the official documentation on <https://docs.docker.com>. Also, Manning has a few books in its catalog on the subject, such as "Learn Docker in a Month of Lunches" by Elton Stoneman (<https://livebook.manning.com/book/learn-docker-in-a-month-of-lunches>) and "Docker in Practice (Second Edition)" by Ian Miell and Aidan Hobson Sayers (<https://livebook.manning.com/book/docker-in-practice-second-edition>).

When you make some changes to an application, you don't want to build and publish a new image manually. That's a job for an automated workflow engine like GitHub Actions. The following section will show you how to complete the commit stage of the deployment pipeline we started in chapter 3.

6.4 Deployment pipeline: Package and publish

In chapter 3, we started implementing a deployment pipeline to support continuous delivery for the Polar Bookshop project. Continuous delivery is a holistic engineering approach for quickly, reliably, and safely delivering high-quality software. The deployment pipeline is the primary pattern to automate the entire journey from code commit to releasable software. We identified three main stages for the deployment pipeline: commit stage, acceptance stage, and production stage.

We'll continue focusing on the commit stage. After a developer commits new code to the mainline, this stage goes through build, unit tests, integration tests, static code analysis, and packaging. At the end of this stage, an executable application artifact is published to an artifact repository. That is a *release candidate*. Chapter 3 covered all the main steps but the final packaging and publishing of a release candidate. That's what we'll see in this section.

6.4.1 Building release candidates in the commit stage

After running static code analysis, compilation, unit tests, and integration tests, it's time to package the application as an executable artifact and publish it. In our case, the executable artifact is a container image that we publish to a container registry.

An essential idea of continuous delivery, also present in the 15-Factors methodology, is to build artifacts only once. At the end of the commit stage, we produce a container image that we're going to reuse in any following stage in the deployment pipeline up to production. If the pipeline proves something is wrong (a test fails) at any point, the release candidate is rejected. If the release candidate goes through all subsequent stages successfully, it's proven to be ready for deployment in production.

After building an executable artifact, we can perform additional operations before publishing it. For example, we could scan it for vulnerabilities. That's what we're going to do with `grype`, similar to what we did for the codebase. A container image includes application libraries but also system libraries that were not included in the previous security analysis. That's why we want to scan both the codebase and the artifact for vulnerabilities. Figure 6.15 illustrates the new steps we will add to the commit stage for building and publishing a release candidate.

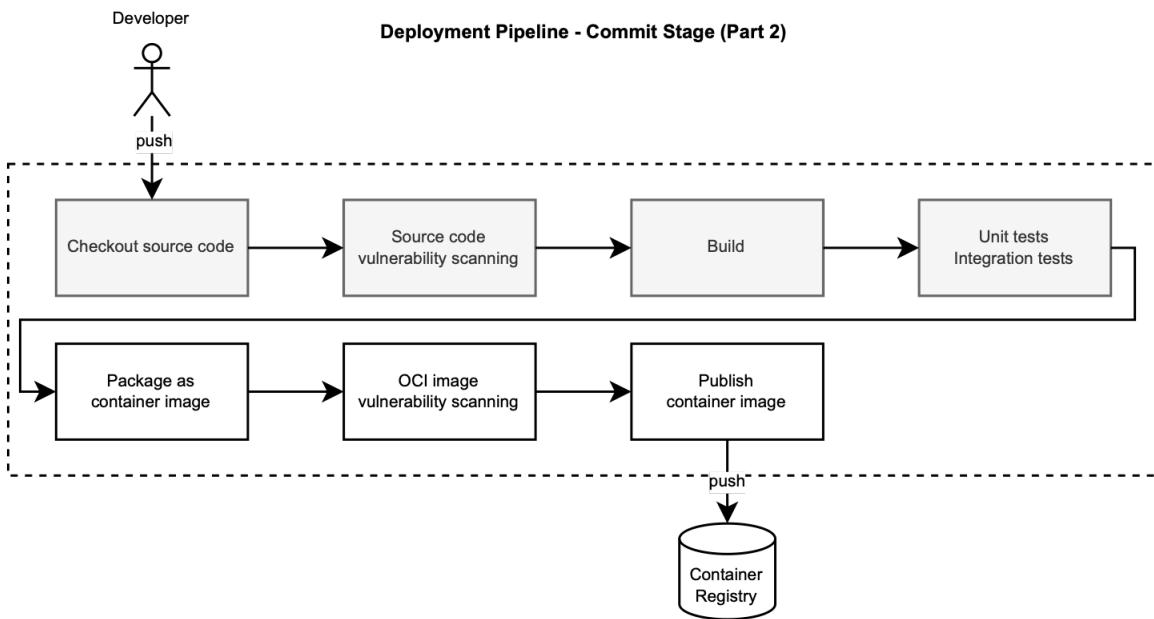


Figure 6.15 At the end of the commit stage, a release candidate is published to an artifact repository. In this case, a container image is published to a container registry.

Once a release candidate is published, several parties can download it and use it, including the next stages in the deployment pipeline. How can we ensure that all interested parties use a legit container image from the Polar Bookshop project and not one that has been compromised? We can achieve that by signing the image. After the publishing step, we could add a new one for signing the release candidate. For example, we could use Sigstore (<https://www.sigstore.dev>), a non-profit service that provides open-source tools to sign, verify, and protect software integrity. If you're interested in this subject, I recommend checking the project website.

In the following section, I'll show you how to implement the new steps in the commit stage of our deployment pipeline.

6.4.2 Publishing container images with GitHub Actions

GitHub Actions is an engine that you can use to automate software workflows directly from your GitHub repositories. Workflow definitions are conventionally stored in a `.github/workflows` directory placed in the root of a repository on GitHub.

In chapter 3, we started working on a workflow to implement the commit stage of the deployment pipeline for Catalog Service. Let's now continue the implementation by adding further steps to package and publish the application.

From your Catalog Service project (`catalog-service`), open the workflow definition for the commit stage (`.github/workflows/commit-stage.yml`) and define a few environment variables to store some essential facts we need when building a container image for the application. By using environment variables, it's easier to change what container registry to use

or the version for the release artifact. Remember to add your GitHub username all in lowercase instead of the placeholder. Chapter 15 will cover software release strategies. Until then, we're going to tag every image with `latest` instead of a version number.

Listing 6.9 Configure facts about the release candidate as environment variables (commit-stage.yml)

```
name: Commit Stage
on: push

env:
  REGISTRY: ghcr.io ①
  IMAGE_NAME: <your_github_username>/catalog-service ②
  VERSION: latest ③

jobs:
  ...

```

- ① Uses the GitHub Container Registry.
- ② The name of the image. Remember to add your GitHub username all in lowercase.
- ③ For the moment, any new image will be tagged as "latest".

Next, let's add a new "Package and Publish" job to the workflow. If the "Build and Test" job completes successfully and the workflow runs on the `main` branch, the new job will be executed. We'll use the same strategy we used locally to package Catalog Service as a container image, relying on the Buildpacks integration provided by the Spring Boot Gradle plugin. Notice how we are not pushing the image directly. That's because we first want to scan the image for vulnerabilities, which we'll do in a moment. For now, update the `commit-stage.yml` file as follows.

Listing 6.10 Package the application as a container image using Buildpacks (commit-stage.yml)

```

name: Commit Stage
on: push

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: <your_github_username>/catalog-service
  VERSION: latest

jobs:
  build:
    ...
    package: ①
      name: Package and Publish
      if: ${{ github.ref == 'refs/heads/main' }} ②
      needs: [ build ] ③
      runs-on: ubuntu-22.04 ④
      permissions:
        contents: read ⑤
        packages: write ⑥
        security-events: write ⑦
      steps:
        - name: Checkout source code
          uses: actions/checkout@v3 ⑧
        - name: Set up JDK
          uses: actions/setup-java@v3 ⑨
          with:
            distribution: temurin
            java-version: 17
            cache: gradle
        - name: Build container image
          run: |
            chmod +x gradlew
            ./gradlew bootBuildImage \
              --imageName
${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:${{ env.VERSION }}

```

- ① The job unique identifier.
- ② Runs the job only on the main branch.
- ③ Runs the job only if the "build" job completes successfully.
- ④ Runs the job on an Ubuntu 22.04 machine.
- ⑤ Permission to check out the current Git repository.
- ⑥ Permission to upload images to GitHub Container Registry.
- ⑦ Permission to submit security events to GitHub.
- ⑧ Checks out the current Git repository ("catalog-service").
- ⑨ Installs and configures a Java runtime.
- ⑩ Relies on the Buildpacks integration in Spring Boot to build a container image and defines the name for the release candidate.

After packaging the application as a container image, let's go ahead and use `grype` to scan it for vulnerabilities and publish a report to GitHub, similar to what we did in chapter 3. Finally, we

can authenticate with the container registry and push the image representing our release candidate.

Listing 6.11 Scan the image for vulnerabilities and publish it to GitHub Container Registry (commit-stage.yml)

```
name: Commit Stage
on: push

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: polarbookshop/catalog-service
  VERSION: latest

jobs:
  build:
    ...
  package:
    ...
    steps:
      - name: Checkout source code
      ...
      - name: Set up JDK
      ...
      - name: Build container image
      ...
      - name: OCI image vulnerability scanning
        uses: anchore/scan-action@v3 ①
        id: scan
        with: ②
          image:
            ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:${{ env.VERSION }}
              fail-build: false ③
              severity-cutoff: high
              acs-report-enable: true
      - name: Upload vulnerability report
        uses: github/codeql-action/upload-sarif@v2 ④
        if: success() || failure()
        with:
          sarif_file: ${{ steps.scan.outputs.sarif }}
      - name: Log into container registry
        uses: docker/login-action@v2 ⑤
        with:
          registry: ${{ env.REGISTRY }} ⑥
          username: ${{ github.actor }} ⑦
          password: ${{ secrets.GITHUB_TOKEN }} ⑧
      - name: Publish container image ⑨
        run: docker push
  ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:${{ env.VERSION }}
```

- ① Scans the release candidate image for vulnerabilities using grype.
- ② The image to scan is the release candidate
- ③ It won't fail the build if vulnerabilities are found in the image.
- ④ Uploads the security vulnerability report to GitHub (SARIF format).
- ⑤ Authenticates with GitHub Container Registry.
- ⑥ The registry value as defined in the environment variable earlier.
- ⑦ The GitHub username of the current user. Provided by GitHub Actions.

- ⑧ The token needed to authenticate with the registry. Provided by GitHub Actions.
- ⑨ Pushes the release candidate to the registry.

In listing 6.10, we are not failing the workflow if serious vulnerabilities are discovered. However, you can find the scanning results in the "Security" section of your `catalog-service` GitHub repository. At the moment of writing, there is no high or critical vulnerability found in the Catalog Service project. Things might be different in the future. As already mentioned in chapter 3, for a real-world scenario, I recommend you configure and tune `grype` carefully and according to your company policies regarding supply chain security, and make the workflow fail if the result is not compliant (setting the `fail-build` property to `true`). For more information, refer to the official project documentation (<https://github.com/anchore/grype>).

After completing the commit stage of the deployment pipeline, make sure your `catalog-service` GitHub repository is public. Then, push your changes to the `main` branch of your remote repository, and see the workflow execution results in the *Actions* tab.

WARNING The action uploading the vulnerability report requires the GitHub repository to be public. It works for private repositories only if you have an enterprise subscription. If you prefer keeping your repository private, you need to skip the "Upload vulnerability report" step. Throughout the book, I'll assume all the repositories we create on GitHub for the Polar Bookshop project are public.

Images published from GitHub Actions and named after a repository are automatically associated. After the workflow completes its execution, in the sidebar of your GitHub `catalog-service` repository main page, you'll find a "Packages" section with a "catalog-service" item (figure 6.16). Click on that, and you'll be directed to the container image repository for Catalog Service.

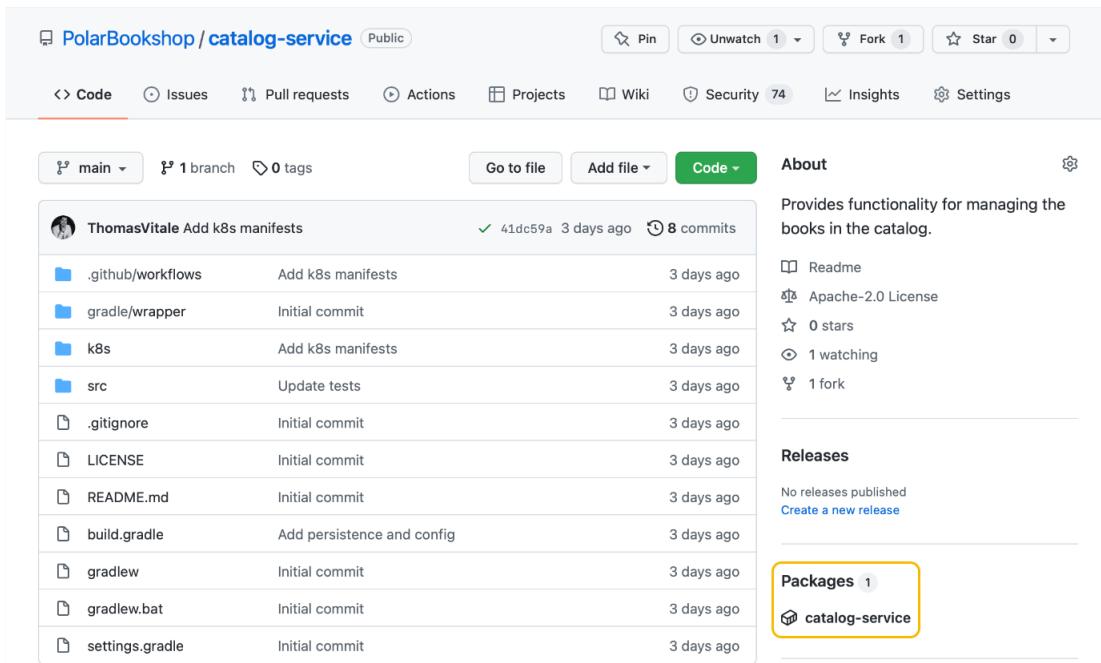


Figure 6.16 When using the GitHub Container Registry, you can store container images next to the source code.

NOTE

Images published to GitHub Container Registry will have the same visibility as the related GitHub code repository. If no repository is associated with the image, it's private by default. Throughout the book, I'll assume that all the images we build for Polar Bookshop are publicly accessible via the GitHub Container Registry. If that's not the case, you can go to the package main page, select "Package settings" from the sidebar menu, scroll to the bottom of the settings page, and make the package public by clicking the "Change visibility" button.

Great job! So far, you built a Spring Boot application exposing a REST API and interacting with a relational database; you wrote unit and integration tests for the application; you handled the database schema with Flyway to be production-ready; and finally, you ran everything within containers and dealt with image generation, Docker, Cloud Native Buildpacks, and vulnerability scanning. The next chapter will complete the first part of this cloud native journey toward production by diving deep into Kubernetes. But before moving on, take a break, congratulate yourself on what you have achieved so far, and perhaps celebrate with a beverage of your choice.

SIDE BAR**Polar Labs**

Feel free to apply what you learned in this chapter to Config Service.

1. Configure the Cloud Native Buildpacks integration and package the application as a container.
2. Update your Docker Compose file to run Config Service as a container.
3. Configure Catalog Service with the Config Service URL via the `SPRING_CLOUD_CONFIG_URI` environment variable, relying on the Docker built-in DNS.
4. Bootstrap a deployment pipeline for Config Service by implementing the workflow for the commit stage using GitHub Actions.

You can refer to the `Chapter06/06-end` folder in the code repository accompanying the book for checking the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

6.5 Summary

- Container images are lightweight executable packages that include everything needed to run the application inside.
- Each image is made up of several layers. Each layer represents a modification produced by the corresponding instruction. The final artifact can be run as a container.
- When you run a container, an additional writable layer is added on top of the image layers.
- The standard way to define a container image is by listing the sequence of instructions in a particular file called Dockerfile.
- A Dockerfile acts as a recipe containing all the steps to build the desired image.
- Performance and security are important concerns when building container images. For example, you shouldn't store secrets in any image layer and never run the container with a root user.
- A container registry is to OCI images what a Maven repository is to Java libraries. Examples of container registries are Docker Hub and GitHub Container Registry.
- You can package Spring Boot applications as container images in different ways.
- Dockerfiles give you maximum flexibility but make it your responsibility to configure everything you need.
- Cloud Native Buildpacks (integrated with the Spring Boot Plugin) let you build OCI images directly from the source code, optimizing security, performance, and storage for you.
- When you run Spring Boot applications as containers, you should consider which ports you want to make available to the outside world (e.g., 8080) and whether containers should communicate with each other. If yes, you can use the Docker DNS server to contact containers in the same network by container name instead of IP or hostname.
- If you want to debug an application running as a container, remember to expose the debug port.
- Docker Compose is a client for interacting with the Docker server and providing a better user experience than Docker CLI. From a YAML file, you can manage all your containers.
- You can use GitHub Actions to automate the process of packaging an application as a container image, scan it for vulnerabilities, and publish it to a container registry. That's part of the commit stage of a deployment pipeline.
- The outcome of the commit stage of a deployment pipeline is a release candidate.



Kubernetes fundamentals for Spring Boot

This chapter covers

- Moving from Docker to Kubernetes
- Deploying Spring Boot applications on Kubernetes
- Understanding service discovery and load balancing
- Building scalable and disposable applications
- Establishing a local Kubernetes development workflow
- Validating Kubernetes manifests with GitHub Actions

In the previous chapter, you learned about Docker and the main characteristics of images and containers. With Buildpacks and Spring Boot, you can build a production-ready image in one command without even having to write your own Dockerfile or install additional tools. With Docker Compose, you can simultaneously control multiple applications, which is convenient for architectures like microservices. What if a container stops working? What if the machine where your containers are running (the Docker host) crashes? What if you want to scale your applications? This chapter will introduce Kubernetes into your workflow to address issues that Docker alone cannot.

As a developer, it's not your job to configure and manage a Kubernetes cluster. You would probably use either a managed service offered by a cloud provider such as Amazon, Microsoft, or Google, or a service managed on-premises by a specialized team in your organization (commonly known as the *platform team*). For now, you'll use a local Kubernetes cluster provisioned with *minikube*. Later in the book, you'll use a managed Kubernetes service offered by a cloud provider.

In our daily job as developers, we don't want to spend too much time on infrastructural concerns, but it's critical to know the basics. Kubernetes has become the de-facto orchestration tool and the

common language to talk about containerized deployments. Cloud vendors have been working on building platforms on top of Kubernetes to provide a better experience to developers. Once you know how Kubernetes works, it'll be straightforward to use those platforms because you'll be familiar with the language and the abstractions.

This chapter will walk you through the main features of Kubernetes and teach you how to create and manage Pods, Deployments, and Services for your Spring Boot applications. Along the way, you'll enable graceful shutdown for your applications, learn how to scale them, and how to use the service discovery and load balancing features provided by Kubernetes. You'll also learn to automate your local development workflow with Tilt, visualize your workloads with Octant, and validate your Kubernetes manifests.

NOTE

The source code for the examples in this chapter is available in the Chapter07/07-begin and Chapter07/07-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

7.1 Moving from Docker to Kubernetes

With Docker Compose, you get the opportunity to manage the deployment of several containers at once, including the configuration of networks and storage. That is extremely powerful, but it's limited to one machine.

Using Docker CLI and Docker Compose, the interaction happens with a single Docker daemon that manages Docker resources on a single machine, called the Docker host. Furthermore, it's not possible to scale a container. All of this is limiting when you need cloud native properties like scalability and resilience for your system. Figure 7.1 shows how you target a single machine when using Docker.

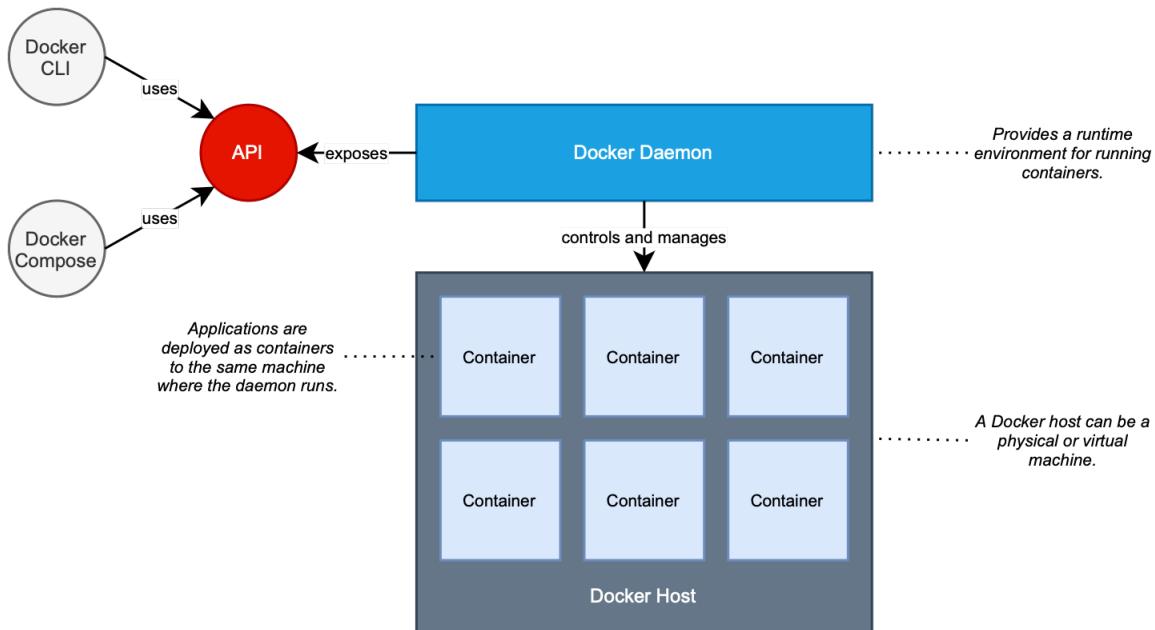


Figure 7.1 Docker clients interact with a Docker daemon that can only manage resources on the machine where it is installed, called the Docker host. Applications are deployed as containers to the Docker host.

You learned in chapter 2 that we change the point of view when we move from a container runtime like Docker to an orchestration platform like Kubernetes. With Docker, we deploy containers to an individual machine. With Kubernetes, we deploy containers to a cluster of machines, enabling scalability and resilience.

Kubernetes clients use an API to interact with the Kubernetes Control Plane, responsible for creating and managing objects in a Kubernetes cluster. In this new scenario, we still send commands to a single entity, but this acts on several machines rather than only one. Figure 7.2 shows the logical infrastructure when we use Kubernetes.

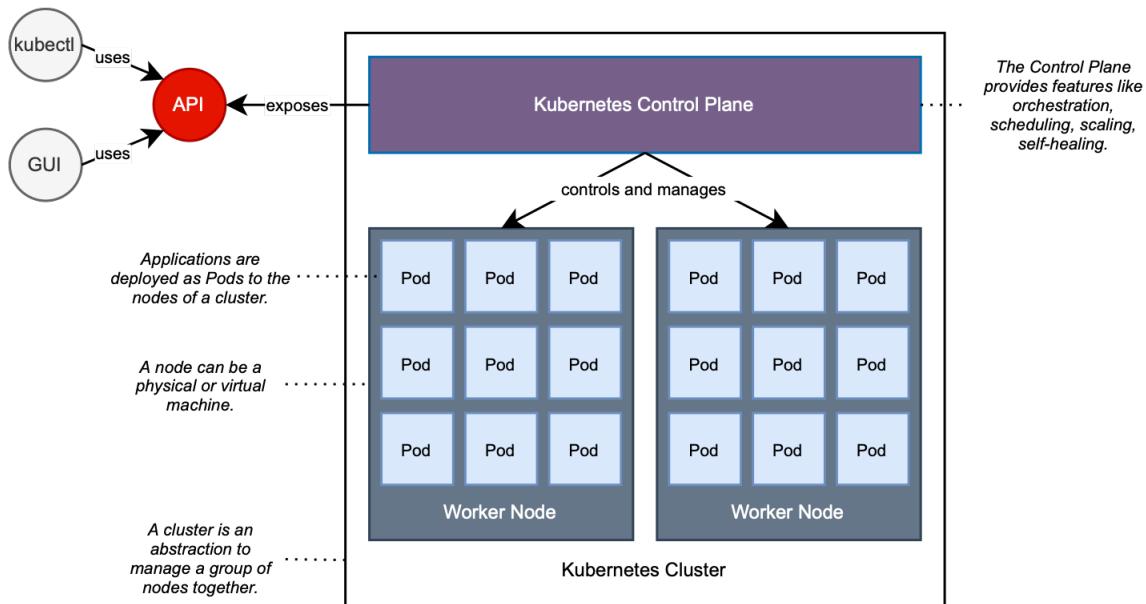


Figure 7.2 Kubernetes clients interact with the Control Plane that manages containerized applications in a cluster comprised of one or more nodes. Applications are deployed as Pods to the nodes of a cluster.

The main components shown in figure 7.2 are:

- **Cluster.** A set of nodes running containerized applications. It hosts the Control Plane and comprises one or more worker nodes.
- **Control Plane.** The cluster component exposing the API and interfaces to define, deploy and manage the lifecycle of Pods. It comprises all the essential elements that implement the typical features of an orchestrator, like cluster management, scheduling, and health monitoring.
- **Worker Nodes.** Physical or virtual machines providing capacity such as CPU, memory, network, and storage so that containers can run and connect to a network.
- **Pod.** The smallest deployable unit wrapping an application container.

With a good understanding of the Kubernetes infrastructure, let's see how to create and manage a Kubernetes cluster on your local machine.

7.1.1 Working with a local Kubernetes cluster

In chapter 2, we worked with *minikube* (<https://minikube.sigs.k8s.io>), a tool for running Kubernetes clusters on local environments. We used the *minikube* CLI to create a local Kubernetes cluster relying on the default configuration. In this section, you'll see how to define a custom configuration for *minikube* to use when initializing a new local Kubernetes cluster for deploying Polar Bookshop.

NOTE

If you haven't installed *minikube* yet, refer to the instructions available in section A.3 of Appendix A.

Since we run `minikube` on top of Docker, remember to start the Docker Engine first. Then, make sure the default cluster is not running by executing the `minikube stop` command. From now on, we won't use the default cluster. Instead, we'll create a custom one for working with Polar Bookshop. With `minikube`, you can create and control multiple clusters identified via *profiles*. When no profile is specified, `minikube` falls back on the default cluster.

WARNING **Running the examples on a local Kubernetes cluster requires Docker to have at least 2 CPUs and 4GB of memory. If you are using Docker Desktop for Mac or for Windows, and you need to increase the resources assigned to the Docker Engine, refer to the product documentation for instructions on how to do that on your specific operating system (<https://docs.docker.com/desktop>).**

Let's create a new Kubernetes cluster named `polar` on top of Docker. This time, we also want to declare the resource limits for CPU and memory.

```
$ minikube start --cpus 2 --memory 4g --driver docker --profile polar
```

You can get a list of all the nodes in the cluster with the following command. The cluster we have just created is composed of a single node, which hosts the Control Plane and acts as a worker node for deploying containerized workloads.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
polar	Ready	control-plane, master	21s	v1.23.3

You can use the same Kubernetes Client (`kubectl`) to interact with different local or remote clusters. The following command will list all the available *contexts* with which you can interact.

```
$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO
*	polar	polar	polar

If you have more than one context, make sure `kubectl` is configured to use `polar`. You can verify which is the current context by running this command.

```
$ kubectl config current-context
polar
```

If the result is different from `polar`, you can change the current context as follows.

```
$ kubectl config use-context polar
Switched to context "polar".
```

For the rest of the chapter, I will assume you have this local cluster up and running. At any time, you can stop the cluster with `minikube stop --profile polar` and start it again with

`minikube start --profile polar`. If you ever want to delete it and start over, you can run the `minikube delete --profile polar` command.

In the next section, you'll complete the setup for your local Kubernetes cluster by deploying a PostgreSQL database.

7.1.2 Managing data services in a local cluster

As you learned in chapter 5, data services are the stateful components of a system and require special care in a cloud environment due to the challenges of handling their storage. Managing persistence and storage in Kubernetes is a complex topic and is not usually the responsibility of developers. When you deploy the Polar Bookshop system in production, you'll rely on the managed data services offered by the cloud provider, so we don't want to spend too much time setting up a database in our local environments.

Therefore, I have already prepared the configuration for deploying PostgreSQL in your local Kubernetes cluster. Check the source code repository accompanying this book (Chapter 07/07-end) and copy the content of the `polar-deployment/kubernetes/platform/development` folder into the same path in your `polar-deployment` repository. The folder contains basic Kubernetes manifests to run a PostgreSQL database.

Open a Terminal window, navigate to the `kubernetes/platform/development` folder located in your `polar-deployment` repository, and run the following command to deploy PostgreSQL in your local cluster.

```
$ kubectl apply -f services
```

NOTE The previous command creates the resources defined in the manifests within the `services` folder. In the next section, you'll learn more about the `kubectl apply` command and Kubernetes manifests.

The result will be a Pod running a PostgreSQL container in your local Kubernetes cluster. You can check it out with the following command.

```
$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
polar-postgres-677b76bfc5-1kkqn	1/1	Running	0	48s

TIP You can check the database logs by running `kubectl logs deployment/polar-postgres`.

NOTE

A popular way of running third-party services in a Kubernetes cluster is through Helm (<https://helm.sh>). Think of it as a package manager. To install software on your computer, you would use one of the operating system package managers, like Apt (Ubuntu), Homebrew (macOS), or Chocolatey (Windows). In Kubernetes, you can similarly use Helm, but we call them charts instead of packages. At this stage of our cloud native journey, using Helm would have been a bit premature and perhaps confusing. To fully understand how it works, it's essential to acquire more familiarity with Kubernetes first.

For the rest of the chapter, I assume you have a PostgreSQL instance running in your local cluster. If at any point you need to undeploy the database, you can run the `kubectl delete -f services` command from the same folder.

The following section will present the main Kubernetes concepts and guide you through deploying Spring Boot applications on your local cluster.

7.2 Kubernetes Deployments for Spring Boot

This section will walk you through the main Kubernetes objects you'll be working with as a developer and the vocabulary necessary to communicate efficiently with the platform team and deploy your applications to a cluster.

You have already gone through the containerization of a Spring Boot application. A Spring Boot application is still packaged as a container, but it runs in a Pod controlled by a Deployment object.

Pods and Deployments are core concepts you need to understand when working with Kubernetes. Let's start by looking at some of their main characteristics, after which you'll practice declaring and creating Kubernetes resources to deploy the Catalog Service application.

7.2.1 From containers to Pods

As we discussed in the previous section, Pods are the smallest deployable units in Kubernetes. When moving from Docker to Kubernetes, we switch from managing containers to managing Pods.

SIDE BAR**DEFINITION**

A Pod is the smallest Kubernetes object and represents a set of running containers in a cluster. It's usually set up to run a single primary container (your application), but it can also run optional helper containers with additional features like logging, monitoring, or security. (<https://kubernetes.io/docs>)

A Pod is usually comprised of one container: the application instance. When that happens, it's not much different from working with containers directly. However, there are some scenarios where your application container needs to be deployed together with some *helper* containers, perhaps performing initialization tasks required by the application or adding extra functionality such as logging. For example, Linkerd (a *service mesh*) adds its own container (a *sidecar*) to Pods to perform operations such as intercepting HTTP traffic and encrypting it to guarantee secure communication between all Pods via mTLS. Figure 7.3 illustrates single-container and multi-container Pods.

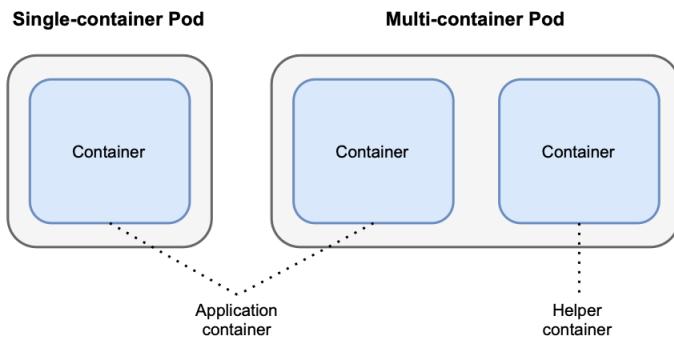


Figure 7.3 Pods are the smallest deployable units in Kubernetes. They run at least one primary container (the application) and might run optional helper containers for additional features like logging, monitoring, or security.

In this book, you'll work with single-container Pods, where the container is the application. Compared to containers, Pods allow you to manage related containers as a single entity. But that's not enough. Directly creating and managing Pods would be no much different than working with plain Docker containers. We need something at a higher level of abstraction to define how we want to deploy and scale our applications. That's where the Deployment objects come into play.

7.2.2 Controlling Pods with Deployments

How can you scale an application to have five replicas running? How can you ensure there are always five replicas up and running even when failures occur? How can you deploy a new version of the application without downtime? With *Deployments*.

SIDEBAR

DEFINITION

A Deployment is an object that manages the lifecycle of a stateless, replicated application. Each replica is represented by a Pod. The replicas are distributed among the nodes of a cluster for better resilience. (<https://kubernetes.io/docs>)

In Docker, you manage your application instances directly by creating/removing containers. In

Kubernetes, you don't manage Pods. You let a Deployment do that for you. Deployment objects have several important and valuable characteristics. You can use them to deploy your applications, roll out upgrades without downtime, roll back to a previous version in case of errors, pause and resume upgrades.

Deployments also let you manage replication. They make use of an object named ReplicaSet to ensure there's always the desired number of Pods up and running in your cluster. If one of them crashes, a new one is created automatically to replace it. Furthermore, replicas are deployed across different nodes in your cluster to ensure even higher availability if one node crashes. Figure 7.4 shows the relationship between containers, Pods, ReplicaSets, and Deployments.

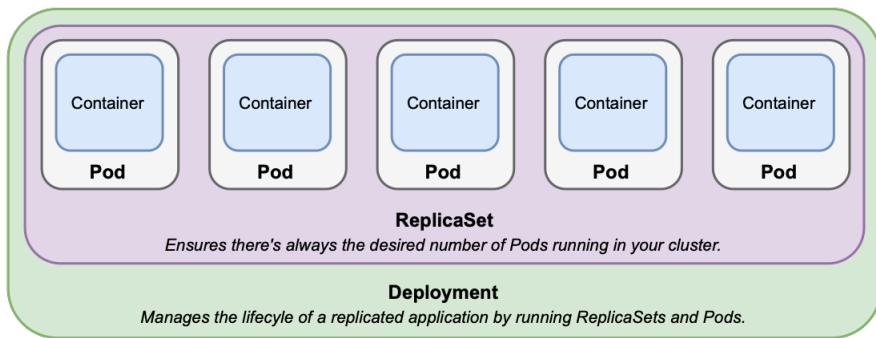


Figure 7.4 A Deployment manages replicated applications with ReplicaSets and Pods in your cluster. A ReplicaSet ensures the desired number of Pods is always up and running. A Pod runs a containerized application.

Deployments provide a convenient abstraction for us to declare what we want to achieve (the *desired state*) and let Kubernetes make it happen. You don't need to worry about how to achieve a specific result. Unlike imperative tools such as Ansible or Puppet, you can just tell Kubernetes what you want, and the orchestrator will figure out how to achieve the desired result and keep it consistent. That's what we call *declarative configuration*.

Kubernetes uses controllers that watch the system and compare the desired state with the actual state. When there is any difference between the two, it acts to make them match again. Deployments and ReplicaSets are controller objects, handling rollout, replication, and self-healing. For example, suppose you declare that you want three replicas deployed of your Spring Boot application. If one crashes, the associated ReplicaSet notices it and creates a new Pod to align the actual state with the desired one.

After packaging a Spring Boot application as an OCI image, all you need to do to run it in a Kubernetes cluster is define a Deployment object. You'll learn how in the next section.

7.2.3 Creating a Deployment for a Spring Boot application

There are a few options to create and manage Kubernetes objects in a cluster. In chapter 2, we used the `kubectl` client directly, but that approach lacks version control and reproducibility. It's the same reason why we prefer Docker Compose over the Docker CLI.

In Kubernetes, the recommended way is to describe an object's desired state in a *manifest* file, typically specified in YAML format. We use **declarative configuration**: we declare what we want instead of how to achieve it. In chapter 2, we *imperatively* used `kubectl` to create and delete objects. When we work with manifests, we *apply* them to the cluster. Then, Kubernetes will automatically reconcile the actual state in the cluster with the desired state in the manifest.

A Kubernetes manifest usually comprises four main sections, as shown in figure 7.5.

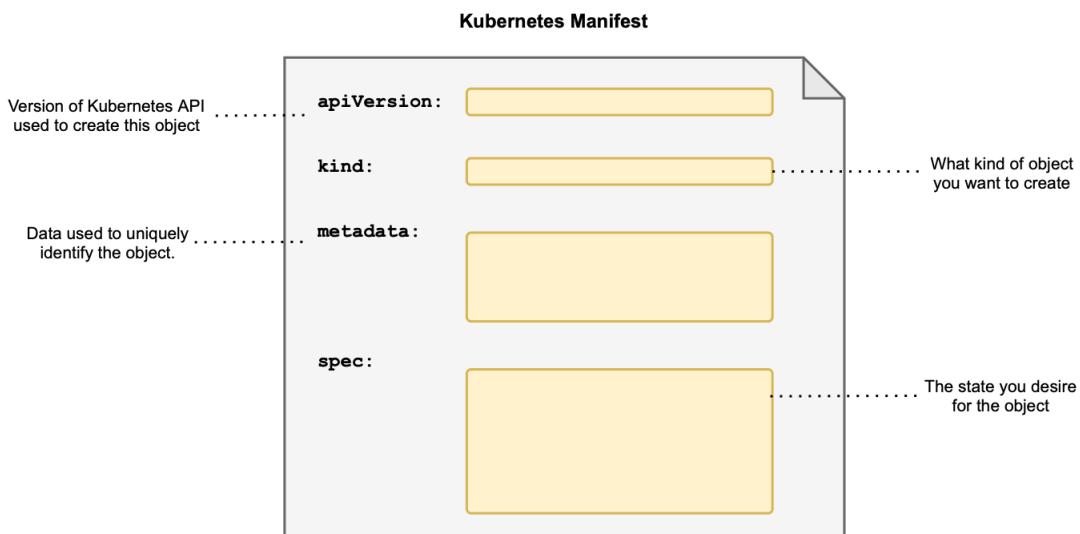


Figure 7.5 A Kubernetes manifest is typically composed of four main sections: `apiVersion`, `kind`, `metadata`, and `spec`.

- `apiVersion` defines the versioned schema of the specific object representation. Core resources such as Pods or Services follow a versioned schema composed of only a version number (such as `v1`). Other resources like Deployments or ReplicaSet follow a versioned schema consisting of a group and a version number (for example, `apps/v1`). If in doubt about which version to use, you can refer to the Kubernetes documentation (<https://kubernetes.io/docs>) or use the `kubectl explain <object_name>` command to get more information about the object, including the API version to use.
- `kind` is the type of Kubernetes object you want to create, such as Pod, ReplicaSet, Deployment, or Service. You can use the `kubectl api-resources` command to list all the objects supported by the cluster.
- `metadata` provides details about the object you want to create, including the name and a set of labels (key/value pairs) used for categorization. For example, you can instruct Kubernetes to replicate all the objects with a specific label attached.
- `spec` is a section specific to each object type and is used to declare the desired configuration.

After getting familiar with the main sections of a Kubernetes manifest, let's define one for a Deployment object to run a Spring Boot application.

DEFINING A DEPLOYMENT MANIFEST WITH YAML

There are different strategies for organizing Kubernetes manifests. For the Catalog Service application, create a `k8s` folder in the project root (`catalog-service`). We will use it to store the manifests for the application.

NOTE

If you haven't followed along with the examples implemented in the previous chapters, you can refer to the repository accompanying the book (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and use the project in Chapter07/07-begin/catalog-service as a starting point.

Let's start by creating a `deployment.yaml` file inside the `catalog-service/k8s` folder. As you saw in figure [7.5](#), the first sections to include are `apiVersion`, `kind`, and `metadata`.

Listing 7.1 Initialize a Deployment manifest for Catalog Service (deployment.yaml)

```
apiVersion: apps/v1 ①
kind: Deployment ②
metadata:
  name: catalog-service ③
  labels: ④
    app: catalog-service ⑤
```

- ① The API version for Deployment objects.
- ② The type of object to create.
- ③ The name of the Deployment.
- ④ A set of labels attached to the Deployment.
- ⑤ This Deployment is labeled with "app=catalog-service".

NOTE

The Kubernetes API can change over time. Make sure you always use the API supported by the version of Kubernetes you're running. If you have followed along so far, you shouldn't have this problem. But in case it happens, `kubectl` will return a very descriptive error message telling you exactly what's wrong and how to fix it. You can also use the `kubectl explain <object_name>` command to check the API version supported by your Kubernetes installation for a given object.

The `spec` section of a Deployment manifest contains a `selector` part to define a strategy for identifying which objects should be scaled by a ReplicaSet (more on this later) and a `template` part describing the specifications for creating the desired Pod and containers.

Listing 7.2 Declare the desired state for the Catalog Service deployment (deployment.yml)

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  labels:
    app: catalog-service
spec:
  selector: ①
    matchLabels:
      app: catalog-service
  template: ②
    metadata:
      labels: ③
        app: catalog-service
    spec:
      containers: ④
        - name: catalog-service ⑤
          image: catalog-service ⑥
          imagePullPolicy: IfNotPresent ⑦
          ports:
            - containerPort: 9001 ⑧
          env: ⑨
            - name: BPL_JVM_THREAD_COUNT ⑩
              value: "50"
            - name: SPRING_DATASOURCE_URL ⑪
              value: jdbc:postgresql://polar-postgres/polardb_catalog
            - name: SPRING_PROFILES_ACTIVE ⑫
              value: testdata

```

- ① Defines the labels used to select the Pods to scale.
- ② The template for creating a Pod.
- ③ Labels attached to the Pod object. They should match the ones used as selectors.
- ④ The list of containers in the Pod. Only one in this case.
- ⑤ The name of the Pod.
- ⑥ The image used to run the container. No tag is defined, so "latest" will be used implicitly.
- ⑦ Instructs Kubernetes to pull the image from a container registry only if it's not present locally yet.
- ⑧ The ports exposed by the container.
- ⑨ The list of environment variables passed to the Pod.
- ⑩ Paketo Buildpacks environment variable to configure the number of threads for memory calculation.
- ⑪ A value for the "spring.datasource.url" property pointing to the PostgreSQL Pod deployed previously.
- ⑫ Enables the "testdata" Spring profile.

The `containers` part should look familiar to you since it resembles how you defined containers in the `services` section of a Docker Compose file. Like you did with Docker, you can use an

environment variable to define the URL for the PostgreSQL instance the application should use. The hostname part of the URL (`polar-postgres`) is the name of the Service object used to expose the database and created previously from the `kubernetes/platform/development` folder. You'll learn more about Services later in the chapter. For now, it's enough to know that `polar-postgres` is the name through which other objects in the cluster can communicate with the PostgreSQL instance.

In a production scenario, the image would be fetched from a container registry. During development, it's more convenient to work with local images. Let's build one for Catalog Service, as you learned in the previous chapter.

Open a Terminal window, navigate to the Catalog Service root folder (`catalog-service`), and build a new container image as follows.

```
$ ./gradlew bootBuildImage
```

By default, `minikube` doesn't have access to your local container images. Therefore, it will not find the image you have just built for Catalog Service. But don't worry: you can manually import it into your local cluster.

```
$ minikube image load catalog-service --profile polar
```

NOTE YAML is an expressive language, but it can make your coding experience quite bad due to its constraints about spaces or perhaps lack of support from your editor. When a `kubectl` command involving a YAML file fails, verify that spaces and indentation are used correctly. For Kubernetes, you can install a plugin in your editor to support you while writing YAML manifests, ensuring you always use the correct syntax, spaces, and indentation. You can find a few plugin options in the `README.md` file in the repository accompanying this book: <https://github.com/ThomasVitale/cloud-native-spring-in-action>.

Now that you have a Deployment manifest, let's move on and see how to apply it to your local Kubernetes cluster.

CREATING A DEPLOYMENT OBJECT FROM A MANIFEST

You can apply Kubernetes manifests to a cluster using the `kubectl` client. Open a Terminal window, navigate to your Catalog Service root folder (`catalog-service`) and run the following command.

```
$ kubectl apply -f k8s/deployment.yml
```

The command is processed by the Kubernetes Control Plane, which will create and maintain all the related objects in the cluster. You can verify which objects have been created with the

following command.

```
$ kubectl get all -l app=catalog-service

NAME                                READY   STATUS    RESTARTS   AGE
pod/catalog-service-68bc5659b8-k6dpb   1/1    Running   0          42s

NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/catalog-service   1/1     1           1          42s

NAME                DESIRED   CURRENT   READY   AGE
replicaset.apps/catalog-service-68bc5659b8   1         1         1        42s
```

Since you used labels consistently in your Deployment manifest, you can use the label `app=catalog-service` to fetch all the Kubernetes objects related to the Catalog Service deployment. As you can see, the declaration in `deployment.yml` resulted in the creation of a Deployment, a ReplicaSet, and a Pod.

To verify that Catalog Service started up correctly, you can check the logs from its Deployment as follows.

```
$ kubectl logs deployment/catalog-service
```

NOTE

You can monitor if Pods have been created successfully by inspecting the STATUS column when you run `kubectl get pods`. Should a Pod fail to be deployed, check that column. Common error statuses are `ErrImagePull` or `ImagePullBackOff`. They happen when Kubernetes can't pull the image used by the Pod from the configured container registry. We're currently working with local images, so make sure you built and loaded a Catalog Service container image into `minikube`. You can use the `kubectl describe pod <pod_name>` command to get more information about the error and `kubectl logs <pod_name>` to get the application logs from a specific Pod instance.

When deploying containers in a cloud environment like a Kubernetes cluster, you want to be sure it has enough resources to operate. In chapter 15, you'll learn how to assign CPU and memory resources to a container running in Kubernetes and how to configure memory for the JVM applying the Java memory calculator provided by Cloud Native Buildpacks. For now, we'll rely on the default resource configuration.

So far, you have created a Deployment for a Spring Boot application and run it in your local Kubernetes cluster. But it's not possible to use it yet since it's isolated inside the cluster. In the next section, you'll learn how to expose your application to the outside world and how to use the service discovery and load balancing functionality provided by Kubernetes.

7.3 Service discovery and load balancing

After talking about Pods and Deployments, let's dig into Services a bit. You got the Catalog Service application running as a Pod in your local Kubernetes cluster, but there are still unanswered questions. How can it interact with the PostgreSQL Pod running in the cluster? How does it know where to find it? How can you expose a Spring Boot application to be used by other Pods in the cluster? How can you expose it outside the cluster?

This section will answer those questions by introducing two important aspects of cloud native systems: service discovery and load balancing. I'll present the two main patterns available to implement them when working with Spring applications: client-side and server-side. Then, you'll apply the latter approach, which is conveniently offered natively by Kubernetes through Service objects, meaning you don't have to change anything in your code to support it (unlike the client-side option). Finally, you'll understand how the communication between the Catalog Service Pod and the PostgreSQL Pod happens, and you'll expose the Catalog Service application as a network service.

7.3.1 Understanding service discovery and load balancing

When a service needs to communicate with another, it must be provided with information about where to find it. For example, it can be an IP address or a DNS name. Let's consider two applications: Alpha App and Beta App. Figure 7.6 shows how the communication between the two would happen if there were only one Beta App instance.

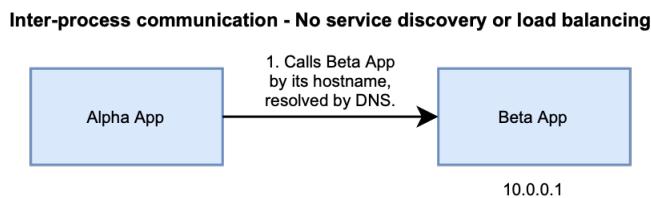


Figure 7.6 If there were only one Beta App instance, the inter-process communication between Alpha App and Beta App would be based on a DNS name resolving to the IP address of Beta App.

In the scenario illustrated in figure 7.6, we say that Alpha App is *upstream* and Beta App is *downstream*. Furthermore, Beta App is a *backing service* with respect to Alpha App. There's only one instance of Beta App running, so the DNS name gets resolved to its IP address.

In the cloud, you probably want to have multiple instances of the same service running. Each service instance has its own IP address. Unlike physical machines or long-running virtual machines, a service instance will not live long in the cloud. Application instances are disposable. They can be removed for different reasons, such as when they are not responsive anymore and

replaced with new ones. You can even enable the auto-scaling feature to automatically scale in and out your application, depending on the workload. Using IP addresses for inter-process communication in the cloud is not an option.

To overcome that issue, you might consider using DNS records relying on a round-robin name resolution pointing to one of the IP addresses assigned to the replicas. Knowing the hostname, you can reach the backing service even if one of the IP addresses changes because the DNS server would be updated with the new ones. However, this approach is not the best fit for cloud environments because the topology changes too often. Some DNS implementations cache the result of a name lookup even after they should have expired. Similarly, some applications cache the DNS lookup response for too long. Either way, there's a high chance of using a hostname/IP address resolution that is no longer valid.

Service discovery in cloud environments requires a different solution. First, we need to keep track of all the service instances running and store that information in a *service registry*. Whenever a new instance is created, an entry should be added to the registry. When it's shut down, it should be removed accordingly. The registry considers that multiple instances of the same application can be up and running. Second, whenever an application needs to call a backing service, it performs a *lookup* in the registry to know which IP address to contact. If multiple instances are available, a *load balancing* strategy is applied to distribute the workload across them.

We distinguish between client-side and server-side service discovery, depending on where the problem is solved. Let's have a look at both options.

7.3.2 Client-side service discovery and load balancing

Client-side service discovery requires applications to register themselves with a service registry upon startup and unregister when shutting down. Whenever they need to call a backing service, they ask the service registry for an IP address. If multiple instances are available, the registry will return the list of IP addresses. The application will choose one of them depending on a load balancing strategy defined by the application itself. Figure 7.7 shows how that works.

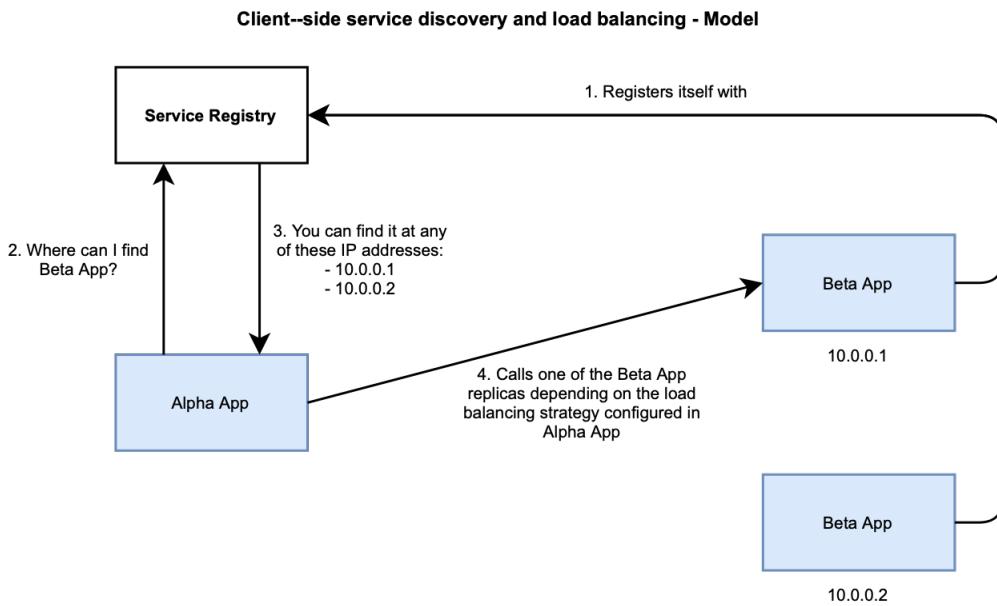


Figure 7.7 The inter-process communication between Alpha App and Beta App is based on the IP address of the specific instance to call, chosen among a list of IP addresses returned upon lookup in the service registry.

The Spring Cloud project offers a few options to add client-side service discovery to your Spring applications. One of the popular choices is Spring Cloud Netflix Eureka, which wraps the Eureka service registry developed by Netflix. Alternatives are Spring Cloud Consul, Spring Cloud Zookeeper Discovery, and Spring Cloud Alibaba Nacos.

Besides managing a service registry explicitly, you also need to add the correct integration to all your applications. For each of the previously mentioned options, Spring Cloud provides a client library you can add to your Spring application to use the service registry with minimal effort. Finally, Spring Cloud Load Balancer can be used for client-side load balancing, the preferred choice over Spring Cloud Netflix Ribbon, no longer maintained.

All those libraries offered by Spring Cloud contributed to making it an excellent choice for building cloud native applications and implementing microservices architectures. A benefit of such a solution is that your applications have complete control over the load balancing strategy. Suppose you need to implement patterns like *hedging*: sending the same request to multiple instances to increase the chance one responds correctly within a specific time limit. In that case, client service discovery can help you with that.

A drawback is that it assigns more responsibility to developers. If your system includes applications built using different languages and frameworks, you'll need to handle the client part for each of them in different ways. Also, it results in one more service to deploy and maintain (the service registry), unless you use PaaS solutions like Azure Spring Cloud or VMware Tanzu Application Service that provide it for you. Server-side discovery solutions solve these issues at the expense of fine-grained control in the application. Let's see how.

7.3.3 Server-side service discovery and load balancing

Server-side service discovery solutions move the responsibility to the deployment platform so that developers can focus on the business logic and rely on the platform for providing all the necessary functionality for service discovery and load balancing. Such solutions automatically register and deregister application instances and rely on a load balancer component to route any incoming request to one of the available instances according to some specific strategy. In this case, the application doesn't need to interact with the service registry, which is updated and managed by the platform. Figure 7.8 shows how that works.

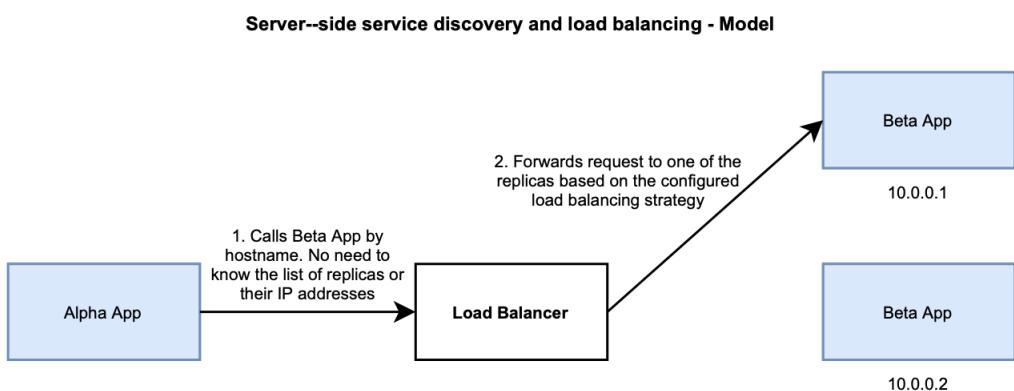


Figure 7.8 The inter-process communication between Alpha App and Beta App is based on a DNS name that gets resolved to one of the instance IP addresses by a load balancer component. The service registration process is handled by the platform transparently.

The Kubernetes implementation of this service discovery pattern is based on Service objects.

SIDE BAR

DEFINITION

A Service is an abstract way to expose an application running on a set of Pods as a network service. (<https://kubernetes.io/docs>)

A Service object is an abstraction targeting a set of Pods (typically using labels) and defining the access policy. When an application needs to contact a Pod exposed by a Service object, it can use the Service name instead of calling the Pod directly. That's what you did to let the Catalog Service application interact with the PostgreSQL instance (polar-postgres was the name of the Service exposing the PostgreSQL Pod). The Service name is then resolved to the IP address of the Service itself by a local DNS server running in the Kubernetes Control Plane.

NOTE

The IP address assigned to a Service is fixed for its lifetime. Therefore, the DNS resolution of a Service name doesn't change as often as it would with application instances.

After resolving the Service name to its IP address, Kubernetes relies on a proxy (called

kube-proxy) which intercepts the connection to the Service object and forwards the request to one of the Pods targeted by the Service. The proxy knows all the replicas available and adopts a load balancing strategy depending on the type of Service and the proxy configuration. There is no DNS resolution involved in this step, solving the problems I explained earlier. The service discovery implementation adopted by Kubernetes is shown in figure 7.9.

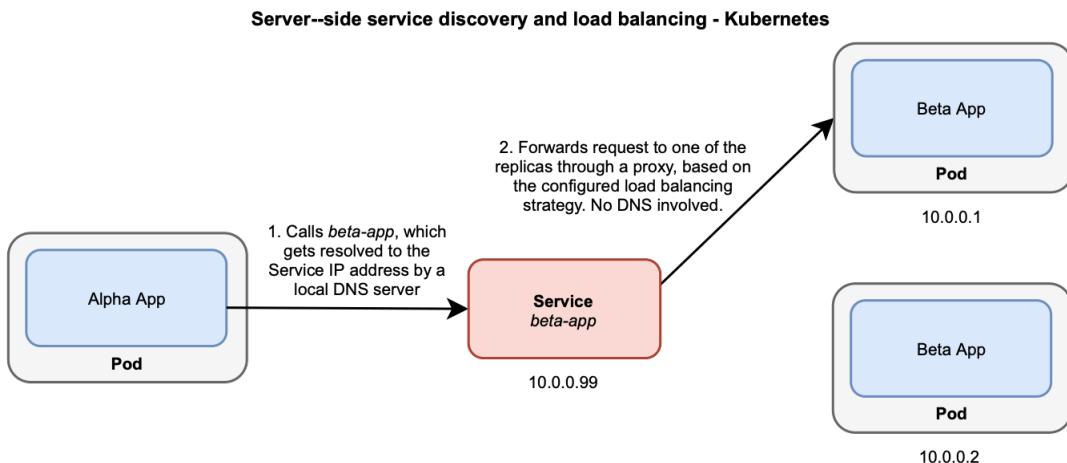


Figure 7.9 In Kubernetes, the inter-process communication between Alpha App and Beta App happens through a Service object. Any request arriving at the Service is intercepted by a proxy that forwards it to one of the replicas targeted by the Service based on a specific load balancing strategy.

This solution is transparent to your Spring Boot applications. Unlike options like Spring Cloud Netflix Eureka, you get service discovery and load balancing out-of-the-box in Kubernetes without requiring any change to your code. That's why it's the preferred option when you use a Kubernetes-based platform for deploying your applications.

SIDEBAR

Service discovery and Spring Cloud Kubernetes

If you need to migrate existing applications that use one of the client-side service discovery options I mentioned in the previous section, maybe you'll be happy to know that you can use Spring Cloud Kubernetes to make the transition smoother. You can keep your existing service discovery and load balancing logic in your application. However, instead of solutions like Spring Cloud Netflix Eureka, you can use the Spring Cloud Kubernetes Discovery Server for the service registry part. It can be a convenient way to migrate applications to Kubernetes without changing too much in your application code. For more information, refer to the project documentation: <https://spring.io/projects/spring-cloud-kubernetes>.

Unless what you're doing requires specific handling of service instances and load balancing in your applications, my recommendation is to migrate over time to using the native service discovery functionality offered by Kubernetes, aiming at removing infrastructural concerns from your applications.

With a general understanding of how service discovery and load balancing are implemented in Kubernetes, let's see how to define a Service to expose a Spring Boot application.

7.3.4 Exposing Spring Boot applications with Kubernetes Services

As you learned in the previous section, Kubernetes Services let you expose a set of Pods via an interface that other applications can call without knowing the details about the single Pod instances. This model provides applications with transparent service discovery and load balancing functionality.

First of all, there are different types of Services depending on which access policy you want to enforce for the application. The default and most common type is called *ClusterIP* and exposes a set of Pods to the cluster. It's what makes it possible for Pods to communicate with each other (for example, Catalog Service and PostgreSQL).

Four pieces of information characterize a ClusterIP Service:

- the label used to match all the Pods that should be targeted and exposed by the Service (`selector`);
- the network protocol used by the Service;
- the port on which the Service is listening (we're going to use port 80 for all our application Services);
- the `targetPort`, the port exposed by the targeted Pods to which Service will forward requests.

Figure 7.10 shows the relationship between a ClusterIP Service and a set of target Pods running applications exposed on port 8080. The name of the Service must be a valid DNS name since it will be used by other Pods as a hostname to access the targeted Pods.

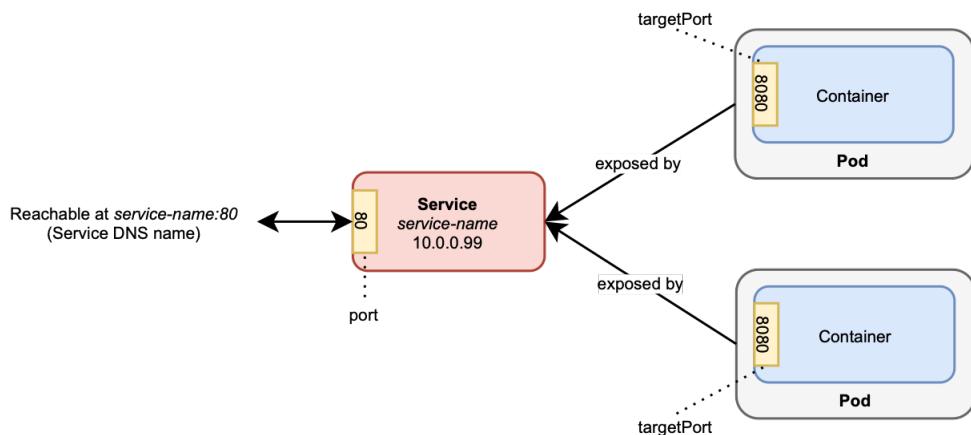


Figure 7.10 A ClusterIP service exposes a set of Pods to the network inside the cluster.

DEFINING A SERVICE MANIFEST WITH YAML

Let's see how to define a manifest for a Service object to expose the Catalog Service application through the DNS name `catalog-service` and port 80. Open the `catalog-service/k8s` folder you created earlier and add a new `service.yml` file.

Listing 7.3 Service manifest for the Catalog Service application

```
apiVersion: v1 ①
kind: Service ②
metadata:
  name: catalog-service ③
  labels:
    app: catalog-service ④
spec:
  type: ClusterIP ⑤
  selector:
    app: catalog-service ⑥
  ports:
    - protocol: TCP ⑦
      port: 80 ⑧
      targetPort: 9001 ⑨
```

- ① The API version for Service objects.
- ② The type of object to create.
- ③ The name of the Service. It must be a valid DNS name.
- ④ A label attached to the Service.
- ⑤ The type of Service.
- ⑥ The label used to match the Pods to target and expose.
- ⑦ The network protocol used by the Service.
- ⑧ The port exposed by the Service.
- ⑨ The port exposed by the Pods targeted by the Service.

CREATING A SERVICE OBJECT FROM A MANIFEST

You can apply a Service manifest as you did for Deployments. Open a Terminal window, navigate to your Catalog Service root folder (`catalog-service`), and run the following command.

```
$ kubectl apply -f k8s/service.yml
```

The command will be processed by the Kubernetes Control Plane, which will create and maintain the Service object in the cluster. You can verify the result with the following command.

```
$ kubectl get svc -l app=catalog-service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
catalog-service	ClusterIP	10.102.29.119	<none>	80/TCP	42s

Since it's of type `ClusterIP`, the Service makes it possible for other Pods within the cluster to communicate with the Catalog Service application, either using its IP address (called cluster IP) or through its name. That will be useful for the applications you'll build in the next chapters, but what about us? How can we expose the application outside the cluster to test it? For now, you'll rely on the port forwarding feature offered by Kubernetes to expose an object (in this case, a Service) to your local machine. You did that already in chapter 2, so the command should look familiar.

```
$ kubectl port-forward service/catalog-service 9001:80
Forwarding from 127.0.0.1:9001 -> 9001
Forwarding from [::1]:9001 -> 9001
```

You can finally call the application from localhost on port 9001, and all requests will be forwarded to the Service object and ultimately to the Catalog Service Pod. Try visiting <http://localhost:9001> from your browser to see the welcome message, or <http://localhost:9001/books> to browse the books available in the catalog.

TIP

The process started by the `kubectl port-forward` command will keep running until you explicitly stop it with `Ctrl+C`. Until then, you need to open another Terminal window if you need to run CLI commands.

Figure 7.11 describes how the communication works between your computer, Catalog Service, and PostgreSQL.

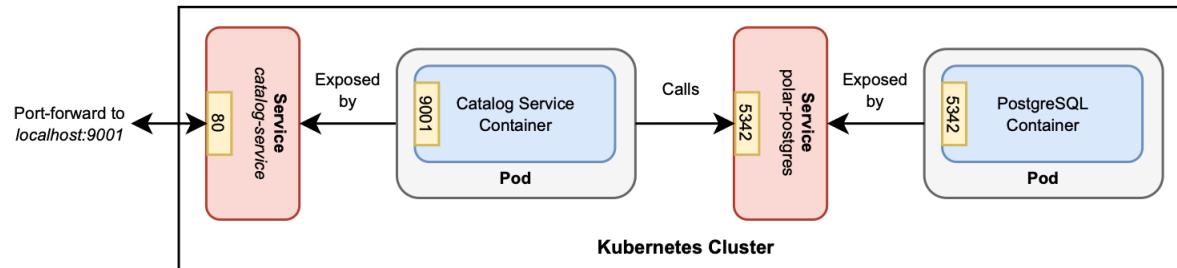


Figure 7.11 The Catalog Service application is exposed to your local machine through port-forwarding. Both Catalog Service and PostgreSQL are exposed to the inside of the cluster through cluster-local hostname, IP address, and port assigned to the Service objects.

So far, we've been working with only one instance of Catalog Service, but we can take advantage of Kubernetes and scale it out. The following section will cover how to scale Spring Boot applications and address aspects such as fast startup and graceful shutdown, essential for cloud native applications.

NOTE Imagine running all those commands every time you change something in your application, and you want to test it locally. It doesn't look very appealing, does it? Don't worry! In section 7.5, I'll show you how to set up a local Kubernetes development workflow to automate all those operations.

7.4 Scalability and disposability

Deploying multiple instances of the same application helps in achieving high availability. When the workload is high, it can be distributed across different replicas. When an instance enters a faulty state and can't process requests anymore, it can be deleted, and a new one created. This continuous and dynamic scaling of application instances requires stateless and disposable applications, as per the 15-Factor methodology.

This section will show you what does it mean for an application to be disposable, how to enable graceful shutdown, and how to scale an application in Kubernetes.

7.4.1 Ensuring disposability: Fast startup

Traditional applications deployed on application servers would take quite some time to start. It's not that rare for them to take several minutes before they're ready to accept connections. On the other hand, cloud native applications should be optimized for starting quickly, taking a few seconds rather than minutes. Spring Boot is already optimized for fast startup, and each new version is shipped with more improvements.

Fast startup is relevant in a cloud environment because applications are disposable and frequently created, destroyed, and scaled. The quicker the startup, the sooner a new application instance is ready to accept connections.

Standard applications like microservices are good with a startup time in the range of a few seconds. On the other hand, serverless applications usually require a faster startup phase in the range of milliseconds rather than seconds. Spring Boot covers both needs, but the second use case might require a bit of additional configuration.

In chapter 16, you'll learn about serverless applications with Spring Cloud Function, and I'll show you how to package them as native images using Spring Native and GraalVM. The result will be an application with almost instant startup time, reduced resource consumption, and reduced image size.

7.4.2 Ensuring disposability: Graceful shutdown

Having applications starting fast is not enough to address the scalability needs. Whenever an application instance is shut down, it must happen gracefully without clients experiencing downtime or error. Gracefully shutting down means the application stops accepting new requests, completes all those still in progress, and closes any open resource like database connections.

All the embedded servers available in Spring Boot support a graceful shutdown mode but in slightly different ways. Tomcat, Jetty, and Netty stop accepting new requests entirely when the shutdown signal is received. On the other hand, Undertow keeps accepting new requests but immediately replies with an HTTP 503 response.

By default, Spring Boot stops the server immediately after receiving a termination signal (`SIGTERM`). You can switch to a graceful mode by configuring the `server.shutdown` property. You can also configure the *grace period*, which is how long the application is allowed to process all the pending requests. After the grace period expires, the application is terminated even if there are still pending requests. By default, the grace period is 30 seconds. You can change it through the `spring.lifecycle.timeout-per-shutdown-phase` property.

Let's configure graceful shutdown for Catalog Service. We could do that via environment variables or have it as a default configuration. We'll go with the second option. Open the `application.yml` located in the `catalog-service/src/main/resources` folder, and update the configuration as follows.

```
server:
  port: 9001
  shutdown: graceful ①
  tomcat:
    connection-timeout: 2s
    keep-alive-timeout: 15s
    threads:
      max: 50
      min-spare: 5

spring:
  application:
    name: catalog-service
  lifecycle:
    timeout-per-shutdown-phase: 15s ②
...
```

- ① Enable graceful shutdown.
- ② Defines a 15s grace period.

Since we modified the application source code, we need to build a new container image and load it into `minikube`. That's not very efficient, isn't it? Later in the chapter, I'll show you a better way. For now, follow the procedure I described earlier to package Catalog Service as a container

image (`./gradlew bootBuildImage`) and load it into the Kubernetes cluster we are using for Polar Bookshop (`minikube image load catalog-service --profile polar`).

After enabling application support for graceful shutdown, you need to update the Deployment manifest accordingly. When a Pod has to be terminated (for example, during a downscaling process or as part of an upgrade), Kubernetes sends a `SIGTERM` signal to it. Spring Boot will intercept that signal and start shutting down gracefully. By default, Kubernetes waits for a grace period of 30 seconds. If the Pod is not terminated after that period, Kubernetes sends a `SIGKILL` signal to force the Pod termination. Since the Spring Boot grace period is lower than the Kubernetes one, the application is in control of when it will terminate.

While sending the `SIGTERM` signal to a Pod, Kubernetes will also inform its own components to stop forwarding requests to the terminating Pod. Since Kubernetes is a distributed system, and the two actions happen in parallel, there is a short time window where the terminating Pod might still receive requests, even if it has already started the graceful shutdown procedure. When that happens, those new requests would be rejected, resulting in errors in the clients. Our premise was to make the shutdown procedure transparent to the clients, so that scenario is unacceptable.

The recommended solution is to delay sending the `SIGTERM` signal to the Pod so that Kubernetes has enough time to spread the news across the cluster. By doing so, when the Pod starts the graceful shutdown procedure, all Kubernetes components already know not to send new requests to it. Technically, the delay can be configured through a `preStop` hook. Let's see how to update the Deployment manifest for Catalog Service to support a transparent and graceful shutdown.

Open the `deployment.yaml` file located in `catalog-service/k8s` and add a `preStop` hook to delay the `SIGTERM` signal by 5 seconds.

Listing 7.4 Configure a delay in Kubernetes before starting the shutdown procedure (deployment.yml)

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  labels:
    app: catalog-service
spec:
  ...
  template:
    metadata:
      labels:
        app: catalog-service
    spec:
      containers:
        - name: catalog-service
          image: catalog-service
          imagePullPolicy: IfNotPresent
          lifecycle:
            preStop: ①
              exec:
                command: [ "sh", "-c", "sleep 5" ]
...

```

- ① Makes Kubernetes wait 5 seconds before sending the SIGTERM signal to the Pod.

Finally, apply the updated version of the Deployment object with `kubectl apply -f k8s/deployment.yml`. Kubernetes will reconcile the new desired state and replace the existing Pod with a new one for which graceful shutdown is fully configured.

NOTE	When a Pod contains multiple containers, the SIGTERM signal is sent to all of them in parallel. Kubernetes will wait up to 30 seconds. If any of the containers in the Pod are not terminated yet, it will shut it down forcefully.
-------------	--

Having configured the graceful shutdown behavior for Catalog Service, you can go ahead and see how to scale it in a Kubernetes cluster.

7.4.3 Scaling Spring Boot applications

Scalability is one of the main properties of a cloud native application, as you learned in chapter 1. To be scalable, applications should be disposable and stateless, as per the 15-Factor methodology.

You handled disposability in the previous section, and Catalog Service is already a stateless application. It has no state but relies on a stateful service (the PostgreSQL database) to permanently store the data about books. Even if an application instance is shut down, the data

will survive. We scale applications in and out, and if they weren't stateless, we would lose the state every time an instance is shut down. The general idea is to keep the applications stateless and rely on data services for storing the state, just like we did for Catalog Service.

In Kubernetes, replication is handled at the Pod level by a ReplicaSet object. As you saw earlier, Deployment objects are already configured to use ReplicaSets. All you need to do is specify how many replicas you want to be deployed. You can do that in the Deployment manifest.

Open the `deployment.yml` file located in `catalog-service/k8s` and define how many replicas of the Pod running Catalog Service you want. Let's go with two.

Listing 7.5 Configure number of replicas for the Catalog Service Pod (deployment.yml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  labels:
    app: catalog-service
spec:
  replicas: 2 ①
  selector:
    matchLabels:
      app: catalog-service
  ...
  
```

- ① How many Pod replicas should be deployed.

The replication is controlled using labels. In listing 7.5, the configuration instructs Kubernetes to manage all Pods with a label `app=catalog-service` so that there are always two replicas running.

Let's check it out. Open a Terminal window, navigate to the `catalog-service` folder and apply the updated version of the Deployment resource.

```
$ kubectl apply -f k8s/deployment.yml
```

Kubernetes will realize that the actual state (one replica) and the desired state (two replicas) don't match and immediately deploy a new replica of Catalog Service. You can verify the result with the following command. Looking at the "age" column, you can tell which Pod is the one that has just been deployed to achieve a state with two replicas.

```
$ kubectl get pods -l app=catalog-service
NAME                  READY   STATUS    RESTARTS   AGE
catalog-service-68bc5659b8-fkpcv  1/1     Running   0          2s
catalog-service-68bc5659b8-kmwm5  1/1     Running   0          3m94s
```

What happens if one of them terminates? Let's find out. Pick one of the two Pod replicas and copy its name. For example, I might use the Pod named `catalog-service-68bc5659b8-kmwm5`

- . Then, from a Terminal window, delete the Pod with the following command.

```
$ kubectl delete pod <pod-name>
```

The Deployment manifest declares two replicas as the desired state. Since there is only one now, Kubernetes will immediately step up to ensure the actual state and the desired state are aligned. If you inspect the Pods again with `kubectl get pods -l app=catalog-service`, you will still see two Pods, but one of them has just been created to replace the deleted Pod. You can identify it by checking its "age".

```
$ kubectl get pods -l app=catalog-service
```

NAME	READY	STATUS	RESTARTS	AGE
catalog-service-68bc5659b8-fkpcv	1/1	Running	0	42s
catalog-service-68bc5659b8-wqchr	1/1	Running	0	3s

Under the hood, a ReplicaSet object keeps checking the number of replicas deployed and ensures they are always in the desired state. That's the basic functionality on top of which you can configure an autoscaler for dynamically increasing or decreasing the number of Pods depending on the workload and without having to update the manifest every time.

Before moving to the next section, make sure to change the number of replicas back to one and clean up your cluster by removing all the resources you have created so far. First, open a Terminal window, navigate to the `catalog-service` folder where you defined the Kubernetes manifests, and delete all the objects created for Catalog Service.

```
$ kubectl delete -f k8s
```

Finally, go to your `polar-deployment` repository, navigate to the `kubernetes/platform/development` folder, and delete the PostgreSQL installation.

```
$ kubectl delete -f services
```

7.5 Local Kubernetes development with Tilt

In the previous sections, you learned the basic Kubernetes concepts and worked with the fundamental objects used to deploy applications to a cluster: Pods, ReplicaSets, Deployments, and Services. After defining the Deployment and Service manifests, you probably don't want to keep rebuilding container images manually and using the `kubectl` client to update the Pods whenever you make any change. Luckily for you, you don't have to.

This section will show you how to set up a local Kubernetes development workflow to automate steps like building images and applying manifests to a Kubernetes cluster. It's part of implementing the *inner development loop* of working with a Kubernetes platform. Tilt takes care

of many infrastructural concerns and lets you focus more on the business logic of your applications. I'll also introduce Octant to visualize and manage your Kubernetes objects through a convenient GUI.

7.5.1 Inner development loop with Tilt

Tilt (<https://tilt.dev>) aims at providing a good developer experience when working on Kubernetes. It's an open-source tool that offers features to build, deploy, and manage containerized workloads in your local environment. We'll use some of its basic features to automate a development workflow for a specific application, but Tilt can also help you orchestrate the deployment of multiple applications and services in a centralized way. You can find information about how to install it in section A.4 of Appendix A.

The goal is to design a workflow that will automate the following steps for you:

- packaging a Spring Boot application as a container image using Cloud Native Buildpacks;
- uploading the image to a Kubernetes cluster (in our case, the one created with *minikube*);
- applying all the Kubernetes objects declared in the YAML manifests;
- enabling the port-forward functionality to access applications from your local computer;
- giving you easy access to the logs from the applications running on the cluster.

Before configuring Tilt, make sure you have a PostgreSQL instance up and running in your local Kubernetes cluster. Open a Terminal window, navigate to the `kubernetes/platform/development` folder in your `polar-deployment` repository, and run the following command to deploy PostgreSQL.

```
$ kubectl apply -f services
```

Let's now see how to configure Tilt to establish that automated development workflow.

Tilt can be configured via a `Tiltfile`, an extensible configuration file written in Starlark (a simplified Python dialect). Go to your Catalog Service project (`catalog-service`) and create a file named `Tiltfile` (no extension) in the root folder. The file will contain three main configurations:

- how to build a container image (Cloud Native Buildpacks);
- how to deploy the application (Kubernetes YAML manifests);
- how to access the application (port forwarding).

Listing 7.6 Tilt configuration for Catalog Service (Tiltfile)

```
# Build
custom_build(
    # Name of the container image
    ref = 'catalog-service',
    # Command to build the container image
    command = './gradlew bootBuildImage --imageName $EXPECTED_REF',
    # Files to watch that trigger a new build
    deps = ['build.gradle', 'src']
)

# Deploy
k8s_yaml(['k8s/deployment.yml', 'k8s/service.yml'])

# Manage
k8s_resource('catalog-service', port_forwards=['9001'])
```

The Tiltfile configures Tilt to use the same approach we used throughout the chapter for building, loading, deploying, and publishing applications on the local Kubernetes cluster. The main difference? It's all automated now! Let's give it a try.

Open a Terminal window, navigate to the root folder of your Catalog Service project, and run the following command to start Tilt.

```
$ tilt up
Tilt started on http://localhost:10350/
```

The process started by the `tilt up` command will keep running until you explicitly stop it with `Ctrl+C`. One of the useful features provided by Tilt is a convenient GUI where you can keep track of the services managed by Tilt, check application logs, and trigger updates manually. Go to the URL where Tilt started its services (by default, it should be <http://localhost:10350>) and monitor the process that Tilt follows to build and deploy Catalog Service (figure 7.12). The first time it can take one or two minutes because the Buildpacks libraries need to be downloaded. The subsequent times it will be much faster.

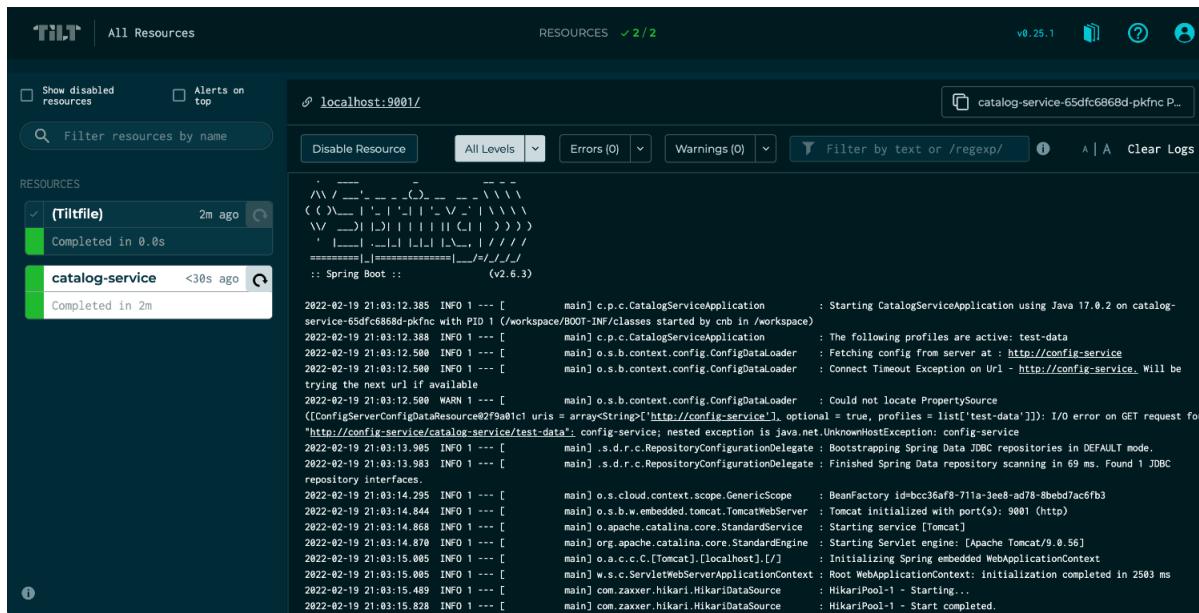


Figure 7.12 Tilt provides a convenient GUI where you can monitor and manage applications.

Besides building and deploying the application, Tilt has also activated a port forwarding to your local machine on port 9001. Go ahead and verify that the application is working correctly.

```
$ http :9001/books
```

Tilt will keep the application in sync with the source code. Whenever you make any change to the application, Tilt will trigger an *update* operation to build and deploy a new container image. All of that happens automatically and continuously.

NOTE

Rebuilding the whole container image every time you change something in your code is not very efficient. You can configure Tilt to synchronize only the changed files and upload them into the current image. To achieve that, you can rely on the features offered by Spring Boot DevTools (<https://mng.bz/nY8v>) and Paketo Buildpacks (<https://mng.bz/v05x>).

When you're done testing the application, stop the Tilt process in the Catalog Service project and run the following command to undeploy the application.

```
$ tilt down
```

7.5.2 Visualizing your Kubernetes workloads with Octant

When you start deploying multiple applications to a Kubernetes cluster, it can become challenging to manage all the related Kubernetes objects or investigate failures when they happen. There are different solutions for visualizing and managing Kubernetes workloads. This section will cover Octant (<https://octant.dev>), an "open source developer-centric web interface for Kubernetes that lets you inspect a Kubernetes cluster and its applications". You can find information about how to install it in section A.4 of Appendix A.

I expect you still have the local Kubernetes cluster you used in the previous section up and running and PostgreSQL deployed. You can also deploy Catalog Service by going into the project root folder and running `tilt up`. Then, open a new Terminal window and run the following command.

```
$ octant
```

The command will open the Octant Dashboard in your browser (usually at <http://localhost:7777>). Figure 7.13 shows the Dashboard. The *Overview* page provides a picture of all the Kubernetes objects running in the cluster. If you followed along, at this point, you should have PostgreSQL and Catalog Service running in the cluster.

The screenshot shows the Octant dashboard with the following details:

- Deployments:**

Name	Labels	Status	Age	Containers	Selector
catalog-service	app:catalog-service, app.kubernetes.io/managed-by:tilt	1/1	2m	catalog-service	app:catalog-service
- Pods:**

Name	Labels	Ready	Phase	Status	Restarts	Node	Age
catalog-service-65dfc6868d-pkfc	app:catalog-s..., app.kubernetes.io/... (with a warning icon)	1/1	Running	Running	0	polar	2m
- ReplicaSets:**

Name	Labels	Status	Age	Containers	Selector
catalog-service-65dfc6868d	app:catalog-service, app.kubernetes.io/manage...	1/1	2m	catalog-service	app:catalog-service

Figure 7.13 Octant offers a web interface for inspecting a Kubernetes cluster and its workloads.

From the *Overview* page, you can expand the objects to get more details. For example, if you click the item corresponding to the Catalog Service Pod, you get access to information about the object, as shown in figure 7.14. You can also perform several operations like enabling port forwarding, reading the logs, modifying the Pod manifest, and investigating failures.

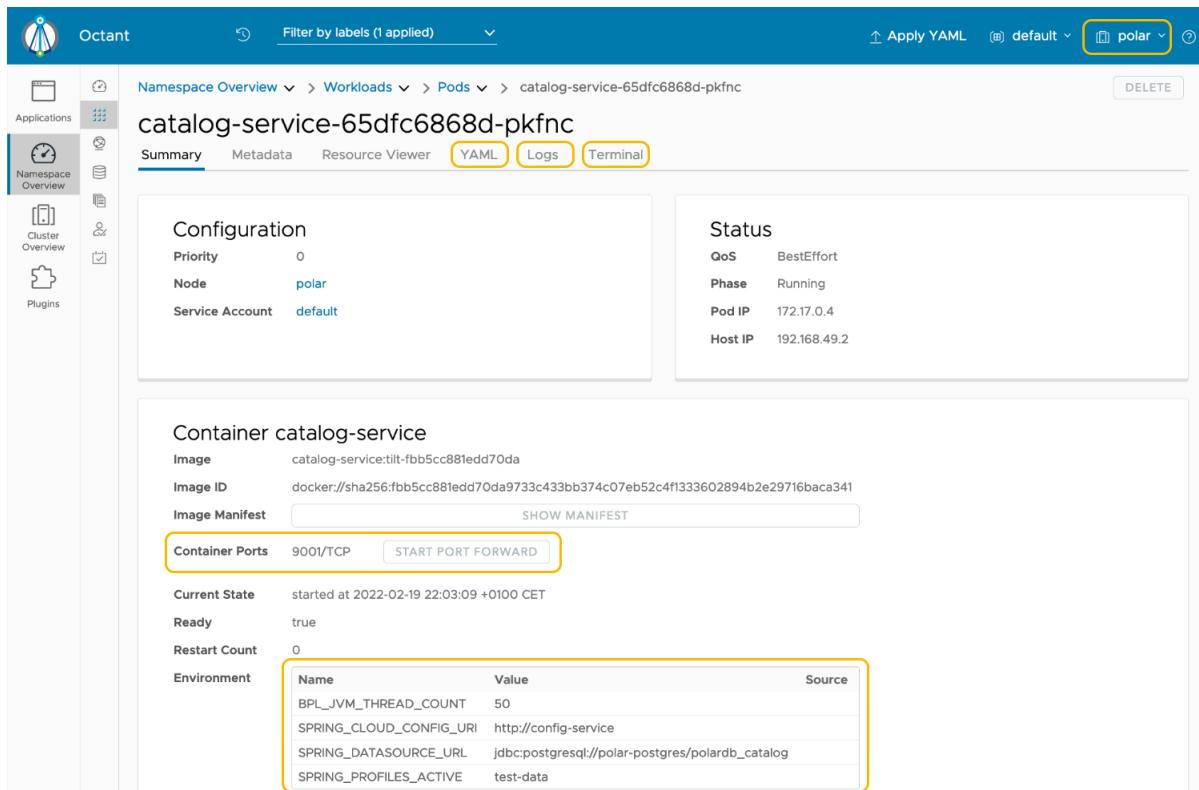


Figure 7.14 Octant lets you access Pod information easily, check their logs, and enable a port forward.

Take your time exploring the many features provided by Octant. It's a convenient tool that you can use to inspect and troubleshoot a local Kubernetes cluster or a remote one. We'll use Octant also to examine the remote production cluster where we'll deploy the Polar Bookshop application. For now, close Octant by stopping its process with `Ctrl+C`.

When you're done, you can stop the Tilt process in the Catalog Service project and run `tilt down` to undeploy the application. Then, go to your `polar-deployment` repository, navigate to the `kubernetes/platform/development` folder, and delete the PostgreSQL installation with `kubectl delete -f services`. Finally, stop the cluster as follows.

```
$ minikube stop --profile polar
```

7.6 Deployment pipeline: Validate Kubernetes manifests

Chapter 3 introduced the concept of deployment pipeline and its importance in the continuous delivery approach for delivering software quickly, reliably, and safely. So far, we have automated the first part of a deployment pipeline: the commit stage. After a developer commits new code to the mainline, this stage goes through build, unit tests, integration tests, static code analysis, and packaging. At the end of this stage, an executable application artifact is published to an artifact repository. That is a *release candidate*.

In this chapter, you learned how to deploy Spring Boot applications on Kubernetes using a declarative approach based on *resource manifests*. They are fundamental to a successful deployment of the release candidate on Kubernetes. Therefore, we should guarantee their correctness. This section will show you how to validate Kubernetes manifests as part of the commit stage.

7.6.1 Validating Kubernetes manifests in the commit stage

Throughout this chapter, we worked with resource manifests for creating Deployments and Services in a Kubernetes cluster.

SIDE BAR

DEFINITION

A manifest is a specification of a Kubernetes API object in JSON or YAML format. It specifies the desired state of an object that Kubernetes will maintain when you apply the manifest. (<https://kubernetes.io/docs>)

Since a manifest specifies the desired state of an object, we should ensure that our specification complies with the API exposed by Kubernetes. It's a good idea to automate this validation in the commit stage of a deployment pipeline to get fast feedback in case of errors (rather than waiting until the acceptance stage, where we need to use those manifests to deploy the application in a Kubernetes cluster). Figure 7.15 illustrates the main steps of the commit stage after including the Kubernetes manifests validation.

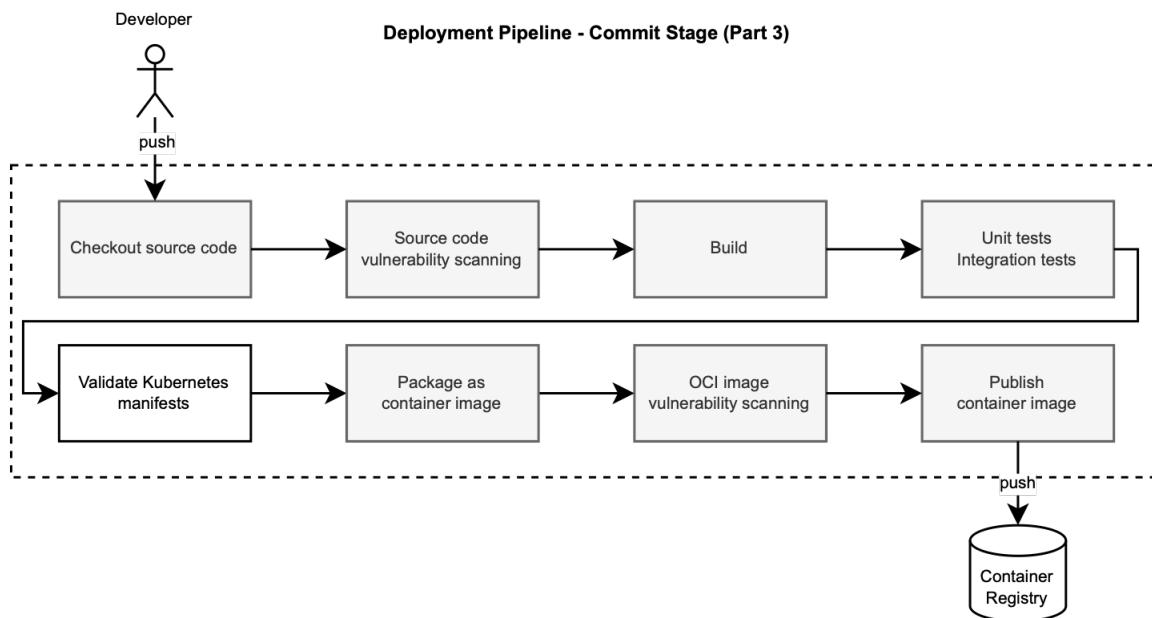


Figure 7.15 When Kubernetes manifests are included in the application repository, a new step in the commit stage is included to validate them.

There are several ways of validating Kubernetes manifests against the Kubernetes API. We'll use

Kubeval (<https://www.kubeval.com>), an open-source tool. You can find information about how to install it in section A.4 of Appendix A.

Let's see how it works. Open a Terminal window and navigate to the root folder of your Catalog Service project (`catalog-service`). Then, use `kubeval` to validate the Kubernetes manifests within the `k8s` directory (`-d k8s`). The `--strict` flag disallows adding additional properties not defined in the object schema.

```
$ kubeval --strict -d k8s

PASS - k8s/deployment.yml contains a valid Deployment (catalog-service)
PASS - k8s/service.yml contains a valid Service (catalog-service)
```

In the next section, you'll see how to use Kubeval in the commit stage workflow we implemented with GitHub Actions.

7.6.2 Automating Kubernetes manifests validation with GitHub Actions

GitHub Actions is the workflow engine we used to implement the commit stage for the Catalog Service deployment pipeline. Let's extend it to include the Kubernetes manifest validation step as illustrated in figure [7.15](#).

Go to your Catalog Service project (`catalog-service`) and open the `commit-stage.yml` file within the `.github/workflows` folder. For implementing the validation step, we'll rely on an action built by Stefan Prodan. He's the maintainer of FluxCD, a CNCF-incubating project providing a continuous deployment solution on Kubernetes based on the GitOps principles. The action lets you install specific versions of useful Kubernetes-related tools. We'll configure the action to install `kubectl` and Kubeval.

Listing 7.7 Add a step to validate the Kubernetes manifests for Catalog Service (commit-stage.yml)

```
name: Commit Stage
on: push
...
jobs:
  build:
    name: Build and Test
    ...
    steps:
      ...
      - name: Validate Kubernetes manifests
        uses: stefanprodan/kube-tools@v1 ①
        with:
          kubectl: 1.23.4 ②
          kubeval: 0.16.1 ③
          command: |
            kubeval --strict -d k8s ④
  package:
    ...

```

- ① An action capable of installing useful tools to work with Kubernetes.
- ② Includes the Kubernetes CLI in the installation.
- ③ Includes Kubeval in the installation.
- ④ Uses Kubeval to validate the Kubernetes manifests in the "k8s" folder.

After updating the `commit-stage.yml` file with the additional validation step, you can commit and push your changes to your `catalog-service` repository on GitHub and verify the commit stage workflow completes successfully, meaning what you included manifests compliant with the Kubernetes API.

SIDE BAR

Polar Labs

Feel free to apply what you learned in this chapter to Config Service and prepare the application for deployment.

1. Configure graceful shutdown and grace period for the application.
2. Write the Deployment and Service manifests for deploying Config Service to a Kubernetes cluster.
3. Update the commit stage of the deployment pipeline for Config Service to validate the Kubernetes manifests.
4. Configure the Catalog Service Deployment with the Config Service URL via the `SPRING_CLOUD_CONFIG_URI` environment variable, relying on the Kubernetes native service discovery feature.
5. Configure Tilt to automate the Config Service deployment to your local Kubernetes cluster bootstrapped with minikube.

When you're done, try deploying all the components of the Polar Bookshop system we've built so far and check their status in Octant. You can refer to the Chapter08/08-end folder in the code repository accompanying the book for checking the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

Congratulations!

7.7 Summary

- Docker works fine when running single-instance containers on a single machine. When your system needs properties like scalability and resilience, you can use Kubernetes.
- Kubernetes provides all the features for scaling containers across a cluster of machines, ensuring resilience both when a container fails and when a machine goes down.
- Pods are the smallest deployable units in Kubernetes.
- Rather than creating Pods directly, you use a Deployment object to declare the desired state for your applications, and Kubernetes ensures it matches the actual state. That includes having the desired number of replicas up and running at any time.
- The cloud is a dynamic environment, and the topology keeps changing. Service discovery and load balancing let you dynamically establish interactions between services, managed either client-side (for example, using Spring Cloud Netflix Eureka) or server-side (for example, using Kubernetes).
- Kubernetes provides a native service discovery and load balancing feature that you can use through the Service objects.
- Each Service name can be used as a DNS name. Kubernetes will resolve the name to the Service IP address and, ultimately, forward the request to one of the instances available.
- You can deploy Spring Boot applications to a Kubernetes cluster by defining two YAML manifests: one for the Deployment object and one for the Service object.
- The `kubectl` client lets you create objects from a file with the command `kubectl apply -f <your-file.yml>`.
- Cloud native applications should be disposable (fast startup and graceful shutdown) and stateless (rely on data services for storing the state).
- Graceful shutdown is supported both by Spring Boot and Kubernetes and is an essential aspect for scalable applications.
- Kubernetes uses ReplicaSet controllers to replicate your application Pods and keep them running.
- Tilt is a tool that automates your local development workflow with Kubernetes: you work on the application while Tilt takes care of building the image, deploying it to your local Kubernetes cluster, and keeping it up-to-date whenever you change something in the code.
- You can start Tilt for your project with `tilt up`.
- The Octant dashboard lets you visualize your Kubernetes workloads.
- Octant is a convenient tool that you can use not only for inspecting and troubleshooting a local Kubernetes cluster but also for a remote one.
- Kubeval is a convenient tool you can use to validate Kubernetes manifests. It's particularly useful when included in your deployment pipeline.



Reactive Spring: Resilience and scalability

This chapter covers

- Understanding reactive programming with Reactor and Spring
- Building reactive servers with Spring WebFlux and Spring Data R2DBC
- Building reactive clients with WebClient
- Improving resilience for applications with Reactor
- Testing reactive applications with Spring and Testcontainers

Polarsophia, the organization behind the Polar Bookshop business, is very happy with the progress of its new software product. Its mission is to spread knowledge and awareness about the North Pole and the Arctic, and making its book catalog available worldwide is an essential part of it.

The Catalog Service application you built so far is a good starting point. It fulfills the requirements of browsing and managing books, and it does that while following cloud native patterns and practices. It's self-contained and stateless. It uses a database as a backing service to store the state. It can be configured externally through environment variables or a configuration server. It respects environment parity. It's verified through the automated execution of tests as part of a deployment pipeline, following continuous delivery practices. For maximum portability, it's also containerized and can be deployed to a Kubernetes cluster using native functionality like service discovery, load balancing, and replication.

Another essential feature of the system is the possibility of purchasing books. In this chapter, you will start working on the Order Service application. This new component will interact not only with a database but also with Catalog Service. When you have applications extensively relying

on IO operations such as database calls or interactions with other services like HTTP request/response communications, the thread-per-request model used in Catalog Service begins to expose its technical limits.

In the thread-per-request model, each request is bound to a thread allocated to its processing exclusively. If database or service calls are part of the processing, the thread will send out a request and then block waiting for a response. During idle time, the resources allocated for that thread are wasted since they cannot be used for anything else. The reactive programming paradigm solves this problem and improves scalability, resilience, and cost-effectiveness for all IO-bound applications.

Reactive applications operate asynchronously and in a non-blocking way, meaning that computational resources are used more effectively. That's a huge advantage in the cloud since you pay for what you use. When a thread sends a call to a backing service, it will not wait idle, but it will move on to executing other operations. This aspect eliminates the linear dependency between the number of threads and the number of concurrent requests, leading to more scalable applications. With the same amount of computational resources, reactive applications can serve more users than their non-reactive counterparts.

Cloud native applications are highly distributed systems deployed in a dynamic environment where change is a constant and failures can and will happen. What if the service is not available? What happens if the request gets lost on its way to the target service? What if the response gets lost on its way back to the caller? Can we guarantee high availability in this context?

Resilience is one of the goals for moving to the cloud and one of the properties characterizing cloud native applications. Our systems should be resilient to failures and stable enough to ensure a certain service level to their users. The integration points between services over a network are among the most critical areas to work on to achieve a stable and resilient system for production. It's so important that Michael T. Nygard spends a large part of his book "Release It! Design and Deploy Production-Ready Software" on the subject.

This chapter will focus on building resilient, scalable, and efficient applications for the cloud using the reactive paradigm. First, I'll introduce the event loop model and the main features of Reactive Streams, Project Reactor, and the Spring reactive stack. Then, you'll build a reactive Order Service application using Spring WebFlux and Spring Data R2DBC.

Order Service will interact with Catalog Service to check the availability of books and their details, so you'll see how to implement a reactive REST client using Spring WebClient. The integration point between the two services is a critical area that needs extra care to reach robustness and fault tolerance. Relying on the Reactor project, you'll adopt stability patterns like retries, timeouts, and failovers. Finally, you'll write autotests to verify the behavior of a reactive application using Spring Boot and Testcontainers.

NOTE

The source code for the examples in this chapter is available in the Chapter08/08-begin and Chapter08/08-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

8.1 Asynchronous and non-blocking architectures with Reactor and Spring

The *Reactive Manifesto* (<https://www.reactivemanifesto.org>) describes a reactive system as responsive, resilient, elastic, and message-driven. Its mission to build loosely-coupled, scalable, resilient, and cost-effective applications is fully compatible with our definition of cloud native. The new part is achieving that goal by using an asynchronous and non-blocking communication paradigm based on message-passing.

Before diving into building reactive applications in Spring, this section will explain the basics of reactive programming, why it matters for cloud native applications, and how it differs from imperative programming. I'll introduce the event loop model to overcome the thread-per-request model shortcomings. Then, you'll learn the essential concepts of the Reactive Streams specification implemented by Project Reactor and the Spring reactive stack.

8.1.1 From thread-per-request to event loop

As you saw in chapter 3, non-reactive applications allocate a thread per request. Until a response is returned, the thread will not be used for anything. That is the **thread-per-request** model. When the request handling involves intensive operations like I/O, the thread will block until they are completed. For example, if a database read is required, the thread will wait until data are returned from the database. During the waiting time, the resources allocated to the handling thread are not used efficiently. If you want to support more concurrent users, you'll have to ensure you have enough threads and resources available. In the end, this paradigm sets constraints on the application scalability and doesn't use computational resources in the most efficient way possible. Figure 8.1 shows how it works.

The Thread-per-Request Model

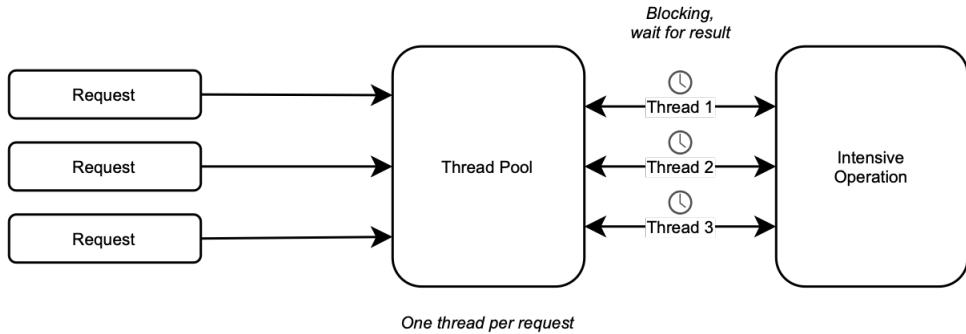


Figure 8.1 In the thread-per-request model, each request is handled by a thread dedicated exclusively to its handling.

Reactive applications are more scalable and efficient by design. Handling requests in a reactive application doesn't involve allocating a given thread exclusively, but it's fulfilled asynchronously based on events. For example, if a database read is required, the thread handling that part of the flow will not wait until data is returned from the database. Instead, a callback is registered, and whenever the information is ready, a notification is sent, and one of the available threads will execute the callback. During that time, the thread can be used to process other requests rather than waiting idle.

This paradigm, called **event loop**, doesn't set hard constraints on the application scalability. It actually makes it easier to scale since an increase in the number of concurrent requests does not strictly depend on the number of threads. As a matter of fact, a default configuration for reactive applications in Spring is to use only one thread per CPU core. With the non-blocking I/O capability and a communication paradigm based on events, reactive applications allow more efficient utilization of computational resources. Figure [8.2](#) shows how it works.

The Event Loop Model

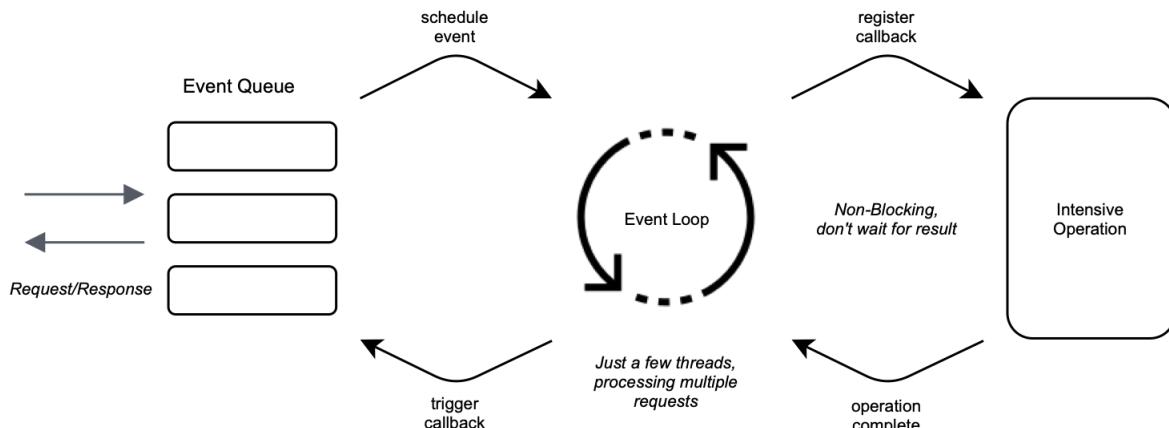


Figure 8.2 In the event loop model, requests are handled by threads that don't block while waiting for an intensive operation, allowing them to process other requests in the meantime.

I wanted to briefly mention the difference between those two paradigms because it helps understand the reasoning behind reactive programming. However, you don't need to know the details of their inner mechanics since we won't have to work at such a low level or implement an event loop. Instead, we're going to rely on convenient higher-level abstractions that let us focus on the business logic of our applications rather than spending time handling the processing at the thread level.

Scale and cost optimization are two critical goals for moving to the cloud, so the reactive paradigm perfectly fits cloud native applications. Scaling applications to support a workload increase is less demanding. Using resources more efficiently, you can save money on the computational resources offered by a cloud provider. Another goal for moving to the cloud is resilience, and reactive applications also help with that.

One of the essential features of reactive applications is to provide non-blocking backpressure (also called *control flow*). It means that consumers can control the amount of data received to lower the risk of producers sending more data than the consumer can handle and causing a DoS attack, slowing the application, cascading the failure or even leading to a total crash.

The reactive paradigm is a solution to the problem of blocking I/O operations that require more threads to handle high concurrency and may lead to slow or entirely unresponsive applications. Sometimes the paradigm is mistaken as a way to increase the speed of an application. Reactive is about improving scalability and resilience, not speed.

With great powers come great troubles, though. Going reactive is an excellent choice when you expect high traffic and concurrency with less computational resources or in streaming scenarios.

However, you should also be aware of the additional complexity introduced by such a paradigm. Besides requiring a mindset shift to think in an event-driven way, reactive applications are more challenging to debug and troubleshoot because of the asynchronous I/O. Before rushing to rewrite all your applications to make them reactive, think twice if that's necessary and consider both benefits and drawbacks.

Reactive programming is not a new concept. It's been used for years. The reason for the recent success of such a paradigm in the Java ecosystem is due to the Reactive Streams specification and its implementations like Project Reactor, RxJava, or Vert.x that provided developers with convenient and high-level interfaces to build asynchronous and non-blocking applications without dealing with the underlying details of how to design a message-driven flow. The following section will introduce Project Reactor, the reactive framework used by Spring.

8.1.2 Project Reactor: Reactive streams with Mono and Flux

Reactive Spring is based on Project Reactor, a framework for building asynchronous, non-blocking applications on the JVM. Reactor is an implementation of the *Reactive Streams* specification aiming at providing "a standard for asynchronous stream processing with non-blocking backpressure" (<https://www.reactive-streams.org>).

Conceptually, reactive streams resemble the Java Stream API in the way of building data pipelines. One of the key differences is that a Java stream is pull-based: consumers process data in an imperative and synchronous fashion. Instead, reactive streams are push-based: consumers are notified by the producers when new data is available, so the processing happens asynchronously.

Reactive streams work according to a producer/consumer paradigm. Producers are called *publishers*. They produce data that might be eventually available. Reactor provides two central APIs implementing the `Producer<T>` interface for objects of type `<T>` and used to compose asynchronous, observable data streams: `Mono<T>` and `Flux<T>`.

- `Mono<T>` represents a single asynchronous value or empty result (0..1).
- `Flux<T>` represents an asynchronous sequence of zero or more items (0..N).

In a Java stream, you would process objects like `Optional<Customer>` or `Collection<Customer>`. In a reactive stream, you would have `Mono<Customer>` or `Flux<Customer>`. The possible outcomes of a reactive stream are empty result, value, or error. All of them are handled as data. When the publisher returns all the data, we say that the reactive stream has been *completed* successfully.

Consumers are called *subscribers* because they subscribe to a publisher and are notified whenever new data is available. As part of the *subscription*, consumers can also define backpressure by informing the publisher they can process only a certain amount of data at a time. That is a powerful feature that puts consumers in control of how much data is received, preventing them from being overwhelmed and becoming unresponsive. Reactive streams are only activated if there's a subscriber.

You can build reactive streams combining data coming from different sources and manipulate it using Reactor's vast collection of *operators* provided. In a Java stream, you can use a fluent API to process data through operators like `map`, `flatMap`, or `filter`, each of which builds a new `Stream` object that keeps the previous step immutable. Similarly, you can build reactive streams using a fluent API and operators to process the data received asynchronously.

Besides the standard operators available to Java streams, you can use more powerful ones to apply backpressure, handle errors, and increase application resilience. For example, you'll see how to use the `retryWhen()` and `timeout()` operators to make the interaction between Order

Service and Catalog Service more robust. Operators perform some actions on a publisher and return a new publisher without modifying the original one, so you can build functional and immutable data streams with ease.

Project Reactor is the foundation of the Spring reactive stack, which lets you implement your business logic in terms of `Mono<T>` and `Flux<T>`. In the next section, you'll learn more about which options you have to build reactive applications with Spring.

8.1.3 Understanding the Spring reactive stack

When you build applications with Spring, you can choose between a servlet stack and a reactive stack. The servlet stack relies on synchronous, blocking I/O and uses the thread-per-request model to handle requests. On the other hand, the reactive stack relies on asynchronous, non-blocking I/O and uses the event loop model to handle requests.

The servlet stack is based on the Servlet API and a Servlet container (e.g., Tomcat). In contrast, the reactive model is based on the Reactive Streams API (implemented by Project Reactor) and either Netty or a Servlet container (version 3.1 as a minimum). Both stacks let you build RESTful applications using either classes annotated as `@RestController` (that you used in chapter 3) or functional endpoints called Router Functions (that you'll learn in chapter 9). The servlet stack uses Spring MVC, while the reactive stack uses Spring WebFlux. Figure 8.3 compares the two stacks (for a broader overview, you can refer to <https://spring.io/reactive>).

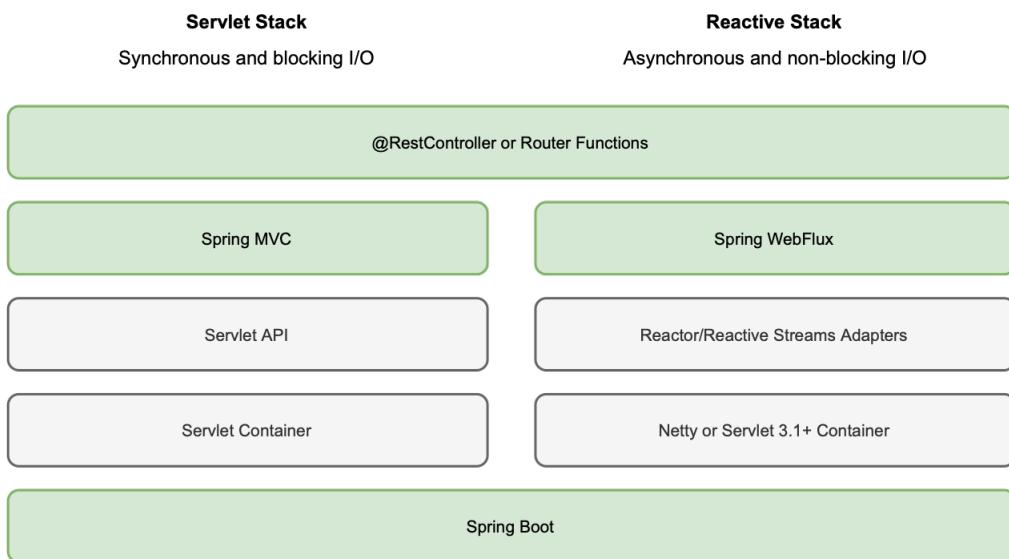


Figure 8.3 The servlet stack is based on the Servlet API and supports synchronous and blocking operations. The reactive stack is based on Project Reactor and supports asynchronous and non-blocking operations.

Tomcat is the default choice for servlet-based applications like Catalog Service. Netty is the preferred choice for reactive applications, providing the best performance.

All the major frameworks in the Spring ecosystem offer both a non-reactive and a reactive option, including Spring Security, Spring Data, and Spring Cloud. Overall, the Spring reactive stack provides a higher-level interface to build reactive applications relying on the familiar Spring projects without being concerned with the underlying implementation of the reactive streams.

8.2 Reactive servers with Spring WebFlux and Spring Data R2DBC

So far, we have worked on Catalog Service, a non-reactive (or *imperative*) application, using Spring MVC and Spring Data JDBC. This section will teach you how to build a reactive web application (Order Service) using Spring WebFlux and Spring Data R2DBC. Order Service will provide functionality for purchasing books. Like Catalog Service, it will expose a REST API and store data in a PostgreSQL database. Unlike Catalog Service, it will use the reactive programming paradigm for improving scalability, resilience, and cost-effectiveness.

You'll see that the principles and patterns you learned in the previous chapters also apply to reactive applications. The main difference is shifting from implementing the business logic in an imperative way to building reactive streams processed asynchronously.

Order Service will also interact with Catalog Service through its REST API to fetch details about books and check their availability. That will be the focus of section 8.3. Figure 8.4 shows the new components of the system.

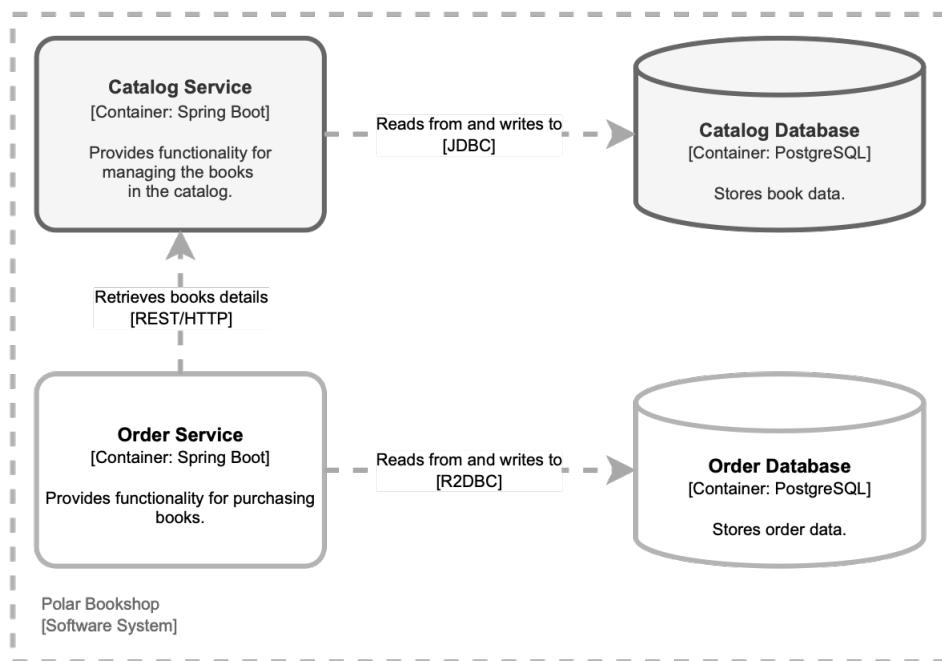


Figure 8.4 The Order Service application exposes an API to submit and retrieve book orders, uses a PostgreSQL database to store data, and communicates with Book Service to fetch book details.

As you learned in chapter 3, we should start with the API first. Order Service will expose a

REST API to retrieve existing book orders and submit new ones. Each order can be related to one book only and up to five copies. The API is described in table [8.1](#).

Table 8.1 Specifications for the REST API that Order Service will expose.

Endpoint	HTTP Method	Req. Body	Status	Res. body	Description
/orders	POST	OrderRequest	200	Order	Submits a new order for a given book in a given quantity.
/orders	GET		200	Order[]	Retrieves all the orders.

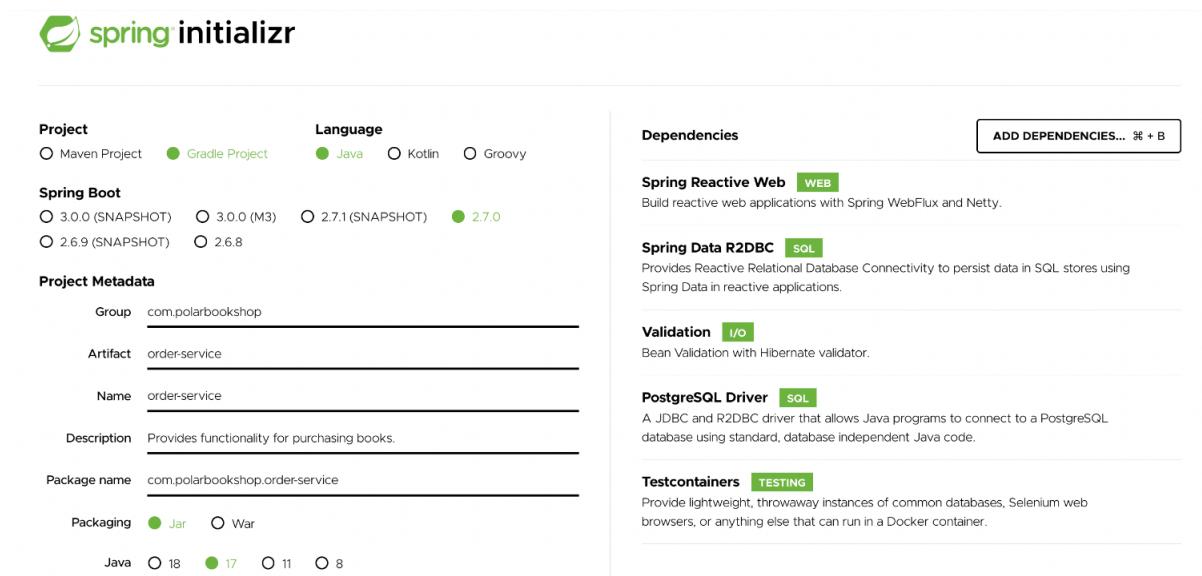
Now, on to the code.

NOTE

If you haven't followed along with the examples implemented in the previous chapters, you can refer to the repository accompanying the book (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and use the project in Chapter 08/08-begin as a starting point.

8.2.1 Bootstrapping a reactive application with Spring Boot

You can initialize the Order Service project from Spring Initializr (<https://start.spring.io>), store the result in a new `order-service` Git repository, and push it to GitHub. The parameters for the initialization are shown in figure [8.5](#).



The screenshot shows the Spring Initializr interface with the following configuration:

- Project:** Maven Project (radio button selected)
- Language:** Java (radio button selected)
- Spring Boot:** 2.7.0 (radio button selected)
- Project Metadata:**
 - Group: com.polarbookshop
 - Artifact: order-service
 - Name: order-service
 - Description: Provides functionality for purchasing books.
 - Package name: com.polarbookshop.order-service
 - Packaging: Jar (radio button selected)
- Dependencies:**
 - Spring Reactive Web** (WEB): Build reactive web applications with Spring WebFlux and Netty.
 - Spring Data R2DBC** (SQL): Provides Reactive Relational Database Connectivity to persist data in SQL stores using Spring Data in reactive applications.
 - Validation** (I/O): Bean Validation with Hibernate validator.
 - PostgreSQL Driver** (SQL): A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
 - Testcontainers** (TESTING): Provide lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.
- Java:** 17 (radio button selected)

Figure 8.5 The parameters for initializing the Order Service project from Spring Initializr.

TIP

In the `begin` folder for this chapter, you can find a `curl` command to run in a Terminal window that downloads a zip file containing all the code you need to get started without going through the manual generation through the Spring Initializr website.

The `dependencies` section of the autogenerated `build.gradle` file looks as follows.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-r2dbc'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'

    runtimeOnly 'org.postgresql:r2dbc-postgresql'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
    testImplementation 'org.testcontainers:junit-jupiter'
    testImplementation 'org.testcontainers:postgresql'
    testImplementation 'org.testcontainers:r2dbc'
}
```

- **Spring Reactive Web** (`org.springframework.boot:spring-boot-starter-webflux`) provides the necessary libraries for building reactive web applications with Spring WebFlux and includes Netty as the default embedded server.
- **Spring Data R2DBC** (`org.springframework.boot:spring-boot-starter-data-r2dbc`) provides the necessary libraries to persist data in relational databases with R2DBC using Spring Data in reactive applications.
- **Validation** (`org.springframework.boot:spring-boot-starter-validation`) provides the necessary libraries for object validation using the Java Bean Validation API.
- **PostgreSQL** (`org.postgresql:r2dbc-postgresql`) provides an R2DBC driver that allows the application to connect to a PostgreSQL database reactively.
- **Spring Boot Test** (`org.springframework.boot:spring-boot-starter-test`) provides several libraries and utilities to test applications, including Spring Test, JUnit, AssertJ, and Mockito. It's automatically included in every Spring Boot project.
- **Reactor Test** (`io.projectreactor:reactor-test`) provides utilities to test reactive applications based on Project Reactor. It's automatically included in every reactive Spring Boot project.
- **Testcontainers** (`org.testcontainers:junit-jupiter, org.testcontainers:postgresql, org.testcontainers:r2dbc`) provides the necessary libraries for testing applications using lightweight Docker containers. In particular, it provides test containers for PostgreSQL supporting the R2DBC driver.

The default and recommended embedded server for reactive applications in Spring Boot is Reactor Netty, built on top of Netty to provide reactive capabilities within Project Reactor. You can configure it either through properties or by defining a `WebServerFactoryCustomizer<NettyReactiveWebServerFactory>` component. Let's use the first approach.

First of all, rename the `application.properties` file generated by Spring Initializr to `application.yml` and define the application name using the `spring.application.name` property. Like you did for Tomcat, you can define the server port through the `server.port` property, configure the graceful shutdown through `server.shutdown`, and the grace period with `spring.lifecycle.timeout-per-shutdown-phase`. Using the specific Netty properties, you can further customize the server behavior. For example, you can define connection and idle timeouts for Netty with the `server.netty.connection-timeout` and `server.netty.idle-timeout` properties.

Listing 8.1 Configure Netty server and graceful shutdown (application.yml)

```
server:
  port: 9002    ①
  shutdown: graceful  ②
  netty:
    connection-timeout: 2s  ③
    idle-timeout: 15s  ④

spring:
  application:
    name: order-service
  lifecycle:
    timeout-per-shutdown-phase: 15s  ⑤
```

- ① The port where the server will accept connections.
- ② Enables graceful shutdown.
- ③ How long to wait for a TCP connection to be established with the server.
- ④ How long to wait before closing a TCP connection if no data is transferred.
- ⑤ Defines a 15s grace period.

Having the basic setup in place, we can now define the domain entity and its persistence.

8.2.2 Persisting data reactively with Spring Data R2DBC

In chapter 5, you learned that the three main aspects of the interaction between a Spring Boot application and a database are database drivers, entities, and repositories. The same concepts you learned with Spring Data JDBC will also apply to Spring Data R2DBC. Spring Data provides common abstractions and patterns, making it straightforward to navigate the different modules.

The main difference for Order Service is the type of database driver. JDBC is the common driver used by Java applications to communicate with a relational database, but it doesn't support reactive programming. There have been a few attempts to provide reactive access to a relational database. A project that stood out and became widely supported fast was the Reactive Relational Database Connectivity (R2DBC) initiated by Pivotal (now VMware Tanzu). There are R2DBC

drivers available for all the major databases (such as PostgreSQL, MariaDB, MySQL, SQL Server, and Oracle DB) and clients for several projects, including Spring Boot with Spring Data R2DBC and Testcontainers.

This section will guide you through defining a domain entity and a persistence layer for Order Service using Spring Data R2DBC and PostgreSQL. Let's get started.

RUN A POSTGRESQL DATABASE FOR ORDER SERVICE

First of all, we need a database. We'll adopt a *database-per-service* approach to keep applications loosely coupled. After deciding that Catalog Service and Order Service will have a database each, we have two options for the actual storage. We could either use the same database server for both databases or two different ones. For convenience, we'll use the same PostgreSQL server we set up in chapter 5 to host both the `polardb_catalog` database used by Catalog Service and the new `polardb_order` database used by Order Service.

Go to your `polar-deployment` repository, and create a new `docker/postgresql` folder. Then, add a new `init.sql` file in the newly created folder. We'll use it to define the initialization script that PostgreSQL should run during the startup phase.

Listing 8.2 Initialize the PostgreSQL server with two databases (postgresql/init.sql)

```
CREATE DATABASE polardb_catalog;
CREATE DATABASE polardb_order;
```

Then, open the `docker-compose.yml` file and update the PostgreSQL container definition to load the initialization script. Remember to remove the value for the `POSTGRES_DB` environment variable since we'll now delegate the database creation to the script. In the book's source code, you can refer to `Chapter08/08-end/polar-deployment/docker` to check the final result.

Listing 8.3 Initialize the PostgreSQL server from an SQL script (docker-compose.yml)

```
version: "3.8"
services:
  ...
  polar-postgres:
    image: "postgres:14.3"
    container_name: "polar-postgres"
    ports:
      - 5432:5432
    environment: ①
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
    volumes: ②
      - ./postgresql/init.sql:/docker-entrypoint-initdb.d/init.sql
```

- ① No value is defined for "POSTGRES_DB" anymore.
- ② Mounts the initialization SQL script to the container as a volume.

Finally, start a new PostgreSQL container based on the new configuration. Open a Terminal window, navigate to the folder where you defined the `docker-compose.yml` file, and run the following command.

```
$ docker-compose up -d polar-postgres
```

For the rest of the chapter, I will assume you have the database up and running.

CONNECTING TO A DATABASE WITH R2DBC

Spring Boot lets you configure the integration of a reactive application with a relational database through the `spring.r2dbc` properties. Open the `application.yml` file for the Order Service project and configure the connection with PostgreSQL. Connection pooling is enabled by default, and you can further configure it by defining connection timeout and sizes, just like you did for JDBC in chapter 5. Since it's a reactive application, the connection pool will probably be smaller than when using JDBC. You can tune the values after monitoring your applications running under normal conditions.

Listing 8.4 Configure the database integration over R2DBC (application.yml)

```
spring:
  r2dbc:
    username: user ①
    password: password ②
    url: r2dbc:postgresql://localhost:5432/polardb_order ③
    pool:
      max-create-connection-time: 2s ④
      initial-size: 5 ⑤
      max-size: 10 ⑥
```

- ① A user with privileges to access the given database.
- ② The password for the given user.
- ③ The R2DBC URL to identify with which database you want to establish a connection.
- ④ The maximum time waiting for getting a connection from the pool.
- ⑤ The initial size of the connection pool.
- ⑥ The maximum number of connections kept in the pool.

Now that you connected a reactive Spring Boot application to a PostgreSQL database through an R2DBC driver, you can go ahead and define the data to persist.

DEFINING PERSISTENT ENTITIES

The Order Service application provides functionality to submit and retrieve orders. That's the *domain entity*. Add a new `com.polarbookshop.orderservice.order.domain` package for the business logic and create an `Order` Java record to represent the domain entity, similarly to how you defined `Book` in Catalog Service.

Following the same approach used in chapter 5, use the `@Id` annotation to mark the field representing the primary key in the database and `@Version` to provide a version number essential to handle concurrent updates and use optimistic locking. You can also add the necessary fields to hold audit metadata using the `@CreatedDate` and `@LastModifiedDate` annotations.

The default strategy for mapping an entity to a relational table is to transform the Java object name into lowercase. In this example, Spring Data would try to map the `Order` record to an `order` table. The problem is that `order` is a reserved word in SQL. It's not recommended to use it as the table name because it would require special handling. You can overcome that issue by naming the table `orders` and configuring the object-relational mapping through the `@Table` annotation (from the `org.springframework.data.relational.core.mapping` package).

Listing 8.5 The Order record defines the domain and persistent entity for the application (Order.java)

```
package com.polarbookshop.orderservice.order.domain;

import java.time.Instant;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.annotation.Version;
import org.springframework.data.relational.core.mapping.Table;

@Table("orders") ①
public record Order (

    @Id
    Long id, ②

    String bookIsbn,
    String bookName,
    Double bookPrice,
    Integer quantity,
    OrderStatus status,

    @CreatedDate
    Instant createdDate, ③

    @LastModifiedDate
    Instant lastModifiedDate, ④

    @Version
    int version ⑤
) {
    public static Order of(
        String bookIsbn, String bookName, Double bookPrice,
        Integer quantity, OrderStatus status
    ) {
        return new Order(
            null, bookIsbn, bookName, bookPrice, quantity, status, null, null, 0
        );
    }
}
```

- ① Configures the mapping between the "Order" object and the "orders" table.
- ② The primary key of the entity.
- ③ When the entity has been created.
- ④ When the entity has been modified the last time.
- ⑤ The entity version number.

Orders can go through different phases. If the requested book is available in the catalog, then the order is *accepted*. If not, it's *rejected*. Once the order is accepted, it can be *dispatched*, as you'll see in chapter 10. You can define these three statuses in an `OrderStatus` enum in the `com.polarbookshop.orderservice.order.domain` package.

Listing 8.6 Enum describing the statuses for the orders (OrderStatus.java)

```
package com.polarbookshop.orderservice.order.domain;

public enum OrderStatus {
    ACCEPTED,
    REJECTED,
    DISPATCHED
}
```

The R2DBC auditing functionality can be enabled in a configuration class using the `@EnableR2dbcAuditing` annotation. Create a `DataConfig` class in a new `com.polarbookshop.orderservice.config` package and enable auditing there.

Listing 8.7 Enable R2DBC auditing via annotation configuration (DataConfig.java)

```
package com.polarbookshop.orderservice.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.r2dbc.config.EnableR2dbcAuditing;

@Configuration ①
@EnableR2dbcAuditing ②
public class DataConfig {}
```

- ① Indicates a class as a source of Spring configuration.
- ② Enables R2DBC auditing for persistent entities.

Having defined the data to persist, you can go ahead and explore how to access it.

USING REACTIVE REPOSITORIES

Spring Data provides the *repository* abstraction for all the modules in the project, including R2DBC. The only difference with what you did in chapter 5 is that you're going to use a reactive repository.

In the `com.polarbookshop.orderservice.order.domain` package, create a new `OrderRepository` interface and make it extend `ReactiveCrudRepository`, specifying the type of data handled (`Order`) and the data type of the `@Id`-annotated field (`Long`).

Listing 8.8 Repository interface for accessing orders (OrderRepository.java)

```
package com.polarbookshop.orderservice.order.domain;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;

public interface OrderRepository
    extends ReactiveCrudRepository<Order, Long> {} ①
```

- ① Extends a reactive repository providing CRUD operations, specifies the type of managed entity (`Order`) and its primary key type (`Long`).

The CRUD operations provided by `ReactiveCrudRepository` are enough for the use cases of the Order Service application, so you don't need to add any custom method. However, we're still missing the `orders` table in the database. Let's define it using Flyway.

MANAGING DATABASE SCHEMAS WITH FLYWAY

Spring Data R2DBC supports initializing data sources through `schema.sql` and `data.sql` files, just like Spring Data JDBC. As you learned in chapter 5, that functionality is convenient for demos and experiments, but we prefer managing schemas explicitly for production use-cases.

For Catalog Service, we used Flyway to create and evolve its database schemas. We can do the same for Order Service. However, Flyway doesn't support R2DBC yet, so we need to provide a JDBC driver to communicate with the database. The Flyway migration tasks are only run at application startup and in a single thread. Using a non-reactive communication approach for this one case doesn't impact the overall application scalability and efficiency.

In the `build.gradle` file for your Order Service project, add new dependencies to Flyway, the PostgreSQL JDBC driver, and Spring JDBC. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 8.9 Add dependency for Flyway and JDBC in Order Service (build.gradle)

```
dependencies {
    ...
    runtimeOnly 'org.flywaydb:flyway-core'      ①
    runtimeOnly 'org.postgresql:postgresql'       ②
    runtimeOnly 'org.springframework:spring-jdbc' ③
}
```

- ① Provides functionality to version-control your database through migrations.
- ② Provides a JDBC driver that allows the application to connect to a PostgreSQL database.
- ③ Provides Spring integration with the JDBC API. It's part of the Spring Framework. Not to be confused with Spring Data JDBC.

Then, you can write the SQL script for creating the `orders` table in a `v1__Initial_schema.sql` file under `src/main/resources/db/migration`. Ensure you type two underscores after the version number.

Listing 8.10 Flyway migration script for schema initialization (V1__Initial_schema.sql)

```
CREATE TABLE orders ( ①
    id          BIGSERIAL PRIMARY KEY NOT NULL, ②
    book_isbn   varchar(255) NOT NULL,
    book_name   varchar(255),
    book_price  float8,
    quantity     int NOT NULL,
    status       varchar(255) NOT NULL,
    created_date timestamp NOT NULL,
    last_modified_date timestamp NOT NULL,
    version      integer NOT NULL
);
```

- ① Definition of the orders table.
- ② Declares the "id" field as the primary key.

Finally, open the `application.yml` file and configure Flyway to use the same database managed with Spring Data R2DBC but using the JDBC driver.

Listing 8.11 Configure the Flyway integration over JDBC (application.yml)

```
spring:
  r2dbc:
    username: user
    password: password
    url: r2dbc:postgresql://localhost:5432/polardb_order
    pool:
      max-create-connection-time: 2s
      initial-size: 5
      max-size: 10
  flyway:
    user: ${spring.r2dbc.username} ①
    password: ${spring.r2dbc.password} ②
    url: jdbc:postgresql://localhost:5432/polardb_order ③
```

- ① Gets the value from the username configured for R2DBC.
- ② Gets the value from the password configured for R2DBC.
- ③ The same database configured for R2DBC but using the JDBC driver.

As you probably noticed, defining domain objects and adding a persistence layer in a reactive application is similar to what you would do with imperative applications. The main differences you encountered in this session are using the R2DBC driver instead of JDBC and having a separate Flyway configuration (at least until R2DBC support is added to the Flyway project: <https://github.com/flyway/flyway/issues/2502>).

In the next section, you'll learn how to use `Mono` and `Flux` in your business logic.

8.2.3 Implementing the business logic with reactive streams

The Spring reactive stack makes it straightforward to build asynchronous, non-blocking applications. In the previous section, we used Spring Data R2DBC and didn't have to deal with any underlying reactive concern. That's generally true for all the reactive modules in Spring. As a developer, you can rely on a familiar, simple, and productive approach for building reactive applications while the framework takes care of all the heavy lifting.

By default, Spring WebFlux assumes that everything is reactive. Such an assumption means that you're expected to interact with the framework by exchanging `Publisher<T>` objects like `Mono<T>` and `Flux<T>`. For example, the `OrderRepository` we created earlier will give access to orders as `Mono<Order>` and `Flux<Order>` instead of returning `Optional<Order>` and `Collection<Order>` like it would do in a non-reactive context. Let's see that in action.

In the `com.polarbookshop.orderservice.order.domain` package, create a new `OrderService` class. For starters, let's implement the logic to read orders through the repository. When multiple orders are involved, you can use a `Flux<Order>` object, representing an asynchronous sequence of zero or more orders.

Listing 8.12 Fetch orders via a reactive stream (OrderService.java)

```
package com.polarbookshop.orderservice.order.domain;

import reactor.core.publisher.Flux;
import org.springframework.stereotype.Service;

@Service ①
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    public Flux<Order> getAllOrders() { ②
        return orderRepository.findAll();
    }
}
```

- ① Stereotype annotation that marks a class to be a service managed by Spring.
- ② A Flux is used to publish multiple orders (0..N).

Then, we need a method to submit orders. Until we have the integration with Catalog Service in place, we can always default to reject submitted orders. `OrderRepository` exposes a `save()` method provided by `ReactiveCrudRepository`. You can build a reactive stream to pass an object of type `Mono<Order>` to `OrderRepository` that will save the order in the database.

Given an ISBN identifying a book and the number of copies to order, you can build a `Mono` object with `Mono.just()` in the same way you would build a Java `Stream` object with

`Stream.of()`. The difference is in the reactive behavior.

You can use the `Mono` object to start a reactive stream and then rely on the `flatMap()` operator to pass the data to `OrderRepository`. Add the following code to the `OrderService` class and complete the business logic implementation.

Listing 8.13 Persist rejected orders upon order request submission (OrderService.java)

```
...
public Mono<Order> submitOrder(String isbn, int quantity) {
    return Mono.just(buildRejectedOrder(isbn, quantity)) ①
        .flatMap(orderRepository::save); ②
}

public static Order buildRejectedOrder(
    String bookIsbn, int quantity
) { ③
    return Order.of(bookIsbn, null, null, quantity, OrderStatus.REJECTED);
}
...
```

- ① Creates a "Mono" out of an "Order" object.
- ② Saves in the database the Order object produced asynchronously by the previous step of the reactive stream.
- ③ When an order is rejected, we only specify ISBN, quantity, and status. Spring Data takes care of adding identifier, version, and audit metadata.

SIDE BAR

map vs. flatMap

When using Reactor, choosing between the `map()` and `flatMap()` operators is usually a source of confusion. Both operators return a reactive stream (either `Mono<T>` or `Flux<T>`). While `map()` maps between two standard Java types, `flatMap()` maps from a Java type to another reactive stream. In the previous code snippet, we map from an object of type `Order` to a `Mono<Order>` (which is returned by `OrderRepository`). Since the `map()` operator expects the target type not to be a reactive stream, it would wrap it in one nevertheless and return a `Mono<Mono<Order>>` object. On the other hand, the `flatMap()` operator expects the target type to be a reactive stream, so it knows how to handle the publisher produced by `OrderRepository` and returns a `Mono<Order>` object correctly.

In the next section, you'll complete the basic implementation of Order Service by exposing an API to fetch and submit orders.

8.2.4 Exposing a REST API with Spring WebFlux

There are two options for defining RESTful endpoints in a Spring WebFlux application: `@RestController` classes or functional beans (i.e., `Router Functions`). For the Order Service application, we'll use the first option. Compared to what we did in chapter 3, the method handlers will return reactive objects.

For the `GET` endpoint, we can use the `Order` domain entity we defined earlier and return a `Flux<Order>` object. When submitting an order, a user must provide the ISBN of the desired book and the number of copies they would like to purchase. We can model that information in an `OrderRequest` record that will act as a Data Transfer Object (DTO). It's also good practice to validate the input, as you learned in chapter 3.

Create a new `com.polarbookshop.orderservice.order.web` package and define an `OrderRequest` record to hold the submitted order information.

Listing 8.14 The OrderRequest DTO class with validation constraints defined for each field (`OrderRequest.java`)

```
package com.polarbookshop.orderservice.order.web;

import javax.validation.constraints.*;

public record OrderRequest (

    @NotBlank(message = "The book ISBN must be defined.")  
    String isbn,  
    ①  
    @NotNull(message = "The book quantity must be defined.")  
    @Min(value = 1, message = "You must order at least 1 item.")  
    @Max(value = 5, message = "You cannot order more than 5 items.")  
    Integer quantity  
    ②
){}
```

- ① Must not be null and must contain at least one non-whitespace character.
- ② Must not be null and contain a value between 1 and 5.

In the same package, create an `OrderController` class to define the two RESTful endpoints exposed by the Order Service application. Since you defined validation constraints for the `OrderRequest` object, you also need to use the familiar `@Valid` annotation to trigger the validation when the method is called.

Listing 8.15 Define handlers to process REST requests (OrderController.java)

```
package com.polarbookshop.orderservice.order.web;

import javax.validation.Valid;
import com.polarbookshop.orderservice.order.domain.Order;
import com.polarbookshop.orderservice.order.domain.OrderService;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import org.springframework.web.bind.annotation.*;

@RestController ①
@RequestMapping("orders") ②
public class OrderController {
    private final OrderService orderService;

    public OrderController(OrderService orderService) {
        this.orderService = orderService;
    }

    @GetMapping
    public Flux<Order> getAllOrders() { ③
        return orderService.getAllOrders();
    }

    @PostMapping
    public Mono<Order> submitOrder(
        @RequestBody @Valid OrderRequest orderRequest ④
    ) {
        return orderService.submitOrder(
            orderRequest.isbn(), orderRequest.quantity()
        );
    }
}
```

- ① Stereotype annotation marking a class as a Spring component and a source of handlers for REST endpoints.
- ② Identifies the root path mapping URI for which the class provides handlers ("orders").
- ③ A Flux is used to publish multiple orders (0..N).
- ④ Accepts an "OrderRequest" object, validated and used to create an order. The created order is returned as a "Mono".

The REST controller completes the basic implementation of the Order Service application. Let's see it in action. First, make sure the PostgreSQL container you created earlier is still running. Then, open a Terminal window, navigate to the Order Service project root folder, and run the application.

```
$ ./gradlew bootRun
```

You can try out the API, for example, by submitting an order. The application will save the order as rejected and return a 200 response to the client.

```
$ http POST :9002/orders isbn=1234567890 quantity=3
HTTP/1.1 200 OK
{
  "bookIsbn": "1234567890",
  "bookName": null,
  "bookPrice": null,
  "createdDate": "2022-06-06T09:40:58.374348Z",
  "id": 1,
  "lastModifiedDate": "2022-06-06T09:40:58.374348Z",
  "quantity": 3,
  "status": "REJECTED",
  "version": 1
}
```

To make it possible to submit orders successfully, we need to make Order Service call Catalog Service to check the book availability and fetch the necessary information for processing the order. That's the focus of the next section. Before moving on, stop the application with `ctrl+C`.

8.3 Reactive clients with Spring WebClient

In a cloud native system, applications can interact in different ways. This section focuses on request/response interactions over HTTP that you'll establish between Order Service and Catalog Service. In this kind of interaction, the client making the request expects to receive a response. In an imperative application, that would translate into a thread blocking until a response is returned. Instead, in a reactive application, we can use resources more efficiently so that no thread will wait for a response, freeing up resources to deal with other processing.

The Spring Framework comes bundled with two clients to perform HTTP requests: `RestTemplate` and `WebClient`. `RestTemplate` is the original Spring REST client that allows blocking HTTP request/response interactions based on a template method API. Since Spring Framework 5.0, it's in maintenance mode and practically deprecated. It's still widely used but will not get any new functionality in future releases.

`WebClient` is the modern alternative to `RestTemplate`. It provides blocking and non-blocking I/O, making it the perfect candidate for both imperative and reactive applications. It can be operated through a functional-style, fluent API that lets you configure any aspect of the HTTP interaction.

This section will teach you how to use `WebClient` to establish non-blocking request/response interactions. I will also explain how to make your application more resilient by adopting patterns like timeouts, retries, and failovers using the Reactor operators `timeout()`, `retryWhen()`, and `onError()`.

8.3.1 Service-to-service communication in Spring

As per the 15-Factor methodology, any backing service should be attached to an application through resource binding. For databases, you relied on the configuration properties provided by Spring Boot to specify credentials and the URL. When a backing service is another application, you need to provide its URL in a similar way. Following the externalized configuration principle, the URL should be configurable, not hard-coded. In Spring, you can achieve that through a `@ConfigurationProperties` bean, as you learned in chapter 4.

In the Order Service project, add a `ClientProperties` record in the `com.polarbookshop.orderservice.config` package. There, define your custom `polar.catalog-service-uri` property to configure the URI for calling the Catalog Service.

Listing 8.16 Define a custom property for the Catalog Service URI (ClientProperties.java)

```
package com.polarbookshop.orderservice.config;

import java.net.URI;
import javax.validation.constraints.NotNull;
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "polar") ①
public record ClientProperties(
    @NotNull
    URI catalogServiceUri ②
){}
```

- ① The prefix for the custom properties.
- ② The property for specifying the Catalog Service URI. It cannot be null.

NOTE

To get auto-completion and type validation checks from your IDE, you need to add a dependency on `org.springframework.boot:spring-boot-configuration-processor` with `scope annotationProcessor` in the `build.gradle` file like you did in chapter 4. You can refer to the Chapter08/08-end/order-service/build.gradle file in the code repository accompanying the book for checking the final result <https://github.com/ThomasVitale/cloud-native-spring-in-action>.

Then, enable the custom configuration properties in your `OrderServiceApplication` class using the `@ConfigurationPropertiesScan` annotation.

Listing 8.17 Enable custom configuration properties (OrderServiceApplication.java)

```
package com.polarbookshop.orderservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationPropertiesScan;

@SpringBootApplication
@ConfigurationPropertiesScan ①
public class OrderServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }
}
```

- ① Loads configuration data beans in the Spring context.

Finally, add a value for the new property to your `application.yml` file. As a default, you can use the URI for the Catalog Service instance running in your local environment.

Listing 8.18 Configure the URI for Catalog Service (application.yml)

```
...
polar:
  catalog-service-uri: "http://localhost:9001"
```

NOTE	When deploying the system with Docker Compose or Kubernetes, you can override the property value via an environment variable taking advantage of the service discovery features offered by the two platforms.
-------------	---

In the next section, you'll use the value configured through this property to call the Catalog Service from Order Service.

8.3.2 Understanding how to exchange data

Whenever a user submits an order for a specific book, Order Service needs to call Catalog Service to check the requested book's availability and fetch its details like the title, the author, and the price. The interaction (HTTP request/response) is described in figure [8.6](#).

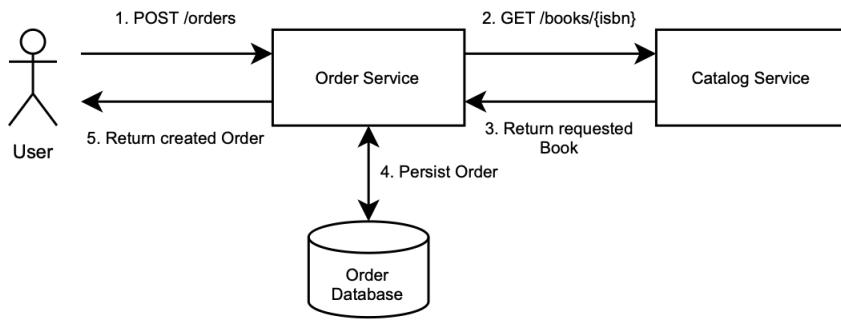


Figure 8.6 When an order is submitted, Order Service calls Catalog Service over HTTP to check the book's availability and fetch its details.

Each order request is submitted for a specific book ISBN. Order Service needs to know the book ISBN, title, author, and price to process an order correctly. Currently, Catalog Service exposes a /books/{bookIsbn} endpoint returning all the available information about a book. In a real scenario, you might expose a different endpoint that returns an object containing only the required information (a DTO). For the sake of the example, we'll reuse the existing endpoint since our focus right now is on building reactive clients.

Having established which endpoint to call, how should you model the exchange between two applications? You've just reached a crossroads. One option is to create a shared library with the classes used by both applications and import it as a dependency into both projects. As per the 15-factor methodology, such a library would be tracked in its own codebase. Doing so would ensure that the model used by both applications is consistent and never out-of-sync. However, it would mean adding implementation coupling.

The other option is to replicate the class into the upstream application. By doing so, you wouldn't have implementation coupling, but you would have to take care of evolving the model as the original one changes in the downstream application. There are a few techniques like consumer-driven contracts to verify when the called API changes through automated tests. Besides checking the data model, those tests would also verify other aspects of the exposed API like HTTP methods, response statuses, headers, variables, and so on. I will not cover this subject here, but I recommend checking the Spring Cloud Contract project (<https://spring.io/projects/spring-cloud-contract>) if you're interested.

Both are viable and used options. Which strategy to adopt is up to your project requirements and your organization's structure. For the Polar Bookshop project, we are going to use the second option.

In a new `com.polarbookshop.orderservice.book` package, create a `Book` record to be used as a DTO and include only the fields used by the order processing logic. As I pointed out earlier, in a real scenario, I would expose a new endpoint in Catalog Service returning book objects modeled as this DTO. For simplicity, we'll use the existing `/books/{bookIsbn}` endpoint, so

any information that doesn't map to any of the fields in this class will be discarded when deserializing the JSON received into the Java object. The fields you do define, make sure have the same name as the `Book` object defined in Catalog Service, or else the parsing will fail. That is something that consumer-driven contract tests can verify for you automatically.

Listing 8.19 The Book record is a DTO to store book information (Book.java).

```
package com.polarbookshop.orderservice.book;

public record Book(
    String isbn,
    String title,
    String author,
    Double price
){}
```

Having a DTO in Order Service ready to hold book information, let's see how to retrieve it from Catalog Service.

8.3.3 Implementing REST clients with WebClient

The modern and reactive choice for REST clients in Spring is `WebClient`. The framework provides several ways for instantiating a `WebClient` object. In this example, you're going to use `WebClient.Builder`. Refer to the official documentation to explore the other options (<https://spring.io/projects/spring-framework>).

In the `com.polarbookshop.orderservice.config` package, create a `ClientConfig` class to configure a `WebClient` bean with the base URL provided by `ClientProperties`.

Listing 8.20 Configure a WebClient bean to call Catalog Service (ClientConfig.java)

```
package com.polarbookshop.orderservice.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class ClientConfig {

    @Bean
    WebClient webClient(
        ClientProperties clientProperties,
        WebClient.Builder webClientBuilder ①
    ) {
        return webClientBuilder ②
            .baseUrl(clientProperties.catalogServiceUri().toString())
            .build();
    }
}
```

- ① An object auto-configured by Spring Boot to build `WebClient` beans.
- ② Configures the `WebClient` base URL to the Catalog Service URL defined as a custom property.

WARNING If you use IntelliJ IDEA, you might get a warning that `WebClient.Builder` cannot be autowired. Don't worry. It's a false positive. You can get rid of the warning by annotating the field with `@SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")`.

Next, create a `BookClient` class in the `com.polarbookshop.orderservice.book` package. That is where you are going to use the `WebClient` bean to send HTTP calls to the `GET /books/{bookIsbn}` endpoint exposed by Catalog Service through its fluent API. The `WebClient` will ultimately return a `Book` object wrapped in a `Mono` publisher.

Listing 8.21 Define a reactive REST client using `WebClient` (`BookClient.java`)

```
package com.polarbookshop.orderservice.book;

import reactor.core.publisher.Mono;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@Component
public class BookClient {
    private static final String BOOKS_ROOT_API = "/books/";
    private final WebClient webClient;

    public BookClient(WebClient webClient) {
        this.webClient = webClient; ①
    }

    public Mono<Book> getBookByIsbn(String isbn) {
        return webClient
            .get() ②
            .uri(BOOKS_ROOT_API + isbn) ③
            .retrieve() ④
            .bodyToMono(Book.class); ⑤
    }
}
```

- ① A `WebClient` bean as configured previously.
- ② The request should use the GET method.
- ③ The target URI of the request is `/books/{isbn}`.
- ④ Sends the request and retrieves the response.
- ⑤ Returns the retrieved object as "`Mono<Book>`".

`WebClient` is a reactive HTTP client. You've just seen how it can return data as reactive publishers. In particular, the result of calling Catalog Service to fetch details about a specific book is a `Mono<Book>` object. Let's see how you can include that in the order processing logic implemented in `OrderService`.

The `submitOrder()` method in the `OrderService` class is currently rejecting orders all the

time. Not for long. You can now autowire a `BookClient` instance and use the underlying `WebClient` to start a reactive stream to process the book information and create an order. The `map()` operator lets you map a `Book` to an accepted `Order`. If `BookClient` returns an empty result, then you can define a rejected `Order` with the `defaultIfEmpty()` operator. Finally, the stream is ended by calling `OrderRepository` to save the order (either accepted or rejected).

**Listing 8.22 Call BookClient to fetch the book details upon an order request submission
(OrderService.java)**

```
package com.polarbookshop.orderservice.order.domain;

import com.polarbookshop.orderservice.book.Book;
import com.polarbookshop.orderservice.book.BookClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import org.springframework.stereotype.Service;

@Service
public class OrderService {
    private final BookClient bookClient;
    private final OrderRepository orderRepository;

    public OrderService(
        BookClient bookClient, OrderRepository orderRepository
    ) {
        this.bookClient = bookClient;
        this.orderRepository = orderRepository;
    }

    ...

    public Mono<Order> submitOrder(String isbn, int quantity) {
        return bookClient.getBookByIsbn(isbn) ①
            .map(book -> buildAcceptedOrder(book, quantity)) ②
            .defaultIfEmpty( ③
                buildRejectedOrder(isbn, quantity)
            )
            .flatMap(orderRepository::save); ④
    }

    public static Order buildAcceptedOrder(Book book, int quantity) {
        return Order.of(book.isbn(), book.title() + " - " + book.author(),
            book.price(), quantity, OrderStatus.ACCEPTED); ⑤
    }

    public static Order buildRejectedOrder(String bookIsbn, int quantity) {
        return Order.of(bookIsbn, null, null, quantity, OrderStatus.REJECTED);
    }
}
```

- ① Calls the Catalog Service to check the book availability.
- ② If the book is available, it accepts the order.
- ③ If not available, it rejects the order.
- ④ Saves the order (either accepted or rejected).
- ⑤ When an order is accepted, we specify ISBN, book name (title + author), quantity, and status. Spring Data takes care of adding identifier, version, and audit metadata.

Let's try that out. First, ensure the PostgreSQL container is up and running by executing this command from the folder where you keep your Docker Compose configuration (`polar-deployment/docker`).

```
$ docker-compose up -d polar-postgres
```

Then, build and run both Catalog Service and Order Service (`./gradlew bootRun`).

WARNING If you're using an Apple Silicon computer, the application logs from Order Service might include some warnings related to DNS resolutions in Netty. In this specific case, the application should still work correctly. If you experience issues, you can add this additional dependency to the Order Service project as `runtimeOnly` to fix the problem:
`io.netty:netty-resolver-dns-native-macos:4.1.77.Final:osx-aarch_64`

Finally, send an order for one of the books created in Catalog Service at startup time. If the book exists, the order should be accepted.

```
$ http POST :9002/orders isbn=1234567891 quantity=3
HTTP/1.1 200 OK
{
  "bookIsbn": "1234567891",
  "bookName": "Northern Lights - Lyra Silverstar",
  "bookPrice": 9.9,
  "createdDate": "2022-06-06T09:59:32.961420Z",
  "id": 2,
  "lastModifiedDate": "2022-06-06T09:59:32.961420Z",
  "quantity": 3,
  "status": "ACCEPTED",
  "version": 1
}
```

When you are done verifying the interaction, stop the applications with `Ctrl+C` and the container with `docker-compose down`.

That concludes the implementation of the order creation logic. If the book exists in the catalog, the order is accepted. If an empty result is returned, it's rejected. What if Catalog Service takes too much time to reply? What if it's momentarily unavailable and can't process any new requests? What if it replies with an error? The following section will answer and handle all these questions.

8.4 Resilient applications with Reactive Spring

Resilience is about keeping a system available and delivering its services, even when failures happen. Since failures will happen and there's no way to prevent them all, it becomes critical to design fault-tolerant applications. The goal is to keep the system available without the user noticing. In the worst-case scenario, it could have degraded functionality (*graceful degradation*), but it should still be available.

The critical point to achieving resilience (or fault-tolerance) is keeping the faulty component isolated until the fault gets fixed. By doing that, you'll prevent what Michael T. Nygard calls *crack propagation*. Think about Polar Bookshop. If Catalog Service enters a faulty state and becomes unresponsive, you don't want Order Service to be affected as well. Integration points between application services should be carefully guarded and made resilient to failures affecting the other party.

There are several patterns to building resilient applications. In the Java ecosystem, a popular library to implement such patterns was Hystrix, developed by Netflix, but as of 2018, it entered the maintenance mode and will not be developed further. Resilience4J gained a lot in popularity to fill the void left by Hystrix. Project Reactor, the Reactive Spring stack foundation, also provides some useful features for resilience.

In this section, you'll make the integration point between Order Service and Catalog Service more robust, using Reactive Spring to configure timeouts, retries, and fallbacks. In the next chapter, you'll learn more about building resilient applications with Resilience4J and Spring Cloud Circuit Breaker.

8.4.1 Timeouts

Whenever your application calls a remote service, you don't know if and when a response will be received. Timeouts (also called *time limiters*) are a simple, yet effective tool to preserve the responsiveness of your application in case a response is not received within a reasonable time period.

Two main reasons for setting up timeouts are:

- if you don't limit the time your client waits, you risk your computational resources being blocked for too long (for imperative applications). In the worst-case scenario, your application will be completely unresponsive because all the available threads are blocked, waiting for a response from a remote service, so there's no thread available to handle new requests anymore;
- if you can't meet your Service Level Agreements (SLAs), there's no reason to keep waiting for an answer, better to fail the request.

Examples of timeouts are:

- **Connection timeout.** It's the time limit for establishing a communication channel with a remote resource. Earlier, you configured the `server.netty.connection-timeout` property to limit the time Netty waits for a TCP connection to be established.
- **Connection pool timeout.** It's the time limit for a client to get a connection from a pool. In chapter 5, you configured a timeout for the Hikari connection pool through the `spring.datasource.hikari.connection-timeout` property.
- **Read timeout.** It's the time limit to read from a remote resource after establishing the initial connection. In the following, you'll define a read timeout for the call to the Catalog Service performed by the `BookClient` class.

In this section, you'll define a timeout for `BookClient` so that if it expires, the Order Service application will throw an exception. You can also specify a failover instead of throwing the exception to the user. Figure 8.7 details how the request/response interaction will work when timeouts and failovers are defined.

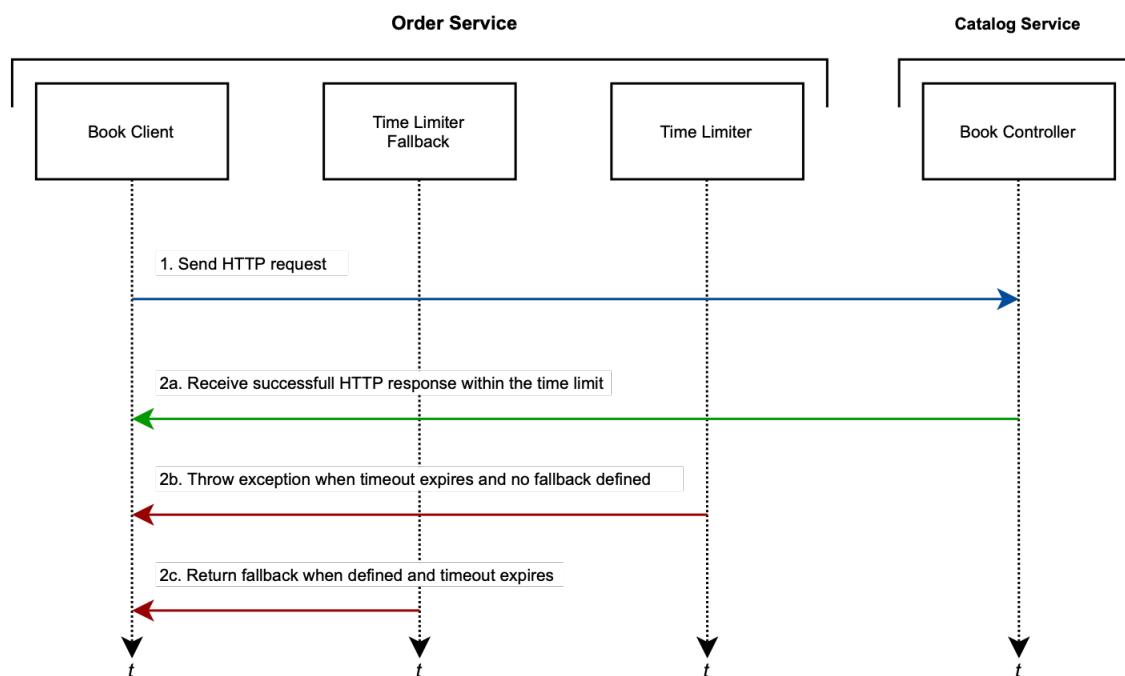


Figure 8.7 When a response is received from the remote service within the time limit, the request is successful. If the timeout expires and no response is received, then a fallback behavior is executed, if any. Otherwise, an exception is thrown.

DEFINING A TIMEOUT FOR WEBCLIENT

Project Reactor provides a `timeout()` operator for defining a time limit for completing an operation. You can chain it with the result of the `WebClient` call to continue the reactive stream. Update the `getBookByIsbn()` method in the `BookClient` class as follows to define a timeout of 3 seconds.

Listing 8.23 Define a timeout for the HTTP interaction (BookClient.java)

```
...
public Mono<Book> getBookByIsbn(String isbn) {
    return webClient
        .get()
        .uri(BOOKS_ROOT_API + isbn)
        .retrieve()
        .bodyToMono(Book.class)
        .timeout(Duration.ofSeconds(3)); ①
}
...
```

- ① Sets a timeout for the GET request to 3 seconds.

Instead of throwing an exception when the timeout expires, you have the chance to provide a fallback behavior. Considering that Order Service can't accept an order if the book availability is not verified, you might think of returning an empty result so that the order will be rejected. You can define a reactive empty result using `Mono.empty()`.

Listing 8.24 Define timeout and fallback for the HTTP interaction (BookClient.java)

```
...
public Mono<Book> getBookByIsbn(String isbn) {
    return webClient
        .get()
        .uri(BOOKS_ROOT_API + isbn)
        .retrieve()
        .bodyToMono(Book.class)
        .timeout(Duration.ofSeconds(3), Mono.empty()); ①
}
...
```

- ① The fallback returns an empty "Mono" object.

NOTE

In a real production scenario, you might want to externalize the timeout configuration by adding a new field to the `ClientProperties`. In that way, you can change its value depending on the environment without having to rebuild the application. It's also essential to monitor any timeout and tune its value if necessary.

UNDERSTANDING HOW TO USE TIMEOUTS EFFECTIVELY

Timeouts improve application resilience and follow the principle of failing fast. Setting a good value for the timeout can be tricky. You should consider your system architecture as a whole. In the previous example, you defined a 2-seconds timeout. It means that a response should get from Catalog Service to Order Service within that time limit. Otherwise, either a failure or a fallback occurs. Catalog Service, in turn, sends a request to the PostgreSQL database to fetch the data about the specific book and waits for a response. A connection timeout itself guards that interaction. You should carefully design a time-limiting strategy for all the integration points in your system to meet your software SLAs and guarantee a good user experience.

If Catalog Service were available, but a response wouldn't get to Order Service within the time limit, the request would likely still be processed by Catalog Service. That is a critical aspect to consider when configuring timeouts. It doesn't matter much for read/query operations because they are idempotent. For write/command operations, you want to ensure proper handling when a timeout expires, including providing the user with the correct status about the operation outcome.

When Catalog Service is overloaded, it can take several seconds to get a JDBC connection from the pool, fetch data from the database, and send a response back to Order Service. In that case, you could think of retrying the request rather than falling back on a default behavior or throwing an exception.

8.4.2 Retries

When a service downstream doesn't respond within a specific time limit or replies with a server error related to its momentary inability to process the request, you can configure your client to try it again. When a service doesn't respond correctly, it's likely because it's going through some issues, and it's unlikely that it will manage to recover immediately. Starting a sequence of retry attempts one after the other risks making the system even more unstable. You don't want to launch a DoS attack on your own applications!

A better approach is using an **exponential backoff** strategy to perform each retry attempt with a growing delay. By waiting for more and more time between one attempt and the next, it's more likely the backing service has had the time to recover and become responsive again. The strategy for computing the delay can be configured.

In this section, you'll configure retries for `BookClient`. Figure 8.8 details how the request/response interaction will work when retries are configured with exponential backoff. For example, the figure shows a scenario where each retry attempt's delay is computed as the number of attempts multiplied by 100ms (the initial backoff value).

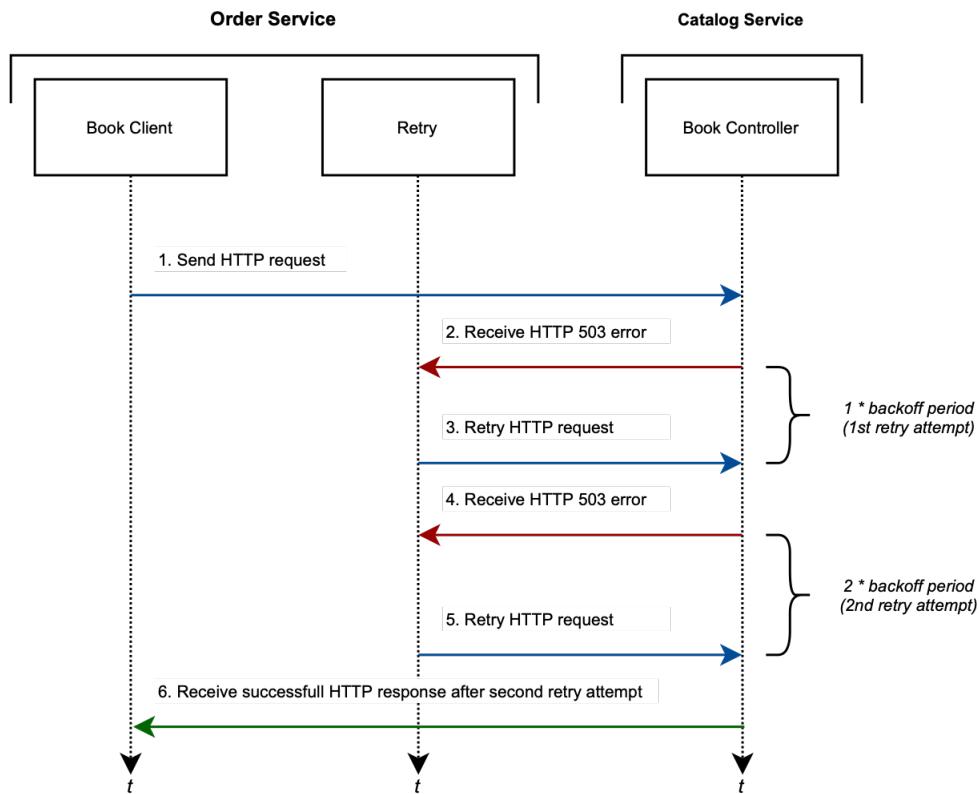


Figure 8.8 When Catalog Service doesn't respond successfully, Order Service will try at most three more times with a growing delay.

DEFINING RETRIES FOR WEBCLIENT

Project Reactor provides a `retryWhen()` operator to retry an operation when it fails. The position where you apply it to the reactive stream matters.

- Placing the `retryWhen()` operator after `timeout()` means that the timeout is applied to each retry attempt.
- Placing the `retryWhen()` operator before `timeout()` means that the timeout is applied to the overall operation (that is, the whole sequence of initial request and retries has to happen within the given time limit).

In `BookClient`, we want the timeout to apply to each retry attempt, so we'll use the first option. The time limiter is applied first. If the timeout expires, the `retryWhen()` operator kicks in and tries the request again. Update the `getBookByIsbn()` method in the `BookClient` class to configure a retry strategy.

You can define the number of attempts and the minimum duration for the first backoff. The delay is computed for each retry as the current attempt number multiplied by the minimum backoff period. A jitter factor can be used to add randomness to the exponential to each backoff. By default, a jitter of at most 50% of the computed delay is used. When you have multiple instances of Order Service running, the jitter factor ensures that the replicas will not retry requests simultaneously.

Listing 8.25 Define retries with exponential backoff for the HTTP interaction (BookClient.java)

```
public Mono<Book> getBookByIsbn(String isbn) {
    return webClient
        .get()
        .uri(BOOKS_ROOT_API + isbn)
        .retrieve()
        .bodyToMono(Book.class)
        .timeout(Duration.ofSeconds(3), Mono.empty())
        .retryWhen(①
            Retry.backoff(3, Duration.ofMillis(100)))
    );
}
```

- ① Exponential backoff is used as the retry strategy. 3 attempts allowed and 100ms initial backoff.

UNDERSTANDING HOW TO USE RETRIES EFFECTIVELY

Retries increase the chance of getting a response back from a remote service when it's momentarily overloaded or unresponsive. Use them wisely. In the context of timeouts, I highlighted the need for handling read and write operations differently. When it comes to retries, it's even more critical. Idempotent requests like read operations can be retried without harm. Even some write requests can be idempotent. For example, a request to change the author of a book with a given ISBN from "S.L. Cooper" to "Sheldon Lee Cooper" is idempotent. You could perform it a few times, but the outcome will not change. You shouldn't retry non-idempotent requests, or you'll risk generating inconsistent states. When you order a book, you don't want to be charged multiple times just because the first attempt failed due to the response being lost in the network and never received.

When retries are configured in a flow where the user is involved, remember to balance resilience and user experience. You don't want users to wait too much while retrying the request behind the scenes. If you can't avoid that, make sure you inform the users and give them feedback about the status of the request.

Retries are a helpful pattern whenever the service downstream is momentarily unavailable or slow due to overloading, but it's likely to heal soon. In this case, you should limit the number of retries and use exponential backoff to prevent adding extra load on an already overloaded service. On the other hand, you shouldn't retry the request if the service fails with a recurrent error, for example, if it's entirely down or returns an acceptable error like 404. The following section will show you how to define fallbacks when specific errors happen.

8.4.3 Fallbacks and error handling

A system is resilient if it keeps providing its services in the face of faults without the user noticing it. Sometimes that's not possible, so the least you can do is ensure a graceful degradation of the service level. A fallback behavior can help you limit the fault to a small area while preventing the rest of the system from misbehaving or entering a faulty state.

When talking about timeouts, you have already dealt with a fallback behavior if no response is received within the time limit. You want to include fallbacks into your general strategy to make your system resilient, not just for a specific case like timeouts. A fallback function can be triggered when some errors or exceptions occur, but they're not all the same.

Some errors are acceptable and semantically meaningful in the context of your business logic. When Order Service calls Catalog Service to fetch information about a specific book, a 404 response might be returned. That's an acceptable response that should be addressed to inform the user that the order cannot be submitted because the book is not available in the catalog.

The retry strategy you defined in the previous section is not limited: it will retry the request as long as an error response is received, including acceptable ones like 404. In that case, you don't want to retry the request. Project Reactor provides an `onErrorResume()` operator to define a fallback when a specific error occurs. You can add it to the reactive stream after the `timeout()` operator and before the `retryWhen()` so that if a 404 response is received (`WebClientResponseException.NotFound` exception), the retry operator is not triggered. Then, you can use the same operator again at the end of the stream to catch any other exception and fall back to an empty `Mono`.

Listing 8.26 Define exception handling and fallbacks for the HTTP interaction (BookClient.java)

```
public Mono<Book> getBookByIsbn(String isbn) {
    return webClient
        .get()
        .uri(BOOKS_ROOT_API + isbn)
        .retrieve()
        .bodyToMono(Book.class)
        .timeout(Duration.ofSeconds(3), Mono.empty())
        .onErrorResume(WebClientResponseException.NotFound.class,
            exception -> Mono.empty()); ①
        .retryWhen(Retry.backoff(3, Duration.ofMillis(100)))
        .onErrorResume(Exception.class,
            exception -> Mono.empty()); ②
}
```

- ① Returns an empty object when a 404 response is received.
- ② If any error happens after the 3 retry attempts, catch the exception and return an empty object.

NOTE

In a real-world scenario, you would probably want to return some contextual information depending on the type of error instead of always returning an empty object. For example, you could add a `reason` field to the `Order` object to describe why it's been rejected. Was it because the book is unavailable in the catalog or because of network problems? In the second case, you could inform the user that the order cannot be processed because it's momentarily unable to check the book availability. A better option would be to save the order in a pending state, queue the order submission request and try it again later, using one of the strategies I'll cover in chapter 10.

The key aspect is designing a resilient system that, in the best-case scenario, can provide its services without the user noticing it. In contrast, in the worst-case scenario, it should still work but with graceful degradation.

NOTE

Spring WebFlux and Project Reactor are exciting subjects in the Spring landscape. If you want to learn more about how reactive Spring works, I recommend checking "Reactive Spring" by Josh Long (<https://reactivespring.io>). In the Manning catalog, you can check part 3 from "Spring in Action" (6th Edition) by Craig Walls (<https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-11>).

In the next section, you'll write automated tests to verify the different aspects of the Order Service application.

8.5 Testing reactive applications with Spring, Reactor, and Testcontainers

When an application depends on a service downstream, you should test the interaction against the API specification of the latter. In this section, you'll first try the `BookClient` class against a mock web server acting as the Catalog Service to ensure the correctness of the client. Then, you'll test the data persistence layer with sliced tests using the `@DataR2dbcTest` annotation and Testcontainers, similarly to what you did in chapter 5 with `@DataJdbcTest`. Finally, you'll write sliced tests for the web layer using the `@WebFluxTest` annotation, which works in the same way as `@WebMvcTest` but for reactive applications.

You already have the necessary dependencies on the Spring Boot test libraries and Testcontainers. What is missing is a dependency on `com.squareup.okhttp3:mockwebserver` that will provide utilities to run a mock web server. Open the `build.gradle` file of the Order Service project and add the missing dependency.

Listing 8.27 Add test dependency for OkHttp MockWebServer in Order Service (build.gradle)

```
dependencies {
    ...
    testImplementation 'com.squareup.okhttp3:mockwebserver'
}
```

Let's start by testing the `BookClient` class.

8.5.1 Testing REST clients with a mock web server

The OkHttp project provides a mock web server that you can use to test HTTP-based request/response interactions with a service downstream. `BookClient` is returning a `Mono<Book>` object, so you can use the convenient utilities provided by Project Reactor for testing reactive applications. The `StepVerifier` object lets you process reactive streams and write assertions in steps through a fluent API.

First, let's set up the mock web server and configure `WebClient` to use it.

Listing 8.28 Prepare the test setup with a mock web server (BookClientTests.java)

```
package com.polarbookshop.orderservice.book;

import java.io.IOException;
import okhttp3.mockwebserver.MockWebServer;
import org.junit.jupiter.api.*;
import org.springframework.web.reactive.function.client.WebClient;

class BookClientTests {
    private MockWebServer mockWebServer;
    private BookClient bookClient;

    @BeforeEach
    void setup() throws IOException {
        this.mockWebServer = new MockWebServer(); ①
        this.mockWebServer.start();

        var webClient = WebClient.builder() ②
            .baseUrl(mockWebServer.url("/").uri().toString())
            .build();
        this.bookClient = new BookClient(webClient);
    }

    @AfterEach
    void clean() throws IOException {
        this.mockWebServer.shutdown(); ③
    }
}
```

- ① Starts the mock server before running a test case.
- ② Uses the mock server URL as the base URL for `WebClient`.
- ③ Shuts the mock server down after completing a test case.

Then, we can define some test cases to validate the client functionality in Order Service.

Listing 8.29 Test the interaction with the Catalog Service application (BookClientTests.java)

```
package com.polarbookshop.orderservice.book;

...
import okhttp3.mockwebserver.MockResponse;
import reactor.core.publisher.Mono;
import reactor.test.StepVerifier;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;

class BookClientTests {
    private MockWebServer mockWebServer;
    private BookClient bookClient;

    ...

    @Test
    void whenBookExistsThenReturnBook() {
        var bookIsbn = "1234567890";

        var mockResponse = new MockResponse() ❶
            .addHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
            .setBody("""
                {
                    "isbn": %s,
                    "title": "Title",
                    "author": "Author",
                    "price": 9.90,
                    "publisher": "Polarisophia"
                }
            """.formatted(bookIsbn));

        mockWebServer.enqueue(mockResponse); ❷

        Mono<Book> book = bookClient.getBookByIsbn(bookIsbn);

        StepVerifier.create(book) ❸
            .expectNextMatches(
                b -> b.isbn().equals(bookIsbn)) ❹
            .verifyComplete(); ❺
    }
}
```

- ❶ Defines the response to be returned by the mock server.
- ❷ Adds a mock response to the queue processed by the mock server.
- ❸ Initializes a StepVerifier object with the object returned by BookClient.
- ❹ Asserts the Book returned has the ISBN requested.
- ❺ Verifies the reactive stream completed successfully.

Let's run the tests and ensure they succeed. Open a Terminal window, navigate to the root folder of your Order Service project, and run the following command.

```
$ ./gradlew test --tests BookClientTests
```

NOTE

When using mocks, there might be situations where the test results depend on the order in which test cases are executed, which tend to be the same on the same operating system. To prevent unwanted execution dependencies, you can annotate the test class with `@TestMethodOrder(MethodOrderer.Random.class)` to ensure that a pseudo-random order is used at each execution.

After testing the REST client part, you can move on and verify the data persistence layer for Order Service.

8.5.2 Testing data persistence with `@DataR2dbcTest` and `Testcontainers`

As you might recall from the previous chapters, Spring Boot allows you to run integration tests by loading only the Spring components used by a specific application slice. For the REST API, you will create tests for the WebFlux slice. Now, I'll show you how to write tests for the R2DBC slice using the `@DataR2dbcTest` annotation.

The approach is the same you used in chapter 5 for testing the data layer in Catalog Service. However, there are two main differences. First, you will use the `StepVerifier` utility to test the `OrderRepository` behavior reactively. Second, you will define a PostgreSQL test container instance explicitly.

For the Catalog Service application, we relied on the test container auto-configuration. In this case, we'll define a test container in the test class and mark it as `@Container`. Then, the `@Testcontainers` annotation on the class will activate the automatic startup and cleanup of the test container. Finally, we'll use the `@DynamicProperties` annotation provided by Spring Boot to pass the test database credentials and URL to the application. This approach of defining test containers and overwriting properties is generic and can be applied to other scenarios.

Now, on to the code. Create a `OrderRepositoryR2dbcTests` class and implement autotests to verify the data persistence layer of the application.

Listing 8.30 Integration tests for the Data R2DBC slice (OrderRepositoryR2dbcTests.java)

```

package com.polarbookshop.orderservice.order.domain;

import com.polarbookshop.orderservice.config.DataConfig;
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
import org.testcontainers.utility.DockerImageName;
import reactor.test.StepVerifier;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.r2dbc.DataR2dbcTest;
import org.springframework.context.annotation.Import;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;

@DataR2dbcTest ①
@Import(DataConfig.class) ②
@Testcontainers ③
class OrderRepositoryR2dbcTests {

    @Container ④
    static PostgreSQLContainer<?> postgresql =
        new PostgreSQLContainer<>(DockerImageName.parse("postgres:14.3"));

    @Autowired
    private OrderRepository orderRepository;

    @DynamicPropertySource ⑤
    static void postgresqlProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.r2dbc.url", OrderRepositoryR2dbcTests::r2dbcUrl);
        registry.add("spring.r2dbc.username", postgresql::getUsername);
        registry.add("spring.r2dbc.password", postgresql::getPassword);
        registry.add("spring.flyway.url", postgresql::getJdbcUrl);
    }

    private static String r2dbcUrl() { ⑥
        return String.format("r2dbc:postgresql://%s:%s/%s",
            postgresql.getContainerIpAddress(),
            postgresql.getMappedPort(PostgreSQLContainer.POSTGRES_PORT),
            postgresql.getDatabaseName());
    }

    @Test
    void createRejectedOrder() {
        var rejectedOrder = OrderService.buildRejectedOrder( "1234567890", 3);
        StepVerifier
            .create(orderRepository.save(rejectedOrder)) ⑦
            .expectNextMatches( ⑧
                order -> order.status().equals(OrderStatus.REJECTED))
            .verifyComplete(); ⑨
    }
}

```

- ① Identifies a test class that focuses on R2DBC components.
- ② Imports R2DBC configuration. Needed to enable auditing.
- ③ Activates automatic startup and cleanup of test containers.
- ④ Identifies a PostgreSQL container for testing.
- ⑤ Overwrites R2DBC and Flyway configuration to point to the test PostgreSQL instance.

- ⑥ Builds an R2DBC connection string since Testcontainers doesn't provide one out-of-the-box as it does for JDBC.
- ⑦ Initializes a StepVerifier object with the object returned by OrderRepository.
- ⑧ Asserts the Order returned has the correct status.
- ⑨ Verifies the reactive stream completed successfully.

Since those slice tests are based on Testcontainers, ensure the Docker Engine is running in your local environment. Then, run the tests.

```
$ ./gradlew test --tests OrderRepositoryR2dbcTests
```

In the next section, you'll write tests for the web slice.

8.5.3 Testing REST controllers with @WebFluxTest

The WebFlux slice can be tested similarly to how you tested the MVC layer in chapter 3 and using the same `webTestClient` utility you used for the integration tests. It's an enhanced version of the standard `webClient` object, containing extra features to simplify tests.

Create a `OrderControllerWebFluxTests` class and annotate it with `@WebFluxTest(OrderController.class)` to collect the slice tests for `OrderController`. As you learned in chapter 3, you can use the Spring annotation `@MockBean` to mock the `OrderService` class and have Spring add it to the Spring context used in the test. That's what makes it injectable.

Listing 8.31 Integration tests for the WebFlux slice (OrderControllerWebFluxTests.java)

```

package com.polarbookshop.orderservice.order.web;

import com.polarbookshop.orderservice.order.domain.Order;
import com.polarbookshop.orderservice.order.domain.OrderService;
import com.polarbookshop.orderservice.order.domain.OrderStatus;
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Mono;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.reactive.server.WebTestClient;
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@WebFluxTest(OrderController.class) ①
class OrderControllerWebFluxTests {

    @Autowired
    private WebTestClient webClient; ②

    @MockBean ③
    private OrderService orderService;

    @Test
    void whenBookNotAvailableThenRejectOrder() {
        var orderRequest = new OrderRequest("1234567890", 3);
        var expectedOrder = OrderService.buildRejectedOrder(
            orderRequest.isbn(), orderRequest.quantity());
        given(orderService.submitOrder(
            orderRequest.isbn(), orderRequest.quantity()))
            .willReturn(Mono.just(expectedOrder)); ④

        webClient
            .post()
            .uri("/orders/")
            .bodyValue(orderRequest)
            .exchange()
            .expectStatus().is2xxSuccessful(); ⑤
            .expectBody(Order.class).value(actualOrder -> {
                assertThat(actualOrder).isNotNull();
                assertThat(actualOrder.status()).isEqualTo(OrderStatus.REJECTED);
            });
    }
}

```

- ① Identifies a test class that focuses on Spring WebFlux components, targeting OrderController.
- ② A WebClient variant with extra features to make testing RESTful services easier.
- ③ Adds a mock of OrderService to the Spring application context.
- ④ Defines the expected behavior for the OrderService mock bean.
- ⑤ Expects the order is created successfully.

Next, run the slice tests for the web layer to ensure they pass.

```
$ ./gradlew test --tests OrderControllerWebFluxTests
```

Great job! You successfully built and tested a reactive application, maximizing scalability, resilience, and cost-effectiveness. In the source code accompanying the book, you can find more test examples, including full integration tests using the `@SpringBootTest` annotation and slice tests for the JSON layer using `@JsonTest`, as you learned in chapter 3.

SIDE BAR**Polar Labs**

Feel free to apply what you learned in the previous chapters and prepare the Order Service application for deployment.

1. Add Spring Cloud Config Client to Order Service to make it fetch configuration data from Config Service.
2. Configure the Cloud Native Buildpacks integration, containerize the application, and define the commit stage of the deployment pipeline.
3. Write the Deployment and Service manifests for deploying Order Service to a Kubernetes cluster.
4. Configure Tilt to automate the Order Service deployment to your local Kubernetes cluster initialized with minikube.

You can refer to the `Chapter08/08-end` folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and deploying the backing services from the manifests available in the `Chapter08/08-end/polar-deployment/kubernetes/platform/development` folder with `kubectl apply -f services`.

The next chapter will continue the discussion on resilience and introduce more patterns like circuit breakers and rate limiters using Spring Cloud Gateway, Spring Cloud Circuit Breaker, and Resilience4J.

8.6 Summary

- When you expect high traffic and concurrency with less computational resources, the reactive paradigm can improve the application scalability, resilience, and cost-effectiveness at the expense of a steeper initial learning curve.
- Choose between a non-reactive and a reactive stack according to your requirements.
- Spring WebFlux is based on Project Reactor and is the core of the reactive stack in Spring. It supports asynchronous, non-blocking I/O.
- Reactive RESTful services can be implemented through `@RestController` classes or `Router Functions`.
- The Spring WebFlux slice can be tested through the `@WebFluxTest` annotation.
- Spring Data R2DBC provides support for reactive data persistence using the R2DBC driver. The approach is the same as for any Spring Data project: database drivers, entities, and repositories.
- Database schemas can be managed with Flyway.
- The persistence slice of a reactive application can be tested using the `@DataR2dbcTest` annotation and Testcontainers.
- A system is resilient if it keeps providing its services in the face of faults without the user noticing it. Sometimes that's not possible, so the least you can do is ensure a graceful degradation of the services.
- WebClient is based on Project Reactor and works with `Mono` and `Flux` publishers.
- You can use the Reactor operators to configure timeouts, retries, fallbacks, and error handling to make the interaction more resilient to any failure in the service downstream or due to the network.

A large, light gray, stylized number '9' is positioned in the upper right corner of the page.

API gateway and circuit breakers

This chapter covers

- Implementing edge services with Spring Cloud Gateway and Reactive Spring
- Configuring circuit breakers with Spring Cloud Circuit Breaker and Resilience4J
- Defining rate limiters with Spring Cloud Gateway and Redis
- Managing distributed sessions with Spring Session Data Redis
- Routing application traffic with a Kubernetes Ingress

In the previous chapter, you learned several aspects of building resilient, scalable, and cost-effective applications using the reactive paradigm. The Spring reactive stack will be the foundation for implementing an API gateway for the Polar Bookshop system. The API gateway is a common pattern in distributed architectures like microservices to decouple the internal APIs from the clients. When establishing such an entry-point to your system, you can also use it to implement cross-cutting concerns, such as security, monitoring, and resilience.

This chapter will teach you how to use Spring Cloud Gateway to build an Edge Service application, implement an API gateway, and some of those cross-cutting concerns. You'll improve the resilience of the system by configuring circuit breakers with Spring Cloud Circuit Breaker, defining rate limiters with Spring Data Redis Reactive, and using retries and timeouts just like you learned in the previous chapter.

Next, I'll cover how to design stateless applications. Some states will need to be saved for the applications to be useful. You have already used relational databases. This chapter will teach you how to store the web session state using Spring Session Redis, a NoSQL, in-memory data store.

Finally, you'll see how to manage external access to the applications running in a Kubernetes cluster relying on the Kubernetes Ingress API.

Figure 9.1 shows how the Polar Bookshop system will look after completing this chapter.

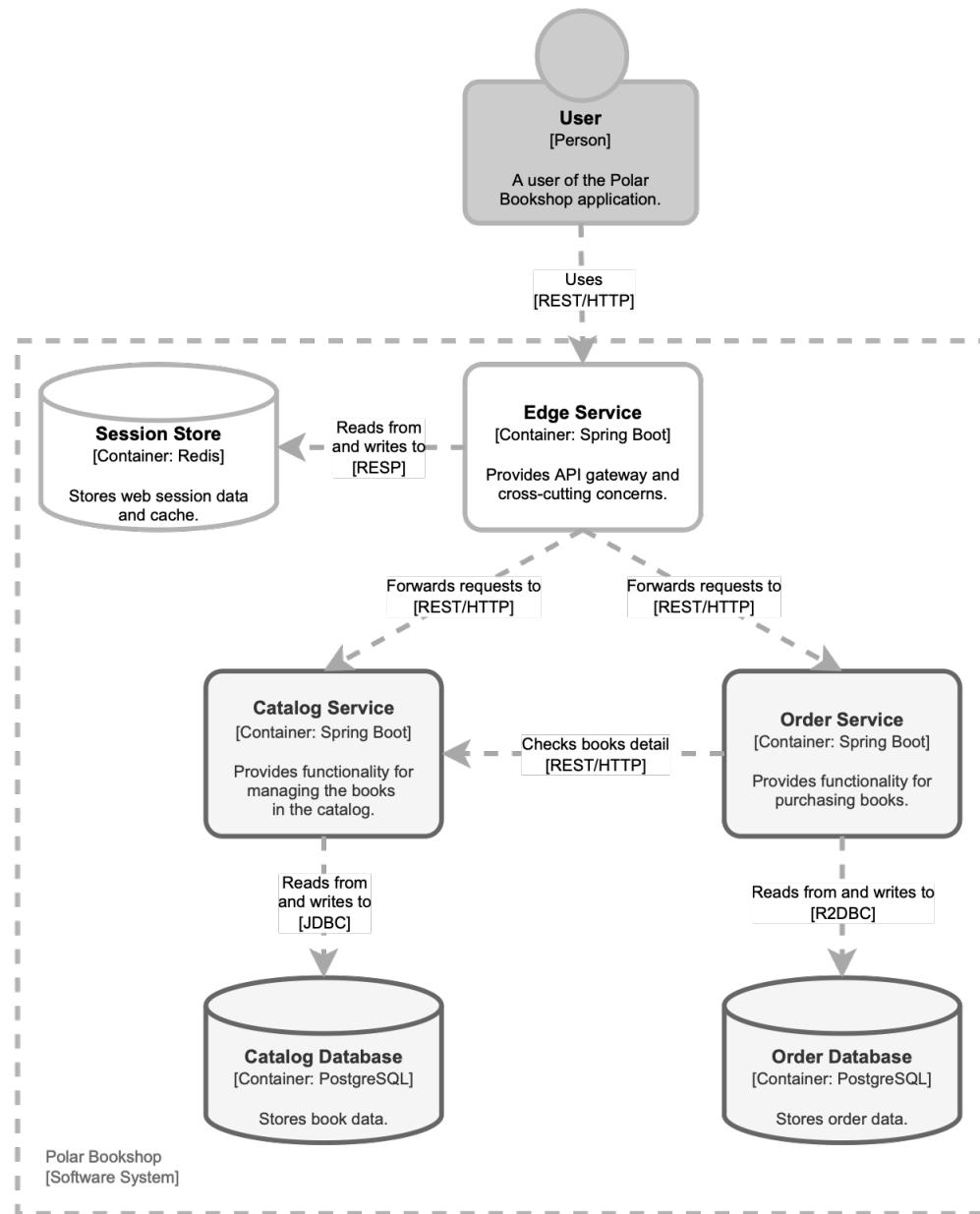


Figure 9.1 The architecture of the Polar Bookshop system after adding Edge Service and Redis.

NOTE

The source code for the examples in this chapter is available in the Chapter09/09-begin and Chapter09/09-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

9.1 Edge server with Spring Cloud Gateway

Spring Cloud Gateway is a project built on top of Spring WebFlux and Project Reactor to provide an API gateway and a central place to implement cross-cutting concerns like security, resilience, and monitoring. It's built for developers and is a good fit in Spring architectures and heterogeneous environments.

An API gateway provides an entry point to your system. In distributed systems like microservices, that's a convenient way to decouple the clients from any change to the internal services APIs. You're free to change how your system is decomposed into services and their APIs, relying on the fact that the gateway can translate from a more stable, client-friendly, public API to the internal one.

Suppose you're in the process of moving from a monolith to microservices. In that case, an API gateway can be used as a **monolith strangler** and wrap your legacy applications until they are migrated to the new architecture so that it's transparent to clients. In case of different client types (single page applications, mobile applications, desktop applications, IoT devices), an API gateway gives you the option to provide a better-crafted API to each of them depending on their needs (also called the **backend-for-frontend** pattern). Sometimes, a gateway can also implement the **API composition** pattern, letting you query and join data from different services before returning the result to a client (for example, using the new Spring for GraphQL project).

Calls are forwarded to downstream services from the gateway according to given routing rules, similar to a reverse proxy. It hides from the client the need to keep track of the different services involved in a transaction, simplifying the client logic, and reducing the number of calls it has to do.

Since it's the entry point to your system, an API gateway can also be an excellent place to implement cross-cutting concerns like security, monitoring, and resilience. We call **edge servers** applications at the edge of a system that implement aspects like API gateways and cross-cutting concerns. You can configure circuit breakers to prevent cascading failures when invoking the services downstream. You can define retries and timeouts for all the calls to the internal services. You can control the ingress traffic and enforce quota policies to limit the usage of your system depending on some criteria (for example, the membership level of your users: basic, premium, pro). You can also implement authentication and authorization at the edge and pass tokens to downstream services (as you'll see in chapters 11 and 12).

However, we should consider that an edge server adds complexity to the system. It's another component to build, deploy, and manage in production. It also adds a new network hop to the system. As a consequence, the response time will increase. Most of the time, that's an insignificant cost, yet you should keep it in mind. Since it's the entry point to the system, it's at risk of becoming a single point of failure. As a basic mitigation strategy, you should deploy at

least two replicas of an edge server following the same approach we discussed for configuration servers in chapter 4.

Spring Cloud Gateway greatly simplifies building edge services, focusing on simplicity and productivity. Furthermore, since it's based on a reactive stack, it can scale efficiently to handle the high workload naturally happening at the edge of a system.

The following section will teach you how to set up an edge server with Spring Cloud Gateway. You'll learn about routes, predicates, and filters, which are the building blocks of the gateway. And you'll apply the retry and timeout patterns you learned in the previous chapter to the interaction between the gateway and the downstream services.

NOTE

If you haven't followed along with the examples implemented in the previous chapters, you can refer to the repository accompanying the book (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and use the project in Chapter09/09-begin as a starting point.

9.1.1 Bootstrapping an edge server with Spring Cloud Gateway

The Polar Bookshop system needs an edge server to route traffic to the internal APIs and address several cross-cutting concerns. You can initialize the Edge Service project from Spring Initializr (<https://start.spring.io>), store the result in a new `edge-service` Git repository, and push it to GitHub. The parameters for the initialization are shown in figure 9.2.

The screenshot shows the Spring Initializr web application. At the top, there's a logo and the text "spring initializr". Below the header, there are three main sections: "Project", "Language", and "Dependencies".

- Project:** "Gradle Project" is selected. Other options include "Maven Project".
- Language:** "Java" is selected. Other options include "Kotlin" and "Groovy".
- Dependencies:** A button labeled "ADD DEPENDENCIES..." with a "⌘ + B" keyboard shortcut.

In the "Dependencies" section, "Gateway" is selected, and "SPRING CLOUD ROUTING" is listed under it. A description states: "Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency."

Project Metadata:

- Group: com.polarbookshop
- Artifact: edge-service
- Name: edge-service
- Description: Provides API gateway and cross-cutting concerns.
- Package name: com.polarbookshop.edgeservice
- Packaging: "Jar" is selected. Other options are "War".
- Java version: "18" is selected. Other options are "17", "11", and "8".

Figure 9.2 The parameters for initializing the Edge Service project.

TIP

In the `begin` folder for this chapter, you can find a `curl` command to run in a Terminal window that downloads a zip file containing all the code you need to get started without going through the manual generation through the Spring Initializr website.

The `dependencies` section of the autogenerated `build.gradle` file looks as follows.

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-gateway'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- **Spring Cloud Gateway** (`org.springframework.cloud:spring-cloud-starter-gateway`) provides utilities to route to APIs and cross-cutting concerns like resilience, security, and monitoring. It's built on top of the Spring reactive stack.
- **Spring Boot Test** (`org.springframework.boot:spring-boot-starter-test`) provides several libraries and utilities to test applications, including Spring Test, JUnit, AssertJ, and Mockito. It's automatically included in every Spring Boot project.

At its core, Spring Cloud Gateway is a Spring Boot application. It provides all the convenient features we've been using in the previous chapters, such as auto-configuration, embedded servers, test utilities, externalized configuration, and so on. It's also built on the Spring reactive stack, so you can use the tools and patterns you learned in the previous chapter regarding Spring WebFlux and Reactor. Let's start by configuring the embedded Netty server.

First, rename the `application.properties` file generated by Spring Initializr (`edge-service/src/main/resources`) to `application.yml`. Then, open the file and configure the Netty server as you learned in the previous chapter.

Listing 9.1 Configure Netty server and graceful shutdown (application.yml)

```
server:
  port: 9000 ①
  netty:
    connection-timeout: 2s ②
    idle-timeout: 15s ③
    shutdown: graceful ④

spring:
  application:
    name: edge-service
  lifecycle:
    timeout-per-shutdown-phase: 15s ⑤
```

- ➊ The port where the server will accept connections.
- ➋ How long to wait for a TCP connection to be established with the server.
- ➌ How long to wait before closing a TCP connection if no data is transferred.

- ④ Enables graceful shutdown.
- ⑤ Defines a 15s grace period.

The application is set up, so you can move on and start exploring the features of Spring Cloud Gateway.

9.1.2 Defining routes and predicates

Spring Cloud Gateway provides three main building blocks:

- **Route.** It's identified by a unique ID, a collection of predicates deciding whether to follow the route, a URI for forwarding the request if the predicates allow, and a collection of filters applied either before or after forwarding the request downstream.
- **Predicate.** It matches anything from the HTTP request, including path, host, headers, query parameters, cookies, and body.
- **Filter.** It modifies an HTTP request or response before or after forwarding the request to the downstream service.

A client sends a request to Spring Cloud Gateway. If the request matches a route through its predicates, the `HandlerMapping` will send the request to the `WebHandler`, which in turn will run the request through a chain of filters.

There are two filter chains. One chain contains the filters to be run before the request is sent to the downstream service. The other chain is run after sending the request downstream and before forwarding the response. You'll learn about the different types of filters in the next section. Figure 9.3 shows how the routing works in Spring Cloud Gateway.

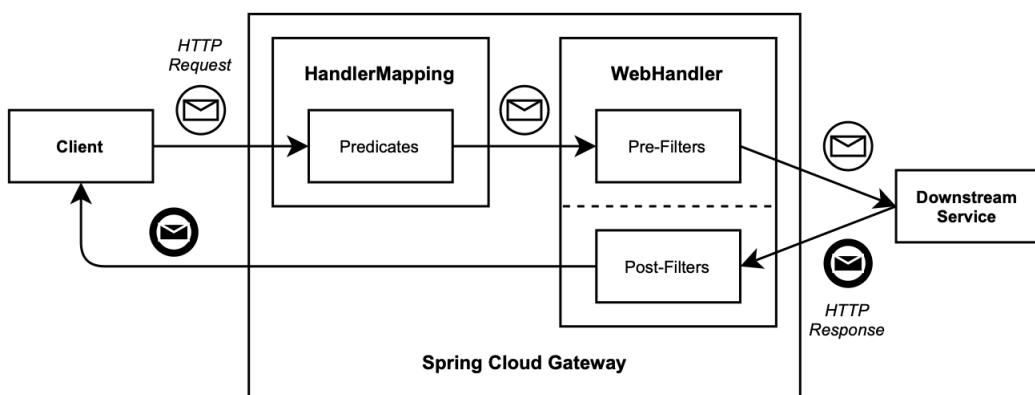


Figure 9.3 Requests are matched against predicates, filtered, and finally forwarded to the downstream service, which replies with a response going through another set of filters before being returned to the client.

In the Polar Bookshop system, we have built two applications with APIs that are meant to be accessible from the outside world (i.e., public APIs): Catalog Service and Order Service. We can use Edge Service to hide them behind an API gateway. For starters, we need to define the routes.

A minimal route must be configured with a unique ID, a URI where to forward the request, and at least one predicate. Open the `application.yml` file for the Edge Service project and configure two routes to Catalog Service and Order Service.

Listing 9.2 Configure routes to downstream services (`application.yml`)

```
spring:
  cloud:
    gateway:
      routes: ①
        - id: catalog-route ②
          uri: ${CATALOG_SERVICE_URL:http://localhost:9001}/books
          predicates:
            - Path=/books/** ③
        - id: order-route
          uri:
            ➔ ${ORDER_SERVICE_URL:http://localhost:9002}/orders ④
          predicates:
            - Path=/orders/**
```

- ① A list of route definitions.
- ② The route ID.
- ③ The predicate is a path to match.
- ④ The URI value comes from an environment variable, or else the default.

Both the routes for Catalog Service and Order Service are matched based on a `Path` predicate. All the incoming requests with a path starting with `/books` will be forwarded to Catalog Service. If the path starts with `/orders`, then Order Service will receive the request. The URIs are computed using the value from an environment variable (`CATALOG_SERVICE_URL` and `ORDER_SERVICE_URL`). If they are not defined, then the default value written after the first `:` symbol will be used. It's an alternate approach compared to how we defined URLs in the previous chapter, based on custom properties. I thought to show you both options.

The project comes built-in with many different predicates that you can use in your route configuration to match against any aspect of an HTTP request, including `Cookie`, `Header`, `Host`, `Method`, `Path`, `Query`, and `RemoteAddr`. You can also combine them to form *AND* conditionals. In the previous example, we used the `Path` predicate. Refer to the official documentation for an extensive list of predicates available in Spring Cloud Gateway: <https://spring.io/projects/spring-cloud-gateway>.

SIDE BAR**Defining routes with the Java/Kotlin DSL**

Spring Cloud Gateway is a very flexible project that lets you configure routes the way it best suits your needs. Here, you have configured routes in a property file (`application.yml` or `application.properties`), but there's also a DSL available to configure routes programmatically in Java or Kotlin. Future versions of the project will also implement a feature to fetch the route configuration from a data source using Spring Data.

Which way to use it is up to you. Routes in configuration properties give you the chance to customize them easily depending on the environment and update them at runtime without the need of re-building and re-deploying the application, for example, when using Spring Cloud Config Server. On the other hand, the DSL for Java and Kotlin lets you define more complex routes. Configuration properties allow you to combine different predicates with an AND logical operator only. The DSL also enables you to use other logical operators like OR and NOT.

Let's verify that it works as intended. We'll use Docker to run the downstream services and PostgreSQL, whereas we'll run Edge Service locally on the JVM to make it more efficient to work with it since we are actively implementing the application.

First, we need both Catalog Service and Order Service up and running. From each project root folder, run `./gradlew bootBuildImage` to package them as container images. Then, start them via Docker Compose. Open a Terminal window, navigate to the folder where your `docker-compose.yml` file is located (`polar-deployment/docker`), and run the following command. Since both applications depend on PostgreSQL, Docker Compose will also run the PostgreSQL container.

```
$ docker-compose up -d catalog-service order-service
```

When the downstream services are all up and running, it's time to start Edge Service. From a Terminal window, navigate to the project root folder (`edge-service`), and run the following command.

```
$ ./gradlew bootRun
```

The Edge Service application will start accepting requests on port 9000. For the final test, try executing operations on books and orders, but this time through the API gateway (that is, using the 9000 port rather than the individual ports to which Catalog Service and Order Service are listening). They should return a 200 OK response.

```
$ http :9000/books
$ http :9000/orders
```

The result is the same as if you called Catalog Service and Order Service directly, but you need to know only one hostname and port this time. When you are done testing the application, stop its execution with `Ctrl+C`. Then, terminate all the containers with Docker Compose.

```
$ docker-compose down
```

Under the hood, Edge Service uses Netty's HTTP client to forward requests to downstream services. As extensively discussed in the previous chapter, whenever an application calls an external service, it's essential to configure a timeout to make it resilient to inter-process communication failures. Spring Cloud Gateway provides dedicated properties to configure the HTTP client timeouts.

Open the Edge Service `application.yml` file once again, and define values for the connection timeout (the time limit for a connection to be established with the downstream service) and for the response timeout (time limit for receiving a response).

Listing 9.3 Configure timeouts for the gateway HTTP client (application.yml)

```
spring:
  cloud:
    gateway:
      httpclient: ①
        connect-timeout: 2000 ②
        response-timeout: 5s ③
```

- ① Configuration properties for the HTTP client.
- ② Time limit for a connection to be established (ms).
- ③ Time limit for a response to be received (Duration).

By default, the Netty HTTP client used by Spring Cloud Gateway is configured with an *elastic* connection pool to increase the number of concurrent connections dynamically as the workload increases. Depending on the number of requests your system will receive simultaneously, you might want to switch to a *fixed* connection pool to have more control over the number of connections. You can configure the Netty connection pool in Spring Cloud Gateway through the `spring.cloud.gateway.httpclient.pool` property group.

Listing 9.4 Configure the connection pool for the gateway HTTP client (application.yml)

```
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 5000
        response-timeout: 5s
      pool:
        type: elastic ①
        max-idle-time: 15s ②
        max-life-time: 60s ③
```

- ① Type of connection pool. It can be elastic, fixed, or disabled.
- ② Idle time after which the communication channel will be closed.
- ③ Time after which the communication channel will be closed.

You can refer to the official Reactor Netty documentation for more details about how the connection pool works, what configuration is available, and tips on the values to use based on the specific scenario (<https://projectreactor.io/docs>).

In the next section, we'll start implementing something more interesting than merely forwarding requests and discovering the power of Spring Cloud Gateway filters.

9.1.3 Processing requests and responses through filters

Routes and predicates alone make the application act as a proxy, but it's filters that make Spring Cloud Gateway really powerful. Filters can run before forwarding incoming requests to a downstream application (*pre-filters*). For example, they're used for:

- manipulating the request headers;
- applying rate limiting and circuit breaking;
- defining retries and timeouts for the proxied request;
- triggering an authentication flow with OAuth2 and OpenID Connect.

Other filters can apply to outgoing responses after being received from the downstream application and before sending them back to the client (*post-filters*). For example, they're used for:

- setting security headers;
- manipulating the response body to remove sensitive information.

Spring Cloud Gateway comes bundled with many filters that you can use to perform different actions, including adding headers to a request, configuring a circuit breaker, saving the web session, retrying the request on failure, or activating a rate limiter.

In the previous chapter, you learned how to use the retry pattern to improve application resilience. You'll now learn how to apply it as a default filter for all GET requests going through the routes defined in the gateway.

USING THE RETRY FILTER

You can define default filters in the `application.yml` file located under `src/main/resources`. One of the filters provided by Spring Cloud Gateway is the `Retry` filter. The configuration is similar to what we did in chapter 8.

Let's define a maximum of 3 retry attempts for all GET requests whenever the error is in the 5xx range (SERVER_ERROR). We don't want to retry requests when the error is in the 4xx range. For example, if the result is a 404 response, it doesn't make sense to retry the request. We can also list the exceptions for which a retry should be attempted, for example, `IOException` and `TimeoutException`.

By now, you know that you shouldn't keep retrying requests one after the other. You should use a backoff strategy instead. By default, the delay is computed using the formula `firstBackoff * (factor ^ n)`. If you set the `basedOnPreviousValue` parameter to `true`, the formula will be `prevBackoff * factor`.

Listing 9.5 Apply the retry filter to all routes (application.yml)

```
spring:
  cloud:
    gateway:
      default-filters: ①
        - name: Retry ②
          args:
            retries: 3 ③
            methods: GET ④
            series: SERVER_ERROR ⑤
            exceptions: java.io.IOException,
java.util.concurrent.TimeoutException ⑥
          backoff: ⑦
            firstBackoff: 50ms
            maxBackOff: 500ms
            factor: 2
            basedOnPreviousValue: false
```

- ① A list of default filters.
- ② The name of the filter.
- ③ Maximum of 3 retry attempts.
- ④ Retries only GET requests.
- ⑤ Retries only when 5xx errors.
- ⑥ Retries only when the given exceptions are thrown.
- ⑦ Retries with a delay computed as "firstBackoff * (factor ^ n)".

The retry pattern is useful when a downstream service is momentarily unavailable. What if it stays down for more than a few instants? At that point, we could directly stop forwarding requests to it until we're sure that it's back. Keeping sending requests wouldn't be beneficial for the caller nor the callee. In that scenario, the circuit breaker pattern comes in handy. That's the topic of the next section.

9.2 Fault tolerance with Spring Cloud Circuit Breaker and Resilience4J

As we know, resilience is a critical property of cloud native applications. One of the principles for achieving resilience is blocking a failure from cascading and affecting other components. Consider a distributed system where an application X depends on an application Y. If application Y fails, will application X fail, too? A circuit breaker can block a failure in one component from propagating to the ones depending on it, protecting the rest of the system. That is accomplished by temporarily stopping the communication with the faulty component until it recovers. This pattern comes from electrical systems for which the circuit is physically open to avoid destroying the entire house when a part of the system fails due to current overload.

In the world of distributed systems, you can establish circuit breakers at the integration points between the components. Think about Edge Service and Catalog Service. In a typical scenario, the circuit is **closed**, meaning that the two services can interact over the network. For each server error response returned by Catalog Service, the circuit breaker in Edge Service would register the failure. When the number of failures exceeds a certain threshold, the circuit breaker trips, and the circuit transitions to **open**.

While the circuit is open, communications between Edge Service and Catalog Service are not allowed. Any request that should be forwarded to Catalog Service will fail right away. In this state, either an error is returned to the client, or fallback logic is executed. After an appropriate amount of time to give the system enough time to recover, the circuit breaker transitions to a **half-open** state, allowing the next call to Catalog Service to go through. That is an exploratory phase to check if there are still issues in contacting the downstream service. If it succeeds, then the circuit breaker is reset, and transitions to *closed*. Otherwise, it goes back to being *open*. Figure 9.4 shows how a circuit breaker changes state.

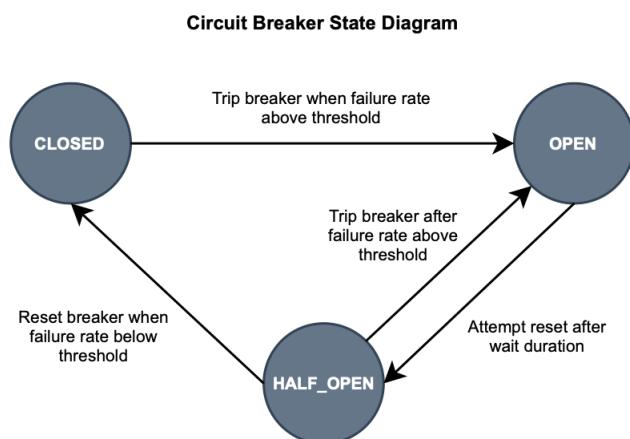


Figure 9.4 A circuit breaker ensures fault tolerance when a downstream service exceeds the maximum number of failures allowed by blocking any communication between upstream and downstream services. The logic is based on three states: **closed**, **open**, and **half-open**.

Unlike retries, when the circuit breaker trips, no calls to the downstream service are allowed anymore. Like retries, its behavior depends on a threshold and a timeout, and it lets you define a fallback method to call. The goal of resilience is to keep a system available to its users, even in the face of failures. In the worst-case scenario, like when a circuit breaker trips, you should guarantee a graceful degradation. You can adopt different strategies for the fallback method. For example, you might decide to return a default value or the last available value from a cache in case of a GET request.

The Spring Cloud Circuit Breaker project provides an abstraction for defining circuit breakers in a Spring application. You can choose between a reactive and non-reactive implementation based on Resilience4J (<https://resilience4j.readme.io>). Netflix Hystrix was the popular choice for microservices architectures, but it entered the maintenance mode back in 2018. After that, Resilience4J became the preferred choice because it provides the same features offered by Hystrix and more.

Spring Cloud Gateway integrates natively with Spring Cloud Circuit Breaker, providing you with a `CircuitBreaker` gateway filter that you can use to protect the interactions with all the downstream services. In the following sections, you'll configure a circuit breaker for the routes to Catalog Service and Order Service from Edge Service.

9.2.1 Introducing circuit breakers with Spring Cloud Circuit Breaker

To use Spring Cloud Circuit Breaker in Spring Cloud Gateway, you need to add a dependency to the specific implementation you'd like to use. In this case, we'll use the Resilience4J reactive version. Go ahead and add the new dependency in the `build.gradle` file for the Edge Service project (`edge-service`). Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 9.6 Add dependency for Spring Cloud Circuit Breaker and Resilience4J (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.cloud:spring-cloud-starter-circuitbreaker-reactor-resilience4j'
}
```

The `CircuitBreaker` filter in Spring Cloud Gateway relies on Spring Cloud Circuit Breaker to wrap a route. Similar to the `Retry` filter, you can choose to apply it to specific routes or define it as a default filter. Let's go with the first option. You can also specify an optional `fallbackUri` to handle the request when the circuit is in an open state. In this example, both routes will be configured with a `CircuitBreaker` filter, but only `catalog-route` will have a `fallbackUri` value so that I can show you both scenarios.

Listing 9.7 Configure circuit breakers for the gateway routes (application.yml)

```

spring:
  cloud:
    gateway:
      routes:
        - id: catalog-route
          uri: ${CATALOG_SERVICE_URL:http://localhost:9001}/books
          predicates:
            - Path=/books/**
          filters:
            - name: CircuitBreaker ①
              args:
                name: catalogCircuitBreaker ②
                fallbackUri: forward:/catalog-fallback ③
        - id: order-route
          uri: ${ORDER_SERVICE_URL:http://localhost:9002}/orders
          predicates:
            - Path=/orders/**
          filters:
            - name: CircuitBreaker ④
              args:
                name: orderCircuitBreaker

```

- ① Name of the filter.
- ② Name of the circuit breaker.
- ③ Forwards request to this URI when the circuit is open.
- ④ No fallback defined for this circuit breaker.

The next step is configuring the circuit breaker.

9.2.2 Configuring a circuit breaker with Resilience4J

After defining to which routes apply the `CircuitBreaker` filter, you need to configure the circuit breakers themselves. As often in Spring Boot, you have two main choices. You can configure circuit breakers through the properties provided by Resilience4J or via a `Customizer` bean. Since we're using the reactive version of Resilience4J, it would be a `Customizer<ReactiveResilience4JCircuitBreakerFactory>`.

Either way, you can choose to define a specific configuration for each circuit breaker you used in your `application.yml` file (in the example, `catalogCircuitBreaker` and `orderCircuitBreaker`) or declare some defaults that will be applied to all of them.

For the current example, we can define circuit breakers to consider a window of 20 calls (`slidingWindowSize`). Each new call will make the window move, dropping the oldest registered call. When at least 50% of the calls in the window produced an error (`failureRateThreshold`), the circuit breaker trips, and the circuit enters the open state. After 15 seconds (`waitDurationInOpenState`), the circuit is allowed to transition to a half-open state in which 5 calls are permitted (`permittedNumberOfCallsInHalfOpenState`). If at least 50% of

them result in an error, the circuit will go back to the open state. Otherwise, the circuit breaker trips to the close state.

Now, on to the code. In the Edge Service project (`edge-service`), at the end of the `application.yml` file, define a default configuration for all Resilience4J circuit breakers.

Listing 9.8 Configure circuit breaker and time limiter (application.yml)

```
resilience4j:
  circuitbreaker:
    configs:
      default: ①
        slidingWindowSize: 20 ②
        permittedNumberOfCallsInHalfOpenState: 5 ③
        failureRateThreshold: 50 ④
        waitDurationInOpenState: 15000 ⑤
    timelimiter:
      configs:
        default: ⑥
          timeoutDuration: 5s ⑦
```

- ① Default configuration bean for all circuit breakers.
- ② The size of the sliding window used to record the outcome of calls when the circuit is closed.
- ③ Number of permitted calls when the circuit is half-open.
- ④ When the failure rate is above the threshold, the circuit becomes open.
- ⑤ Waiting time before moving from open to half-open (ms).
- ⑥ Default configuration bean for all time limiters.
- ⑦ Configures a timeout (seconds).

We configure both the circuit breaker and a time limiter, a required component when using the Resilience4J implementation of Spring Cloud Circuit Breaker. The timeout configured via Resilience4J will take precedence over the response timeout we defined in the previous section for the Netty HTTP client (`spring.cloud.gateway.httpclient.response-timeout`).

When a circuit breaker switches to the open state, the least we can do is degrade the service level gracefully and make the user experience as pleasant as possible. I'll show you how to do that in the next section.

9.2.3 Defining fallback REST APIs with Spring WebFlux

When we added the `CircuitBreaker` filter to `catalog-route`, we defined a value for the `fallbackUri` property to forward the requests to the `/catalog-fallback` endpoint when the circuit is in an open state. Since the `Retry` filter is also applied to that route, the fallback endpoint will be invoked even when all retry attempts fail for a given request. It's time to define that endpoint.

As I mentioned in previous chapters, Spring supports defining REST endpoints either using `@RestController` classes or Router Functions. Let's use the functional way for declaring the fallback endpoints.

In a new `com.polarbookshop.edgeservice.web` package in the Edge Service project, create a new `WebEndpoints` class. Functional endpoints in Spring WebFlux are defined as routes in a `RouterFunction<ServerResponse>` bean, using the fluent API provided by `RouterFunctions`. For each route, you need to define the endpoint URL, a method, and a handler.

Listing 9.9 Configure fallback endpoints for when the Catalog Service is unavailable (`WebEndpoints.java`)

```
package com.polarbookshop.edgeservice.web;

import reactor.core.publisher.Mono;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerResponse;

@Configuration
public class WebEndpoints {

    @Bean ①
    public RouterFunction<ServerResponse> routerFunction() {
        return RouterFunctions.route() ②
            .GET("/catalog-fallback", request -> ③
                ServerResponse.ok().body(Mono.just(""), String.class))
            .POST("/catalog-fallback", request -> ④
                ServerResponse.status(HttpStatus.SERVICE_UNAVAILABLE).build())
            .build(); ⑤
    }
}
```

- ① Functional REST endpoints are defined in a bean.
- ② Offers a fluent API to build routes.
- ③ Fallback response used to handle the GET endpoint.
- ④ Fallback response used to handle the POST endpoint.
- ⑤ Builds the functional endpoints.

For simplicity, the fallback for GET requests returns an empty string, whereas the fallback for POST requests returns an HTTP 503 error. In a real scenario, you might want to adopt different fallback strategies depending on the context, including throwing a custom exception to be handled from the client or returning the last value saved in cache for the original request.

So far, we have used retries, timeouts, circuit breakers, and failovers (fallbacks). In the next section, I'll expand on how to work with all those resilience patterns together.

9.2.4 Combining circuit breakers, retries, and time limiters

When you combine multiple resilience patterns, the sequence in which they are applied is fundamental. Spring Cloud Gateway takes care of applying the `TimeLimiter` first (or the timeout on the HTTP client), then the `CircuitBreaker` filter, and finally `Retry`. Figure 9.5 shows how these patterns work together to increase the application resilience.

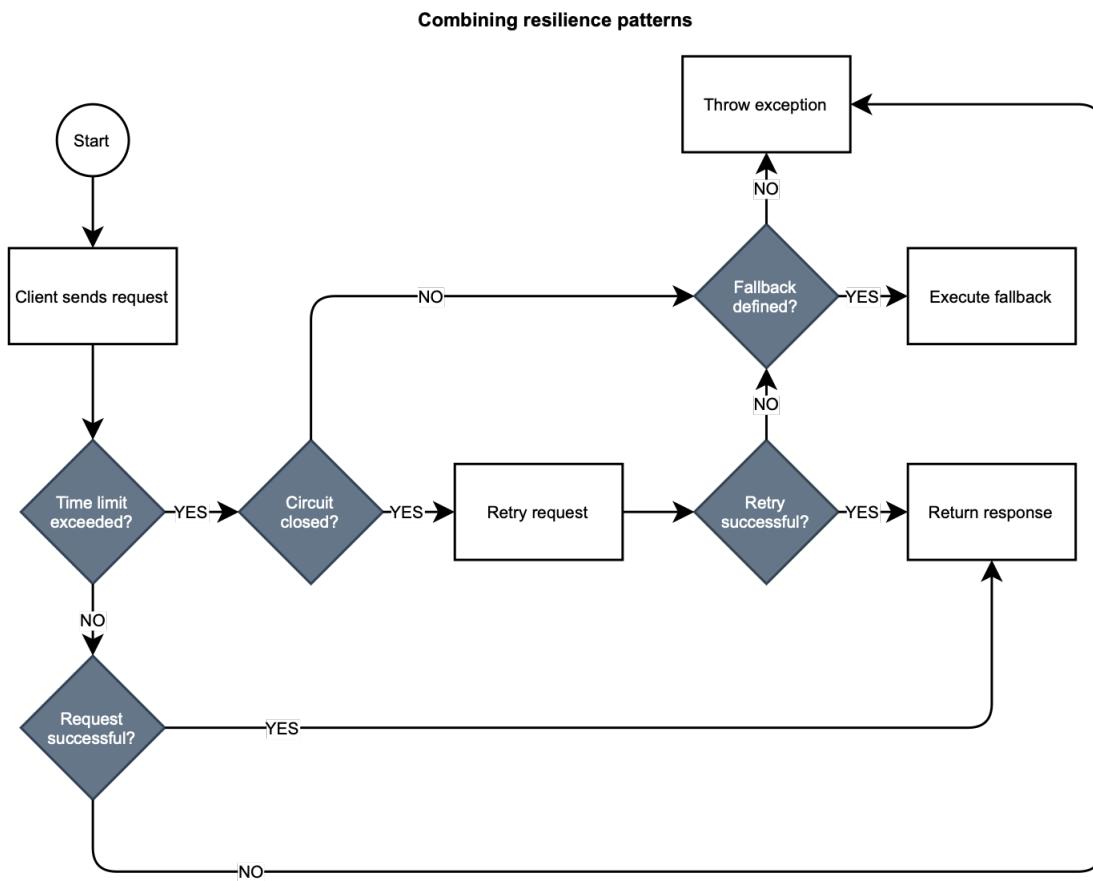


Figure 9.5 When multiple resilience patterns are implemented, they are applied in a specific sequence.

You can verify the result of applying these patterns to Edge Service using a tool like Apache Benchmark (<https://httpd.apache.org/docs/2.4/programs/ab.html>). If you’re using macOS or Linux, you might have this tool already installed. Otherwise, you can follow the instructions on the official website and install it.

Make sure both Catalog Service and Order Service are not running so that we can test circuit breakers in a failure scenario. Then, enable debug logging for Resilience4J to follow the state transitions of the circuit breaker. At the end of the `application.yml` file in your Edge Service project, add the following configuration.

Listing 9.10 Enable debug logging for Resilience4J (application.yml)

```
logging:
  level:
    io.github.resilience4j: DEBUG
```

Next, build and run Edge Service (`./gradlew bootRun`). Since all downstream services are not running (if they are, you should stop them), all the requests sent to them from Edge Service will result in an error. Let's see what happens if we run 21 sequential POST requests (`-n 21 -c 1 -m POST`) to the `/orders` endpoint. Remember that POST requests have no retry configuration, and `order-route` has no fallback, so the result will only be affected by timeout and circuit breaker.

```
$ ab -n 21 -c 1 -m POST http://localhost:9000/orders
```

From the `ab` output, you can see that all requests returned an error.

```
Complete requests: 21
Non-2xx responses: 21
```

The circuit breaker is configured to trip to the open state when at least 50% of the calls in a 20-sized time window fails. Since you have just started the application, the circuit will transition to the open state after 20 requests. In the application logs, you can analyze how the requests have been handled. All requests fail, so the circuit breaker registers an `ERROR` event for each of them.

```
Event ERROR published: CircuitBreaker 'orderCircuitBreaker'
recorded an error.
```

At the 20th request, a `FAILURE_RATE_EXCEEDED` event is recorded because it exceeded the failure threshold. That will result in a `STATE_TRANSITION` event that will open the circuit.

```
Event ERROR published: CircuitBreaker 'orderCircuitBreaker'
recorded an error.
Event FAILURE_RATE_EXCEEDED published: CircuitBreaker 'orderCircuitBreaker'
exceeded failure rate threshold.
Event STATE_TRANSITION published: CircuitBreaker 'orderCircuitBreaker'
changed state from CLOSED to OPEN
```

The 21st request will not even try contacting Order Service: the circuit is open, so it cannot go through. A `NOT_PERMITTED` event is registered to signal why the request failed.

```
Event NOT_PERMITTED published: CircuitBreaker 'orderCircuitBreaker'
recorded a call which was not permitted.
```

NOTE

Monitoring the status of circuit breakers in production is a critical task. In chapter 13, I'll show you how to export that information as Prometheus metrics you can visualize in a Grafana dashboard instead of checking the logs. In the meantime, for a more visual explanation, feel free to watch my session on circuit breakers at GOTO Copenhagen (<http://mng.bz/1jyg>).

Now, let's see what happens when calling a GET endpoint for which both retries and fallback have been configured. Before proceeding, rerun the application to start with a clear circuit breaker state (`./gradlew bootRun`). Then, run the following command.

```
$ ab -n 21 -c 1 -m GET http://localhost:9000/books
```

If you check the application logs, you can see how the circuit breaker behaves precisely like before: 20 allowed requests (close circuit), followed by a non-permitted request (open circuit). However, the result of the previous command shows 21 requests completed with no errors. This time, all requests have been forwarded to the fallback endpoint, so the client didn't experience any error.

```
Complete requests: 21
Failed requests: 0
```

We configured the `Retry` filter to be triggered when an `IOException` or `TimeoutException` occurs. In this case, since the downstream service is not running, the exception thrown is of type `ConnectException`, so the request is conveniently not retried, allowing me to show you the combined behavior of circuit breakers and fallbacks without retries.

So far, we have looked at patterns to make the interaction between Edge Service and the downstream applications more resilient. What about the entry point of the system? The next section will introduce rate limiters to control the request flow coming into the system through the Edge Service application. Before proceeding, stop the application execution with `Ctrl+C`.

9.3 Request rate limiting with Spring Cloud Gateway and Redis

Rate limiting is a pattern used to control the rate of traffic sent to or received from an application, helping to make your system more resilient and robust. In the context of HTTP interactions, you can apply this pattern to control outgoing or incoming network traffic using client-side and server-side rate limiters, respectively.

Client-side rate limiters are for constraining the number of requests sent to a downstream service in a given period. It's a useful pattern to adopt when third-party organizations like cloud providers manage and offer the downstream service. You want to avoid incurring extra costs for having sent more requests than the ones allowed by your subscription. In case of pay-per-use services, it helps prevent unexpected expenses.

If the downstream service belongs to your system, you might use a rate limiter to avoid DoS yourself. In this last case, though, a *bulkhead* pattern (or *concurrent request limiter*) would be a better fit, setting constraints on how many concurrent requests are allowed and queuing up the blocked ones. Even better if it's an adaptive bulkhead, for which the concurrency limits are dynamically updated by an algorithm to better adapt to the elasticity of cloud infrastructures.

Server-side rate limiters are for constraining the number of requests received by an upstream service (or client) in a given period. This pattern is handy when implemented in an API gateway to protect the whole system from overloading or DoS attacks. When the number of users increases, the system should scale in a resilient way, ensuring an acceptable quality of service to all users. Sudden increases in user traffic are expected, and at the beginning, they are usually addressed by adding more resources to the infrastructure or more application instances. Over time, though, they can become a problem and even lead to service outages. Server-side rate limiters help with that.

When a user has exceeded the number of allowed requests in a specific time window, all the extra requests are rejected with an `HTTP 429 - Too Many Requests` status. The limit is applied according to a given strategy. For example, you can limit requests per session, per IP address, per user, or per tenant. The overall goal is to keep the system available for all users in case of adversities. That is the definition of resilience. This pattern is also handy for offering services to users depending on their subscription tiers. For example, you might define different rate limits for basic, premium, and enterprise users.

Resilience4J supports the client-side rate limiter and bulkhead patterns for both reactive and non-reactive applications. Spring Cloud Gateway supports the server-side rate limiter pattern. This section will show you how to use the server-side rate limiter pattern for Edge Service using Spring Cloud Gateway and Spring Data Redis Reactive. Let's start with setting up a Redis container.

9.3.1 Running Redis as a container

Imagine you want to limit the access to your API so that each user can only perform 10 requests per second. Implementing such a requirement would require a storage mechanism to track the number of requests each user performs every second. When the limit is reached, the following requests should be rejected. When the second is over, each user can perform 10 more requests within the next second. The data used by the rate-limiting algorithm is small and temporary, so you might think of saving it in memory inside the application itself.

However, that would make the application stateful and lead to errors since each application instance would limit requests based on a partial data set. It would mean letting users perform 10 requests per second per instance rather than overall because each instance would only keep track of its own incoming requests. The solution is using a dedicated data service to store the rate-limiting state and make it available to all the application replicas. Enter Redis.

Redis (<https://redis.com>) is an in-memory store that is commonly used as a cache, message broker, or database. In Edge Service, we'll use it as the data service backing the request rate limiter implementation provided by Spring Cloud Gateway. The Spring Data Reactive Redis project provides the integration between a Spring Boot application and Redis.

Let's first define a Redis container. Open the `docker-compose.yml` file you created in your `polar-deployment` repository. If you haven't followed along with the examples, you can use `Chapter09/09-begin/polar-deployment/docker/docker-compose.yml` from the source code accompanying the book as a starting point. Then, add a new service definition using the Redis official image and expose it through port 6379.

Listing 9.11 Define a Redis container (`docker-compose.yml`)

```
version: "3.8"
services:
  ...
  polar-redis:
    image: "redis:7.0"      ①
    container_name: "polar-redis"
    ports:
      - 6379:6379          ②
```

- ① Uses Redis 7.0.
- ② Exposes Redis through port 6379.

Next, open a Terminal window, navigate to the folder where your `docker-compose.yml` file is located, and run the following command to start a Redis container.

```
$ docker-compose up -d polar-redis
```

In the following section, you'll configure the Redis integration with Edge Service.

9.3.2 Integrating Spring with Redis

The Spring Data project has modules supporting several database options. In the previous chapters, we worked with Spring Data JDBC and Spring Data R2DBC to use relational databases. Now, we'll use Spring Data Redis, which provides support for this in-memory, non-relational data store. Both imperative and reactive applications are supported.

First, we need to add a new dependency on Spring Data Reactive Redis in the `build.gradle` file of the Edge Service project (`edge-service`). Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 9.12 Add dependency for Spring Data Reactive Redis in Edge Service (`build.gradle`)

```
dependencies {
  ...
  implementation
  'org.springframework.boot:spring-boot-starter-data-redis-reactive'
}
```

Then, in the `application.yml` file, configure the Redis integration through the properties provided by Spring Boot. Besides `spring.redis.host` and `spring.redis.port` for defining where to reach Redis, you can also specify connection and read timeouts using

`spring.redis.connect-timeout` and `spring.redis.timeout` respectively.

Listing 9.13 Configure the Redis integration (application.yml)

```
spring:
  redis:
    connect-timeout: 2s ①
    host: localhost ②
    port: 6379 ③
    timeout: 1s ④
```

- ① Time limit for a connection to be established.
- ② Default Redis host.
- ③ Default Redis port.
- ④ Time limit for a response to be received.

In the next section, you'll see how to use it to back the `RequestRateLimiter` gateway filter that provides server-side rate limiting support.

9.3.3 Configuring a request rate limiter

Depending on the requirements, you can configure the `RequestRateLimiter` filter for specific routes or as a default filter. In this case, we'll configure it as a default filter so that it's applied to all routes, current and future.

The implementation of `RequestRateLimiter` on Redis is based on the **token bucket algorithm**. Each user is assigned a bucket inside which tokens are dripped over time at a specific rate (*replenish rate*). Each bucket has a maximum capacity (*burst capacity*). When a user makes a request, a token is removed from its bucket. When there are no more tokens left, the request is not permitted, and the user will have to wait until more tokens are dripped into its bucket.

NOTE

If you want to know more about the token bucket algorithm, I recommend reading Paul Tarjan's article about how they use it to implement rate limiters at Stripe. (<https://stripe.com/blog/rate-limiters>)

For this example, let's configure the algorithm so that each request costs 1 token (`redis-rate-limiter.requestedTokens`). Tokens are dripped in the bucket following the configured replenish rate (`redis-rate-limiter.replenishRate`). Let's make it 10 tokens per second. Sometimes, there might be some spikes resulting in a larger number of requests than usual. You can allow temporary bursts by defining a larger capacity for the bucket (`redis-rate-limiter.burstCapacity`), for example, 20. It means that when a spike occurs, up to 20 requests are allowed per second. Since the replenish rate is lower than the burst capacity,

subsequent bursts are not allowed. If two spikes happen sequentially, only the first one will succeed, while the second one will result in some requests being dropped with an `HTTP 429 - Too Many Requests` response.

Listing 9.14 Configure a request rate limiter as a gateway filter (`application.yml`)

```
spring:
  cloud:
    gateway:
      default-filters:
        name: RequestRateLimiter
        args:
          redis-rate-limiter:
            replenishRate: 10 ①
            burstCapacity: 20 ②
            requestedTokens: 1 ③
```

- ① Number of tokens dripped in the bucket each second.
- ② Allows request bursts of up to 20 requests.
- ③ How many tokens a request costs.

There's no general rule to follow in coming up with good numbers for the request rate limiter. You should start from your application requirements and go with a trial and error approach: analyze your production traffic, tune the configuration, and then all over again until you achieve a good setup that keeps your system available while not affecting the user experience badly. Even after that, you should keep monitoring the status of your rate limiters since things can change in the future.

Spring Cloud Gateway relies on Redis to keep track of the number of requests happening each second. By default, each user is assigned a bucket. However, we haven't introduced any authentication mechanism yet, so we'll use a single bucket for all requests until we address the security concern in chapters 11 and 12.

NOTE

What happens if Redis becomes unavailable? Spring Cloud Gateway has been built with resilience in mind. It will keep its service level, but the rate limiters will be momentarily disabled until Redis is up and running again.

The `RequestRateLimiter` filter relies on a `KeyResolver` bean to derive the bucket to use per request. By default, it uses the currently authenticated user in Spring Security. Until we add security to Edge Service, let's define a custom `KeyResolver` bean and make it return a constant value (for example, `anonymous`) so that any request will be mapped to the same bucket.

In your Edge Service project, create a `RateLimiterConfig` class in a new `com.polarbookshop.edgeservice.config` package, and declare a `KeyResolver` bean implementing a strategy to return a constant key.

Listing 9.15 Define a strategy to resolve the bucket to use for each request (RateLimiterConfig.java)

```
package com.polarbookshop.edgeservice.config;

import reactor.core.publisher.Mono;
import org.springframework.cloud.gateway.filter.ratelimit.KeyResolver;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RateLimiterConfig {

    @Bean
    public KeyResolver keyResolver() {
        return exchange -> Mono.just("anonymous"); ①
    }
}
```

- ① Rate limiting is applied to requests using a constant key.

Spring Cloud Gateway is configured to append headers with details about the rate-limiting to each HTTP response that we can use to verify its behavior. Re-build and run Edge Service (`./gradlew bootRun`), then try calling one of the endpoints.

```
$ http :9000/books
```

The response body depends on whether Catalog Service is running or not, but that doesn't matter in this example. The interesting aspect to notice is the HTTP headers of the response. They show the rate limiter's configuration and the number of remaining requests allowed within the time window (i.e., one second).

```
HTTP/1.1 200 OK
Content-Type: application/json
X-RateLimit-Burst-Capacity: 20
X-RateLimit-Remaining: 19
X-RateLimit-Replenish-Rate: 10
X-RateLimit-Requested-Tokens: 1
```

You might not want to expose this information to clients in some cases where the information could help bad actors craft attacks against your system. Or you might need different header names. Either way, you can use the `spring.cloud.gateway.redis-rate-limiter` property group to configure that behavior. When you're done testing the application, stop it with `Ctrl+C`.

NOTE When the rate limiter pattern is combined with other patterns like time limiters, circuit breakers, and retries, the rate limiter is applied first. If the user request exceeds the rate limit, then it is rejected right away.

Redis is an efficient data store ensuring fast data access, high availability, and resilience. In this section, we used it to provide storage for the rate limiters. The following section will show you

how to use it in another common scenario: session management.

9.4 Distributed session management with Redis

In the previous chapters, I highlighted many times how cloud native applications should be stateless. We scale them in and out. If they weren't stateless, we would lose the state every time an instance is shut down. Some state needs to be saved, or else the applications would probably be useless. Catalog Service and Order Service are stateless themselves but rely on a stateful service (the PostgreSQL database) to permanently store the data about books and orders. Even if the applications are shut down, the data will survive and be accessed by all the application instances.

Edge Service is not dealing with any business entity to store, but it still needs a stateful service (Redis) to store the state related to the `RequestRateLimiter` filter. When Edge Service is replicated, it's important to keep track of how many requests are left before exceeding the threshold. Using Redis, the rate limiter functionality is guaranteed consistently and safely.

Furthermore, in chapter 11, you'll expand Edge Service to add authentication and authorization. Since it's the entry point to the Polar Bookshop system, it makes sense to authenticate the user there. Data about the authenticated session will have to be saved outside the application for the same reason as the rate limiter information. If it didn't, the user might have to authenticate themselves every time the request hits a different Edge Service instance.

The general idea is to keep the applications stateless and use data services for storing the state. As you learned in chapter 5, data services need to guarantee high availability, replication, and durability. In your local environment, you can ignore that aspect. In production, you'll rely on the data services offered by cloud providers, both for PostgreSQL and Redis.

The following section will cover how to work with Spring Session Data Redis to establish distributed session management.

9.4.1 Handling sessions with Spring Session Data Redis

Spring provides session management features with the Spring Session project. By default, session data is stored in memory, but that's not feasible in a cloud native application. You want to keep it in an external service so that they survive the application shutdown. Another fundamental reason for using a *distributed session store* is that you usually have multiple instances for the same application. You want them to access the same session data to provide a seamless experience to the user.

Redis is a popular option for session management and is supported by Spring Session Data Redis. Furthermore, you have already set it up for the rate limiters. You can add it to Edge Service with minimal configuration.

First, you need to add a new dependency on Spring Session Data Redis to the `build.gradle` file for the Edge Service project. You can also add the Testcontainers library for using a lightweight Redis container when writing integration tests. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 9.16 Add dependency for Spring Session and Testcontainers in Edge Service (build.gradle)

```
ext {
    ...
    set('testcontainersVersion', "1.17.2")
}

dependencies {
    ...
    implementation 'org.springframework.session:spring-session-data-redis'
    testImplementation 'org.testcontainers:junit-jupiter'
}

dependencyManagement {
    imports {
        ...
        mavenBom
        "org.testcontainers:testcontainers-bom:${testcontainersVersion}"
    }
}
```

Then, you need to instruct Spring Boot to use Redis for session management (`spring.session.store-type`) and define a unique namespace to prefix all session data coming from Edge Service (`spring.session.redis.namespace`). You can also define a timeout for the session (`spring.session.timeout`). If you don't specify any, the default is 30 minutes.

Configure Spring Session in the `application.yml` file as follows.

Listing 9.17 Configure Spring Session to store data in Redis (application.yml)

```
spring:
  session:
    store-type: redis
    timeout: 10m
    redis:
      namespace: polar:edge
```

Managing web sessions in a gateway requires some additional care to ensure we save the right state at the right time. In this example, we want the session to be saved in Redis before forwarding a request downstream. How can we do that? If you were thinking about whether there's a gateway filter for it, you would be right!

In the `application.yml` file for the Edge Service project, add `SaveSession` as a default filter to instruct Spring Cloud Gateway always to save the web session before forwarding requests downstream.

Listing 9.18 Configure the gateway to save the session data before forwarding requests (application.yml)

```
spring:
  cloud:
    gateway:
      default-filters:
        - SaveSession ①
```

- ① Enforces the session data to be saved before forwarding a request downstream.

That's a critical point when Spring Session is combined with Spring Security. Chapters 11 and 12 will cover more details about session management. For now, let's set up an integration test to verify the Spring context in Edge Service loads correctly, including the integration with Redis.

The approach is similar to the one used for defining PostgreSQL test containers in the previous chapter. Let's extend the existing `EdgeServiceApplicationTests` class generated by Spring Initializr, and configure a Redis test container. For this example, it's enough to verify that the Spring context loads correctly when Redis is used for storing web sessions-related data.

Listing 9.19 Define a Redis test container to test the Spring context loading (EdgeServiceApplicationTests.java)

```
package com.polarbookshop.edgeservice;

import org.junit.jupiter.api.Test;
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
import org.testcontainers.utility.DockerImageName;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;

@SpringBootTest( ❶
    webEnvironment = SpringApplication.WebEnvironment.RANDOM_PORT
)
@Testcontainers ❷
class EdgeServiceApplicationTests {

    private static final int REDIS_PORT = 6379;

    @Container
    static GenericContainer<?> redis = ❸
        new GenericContainer<?>(DockerImageName.parse("redis:7.0"))
            .withExposedPorts(REDIS_PORT);

    @DynamicPropertySource ❹
    static void redisProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.redis.host",
            () -> redis.getHost());
        registry.add("spring.redis.port",
            () -> redis.getMappedPort(REDIS_PORT));
    }

    @Test
    void verifyThatSpringContextLoads() { ❺
    }
}
```

- ❶ Loads a full Spring web application context and a web environment listening on a random port.
- ❷ Activates automatic startup and cleanup of test containers.
- ❸ Defines a Redis container for testing.
- ❹ Overwrites the Redis configuration to point to the test Redis instance.
- ❺ Empty test used to verify the application context is loaded correctly and that a connection with Redis has been established successfully.

Finally, run the integration tests as follows.

```
$ ./gradlew test --tests EdgeServiceApplicationTests
```

Should you want to disable the session management through Redis in some of your tests, you can do so by setting the `spring.session.store-type` property to `none` in a specific test class using the `@TestPropertySource` annotation or in a property file to make it apply to all test

classes.

SIDE BAR
Polar Labs

Feel free to apply what you learned in the previous chapters and prepare the Edge Service application for deployment.

1. Add Spring Cloud Config Client to Edge Service to make it fetch configuration data from Config Service.
2. Configure the Cloud Native Buildpacks integration, containerize the application, and define the commit stage of the deployment pipeline, as you learned in chapters 3 and 6.
3. Write the Deployment and Service manifests for deploying Edge Service to a Kubernetes cluster.
4. Configure Tilt to automate the Edge Service deployment to your local Kubernetes cluster initialized with minikube.

You can refer to the `Chapter09/09-end` folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and deploying the backing services from the manifests available in the `Chapter09/09-end/polar-deployment/kubernetes/platform/development` folder with `kubectl apply -f services`.

9.5 Managing external access with Kubernetes Ingress

Spring Cloud Gateway helps define an edge service where to implement several patterns and cross-cutting concerns at the ingress point of a system. In the previous sections, you saw how to use it as an API gateway, implement resilience patterns like rate limiting and circuit breakers, and define distributed sessions. In chapters 11 and 12, we'll also add authentication and authorization features to Edge Service.

Edge Service represents the entry point to the Polar Bookshop system. However, when deployed in a Kubernetes cluster, it's only accessible from within the cluster itself. In chapter 7, we used the *port-forward* feature to expose a Kubernetes Service defined in a *minikube* cluster to your local computer. That's a useful strategy during development, but it's not suitable for production.

This section will cover how to manage external access to applications running in a Kubernetes cluster using the Ingress API.

NOTE

This section assumes you have gone through the tasks listed in the previous "Polar Labs" card and prepared Edge Service for deployment on Kubernetes.

9.5.1 Understanding Ingress API and Ingress Controller

When it comes to exposing applications inside a Kubernetes cluster, we can use a Service object of type `clusterIP`. That's what we did so far to make it possible for Pods to interact with each other within the cluster. For example, that's how Catalog Service Pods can communicate with the PostgreSQL Pod.

A Service object can also be of type `LoadBalancer`, which relies on an external load balancer provisioned by a cloud provider to expose an application to the Internet. We could define a `LoadBalancer` Service for Edge Service instead of the `clusterIP` one. When running the system in a public cloud, the vendor would provision a load balancer, assign a public IP address, and all the traffic coming from that load balancer would be directed to the Edge Service Pods. It's a flexible approach that lets you expose a service directly to the Internet and works with different types of traffic.

The `LoadBalancer` Service approach results in assigning a different IP address to each service we decide to expose to the Internet. Since services are directly exposed, we don't have a chance to apply any further network configuration such as TLS termination. We could configure HTTPS in Edge Service, route all traffic directed to the cluster through the gateway (even platform services that don't belong to Polar Bookshop), and apply further network configuration there. The Spring ecosystem provides everything we need to address those concerns, and it's probably what we would do in many scenarios. However, since we want to run our system on Kubernetes, we can manage those infrastructural concerns at the platform level and keep our applications simpler and more maintainable. That's where the Ingress API comes handy.

SIDE BAR

DEFINITION

An Ingress is an object that "manages external access to the services in a cluster, typically HTTP. Ingress may provide load balancing, SSL termination, and name-based virtual hosting". (<https://kubernetes.io/docs>)

An Ingress object acts as an entry point into a Kubernetes cluster and is capable of routing traffic from a single external IP address to multiple services running inside the cluster. We can use an Ingress object to perform load balancing, accept external traffic directed to a specific URL, and manage the TLS termination to expose the application services via HTTPS.

Ingress objects alone don't accomplish anything by themselves. We use an Ingress object to declare the *desired state* in terms of routing and TLS termination. The actual component that enforces those rules and routes traffic from outside the cluster to the applications inside is the *Ingress Controller*. Since multiple implementations are available, there's no default Ingress

Controller included in the core Kubernetes distribution, but it's up to you to install one. Ingress Controllers are applications usually built using reverse proxies like NGINX, HAProxy, or Envoy. Examples are Ambassador Emissary, Contour, and Ingress NGINX.

In production, the cloud platform or dedicated tools are used to configure an Ingress Controller. In a local environment, we need some additional configuration to make the routing work. For the Polar Bookshop example, we'll use Ingress NGINX (<https://github.com/kubernetes/ingress-nginx>) in both environments.

NOTE

There are two popular Ingress Controllers based on NGINX. The Ingress NGINX project (<https://github.com/kubernetes/ingress-nginx>) is developed, supported, and maintained in the Kubernetes project itself. It's open-source, and it's what we'll use in this book. The NGINX Controller (<https://www.nginx.com/products/nginx-controller>) is a product developed and maintained by the F5 NGINX company, and it comes with free and commercial options.

Let's see how to use the Ingress NGINX on our local Kubernetes cluster. An Ingress Controller is a workload just like any other application running on Kubernetes and can be deployed in different ways. The simplest option would be using `kubectl` to apply its deployment manifests to the cluster. Since we use `minikube` to manage a local Kubernetes cluster, we can rely on a built-in addon to enable the Ingress functionality based on Ingress NGINX.

First, let's start the `polar` local cluster we introduced in chapter 7. Since we configured `minikube` to run on Docker, make sure your Docker Engine is up and running.

```
$ minikube start --cpus 2 --memory 4g --driver docker --profile polar
```

Then, we can enable the ingress addon, which will make sure that Ingress NGINX is deployed to our local cluster.

```
$ minikube addons enable ingress --profile polar
```

In the end, you can get information about the different components deployed with Ingress NGINX as follows.

```
$ kubectl get all -n ingress-nginx
```

The previous command contains an argument we haven't encountered yet: `-n ingress-nginx`. That means that we want to fetch all objects created in the `ingress-nginx` *namespace*.

SIDE BAR**DEFINITION**

A Namespace is an abstraction used by Kubernetes to support isolation of groups of resources within a single cluster. Namespaces are used to organize objects in a cluster and provide a way to divide cluster resources. (<https://kubernetes.io/docs>)

We use namespaces to keep our cluster organized and define network policies to keep certain resources isolated for security reasons. So far, we've been working with the default namespace, and we'll keep doing that for all our Polar Bookshop applications. However, when it comes to platform services such as Ingress NGINX, we'll rely on dedicated namespaces to keep those resources isolated.

After Ingress NGINX is installed, let's go ahead and deploy the backing services used by our Polar Bookshop applications. Check the source code repository accompanying this book (Chapter 09/09-end) and copy the content of the polar-deployment/kubernetes/platform/development folder into the same path in your polar-deployment repository, overwriting any existing file we used in previous chapters. The folder contains basic Kubernetes manifests to run PostgreSQL and Redis.

Open a Terminal window, navigate to the kubernetes/platform/development folder located in your polar-deployment repository, and run the following command to deploy PostgreSQL and Redis in your local cluster.

```
$ kubectl apply -f services
```

You can verify the result with the following command.

```
$ kubectl get deployment
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
polar-postgres   1/1     1           1          73s
polar-redis     1/1     1           1          73s
```

TIP

For your convenience, I prepared a script for performing all the previous operations with a single command. You can run it to create a local Kubernetes cluster with minikube, enable the Ingress NGINX addon, and deploy the backing services used by Polar Bookshop. You'll find two create-cluster.sh and destroy-cluster.sh files in the kubernetes/platform/development folder that you have just copied over your polar-deployment repository. On macOS and Linux, you might need to make the scripts executable via the chmod +x create-cluster.sh command.

Let's conclude this section by packaging Edge Service as a container image and loading the

artifact to the local Kubernetes cluster. Open a Terminal window, navigate to the Edge Service root folder (`edge-service`), and run the following commands.

```
$ ./gradlew bootBuildImage
$ minikube image load edge-service --profile polar
```

In the next section, you'll define an Ingress object and configure it to manage external access to the Polar Bookshop system running in a Kubernetes cluster.

9.5.2 Working with Ingress objects

Edge Service takes care of application routing, but it should not be concerned with the underlying infrastructure and network configuration. Using an Ingress resource, we can decouple the two responsibilities. Developers would maintain Edge Service while the platform team would manage the Ingress Controller and the network configuration (perhaps relying on a service mesh like Linkerd or Istio). Figure 9.6 shows the deployment architecture of Polar Bookshop after introducing an Ingress.

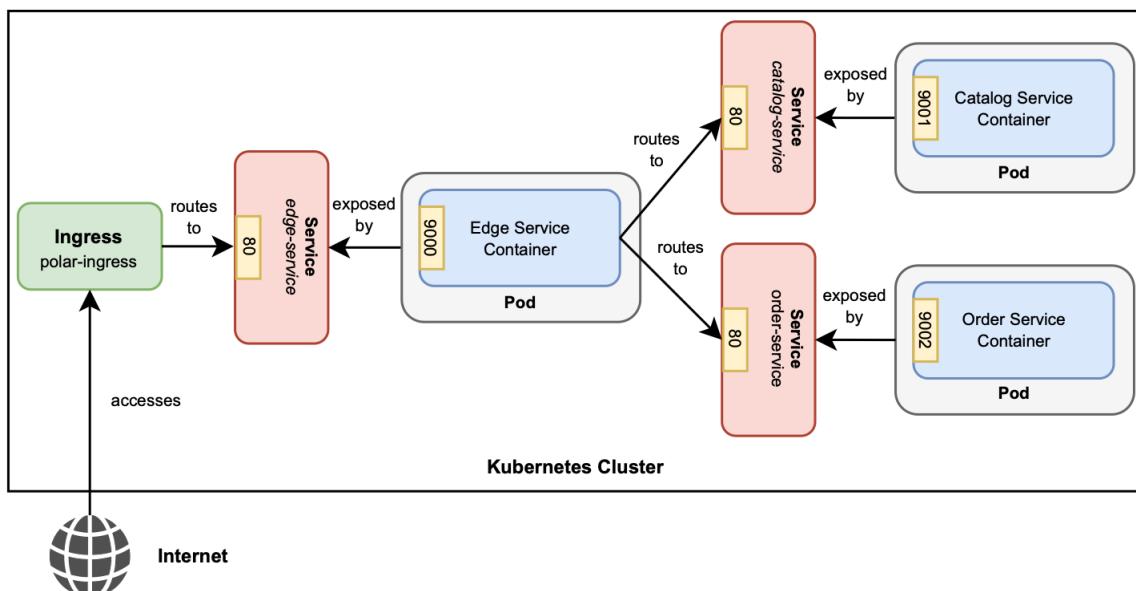


Figure 9.6 The deployment architecture of the Polar Bookshop system after introducing an Ingress to manage external access to the cluster.

Let's define an Ingress to route all HTTP traffic coming from outside the cluster to Edge Service. It's common to define Ingress routes and configurations based on the DNS name used to send the HTTP request. Since we are working locally and assuming we don't have a DNS name, we can call the external IP address provisioned for the Ingress to be accessible from outside the cluster. On Linux, you can use the IP address assigned to the `minikube` cluster. You can retrieve the value by running the following command.

```
$ minikube ip --profile polar
192.168.49.2
```

On macOS and Windows, the `ingress` addon doesn't support using the `minikube` cluster IP address yet when running on Docker. Instead, we need to use the `minikube tunnel --profile polar` command to expose the cluster to the local environment, and then use the `127.0.0.1` IP address to call the cluster. It's similar to the `kubectl port-forward` command, but it applies to the whole cluster instead of a specific service.

After identifying the IP address to use, let's define the Ingress object for Polar Bookshop. In the Edge Service project, create a new `ingress.yaml` file in the `k8s` folder.

Listing 9.20 Expose Edge Service outside the cluster via an Ingress (`ingress.yaml`)

```
apiVersion: networking.k8s.io/v1 ①
kind: Ingress ②
metadata:
  name: polar-ingress ③
spec:
  ingressClassName: nginx ④
  rules:
    - http: ⑤
      paths:
        - path: / ⑥
          pathType: Prefix
          backend:
            service:
              name: edge-service ⑦
              port:
                number: 80 ⑧
```

- ① The API version for Ingress objects.
- ② The type of object to create.
- ③ The name of the Ingress.
- ④ Configures the Ingress Controller responsible for managing this object.
- ⑤ Ingress rules for HTTP traffic.
- ⑥ A default rule for all requests.
- ⑦ The name of the Service object where to forward traffic.
- ⑧ The port number for the Service where to forward traffic.

At this point, we are ready to deploy Edge Service and the Ingress to the local Kubernetes cluster. Open a Terminal window, navigate to the Edge Service root folder (`edge-service`), and run the following command.

```
$ kubectl apply -f k8s
```

Let's verify the Ingress object has been created correctly with the following command.

```
$ kubectl get ingress
```

NAME	CLASS	HOSTS	PORTS	AGE
polar-ingress	nginx	*	80	21s

It's time to test that Edge Service is correctly available through the Ingress. If you're on Linux, you don't need any further preparation steps. If you're on macOS or Windows, open a new Terminal window and run the following command to expose your `minikube` cluster to your localhost. The command will continue running for the tunnel to be accessible, so make sure you keep the Terminal window open. The first time you run this command, you might be asked to input your machine password to authorize the tunneling to the cluster.

```
$ minikube tunnel --profile polar
```

Finally, open a new Terminal window and run the following command to test the application (on Linux, use the `minikube` IP address instead of `127.0.0.1`).

```
$ http 127.0.0.1/books
```

Since Catalog Service is not running, Edge Service will execute the fallback behavior we configured earlier and return a `200 OK` response with an empty body. That's what we expected, and it proves that the Ingress configuration works.

When you are done trying out the deployment, you can stop and delete the local Kubernetes cluster with the following commands.

```
$ minikube stop --profile polar
$ minikube delete --profile polar
```

TIP For your convenience, you can also use the `destroy-cluster.sh` script available in the `kubernetes/platform/development` folder of your `polar-deployment` repository that you copied earlier from the book source code. On macOS and Linux, you might need to make the script executable via the `chmod +x destroy-cluster.sh` command.

Good job! We're now ready to make Edge Service even better by adding authentication and authorization. Before configuring security, though, we still need to complete the Polar Bookshop business logic for dispatching orders. In the next chapter, you'll do that while learning event-driven architectures, Spring Cloud Function, and Spring Cloud Stream with RabbitMQ.

9.6 Summary

- An API gateway provides several benefits in a distributed architecture, including decoupling the internal services from the external API and offering a central, convenient place for implementing cross-cutting concerns like security, monitoring, and resilience.
- Spring Cloud Gateway is based on the Spring reactive stack, provides an API gateway implementation, and integrates with the other Spring projects to add cross-cutting concerns to the application, including Spring Security, Spring Cloud Circuit Breaker, and Spring Session.
- Routes are the core of Spring Cloud Gateway. They are identified by a unique ID, a collection of predicates deciding whether to follow the route, a URI towards forwarding the request if the predicates allow and a collection of filters applied before or after forwarding the request downstream.
- The `Retry` filter is for configuring retry attempts for specific routes.
- The `RequestRateLimiter` filter, integrated with Spring Data Redis Reactive, limits the number of requests that can be accepted within a specific time window.
- The `CircuitBreaker` filter, based on Spring Cloud Circuit Breaker and Resilience4J, defines circuit breakers, time limiters, and fallbacks to specific routes.
- Cloud native applications should be stateless. Data services should be used for storing the state. For example, PostgreSQL is used for persistence storage; Redis for cache and session data.
- A Kubernetes Ingress resource allows managing external access to applications running inside the Kubernetes cluster.
- The routing rules are enforced by an Ingress Controller, an application that also runs in the cluster.

10

Event-driven applications and functions

This chapter covers

- Understanding event-driven architectures
- Using RabbitMQ as the message broker
- Implementing functions with Spring Cloud Function
- Processing events with Spring Cloud Stream
- Producing and consuming events with Spring Cloud Stream

In the previous chapters, we worked on a system of distributed applications that interact according to the request/response pattern, a type of synchronous communication. You saw how to design the interaction both in an imperative and a reactive way. In the first case, processing threads would block waiting for a response from an IO operation. In the second case, threads would not wait. A response would be processed by any available thread asynchronously once received.

Even if the reactive programming paradigm lets you subscribe to producers and process the incoming data asynchronously, the interaction between the two applications is synchronous. The first application (the client) sends a request to the second one (the server) and expects a response to arrive in a short time. How the client processes the response (imperative or reactive) is an implementation detail that doesn't affect the interaction itself. No matter what, a response is expected to arrive.

Cloud native applications should be loosely coupled. The microservices expert Sam Newman identifies a few different types of coupling, including implementation, deployment, and temporal. Let's consider the Polar Bookshop system on which we've been working so far.

We can change the implementation of any of the applications without having to change the

others. For example, we can re-implement Catalog Service using the reactive paradigm without affecting Order Service. Using a service interface like a REST API, we hide the implementation details, improving loose coupling. All the applications can be deployed independently. They're not coupled, hence reducing risks and increasing agility.

However, if you think about how the applications we built so far interact, you'll notice they need other components of the systems to be available. Order Service needs Catalog Service to ensure that a user can order a book successfully. We know that failures happen all the time, so we adopted several strategies to ensure resilience even in the face of adversities, or at least a graceful degradation of functionality. That's a consequence of *temporal coupling*: Order Service and Catalog Service need to be available at the same time to fulfill the system requirements.

Event-driven architectures describe distributed systems that interact by *producing* and *consuming* events. The interaction is asynchronous, solving the problem of temporal coupling. This chapter will cover the basics of event-driven architectures and event brokers. You'll then learn how to implement business logic using the functional programming paradigm and Spring Cloud Function. Finally, you'll use Spring Cloud Stream to expose the functions as message channels via RabbitMQ, building event-driven applications through the publisher/subscriber model.

NOTE

The source code for the examples in this chapter is available in the Chapter10/10-begin, Chapter10/10-intermediate, and Chapter10/10-end folders, containing the initial, intermediate, and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

10.1 Event-driven architectures

An event is an occurrence. It's something relevant that happened in a system, like a state change. There can be many sources of events. This chapter will focus on applications, but events can very well be happening in IoT devices, sensors, or networks. When an event occurs, interested parties can be notified. Event notification is usually done through messages, which are a data representation of an event.

In an event-driven architecture, we identify event producers and event consumers. A producer is a component detecting the event and sending a notification. A consumer is a component getting notified when a specific event occurs. Producers and consumers don't know each other and work independently. A producer sends an event notification by publishing a message to a channel operated by an event broker responsible for collecting and routing messages to consumers. A consumer is notified by the broker when an event occurs and can act upon it.

Producers and consumers have minimal coupling when using a broker that takes the processing and distribution of events on itself. In particular, they are temporally decoupled because the interaction is asynchronous. Consumers can fetch and process messages at any time without

affecting the producers whatsoever.

In this section, you'll learn the fundamentals of event-driven models and how they can help build more resilient and loosely coupled applications in the cloud.

10.1.1 Understanding the event-driven models

Event-driven architectures can be based on two main models:

- **Publisher/Subscriber** (Pub/Sub). This model is based on subscriptions. Producers publish events that are sent to all subscribers to be consumed. Events cannot be replayed after being received, so new consumers joining will not be able to get the past events.
- **Event Streaming**. In this model, events are written to a log. Producers publish events as they occur, and they are all stored in an ordered fashion. Consumers don't subscribe to them, but they can read from any part of the event stream. In this model, events can be replayed. Clients can join at any time and receive all the past events.

In a basic scenario, consumers receive and process events as they arrive. For specific use cases like pattern matching, they can also process a series of events over a time window. In the event streaming model, consumers have the additional possibility to process event streams. At the core of event-driven architectures, there are platforms that can process and route events. For example, RabbitMQ is a common choice in combination with the pub/sub model. Apache Kafka is a powerful platform for event stream processing.

The event streaming model is fascinating and growing in popularity thanks to the many technologies developed in the last few years, allowing you to build real-time data pipelines. It's a complex model, though, that deserves its own book to be taught effectively. In this chapter, I will cover the pub/sub model.

Before analyzing this model in more detail, I'll define some requirements for the Polar Bookshop system to use as a means to explore event-driven architectures with the pub/sub model.

10.1.2 Event-driven with pub/sub

In the Polar Bookshop system, we need to implement an event-driven solution to allow different applications to communicate with each other asynchronously while reducing their coupling. The requirements are the following.

- When an order is accepted:
 - Order Service should notify interested consumers of the event;
 - Dispatcher Service should execute some logic to dispatch the order.
- When an order is dispatched:
 - Dispatcher Service should notify consumers interested in such an event;
 - Order Service should update the order status in the database.

If you paid attention, you probably noticed that the requirements don't specify which applications Order Service should notify upon order creation. In our example, it will be just the new Dispatcher Service application interested in those events. Still, there might be more applications subscribing to the order creation events in the future. The beauty of it is that you can evolve a software system and add more applications without affecting the existing ones at all. For example, you could add a Mail Service that sends an email to users whenever an order they made has been accepted, and Order Service wouldn't even be aware of it.

This type of interaction should be asynchronous and can be modeled with the pub/sub model. Figure 10.1 illustrates the interaction and describes three flows for accepting, dispatching, and updating an order. They are temporally decoupled and executed asynchronously.

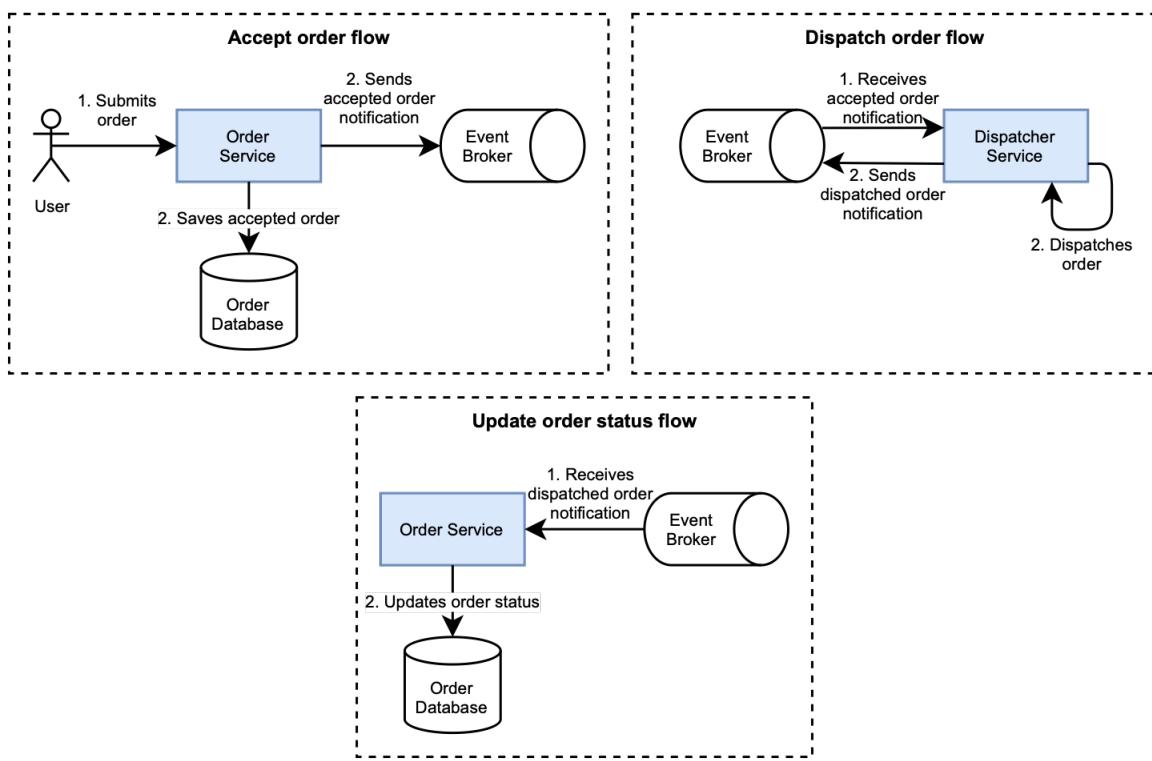


Figure 10.1 Order Service and Dispatcher Service communicate asynchronously and indirectly through producing and consuming events collected and distributed by an event broker (RabbitMQ).

In the rest of the chapter, you'll learn a few technologies and patterns to implement this event-driven design for Polar Bookshop. RabbitMQ will be the event processing platform responsible for collecting, routing, and distributing messages to consumers. Figure 10.2 highlights the event-driven part of the Polar Bookshop system after introducing the Dispatcher Service application and RabbitMQ.

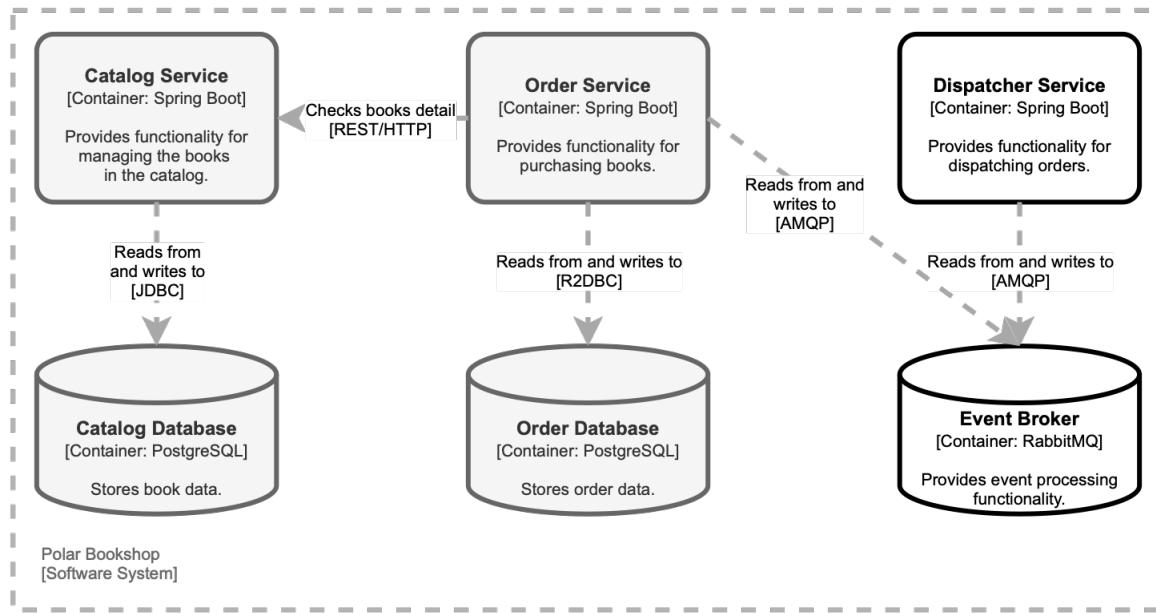


Figure 10.2 In the Polar Bookshop system, Order Service and Dispatcher Service communicate asynchronously based on events distributed by RabbitMQ.

The following section will introduce the basic concepts of RabbitMQ, its protocol, and how to run it in your local environment.

10.2 Message brokers with RabbitMQ

A messaging system requires two main things: a message broker and a protocol. The Advanced Message Queuing Protocol (AMQP) ensures interoperability across platforms and reliable message delivery. It has become widely used in modern architectures, and it's a good fit in the cloud, where we need resilience, loose coupling, and scalability. RabbitMQ is a popular open-source message broker that relies on AMQP and provides flexible asynchronous messaging, distributed deployment, and monitoring. Recent RabbitMQ versions have also introduced event streaming features.

Spring provides broad support for the most used messaging solutions. The Spring Framework itself has built-in support for the Java Messaging System (JMS) API. The Spring AMQP project (<https://spring.io/projects/spring-amqp>) adds support for this messaging protocol and provides integration with RabbitMQ. Apache Kafka is another technology that has become increasingly used in the last few years, for example, to implement the event sourcing pattern or real-time stream processing. The Spring Kafka project (<https://spring.io/projects/spring-kafka>) provides that integration.

This section will cover the fundamental aspects of the AMQP protocol and RabbitMQ, the message broker we'll use to implement messaging in the Polar Bookshop system. On the application side, we'll use Spring Cloud Stream, which offers convenient and robust integration

with RabbitMQ by relying on the Spring AMQP project.

10.2.1 Understanding AMQP for messaging systems

When using an AMQP-based solution like RabbitMQ, the actors involved in the interaction can be categorized as follows:

- **Producer.** The entity sending messages (publisher).
- **Consumer.** The entity receiving messages (subscriber).
- **Message Broker.** The middleware accepting messages from producers and routing them to consumers.

Figure 10.3 describes the interaction between the actors. From the protocol point of view, we can also say that the broker is the *server* while producers and consumers are the *clients*.

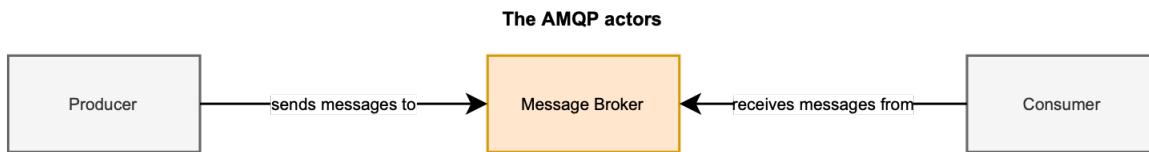


Figure 10.3 In AMQP, a broker accepts messages from producers and routes them to consumers.

NOTE

RabbitMQ was initially developed to support AMQP, but it also supports other protocols, including STOMP, MQTT, and even WebSockets to deliver messages over HTTP. Since version 3.9, it also supports event streaming.

The AMQP messaging model is based on *exchanges* and *queues*, as illustrated in figure 10.4. Producers send messages to an exchange. RabbitMQ computes which queues should receive a copy of the message according to a given routing rule. Consumers read messages from a queue.

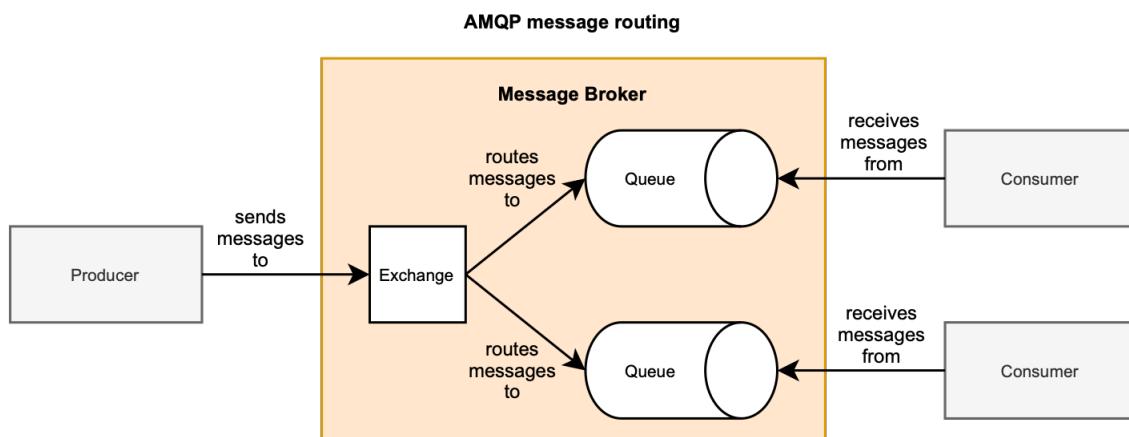


Figure 10.4 Producers publish messages to an exchange. Consumers subscribe to queues. Exchanges route messages to queues according to a routing algorithm.

The protocol establishes that a message comprises attributes and a payload, as shown in figure 10.5. AMQP defines some attributes, but you can add your own to pass the information necessary to route the message correctly. The payload must be of a binary type and has no further constraints besides that.

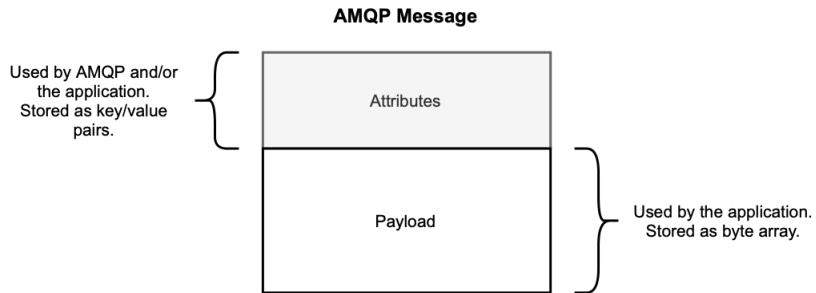


Figure 10.5 Producers publish messages to an exchange. Consumers subscribe to queues. Exchanges route messages to queues according to a routing algorithm.

Now that you know the basics of AMQP, let's get RabbitMQ up and running.

10.2.2 Using RabbitMQ for publish/subscribe communications

RabbitMQ, on top of AMQP, provides a simple yet effective solution to implement a publish/subscribe interaction that is precisely the one we want to establish between Order Service and Dispatcher Service. Besides the functionality itself, it's essential to look for those properties I addressed in the previous chapters for cloud systems and data services, including resilience, high availability, and data replication. RabbitMQ offers all of that. For example, it provides delivery acknowledgment, clustering, monitoring, queue durability, and replication. Furthermore, several cloud providers offer integrations with managed RabbitMQ services.

For now, you'll run RabbitMQ as a container on your local machine. First, make sure your Docker Engine is running. Then, open the `docker-compose.yml` file located in your `polar-deployment` repository.

NOTE

If you haven't followed along with the examples, you can use Chapter10/10-begin/polar-deployment/docker/docker-compose.yml from the source code accompanying the book as a starting point.

Then, add a new service definition using the RabbitMQ official image (including the management plugin) and expose it through port 5672 (for AMQP) and 15672 (for the management console). The RabbitMQ management plugin is convenient for inspecting exchanges and queues from a browser-based UI.

Listing 10.1 Define a container for RabbitMQ (docker-compose.yml)

```
version: "3.8"
services:
  ...
  polar-rabbitmq:
    image: rabbitmq:3.10-management ①
    container_name: polar-rabbitmq
    ports:
      - 5672:5672 ②
      - 15672:15672 ③
    volumes: ④
      - ./rabbitmq/rabbitmq.conf:/etc/rabbitmq/rabbitmq.conf
```

- ① The official RabbitMQ image with the management plugin enabled.
- ② The port where RabbitMQ listens for AMQP requests.
- ③ The port to expose the management GUI.
- ④ Configuration file mounted as a volume.

The configuration is based on a file mounted as a volume, similar to how we configured PostgreSQL. Create a `docker/rabbitmq` folder within your `polar-deployment` repository and add a new `rabbitmq.conf` file to configure the default account.

Listing 10.2 Configure RabbitMQ default account (rabbitmq.conf)

```
default_user = user
default_pass = password
```

Next, open a Terminal window, navigate to the folder where your `docker-compose.yml` file is located, and run the following command to start RabbitMQ.

```
$ docker-compose up -d polar-rabbitmq
```

Finally, open a browser window and navigate to <http://localhost:15672> to access the RabbitMQ management console. Log in using the credentials we defined in the configuration file (`user/password`) and have a look around. In the following sections, you'll be able to follow the message flows between Order Service and Dispatcher Service in the "Exchanges" and "Queues" areas of the management console.

When you're done exploring the RabbitMQ management console, you can shut it down as follows.

```
$ docker-compose down
```

Spring Cloud Stream helps integrate applications with event brokers like RabbitMQ seamlessly. But before getting to it, we need to define the logic that will process messages. In the next section, you'll learn about Spring Cloud Function and how to implement the business logic of the

new order flow in terms of suppliers, functions, and consumers.

10.3 Functions with Spring Cloud Function

Oleg Zhurakousky, project lead for Spring Cloud Function and Spring Cloud Stream, often asks this question to conference audiences: Is there any business feature that you cannot define in terms of suppliers, functions, and consumers? It's an interesting and challenging question. Can you think of anything? Most software requirements can be expressed with functions.

Why use functions in the first place? They are a simple, uniform, and portable programming model that is a perfect fit for event-driven architectures, inherently based on these concepts.

Spring Cloud Function promotes the implementation of business logic via functions based on the standard interfaces introduced by Java 8: `Supplier`, `Function`, and `Consumer`.

- **Supplier.** A supplier is a function with only output, no input. It's also known as a *producer*, *publisher*, or *source*.
- **Function.** A function has both input and output. It's also known as a *processor*.
- **Consumer.** A consumer is a function with input but no output. It's also known as a *subscriber* or *sink*.

In this section, you'll learn how Spring Cloud Function works and how to implement business logic via functions.

10.3.1 Using the functional paradigm in Spring Cloud Function

Let's get started with functions by considering the business requirements I listed earlier for the Dispatcher Service application. Whenever an order is accepted, Dispatcher Service should be responsible for packing and labeling the order, and notifying interested parties (in this case, the Order Service) once the order has been dispatched. For simplicity, let's assume that both the *pack* and *label* actions are performed by the application itself and consider how to implement the business logic via functions before even thinking about frameworks.

The two actions to be performed as part of dispatching an order could be represented as functions.

- The *pack* function takes the identifier of an accepted order as input, packs the order (in the example, the processing is represented by a log message), and returns the order identifier as output, ready to be labeled.
- The *label* function takes the identifier of a packed order as input, labels the order (in the example, the processing is represented by a log message), and returns the order identifier as output, completing the dispatch.

The composition of these two functions in sequence gives the full implementation of the business logic for Dispatcher Service, as shown in figure [10.6](#).

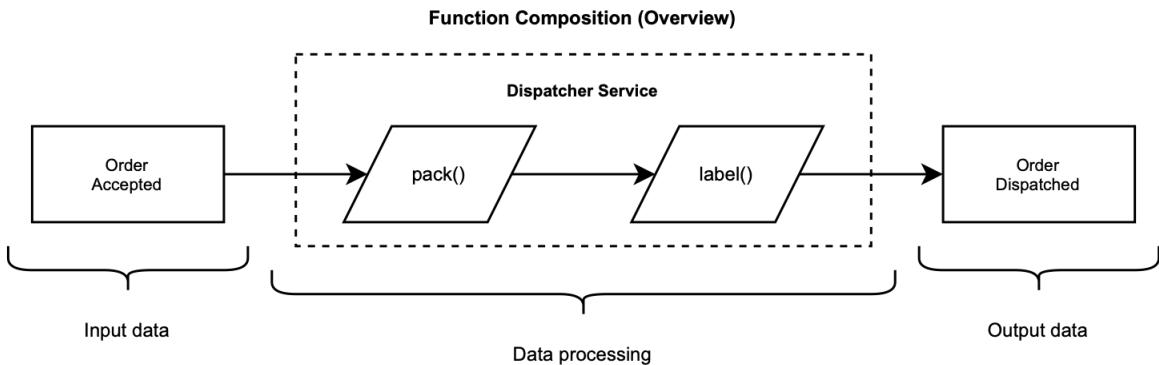


Figure 10.6 The business logic for Dispatcher Service is implemented as a composition of two functions: pack and label.

Let's see how to implement the functions and what Spring Cloud Function brings to the table.

INITIALIZING A SPRING CLOUD FUNCTION PROJECT

You can initialize the Dispatcher Service project from Spring Initializr (<https://start.spring.io>) and store the result in a new `dispatcher-service` Git repository. The parameters for the initialization are shown in figure 10.7.

The screenshot shows the Spring Initializr interface with the following settings:

- Project:** Gradle Project
- Language:** Java
- Spring Boot:** 2.7.0
- Dependencies:** Function (Spring Cloud)
- Project Metadata:**
 - Group: com.polarbookshop
 - Artifact: dispatcher-service
 - Name: dispatcher-service
 - Description: Provides functionality for dispatching orders.
 - Package name: com.polarbookshop.dispatcherservice
 - Packaging: Jar
 - Java version: 17

Figure 10.7 The parameters for initializing the Dispatcher Service project.

TIP

In the `begin` folder for this chapter, you can find a `curl` command to run in a Terminal window that downloads a zip file containing all the code you need to get started without going through the manual generation through the Spring Initializr website.

The resulting dependencies section of the `build.gradle` file looks as follows.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    implementation 'org.springframework.cloud:spring-cloud-function-context'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- **Spring Boot** (`org.springframework.boot:spring-boot-starter`) provides basic Spring Boot libraries and auto-configuration.
- **Spring Cloud Function** (`org.springframework.cloud:spring-cloud-function-context`) provides the Spring Cloud Function libraries to promote and support business logic implementation via functions.
- **Spring Boot Test** (`org.springframework.boot:spring-boot-starter-test`) provides several libraries and utilities to test applications, including Spring Test, JUnit, AssertJ, and Mockito. It's automatically included in every Spring Boot project.

Then, rename the autogenerated `application.properties` file to `application.yml`, and configure the server port and application name. At the moment, the application doesn't contain a web server. Nevertheless, we configure the server port number because it will be used when adding monitoring capabilities to the application in chapter 13.

Listing 10.3 Configure server and application name (application.yml)

```
server:
  port: 9003 ①

spring:
  application:
    name: dispatcher-service ②
```

- ① The port that will be used by the embedded web server.
- ② The name of the application.

Next, let's see how to implement the business logic using functions.

IMPLEMENTING THE BUSINESS LOGIC VIA FUNCTIONS

The business logic can be implemented in a standard way by using the Java `Function` interface. No Spring needed. Let's first consider the `pack` function. The input of the function should provide the identifier of an order that has previously been accepted. We can model such data via a simple DTO.

In the `com.polarbookshop.dispatcherservice` package, create an `OrderAcceptedMessage` record to hold the order identifier.

Listing 10.4 A DTO representing the event about orders being accepted (OrderAcceptedMessage.java)

```
package com.polarbookshop.dispatcherservice;

public record OrderAcceptedMessage ( ①
    Long orderId
){}
```

- ① DTO containing the order identifier as a Long field.

NOTE

Modeling events is a fascinating topic that goes beyond Spring and would require a few chapters to be covered properly. If you'd like to learn more about this subject, I recommend reading these articles by Martin Fowler: "Focusing on Events" (<https://martinfowler.com/eaaDev/EventNarrative.html>), "Domain Events" (<https://martinfowler.com/eaaDev/DomainEvent.html>) and "What do you mean by Event-Driven?" (<https://martinfowler.com/articles/201701-event-driven.html>).

The output of the function can be the simple identifier of the packed order represented as a `Long` object. Now that input and output are clear, it's time to define the function. Create a new `DispatchingFunctions` class and add a `pack()` method to implement the order packing as a function.

Listing 10.5 Implement the "pack" action as a function (DispatchingFunctions.java)

```
package com.polarbookshop.dispatcherservice;

import java.util.function.Function;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class DispatchingFunctions {
    private static final Logger log =
        LoggerFactory.getLogger(DispatchingFunctions.class);

    public Function<OrderAcceptedMessage, Long> pack() { ①
        return orderAcceptedMessage -> { ②
            log.info("The order with id {} is packed.", ③
                orderAcceptedMessage.orderId());
            return orderAcceptedMessage.orderId();
        };
    }
}
```

- ① Function implementing the order packing business logic.
- ② It takes an "OrderAcceptedMessage" object as input.
- ③ Returns an order identifier ("Long") as output.

You can see how there's only standard Java code. I strive to provide real-world examples in this book, so you might wonder what's happening here. In this case, I decided to focus on the

essential aspects of using the functional programming paradigm in the context of an event-driven application. Inside the function, you can add any processing logic you like. It doesn't matter. What matters here is the contract provided by the function, its signature: inputs and outputs. After defining that, you're free to implement the function as needed. I could have provided some more real-world implementation of this function, but it would have added nothing valuable considering the goal of this chapter. It doesn't even have to be Spring-based code. And in this example, it's not: it's plain Java code.

Spring Cloud Function is capable of managing functions defined in different ways, as long as they adhere to the standard Java interfaces `Function`, `Supplier`, and `Consumer`. You can make Spring Cloud Function aware of your functions by registering them as beans. Go ahead and do that for the `pack()` function by annotating the `DispatchingFunctions` class as `@Configuration` and the method as `@Bean`.

Listing 10.6 Register a standard function as a bean so that it's managed by Spring Cloud Function (`DispatchingFunctions.java`)

```
@Configuration ①
public class DispatchingFunctions {
    private static final Logger log =
        LoggerFactory.getLogger(DispatchingFunctions.class);

    @Bean ②
    public Function<OrderAcceptedMessage, Long> pack() {
        return orderAcceptedMessage -> {
            log.info("The order with id {} is packed.",
                orderAcceptedMessage.orderId());
            return orderAcceptedMessage.orderId();
        };
    }
}
```

- ① Functions are defined in a configuration class.
- ② Functions defined as beans can be discovered and managed by Spring Cloud Function.

As you'll see later, functions registered as beans are enhanced with extra features by the Spring Cloud Function framework. The beauty of it is that the business logic itself is not aware of the surrounding framework. So you can evolve it independently and test it without being concerned with framework-related issues.

USING IMPERATIVE AND REACTIVE FUNCTIONS

Spring Cloud Function supports both imperative and reactive code, so you're free to implement functions using reactive APIs like `Mono` and `Flux`. You can also mix and match. For the sake of the example, let's implement the `label` function using Project Reactor. The input of the function will be the identifier of an order that has been packed, represented as a `Long` object. The output of the function will be the identifier of the order that has been labeled, resulting in the dispatching process being complete. We can model such data via a simple DTO, just like we did for `OrderAcceptedMessage`.

In the `com.polarbookshop.dispatcherservice` package, create an `OrderDispatchedMessage` record to hold the identifier for a dispatched order.

Listing 10.7 A DTO representing the event about orders being dispatched (`OrderDispatchedMessage.java`)

```
package com.polarbookshop.dispatcherservice;

public record OrderDispatchedMessage ( ❶
    Long orderId
){}
```

- ❶ DTO containing the order identifier as a `Long` field.

Now that input and output are clear, it's time to define the function. Open the `DispatchingFunctions` class and add a `label()` method to implement the order labeling as a function. Since we want it to be reactive, both input and output are wrapped in a `Flux` publisher.

Listing 10.8 Implement the "label" action as a function (`DispatchingFunctions.java`)

```
package com.polarbookshop.dispatcherservice;

import java.util.function.Function;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import reactor.core.publisher.Flux;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DispatchingFunctions {
    private static final Logger log =
        LoggerFactory.getLogger(DispatchingFunctions.class);

    ...
    @Bean ❶
    public Function<Flux<Long>, Flux<OrderDispatchedMessage>> label() {
        return orderFlux -> orderFlux.map(orderId -> { ❷
            log.info("The order with id {} is labeled.", orderId);
            return new OrderDispatchedMessage(orderId); ❸
        });
    }
}
```

- ① Function implementing the order labeling business logic.
- ② It takes an order identifier ("Long") as input.
- ③ Returns an "OrderDispatchedMessage" as output.

We have just implemented both functions, so let's see how to combine and use them.

10.3.2 Composing and integrating functions: REST, serverless, data streams

The implementation of the business logic for Dispatcher Service is almost done. We still need a way to compose the two functions. Based upon our requirements, dispatching an order consists of two steps to be executed in sequence: `pack()` first and `label()` after.

Java provides features to compose `Function` objects in sequence using the `andThen()` or `compose()` operators. The problem is that you can use them only when the output type of the first function is the same as the second function input. Spring Cloud Function provides a solution to that problem and lets you compose functions seamlessly through transparent type conversion, even between imperative and reactive functions like those we defined earlier.

Composing functions with Spring Cloud is as simple as defining a property in your `application.yml` (or `application.properties`) file. Open the file `application.yml` in your Dispatcher Service project, and configure Spring Cloud Function to manage and compose the `pack()` and `label()` functions.

Listing 10.9 Declare the functions managed by Spring Cloud (application.yml)

```
spring:
  cloud:
    function:
      definition: pack|label ①
```

- ① Definition of the function managed by Spring Cloud Function.

The `spring.cloud.function.definition` property lets you declare which function(s) you want Spring Cloud Function to manage and integrate, resulting in a specific data flow. In the previous section, we implemented the basic `pack()` and `label()` functions. Now, we can instruct Spring Cloud Function to use them as building blocks and produce a new function that comes from the composition of those two.

In a serverless application like those meant to be deployed on a FaaS platform (e.g., AWS Lambda, Azure Functions, Google Cloud Functions, Knative), you would usually have one function defined per application. The cloud function definition can be mapped one-to-one to a function declared in your application, or you can use the `|` operator (*pipe*) to compose functions together in a data flow. If you need to define multiple functions, you can use the `:` character as the separator instead of `|`.

To sum up, you only need to implement standard Java functions and then configure Spring Cloud Function to use them as they are or after combining them. The framework will do the rest, including transparently converting input and output types to make the composition possible. Figure 10.8 illustrates the function composition.

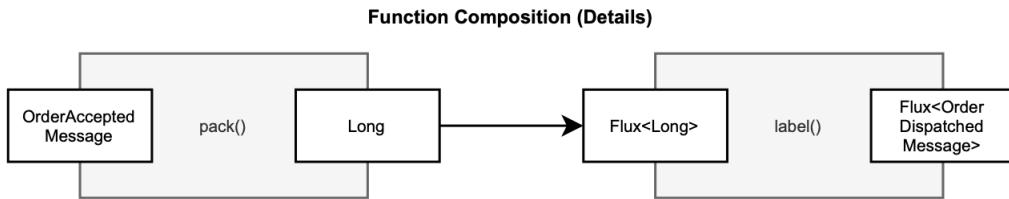


Figure 10.8 You can combine functions with different input/output types and mix imperative and reactive types as well. Spring Cloud Function will transparently handle any type conversion.

At this point, you’re probably wondering how to use these functions. That’s my favorite part. Once you define the functions, the framework can expose them in different ways depending on your needs. For example, Spring Cloud Function can automatically expose the function(s) defined in `spring.cloud.function.definition` as REST endpoint(s). Then you can directly package the application, deploy it on a FaaS platform like Knative, and voilà: you’ve got your first serverless Spring Boot application. That’s what we’ll do in chapter 16 when building serverless applications. Or you can use one of the adapters provided by the framework to package the application and deploy it on AWS Lambda, Azure Functions, or Google Cloud Functions. Or you can combine it with Spring Cloud Stream and bind the function to message channels in an event broker like RabbitMQ or Kafka.

Before exploring the integration with RabbitMQ using Spring Cloud Stream, I want to show you how to test the functions and their composition in isolation. Once the business logic is implemented as functions and tested, we can be sure it will work the same, whether triggered by a REST endpoint or an event notification.

10.3.3 Writing integration tests with `@FunctionalSpringBootTest`

Using the functional programming paradigm, we can implement business logic in standard Java and write unit tests with JUnit without being affected by the framework. At that level, there is no Spring code, just plain Java. Once you’ve ensured that each function works, you’ll want to write some integration tests to verify your application’s overall behavior when your functions are processed by Spring Cloud Function and exposed the way you configured.

Spring Cloud Function provides a `@FunctionalSpringBootTest` annotation you can use to set up the context for your integration tests. Unlike unit tests, you don’t want to invoke the function directly but ask the framework to provide that for you. All the functions managed by the framework are available through the `FunctionCatalog`, an object that acts as a function registry. When the framework serves the function, it doesn’t only contain the implementation

you wrote, but it's enhanced with extra features offered by Spring Cloud Function like transparent type conversion and function composition. Let's see how it works.

First, you need to add a test dependency on Reactor Test in the `build.gradle` file since part of the business logic is implemented using Reactor. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 10.10 Add dependency for Reactor Test in Dispatcher Service (`build.gradle`)

```
dependencies {
    ...
    testImplementation 'io.projectreactor:reactor-test'
}
```

Then, in the `src/test/java` folder of your Dispatcher Service project, create a new `DispatchingFunctionsIntegrationTests` class. You can write integration tests for the two functions individually, but it's more interesting verifying the behavior of the composed function `pack() + label()` as provided by Spring Cloud Function.

Listing 10.11 Integration tests for a function composition (`DispatchingFunctionsIntegrationTests.java`)

```
package com.polarbookshop.dispatcherservice;

import java.util.function.Function;
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.function.context.FunctionCatalog;
import org.springframework.cloud.function.context.test.FunctionalSpringBootTest;

@Autowired
class DispatchingFunctionsIntegrationTests {

    @Test
    void packAndLabelOrder() {
        Function<OrderAcceptedMessage, Flux<OrderDispatchedMessage>>
            packAndLabel = catalog.lookup(
                Function.class,
                "pack|label"); ①
        long orderId = 121;

        StepVerifier.create(packAndLabel.apply(
            new OrderAcceptedMessage(orderId) ②
        ))
            .expectNextMatches(dispatchedOrder -> ③
                dispatchedOrder.equals(new OrderDispatchedMessage(orderId)))
            .verifyComplete();
    }
}
```

- ① Gets the composed function from the `FunctionCatalog`.
- ② Defines an "OrderAcceptedMessage", which is the input to the function.

- ③ Asserts the output of the function is the expected "OrderDispatchedMessage" object.

Finally, open a Terminal window, navigate to the Dispatcher Service project root folder, and run the tests.

```
$ ./gradlew test --tests DispatchingFunctionsIntegrationTests
```

This type of integration test ensures the correct behavior of the defined cloud function, independently of how it will be exposed. In the source code accompanying the book, you'll find a broader set of autotests ([Chapter10/10-intermediate/dispatcher-service](#)).

Functions are a simple yet effective way to implement business logic and delegate infrastructural concerns to the framework. In the next section, you'll learn how to bind functions to message channels on RabbitMQ using Spring Cloud Stream.

10.4 Processing messages with Spring Cloud Stream

The principles that drive the Spring Cloud Function framework can also be found in Spring Cloud Stream. The idea is that you as a developer are responsible for the business logic, while the framework handles infrastructural concerns like how to integrate a message broker.

Spring Cloud Stream is a framework for building scalable, event-driven, and streaming applications. It's built on top of Spring Integration which offers the communication layer with message brokers, Spring Boot which provides auto-configuration for the middleware integration, and Spring Cloud Function to produce, process, and consume events. It relies on the native features of each message broker, but it also provides an abstraction to ensure a seamless experience independently from the underlying middleware. For example, features like consumer groups and partitions (native in Apache Kafka) are not present in RabbitMQ, but you can still use them thanks to the framework providing them for you.

My favorite Spring Cloud Stream feature is that it's enough to drop a dependency in a project like Dispatcher Service to get functions automatically bound to an external message broker. The best part of it? You don't have to change any code in the application, just the configuration in `application.yml` or `application.properties`. In previous versions of the framework, it was necessary to use dedicated annotations to match the business logic with the Spring Cloud Stream components. Now, it's completely transparent.

The framework supports integrations with RabbitMQ, Apache Kafka, Kafka Streams, and Amazon Kinesis. There are also integrations maintained by partners for Google PubSub, Solace PubSub+, Azure Event Hubs, and Apache RocketMQ.

This section will cover how to expose the composed function we defined in Dispatcher Service

through message channels in RabbitMQ.

10.4.1 Configuring the integration with RabbitMQ

Spring Cloud Stream is based on a few essential concepts:

- **Destination binder.** The component providing the integration with external messaging systems like RabbitMQ or Kafka.
- **Destination binding.** The bridge between the external messaging system entities like queues and topics, and the application-provided producers and consumers.
- **Message.** The data structure used by the application producers and consumers to communicate with the destination binders, and therefore with the external messaging systems.

All three of them are handled by the framework itself. The core of your application, the business logic, is not aware of the external messaging system. Destination binders are responsible for letting the application communicate with the external message brokers, including any vendor-specific concern. The bindings are autoconfigured by the framework, but you can still provide your own configuration to adapt them to your needs, as we'll do for Dispatcher Service. Figure 10.9 shows the model of a Spring Boot application using Spring Cloud Stream.

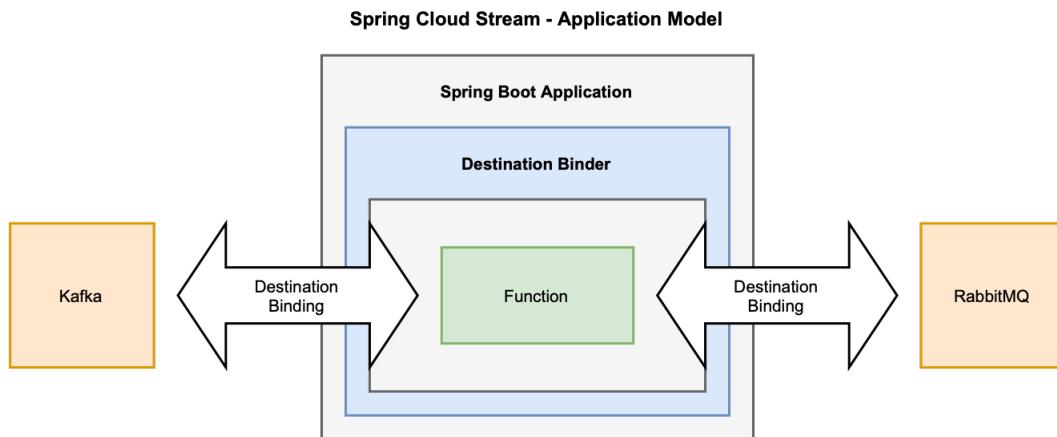


Figure 10.9 In Spring Cloud Stream, a destination binder provides integration with external messaging systems and establishes message channels with them.

Once you have defined the business logic of your application as functions and configured Spring Cloud Function to manage them (like we did for Dispatcher Service), you can expose the functions through a message broker by adding a dependency on the Spring Cloud Stream binder project specific to the broker you want to use. I'll show you how to work with RabbitMQ both for input and output message channels, but you can also bind with multiple messaging systems within the same application.

INTEGRATING RABBITMQ WITH SPRING

First, open the `build.gradle` file for the Dispatcher Service project (`dispatcher-service`), and replace the Spring Cloud Function dependency with the RabbitMQ binder for Spring Cloud Stream. Since Spring Cloud Function is already included in Spring Cloud Stream, you don't need to add it explicitly. You can also remove the dependency on Spring Boot Starter, which is also included in the Spring Cloud Stream dependency. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 10.12 Update dependencies in Dispatcher Service (build.gradle)

```
dependencies {
    implementation
    'org.springframework.cloud:spring-cloud-stream-binder-rabbit'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
}
```

Next, open the `application.yml` file and add the following configuration for the RabbitMQ integration. Port, username, and password are the ones we previously defined in Docker Compose.

Listing 10.13 Configure the RabbitMQ integration (application.yml)

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: user
    password: password
    connection-timeout: 5s
```

That's it. If you run the Dispatcher Service, you'll notice it works perfectly already without further configuration. Spring Cloud Stream would auto-generate and configure the bindings to exchanges and queues in RabbitMQ. That's great to get up and running quickly, but you would probably add your own configuration to customize the behavior for a real production scenario. The following section will show you how to do that. Once again, without changing any code in your business logic. How great is that?

10.4.2 Binding functions to message channels

Getting started with Spring Cloud Stream is straightforward, but there's a chance of getting confused between concepts called with similar names. In the context of message brokers and Spring Cloud Stream, the term *binding* and its variations are used a lot and can lead to misunderstanding. Figure 10.10 shows all the entities in place.

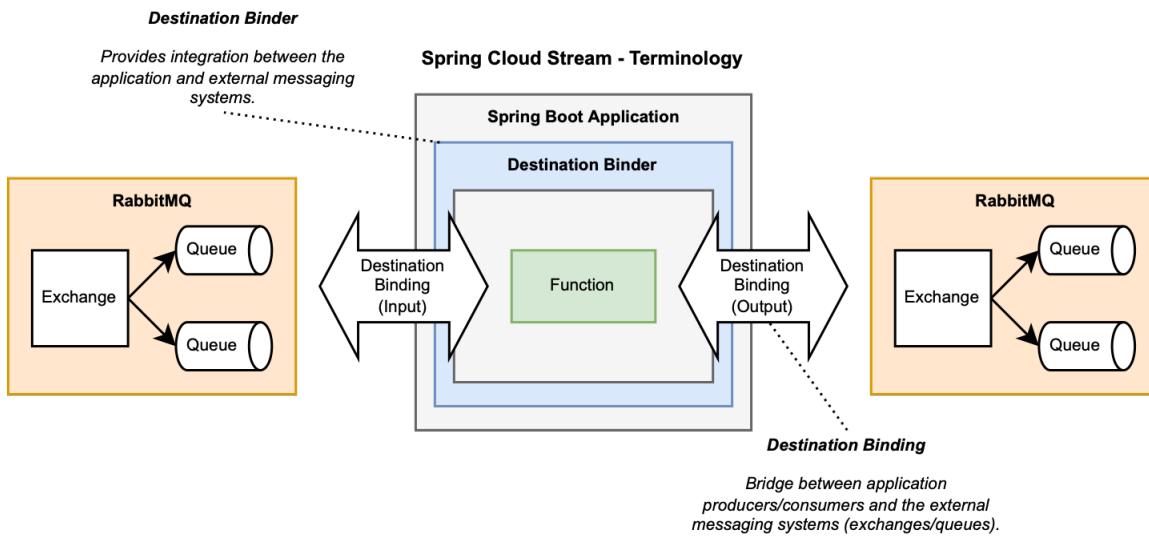


Figure 10.10 In Spring Cloud Stream, bindings establish message channels between application and message brokers.

Spring Cloud Stream provides a Spring Boot application with a **destination binder** that integrates with an external messaging system. The binder is also responsible for establishing communication channels between the application producers/consumers and the messaging system entities (exchanges and queues for RabbitMQ). These communication channels are called *destination bindings*, and they are bridges between application and broker.

A **destination binding** can be either an input channel or an output channel. By default, Spring Cloud Stream maps each binding (both input and output) to an exchange in RabbitMQ (a *topic exchange*, to be more precise). Furthermore, for each input binding, it binds a queue to the related exchange. That's the queue from which consumers receive and process events. This setup provides all the plumbing for implementing event-driven architectures based on the publisher/subscriber model.

In the following sections, I'll tell you more about destination bindings in Spring Cloud Stream and how they relate to exchanges and queues in RabbitMQ.

UNDERSTANDING DESTINATION BINDINGS

As you can see in figure 10.10, destination bindings are an abstraction representing a bridge between application and broker. When using the functional programming model, Spring Cloud Stream generates an input binding for each function accepting input data, and an output binding for each function returning output data. Each binding is assigned a logical name following this convention:

- Input binding: <functionName> + -in- + <index>
- Output binding: <functionName> + -out- + <index>

Unless you use partitions (for example, with Kafka), the `<index>` part of the name will always be `0`. The `<functionName>` is computed from the value of the `spring.cloud.function.definition` property. In case of a single function, there is a one-to-one mapping. For example, if in Dispatcher Service we only had one function called `dispatch`, then the related binding would be named `dispatch-in-0` and `dispatch-out-0`. We actually use a composed function (`pack|label`), so the binding names are generated by combining the name of all the functions involved in the composition.

- Input binding: `packlabel-in-0`
- Output binding: `packlabel-out-0`

These names are only relevant for configuring the bindings themselves in the application. They're like unique identifiers that let you reference a specific binding and apply custom configuration. Notice that these names exist only in Spring Cloud Stream, they're logical names. RabbitMQ doesn't know about them.

CONFIGURING DESTINATION BINDINGS

By default, Spring Cloud Stream uses the binding names to generate the names for exchanges and queues in RabbitMQ. But in a production scenario, you probably want to manage them explicitly for several reasons. For example, it's likely that both exchanges and queues already exist in production. You also want to control different options for exchanges and queues, like durability or routing algorithms.

For Dispatcher Service, I'll show you how to configure input and output bindings. At startup, Spring Cloud Stream will check if the related exchanges and queues already exist in RabbitMQ. If they don't, it will create them according to our configuration.

Let's start by defining the destination names that will be used to name exchanges and queues in RabbitMQ. In your Dispatcher Service project, update the `application.yml` file as follows.

Listing 10.14 Configure Cloud Stream bindings and RabbitMQ destinations (`application.yml`)

```
spring:
  cloud:
    function:
      definition: pack|label
    stream:
      bindings:
        ①
        packlabel-in-0: ②
          destination: order-accepted ③
          group: ${spring.application.name} ④
        packlabel-out-0: ⑤
          destination: order-dispatched ⑥
```

- ① Section for configuring destination bindings.
- ② The input binding.

- ③ The actual name at the broker that the binder binds to (the exchange in RabbitMQ).
- ④ The consumer group interested in the destination. Same as the application name.
- ⑤ The output binding.
- ⑥ The actual name at the broker that the binder binds to (the exchange in RabbitMQ).

The output binding (`packlabel-out-0`) will be mapped to an `order-dispatched` exchange in RabbitMQ. The input binding (`packlabel-in-0`) will be mapped to an `order-accepted` exchange and an `order-accepted.dispatcher-service` queue in RabbitMQ. If they don't exist already in RabbitMQ, the binder will create them. The queue naming strategy (`<destination>.<group>`) includes a parameter called *consumer group*.

The idea of **consumer groups** has been borrowed from Kafka and is very useful. In a standard publish/subscribe model, all consumers receive a copy of the messages sent to the queue they're subscribed to. That is convenient when different applications need to process the message. But in a cloud native context, where multiple instances of an application are running simultaneously for scaling and resilience, that would be a problem. If you have numerous Dispatcher Service instances, you don't want an order to be dispatched from all of them. That would lead to errors and an inconsistent state.

Consumer groups solve the problem. All consumers in the same group share one single subscription. As a consequence, each message arriving at the queue to which they're subscribed will be processed by one consumer only. Assume we have two applications (Dispatcher Service and Mail Service) interested in receiving events about accepted orders and deployed in a replicated fashion. Using the application name to configure consumer groups, we can ensure that each event is received and processed by one single instance of Dispatcher Service and one single instance of Mail Service, as shown in figure [10.11](#).

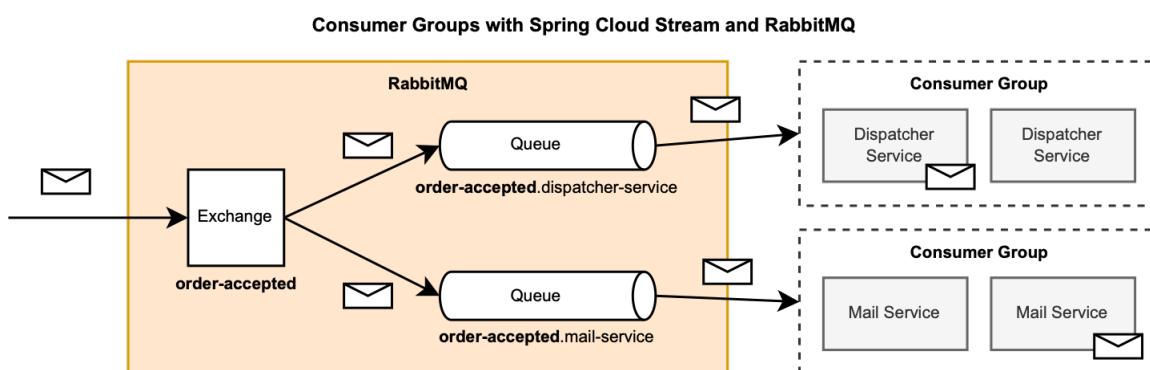


Figure 10.11 Consumer groups ensure that each message is received and processed by one consumer only within the same group.

EXPLORING EXCHANGES AND QUEUES IN RABBITMQ

After configuring the integration with RabbitMQ through Spring Cloud Stream, it's time to try running Dispatcher Service.

First, start a RabbitMQ container. Open a Terminal window, navigate to the folder in your `polar-deployment` repository where you keep your `docker-compose.yml` file (`polar-deployment/docker`), and run the following command.

```
$ docker-compose up -d polar-rabbitmq
```

Then, open another Terminal window, navigate to the Dispatcher Service project root folder (`dispatcher-service`), and run the application as follows.

```
$ ./gradlew bootRun
```

The application logs already give you a hint of what happened, but for a clearer understanding, let's check the RabbitMQ management console (exposed through port 15672).

Open a browser window and navigate to <http://localhost:15672>. The credentials are the same we defined in Docker Compose (`user/password`). Then, go to the "Exchanges" section. As shown in figure 10.12, there's a list of default exchanges provided by RabbitMQ and the two exchanges generated by our application: `order-accepted` and `order-dispatched`. Spring Cloud Stream maps them to the `packlabel-in-0` and `packlabel-out-0` bindings, respectively. The exchanges are *durable* (denoted by the `D` icon in the management console), meaning they will survive a broker restart.

The screenshot shows the RabbitMQ Management Console interface. At the top, there are tabs for Overview, Connections, Channels, Exchanges (which is selected), Queues, and Admin. On the right, there are refresh, virtual host, cluster, and user information. Below the tabs, the title is 'Exchanges' with a dropdown for 'All exchanges (9)'. There are filters for page, filter, and regex, and a note about displaying 9 items. A table lists the exchanges:

Name	Type	Features	Message rate in	Message rate out	Operations
(AMQP default)	direct	<code>D</code>			+/-
amq.direct	direct	<code>D</code>			
amq.fanout	fanout	<code>D</code>			
amq.headers	headers	<code>D</code>			
amq.match	headers	<code>D</code>			
amq.rabbitmq.trace	topic	<code>D I</code>			
amq.topic	topic	<code>D</code>			
order-accepted	topic	<code>D</code>			
order-dispatched	topic	<code>D</code>			

Figure 10.12 Spring Cloud Stream maps the two destination binding to two exchanges in RabbitMQ.

Next, let's have a look at the queues. In Dispatcher Service, we configured a `packlabel-in-0` binding and a consumer group. That's the only input channel for the application, so it should result in one single queue. Let's verify that. In the RabbitMQ management console, as illustrated in figure 10.13, you can see a durable `order-accepted.dispatcher-service` queue in the "Queues" section.

Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
<code>order-accepted.dispatcher-service</code>	classic	D	idle	0	0	0			

Figure 10.13 Spring Cloud Stream maps each input binding to a queue, named according to the configured consumer group.

NOTE

No queue has been created for the `packlabel-out-0` binding because no consumer subscribed to it. Later, you'll see that a queue will be created after configuring Order Service to listen to it.

We can verify that the integration works by manually sending a message to the `order-accepted` exchange. If everything is configured correctly, Dispatcher Service will read the message from the `order-accepted.dispatcher-service` queue, process it through the composed function `pack|label`, and finally, send it to the `order-dispatched` exchange.

Go to the "Exchanges" section again, select the `order-accepted` exchange, and in the "Publish message" panel, insert an `OrderAcceptedMessage` object in JSON format as shown in figure 10.14. When you're done, click "Publish message".

The screenshot shows the RabbitMQ Management UI with the 'Exchanges' tab selected. Under the 'order-accepted' exchange, the 'Publish message' section is active. The 'Payload' field contains the JSON message: {"orderId": 121}. The 'Publish message' button at the bottom is highlighted with a yellow box.

Figure 10.14 You can trigger the data flow in Dispatcher Service by sending a message to the "order-accepted" exchange.

In the application logs, you should see the following messages signaling that the data flow happened correctly.

```
2022-02-22 22:02:22 INFO 30073 --- [tcher-service-1]
➡ c.p.d.DispatchingFunctions: The order with id 394 is packed.
2022-02-22 22:02:22 INFO 30073 --- [tcher-service-1]
➡ c.p.d.DispatchingFunctions: The order with id 394 is labeled.
```

The output message has been sent to the `order-dispatched` exchange, but it's not been routed to any queue because no consumer has subscribed. In the final part of this chapter, we'll complete the flow by defining a supplier in Order Service to publish messages to the `order-accepted` exchange whenever an order is accepted and a consumer to read messages from the `order-dispatched` queue whenever an order is dispatched. But first, let's add some tests to verify the integration with the Spring Cloud Stream binder.

Before moving on, stop the application process with `Ctrl+C` and the RabbitMQ container with `docker-compose down`.

10.4.3 Writing integration tests with a test binder

As I stressed several times, Spring Cloud Function and Spring Cloud Stream's whole philosophy is about keeping the application business logic infrastructure- and middleware-neutral. After defining the original `pack()` and `label()` functions, all we did was update dependencies in Gradle and modify the configuration in `application.yml`.

It's a good idea to have unit tests covering the business logic independent from the framework. But it's worth adding a few integration tests to cover the application behavior in a Spring Cloud Stream context. You should disable the integration tests you wrote earlier in the

`DispatchingFunctionsIntegrationTests` class since now you want to test the integration with the external messaging system.

The framework provides a binder specifically to implement integration tests focusing on the business logic rather than the middleware. Let's see how it works using Dispatcher Service as an example.

NOTE

The test binder provided by Spring Cloud Stream is meant to verify the correct configuration and integration with a technology-agnostic destination binder. If you want to test the application against a specific broker (in our case, it would be RabbitMQ), you can rely on Testcontainers, as you learned in the previous chapter. I'll leave that up to you as an exercise.

First, add a dependency to the test binder in the `build.gradle` file of your Dispatcher Service project. Unlike the other dependencies we've been working on so far, the test binder requires a more elaborated syntax to be included. For more information, refer to the Spring Cloud Stream documentation (<https://spring.io/projects/spring-cloud-stream>). Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 10.15 Add dependency for the test binder in Dispatcher Service (build.gradle)

```
dependencies {
    ...
    testImplementation("org.springframework.cloud:spring-cloud-stream") {
        artifact {
            name = "spring-cloud-stream"
            extension = "jar"
            type = "test-jar"
            classifier = "test-binder"
        }
    }
}
```

Then, create a new `FunctionsStreamIntegrationTests` class for testing. The test setup consists of three steps:

- Import the `TestChannelBinderConfiguration` class providing configuration for the test binder.
- Inject an `InputDestination` bean representing the input binding `packlabel-in-0` (by default since it's the only one).
- Inject an `OutputDestination` bean representing the output binding `packlabel-out-0` (by default since it's the only one).

The data flow is based on `Message` objects (from the `org.springframework.messaging` package). The framework handles type conversion for you transparently when running the application. However, in this type of test, you need to provide `Message` objects explicitly. You can use `MessageBuilder` to create the input message and the `ObjectMapper` utility to perform the type conversion from the binary format used for storing message payloads in a broker.

Listing 10.16 Testing the integration with external messaging systems (FunctionsStreamIntegrationTests.java)

```

package com.polarbookshop.dispatcherservice;

import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.cloud.stream.binder.test.InputDestination;
import org.springframework.cloud.stream.binder.test.OutputDestination;
import org.springframework.cloud.stream.binder.test.
➥TestChannelBinderConfiguration;
import org.springframework.context.annotation.Import;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;
import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
@Import(TestChannelBinderConfiguration.class) ①
class FunctionsStreamIntegrationTests {

    @Autowired
    private InputDestination input; ②

    @Autowired
    private OutputDestination output; ③

    @Autowired
    private ObjectMapper objectMapper; ④

    @Test
    void whenOrderAcceptedThenDispatched() throws IOException {
        long orderId = 121;
        Message<OrderAcceptedMessage> inputMessage = MessageBuilder
            .withPayload(new OrderAcceptedMessage(orderId)).build();
        Message<OrderDispatchedMessage> expectedOutputMessage = MessageBuilder
            .withPayload(new OrderDispatchedMessage(orderId)).build();

        this.input.send(inputMessage); ⑤
        assertThat(objectMapper.readValue(output.receive().getPayload(),
            OrderDispatchedMessage.class))
            .isEqualTo(expectedOutputMessage.getPayload()); ⑥
    }
}

```

- ① Configures the test binder.
- ② Represents the input binding packlabel-in-0.
- ③ Represents the output binding packlabel-out-0.
- ④ Uses Jackson to deserialize JSON message payloads to Java objects.
- ⑤ Sends a message to the input channel.
- ⑥ Receives and asserts a message from the output channel.

WARNING If you use IntelliJ IDEA, you might get a warning that `InputDestination`, `OutputDestination`, and `ObjectMapper` cannot be autowired. Don't worry. It's a false positive. You can get rid of the warning by annotating the field with `@SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")`

Message brokers like RabbitMQ deal with binary data, so any data flowing through them is mapped to `byte[]` in Java. The conversion between bytes and DTOs is handled by Spring Cloud Stream transparently. But just like for messages, in this test scenario, we need to handle that explicitly when asserting the content of the message received from the output channel.

After writing the integration tests, open a Terminal window, navigate to the Dispatcher Service project root folder, and run the tests.

```
$ ./gradlew test --tests FunctionsStreamIntegrationTests
```

The next section will go through some considerations about resilient integrations with messaging systems.

10.4.4 Making messaging resilient to failures

Event-driven architectures solve some issues affecting synchronous request/response interactions. For example, if you remove the temporal coupling between applications, you won't need to adopt patterns like circuit breakers since the communication will be asynchronous. If the consumer is momentarily unavailable while the producer sends a message, it doesn't matter. It will receive the message once it's up and running again.

In software engineering, there are no silver bullets. Everything comes at a cost. On the one hand, applications that are decoupled can operate more independently. On the other hand, you introduced a new component in your system that needs to be deployed and maintained: the message broker.

Assuming that part is taken care of by the platform, there's still something for you to do as the application developer. When an event happens and your application wants to publish a message, something might go wrong. Retries and timeouts are still helpful, but this time they will be configured to make the interaction between application and broker more resilient. Spring Cloud Stream uses the retry pattern with an exponential backoff strategy by default, relying on the Spring Retry library for imperative consumers and the `retryWhen()` Reactor operator for reactive consumers (the one you learned in chapter 7). As usual, you can customize it via configuration properties.

Spring Cloud Stream defines several defaults to make the interaction more resilient, including

error channels and graceful shutdown. You can configure different aspects of message processing, including dead-letter queues, acknowledgment flows, and republishing messages on error.

RabbitMQ itself has several features in place to improve reliability and resilience. Among the other things, it guarantees that each message is delivered at least once. Be aware that consumers in your applications might receive the same message twice, so your business logic should know how to identify and handle duplicates.

I won't go further into details since this is an extensive subject that would require several dedicated chapters to be adequately covered. Instead, I encourage you to read the documentation for the different projects involved in your event-driven architecture: RabbitMQ (<https://www.rabbitmq.com/>), Spring AMQP (<https://spring.io/projects/spring-amqp>), and Spring Cloud Stream (<https://spring.io/projects/spring-cloud-stream>).

In the last part of the chapter, you'll work with suppliers and consumers and complete the order flow for the Polar Bookshop system.

10.5 Producing and consuming messages with Spring Cloud Stream

In the previous sections, you learned about the functional programming paradigm and how it fits in the Spring ecosystem using Spring Cloud Function and Spring Cloud Stream. This last section will guide you through the implementation of producers and consumers.

As you'll see, consumers are not that different from the functions you wrote in Dispatcher Service. On the other hand, producers are slightly different because, unlike functions and consumers, they are not naturally activated. I'll show you how to use them both in Order Service while implementing the last part of the order flow for the Polar Bookshop system.

10.5.1 Implementing event consumers and the problem of idempotency

The Dispatcher Service application we previously built produces messages when orders are dispatched. The Order Service should be notified when that happens so that it can update the order status in the database.

First, open your Order Service project (`order-service`) and add the dependencies on Spring Cloud Stream and the test binder in the `build.gradle` file. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 10.17 Add dependency for Spring Cloud Stream and test binder in Order Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.cloud:spring-cloud-stream-binder-rabbit'
    testImplementation("org.springframework.cloud:spring-cloud-stream") {
        artifact {
            name = "spring-cloud-stream"
            extension = "jar"
            type = "test-jar"
            classifier = "test-binder"
        }
    }
}
```

Next, we need to model the event we'd like Order Service to listen to. Create a new `com.polarbookshop.orderservice.order.event` package and add an `OrderDispatchedMessage` class to hold the identifier for a dispatched order.

Listing 10.18 A DTO representing the event about orders being dispatched (OrderDispatchedMessage.java)

```
package com.polarbookshop.orderservice.order.event;

public record OrderDispatchedMessage (
    Long orderId
){}
```

Then, we can implement the business logic using a functional approach. Create an `OrderFunctions` class (`com.polarbookshop.orderservice.order.event` package) and implement a function to consume the messages produced by the Dispatcher Service application when an order is dispatched. The function will be a `Consumer` responsible for listening to the incoming messages and updating the database entities accordingly. `Consumer` objects are functions with input but no output. To keep the function clean and readable, we move the processing of `OrderDispatchedMessage` objects to the `OrderService` class (that we'll implement in a minute).

Listing 10.19 Consume messages from RabbitMQ (OrderFunctions.java)

```

package com.polarbookshop.orderservice.order.event;

import java.util.function.Consumer;
import com.polarbookshop.orderservice.order.domain.OrderService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import reactor.core.publisher.Flux;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OrderFunctions {

    private static final Logger log =
        LoggerFactory.getLogger(OrderFunctions.class);

    @Bean
    public Consumer<Flux<OrderDispatchedMessage>> dispatchOrder(
        OrderService orderService
    ) {
        return flux ->
            orderService.consumeOrderDispatchedEvent(flux) ❶
                .doOnNext(order -> log.info("The order with id {} is dispatched",
                    order.id())) ❷
                .subscribe(); ❸
    }
}

```

- ❶ For each dispatched message, it updates the related order in the database.
- ❷ For each order updated in the database, it logs a message.
- ❸ Subscribes to the reactive stream in order to activate it. Without a subscriber, no data flows through the stream.

Order Service is a reactive application. Therefore, the `dispatchOrder` function will consume messages as a reactive stream (a `Flux` of `OrderDispatchedMessage`). Reactive streams are *activated* only if there's a subscriber interested in receiving the data. For that reason, it's critical that we end the reactive stream by subscribing to it, or else no data will ever be processed. In previous examples, the subscription part was handled transparently by the framework (for example, when using reactive streams to return data via a REST endpoint or to send data to a backing service). In this case, we have to explicitly do that with the `subscribe()` clause.

Next, let's implement the `consumeOrderDispatchedMessageEvent()` method in the `OrderService` class to update the status in the database for an existing order after being dispatched.

Listing 10.20 Implement the logic for updating an order as dispatched (OrderService.java)

```

@Service
public class OrderService {
    ...

    public Flux<Order> consumeOrderDispatchedEvent(
        Flux<OrderDispatchedMessage> flux
    ) {
        return flux ①
            .flatMap(message ->
                orderRepository.findById(message.orderId()) ②
            .map(this::buildDispatchedOrder) ③
            .flatMap(orderRepository::save); ④
    }

    private Order buildDispatchedOrder(Order existingOrder) {
        return new Order(⑤
            existingOrder.id(),
            existingOrder.bookIsbn(),
            existingOrder.bookName(),
            existingOrder.bookPrice(),
            existingOrder.quantity(),
            OrderStatus.DISPATCHED,
            existingOrder.createdDate(),
            existingOrder.lastModifiedDate(),
            existingOrder.version()
        );
    }
}

```

- ① Accepts a reactive stream of OrderDispatchedMessage objects as input.
- ② For each object emitted to the stream, it reads the related order from the database.
- ③ Updates the order with the "dispatched" status.
- ④ Saves the updated order in the database.
- ⑤ Given an order, it returns a new record with the "dispatched" status.

Consumers are triggered by a message arriving in the queue. RabbitMQ provides an *at-least-one delivered* guarantee, so you need to be aware of possible duplicates. The code we implemented updates the status of the specific order to be DISPATCHED, an operation that can be executed several times with the same result. Since the operation is idempotent, the code is resilient to duplicates. A further optimization would be to check for the status and skip the update operation if it's already dispatched.

Finally, we need to configure Spring Cloud Stream in the `application.yml` file so that the `dispatchOrder-in-0` binding (inferred from the `dispatchOrder` function name) is mapped to the `order-dispatched` exchange in RabbitMQ. Also, remember to define `dispatchOrder` as the function Spring Cloud Function should manage, and the integration with RabbitMQ.

Listing 10.21 Configure Cloud Stream bindings and RabbitMQ integration (application.yml)

```

spring:
  cloud:
    function:
      definition: dispatchorder ①
    stream:
      bindings:
        dispatchOrder-in-0: ②
          destination: order-dispatched ③
          group: {spring.application.name} ④
  rabbitmq: ⑤
    host: localhost
    port: 5672
    username: user
    password: password
    connection-timeout: 5s

```

- ① Definition of the function managed by Spring Cloud Function.
- ② The input binding.
- ③ The actual name at the broker that the binder binds to (the exchange in RabbitMQ).
- ④ The consumer group interested in the destination. Same as the application name.
- ⑤ Configures the integrations with RabbitMQ.

As you can see, it works the same way as the functions in Dispatcher Service. The consumers in Order Service will be part of the `order-service` consumer group and Spring Cloud Stream will define a message channel between them and an `order-dispatched.order-service` queue in RabbitMQ.

Next, we'll complete the order flow by defining a supplier responsible for triggering the whole procedure.

10.5.2 Implementing event producers and the problem of atomicity

Suppliers are message sources. They produce messages when an event happens. In Order Service, a supplier should notify the interested parties (in this case, Dispatcher Service) whenever an order has been accepted. Differently from functions and consumers, suppliers need to be activated. They only act upon invocation.

Spring Cloud Stream provides a few ways to define suppliers and cover different scenarios. In our case, the event source is not a message broker, but a REST endpoint. When a user sends a POST request to Order Service for purchasing a book, we want to publish an event signaling whether the order has been accepted.

Let's start by modeling that event as a DTO. It will be the same as the `OrderAcceptedMessage` record we used in Dispatcher Service. Add the record to the `com.polarbookshop.orderservice.order.event` package in your Order Service project (

order-service).

**Listing 10.22 A DTO representing the event about orders being accepted
(OrderAcceptedMessage.java)**

```
package com.polarbookshop.orderservice.order.event;

public record OrderAcceptedMessage (
    Long orderId
){}
```

We can bridge the REST layer with the stream part of the application using a `StreamBridge` object that allows sending data to a specific destination imperatively. Let's break this new functionality down. First, we can implement a method that accepts an `Order` object as input, verifies it's accepted, builds an `OrderAcceptedMessage` object, and sends it to a RabbitMQ destination using `StreamBridge`.

Open the `OrderService` class, autowire a `StreamBridge` object, and define a new `publishOrderAcceptedEvent` method.

Listing 10.23 Implement the logic for publishing events to a given destination (OrderService.class)

```

package com.polarbookshop.orderservice.order.domain;

import com.polarbookshop.orderservice.book.BookClient;
import com.polarbookshop.orderservice.order.event.OrderAcceptedMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.stream.function.StreamBridge;
import org.springframework.stereotype.Service;
...

@Service
public class OrderService {
    private static final Logger log =
        LoggerFactory.getLogger(OrderService.class);

    private final BookClient bookClient;
    private final OrderRepository orderRepository;
    private final StreamBridge streamBridge;

    public OrderService(BookClient bookClient,
        StreamBridge streamBridge, OrderRepository orderRepository
    ) {
        this.bookClient = bookClient;
        this.orderRepository = orderRepository;
        this.streamBridge = streamBridge;
    }

    ...

    private void publishOrderAcceptedEvent(Order order) {
        if (!order.status().equals(OrderStatus.ACCEPTED)) {
            return; ①
        }
        var orderAcceptedMessage =
            new OrderAcceptedMessage(order.id()); ②
        log.info("Sending order accepted event with id: {}", order.id());
        var result = streamBridge.send("acceptOrder-out-0",
            orderAcceptedMessage); ③
        log.info("Result of sending data for order with id {}: {}",
            order.id(), result);
    }
}

```

- ① If the order is not accepted, it does nothing.
- ② Builds a message to notify an order has been accepted.
- ③ Explicitly sends a message to the "acceptOrder-out-0" binding.

Since the data source is a REST endpoint, there is no `Supplier` bean we can register with Spring Cloud Function, and therefore there is no trigger for the framework to create the necessary bindings with RabbitMQ. Yet, in listing [10.23](#), `StreamBridge` is used to send data to an `acceptOrder-out-0` binding. Where does it come from? There is no `acceptOrder` function!

At startup time, Spring Cloud Stream will notice that `StreamBridge` wants to publish messages via an `acceptOrder-out-0` binding, and therefore it will create one automatically. Similar to the bindings created from functions, we can configure the destination name in RabbitMQ. Open

the `application.yml` file and configure the binding as follows.

Listing 10.24 Configure Cloud Stream output binding (`application.yml`)

```
spring:
  cloud:
    function:
      definition: dispatchOrder
    stream:
      bindings:
        dispatchOrder-in-0:
          destination: order-dispatched
          group: ${spring.application.name}
        acceptOrder-out-0: ①
          destination: order-accepted ②
```

- ① Output binding created and managed by StreamBridge.
- ② The actual name at the broker that the binder binds to (the exchange in RabbitMQ).

All that is left now is calling the method whenever a submitted order is accepted. That is a very critical point and one of the aspects characterizing the **saga pattern**, a popular alternative to distributed transactions in microservices architectures. To ensure consistency in your system, persisting an order in the database and sending a message about it must be done atomically. Either both operations succeed, or they both fail. A simple yet effective way to ensure atomicity is by wrapping the two operations in a local transaction. To do that, we can rely on the built-in Spring transaction management functionality.

NOTE

The saga pattern is described extensively in chapter 4 of the "Microservices Patterns" book written by Chris Richardson and published by Manning (<https://livebook.manning.com/book/microservices-patterns/chapter-4>). I recommend you check it out if you're interested in designing business transactions that span multiple applications.

In the `OrderService` class, modify the `submitOrder()` method to call the `publishOrderAcceptedEvent` method, and annotate it with `@Transactional`.

Listing 10.25 Define a saga transaction with database update and message production (OrderService.java)

```

@Service
public class OrderService {
    ...

    @Transactional ①
    public Mono<Order> submitOrder(String isbn, int quantity) {
        return bookClient.getBookByIsbn(isbn)
            .map(book -> buildAcceptedOrder(book, quantity))
            .defaultIfEmpty(buildRejectedOrder(isbn, quantity))
            .flatMap(orderRepository::save) ②
            .doOnNext(this::publishOrderAcceptedEvent); ③
    }

    private void publishOrderAcceptedEvent(Order order) {
        if (!order.status().equals(OrderStatus.ACCEPTED)) {
            return;
        }
        var orderAcceptedMessage = new OrderAcceptedMessage(order.id());
        log.info("Sending order accepted event with id: {}", order.id());
        var result = streamBridge.send("acceptOrder-out-0",
            orderAcceptedMessage);
        log.info("Result of sending data for order with id {}: {}",
            order.id(), result);
    }
}

```

- ① Executes the method in a local transaction.
- ② Saves the order in the database.
- ③ Publishes an event if the order is accepted.

Spring Boot comes preconfigured with transaction management functionality and can handle transactional operations involving relational databases (as you learned in chapter 5). However, the channel established with RabbitMQ for the message producer is not transactional by default. To make the event-publishing operation join the existing transaction, we need to enable the RabbitMQ transactional support for the message producer in the `application.yml` file.

Listing 10.26 Configure the exchange mapped to the output binding to be transactional (application.yml)

```

spring:
  cloud:
    function:
      definition: dispatchOrder
    stream:
      bindings:
        dispatchOrder-in-0:
          destination: order-dispatched
          group: ${spring.application.name}
        acceptOrder-out-0:
          destination: order-accepted
    rabbit: ①
      bindings:
        acceptOrder-out-0:
          producer:
            transacted: true ②

```

- ① RabbitMQ-specific configuration for the Spring Cloud Stream bindings.
- ② Makes the `acceptOrder-out-0` binding transactional.

Now, you can write new integration tests for the supplier and the consumer like we did for the functions in Dispatcher Service. I'll leave the autotests to you since you have the necessary tools now. If you need inspiration, check out the source code accompanying this book (Chapter10/10-end/order-service).

You will also need to import the configuration for the test binder (`@Import(TestChannelBinderConfiguration.class)`) in the existing `OrderServiceApplicationTests` class to make it work.

It's been a nice journey through event-driven models, functions, and messaging systems. Before wrapping up, let's see the order flow in action. First, start RabbitMQ, PostgreSQL (`docker-compose up -d polar-rabbitmq polar-postgres`), and Dispatcher Service (`./gradlew bootRun`). Then, run also Catalog Service and Order Service (`./gradlew bootRun` or from Docker Compose after building the images first).

Once all those services are up and running, add a new book to the catalog.

```
$ http POST :9001/books author="Jon Snow" \
  title="All I don't know about the Arctic" isbn="1234567897" \
  price=9.90 publisher="Polarsophia"
```

Then, order three copies of that book.

```
$ http POST :9002/orders isbn=1234567897 quantity=3
```

If you order a book that exists, then the order is accepted, and Order Service publishes an `OrderAcceptedEvent` message. Dispatcher Service, subscribed to that same event, processes the

order, and publishes an `OrderDispatchedEvent` message. Order Service is notified and updates the order status in the database.

TIP

You can follow the message flow by checking the application logs from Order Service and Dispatcher Service.

The moment of the truth. Fetch the order from Order Service.

```
$ http :9002/orders
```

The status should be `DISPATCHED`.

```
{
  "bookIsbn": "1234567897",
  "bookName": "All I don't know about the Arctic - Jon Snow",
  "bookPrice": 9.9,
  "createdDate": "2022-06-06T19:40:33.426610Z",
  "id": 1,
  "lastModifiedDate": "2022-06-06T19:40:33.866588Z",
  "quantity": 3,
  "status": "DISPATCHED",
  "version": 2
}
```

And it is. Great job! When you're done testing the system, stop all the applications (`Ctrl+C`) and Docker containers (`docker-compose down`).

That concludes the main implementation of the business logic for the Polar Bookshop system. The next chapter will cover security for cloud native applications using Spring Security, OAuth 2.0, and OpenID Connect.

SIDE BAR**Polar Labs**

Feel free to apply what you learned in the previous chapters and prepare the Dispatcher Service application for deployment.

1. Add Spring Cloud Config Client to Dispatcher Service to make it fetch configuration data from Config Service.
2. Configure the Cloud Native Buildpacks integration, containerize the application, and define the commit stage of the deployment pipeline.
3. Write the Deployment and Service manifests for deploying Dispatcher Service to a Kubernetes cluster.
4. Configure Tilt to automate the Dispatcher Service deployment to your local Kubernetes cluster initialized with minikube.

Then, update the Docker Compose specification and the Kubernetes manifests to configure the RabbitMQ integration for Order Service.

You can refer to the Chapter10/10-end folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and deploying the backing services from the manifests available in the Chapter10/10-end/polar-deployment/kubernetes/platform/development folder with `kubectl apply -f services`.

10.6 Summary

- Event-driven architectures describes distributed systems that interact with each other through producing and consuming events.
- An event is something relevant that happened in a system.
- In the pub/sub model, producers publish events sent to all subscribers to be consumed.
- Event processing platforms like RabbitMQ or Kafka are responsible for collecting events from the producers, routing, and distributing them to the interested consumers.
- In the AMQP protocol, producers send messages to an exchange in a broker that forwards them to queues according to specific routing algorithms.
- In the AMQP protocol, consumers receive messages from the queues in the broker.
- In the AMQP protocol, messages are data structures composed of key/value attributes and a binary payload.
- RabbitMQ is a message broker based on the AMQP protocol that you can use to implement event-driven architectures based on the publisher/subscriber model.
- RabbitMQ provides high availability, resilience, and data replication.
- Spring Cloud Function enables you to implement your business logic using the standard Java Function, Supplier, and Consumer interfaces.
- Spring Cloud Function wraps your function and provides several exciting features like transparent type conversion and function composition.
- Functions implemented in the context of Spring Cloud Function can be exposed and integrated with external systems in different ways.
- Functions can be exposed as REST endpoints, packaged, and deployed in a FaaS platform as serverless applications (Knative, AWS Lambda, Azure Function, Google Cloud Functions), or bound to message channels.
- Spring Cloud Stream, built on top of Spring Cloud Function, provides you with all the necessary plumbing for integrating your functions with external messaging systems like RabbitMQ or Kafka.
- Once you implement your functions, you don't have to make any changes to your code. You only need to add a dependency on Spring Cloud Stream and configure it to adapt to your needs.
- In Spring Cloud Stream, destination binders provide integration with external messaging systems.
- In Spring Cloud Stream, destination bindings (input and output) bridge between the producers and consumers in your applications with exchanges and queues in a message broker like RabbitMQ.
- Functions and consumers are activated automatically when new messages arrive.
- Suppliers need to be explicitly activated, for example, by explicitly sending a message to a destination binding.

11

Security: Authentication and SPA

This chapter covers

- Understanding the Spring Security fundamentals
- Managing user accounts with Keycloak
- Working with OpenID Connect, JWT, and Keycloak
- Authenticating users with Spring Security and OpenID Connect
- Testing Spring Security and OpenID Connect

Security is one of the most critical aspects of web applications and probably the one with the most catastrophic effects when done wrong. For educational purposes, I'm introducing this topic only now. In a real-world scenario, I recommend considering security from the beginning of each new project or feature and never letting it go until the application is retired.

Access control systems allow users access to resources only when their identity has been proven, and they have the required permissions. To accomplish that, we need to follow three pivotal steps: identification, authentication, and authorization.

1. **Identification** happens when a user (human or machine) claims an identity. In the physical world, that's when I introduce myself by stating my name. In the digital world, I would do that by providing my username or email address.
2. **Authentication** is about verifying the user's claimed identity through factors like a passport, a driver's license, a password, a certificate, or a token. When multiple factors are used to verify the user identity, we talk about *multi-factor authentication*.
3. **Authorization** always happens after authentication and checks what the user is allowed to do in a given context.

This chapter and the next one will cover how to implement access control systems in cloud native applications. In the following, you'll see how to add authentication to a system like Polar

Bookshop and use a dedicated identity and access management solution like Keycloak. I'll show you how to use Spring Security to secure applications and adopt standards like JWT, OAuth2, and OpenID Connect. During the progress, you'll also add an Angular frontend to the system and learn the best practices for security when a Single-Page Application (SPA) is involved.

NOTE

The source code for the examples in this chapter is available in the Chapter11/11-begin and Chapter11/11-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

11.1 Understanding the Spring Security fundamentals

Spring Security (<https://spring.io/projects/spring-security>) is the de-facto standard for securing Spring applications, supporting imperative and reactive stacks. It provides authentication and authorization features as well as protection against the most common attacks.

The framework provides its main functionality by relying on *filters*. Let's consider a possible requirement for adding authentication to a Spring Boot application. Users should be able to authenticate with their username and password through a login form. When we configure Spring Security to enable such a feature, the framework adds a filter that intercepts any incoming HTTP request. If the user is already authenticated, it sends the request through to be processed by a given web handler, such as a `@RestController` class. If the user is not authenticated, it forwards the user to a login page and prompts for their username and password.

NOTE

In imperative Spring applications, a filter is implemented as a `Servlet Filter` class. In reactive applications, the `WebFilter` class is used.

Most Spring Security features, when enabled, are handled via filters. The framework establishes a chain of filters that are executed according to a well-defined and sensible order. For instance, the filter handling authentication runs before the one checking for authorization because we can't verify a user's authority before knowing who it is.

Let's start with a basic example to understand better how Spring Security works. We want to add authentication to the Polar Bookshop system. Since Edge Service is the entry point, it makes sense to implement cross-cutting concerns like security there. Users should be able to authenticate with a username and password via a login form.

First, add a new dependency on Spring Security in the `build.gradle` file for the Edge Service project (`edge-service`). Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 11.1 Add dependency for Spring Security in Edge Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-security'
}
```

The central place for defining and configuring security policies in Spring Security is a `SecurityWebFilterChain` bean. That object tells the framework which filters should be enabled. You can build a `SecurityWebFilterChain` bean through the DSL provided by `ServerHttpSecurity`. For now, we want to comply with the following requirements:

- all endpoints exposed by Edge Service must require user authentication;
- the authentication must happen via a login form page.

To collect all the configurations related to security, create a `SecurityWebFilterChain` bean in a new `SecurityConfig` class (`com.polarbookshop.edgeservice.config` package).

```
@Bean ①
SecurityWebFilterChain springSecurityFilterChain(
    ServerHttpSecurity http
) {}
```

- ① The "SecurityWebFilterChain" bean is used to define and configure security policies for the application.

The `ServerHttpSecurity` object, autowired by Spring, provides a convenient DSL to configure Spring Security and build a `SecurityWebFilterChain` bean. With `authorizeExchange()`, you can define access policies for any request (called `exchange` in reactive Spring). In this case, we want all requests to require authentication (`authenticated()`).

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    return http
        .authorizeExchange(exchange ->
            exchange.anyExchange().authenticated() ①
        .build();
}
```

- ① All requests require authentication.

Spring Security provides several authentication strategies, including HTTP Basic, login form, SAML, and OpenID Connect. For this example, we want to use the login form strategy, which we can enable via the `formLogin()` method exposed by the `ServerHttpSecurity` object. We'll go with the default configuration (available via the Spring Security `Customizer` interface), which includes a login page provided by the framework out-of-the-box and an automatic redirect to it whenever a request is not authenticated.

```

@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    return http
        .authorizeExchange(exchange -> exchange.anyExchange().authenticated())
        .formLogin(Customizer.withDefaults()) ①
        .build();
}

```

- ① Enables user authentication via a login form.

Next, annotate the `SecurityConfig` class with `@EnableWebFluxSecurity` to enable Spring Security WebFlux support. The final security configuration is shown in listing 11.2.

Listing 11.2 Require authentication for all endpoints through a login form (`SecurityConfig.java`)

```

package com.polarbookshop.edgeservice.config;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.reactive.
➥EnableWebFluxSecurity;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http
    ) {
        return http
            .authorizeExchange(exchange ->
                exchange.anyExchange().authenticated()) ①
            .formLogin(Customizer.withDefaults()) ②
            .build();
    }
}

```

- ① All requests require authentication.
- ② Enables user authentication via a login form.

Let's verify that it works correctly. First, start your Redis container required by Edge Service. Open a Terminal window, navigate to the folder where you keep your Docker Compose file (`polar-deployment/docker/docker-compose.yml`) and run the following command.

```
$ docker-compose up -d polar-redis
```

Then, run the Edge Service application (`./gradlew bootRun`), open a browser window, and head to <http://localhost:9000/books>. You should get redirected to a login page served by Spring Security where you can authenticate.

Wait a minute! How can we authenticate without defining any user in the system? By default,

Spring Security defines a user account in memory with the username `user` and a password randomly generated and printed out in the application logs. You should look for a log entry like the following.

```
Using generated security password: ee60bdf6-fb82-439a-8ed0-8eb9d47bae08
```

Now, you can authenticate with the pre-defined user account created by Spring Security. After successfully authenticating, you are redirected to the `/books` endpoint. Since Catalog Service is down and Edge Service has a fallback method to return an empty list when querying books (implemented in chapter 9), you'll see a blank page. That's expected.

NOTE

I recommend you open a new incognito browser window every time you test the application from now on. Since you're going to try out different security scenarios, the incognito mode will prevent you from having issues related to browser cache and cookies from previous sessions.

The crucial point of this test is the following. A user tried to access a protected endpoint exposed by Edge Service. The application redirected the user to a login page, showed a login form, and asked the user to provide a username and password. Then, Edge Service validated the credentials against its internal user database (auto-generated in memory) and, if valid, started an authenticated session with the browser. Since HTTP is a stateless protocol, the user session is kept alive through a cookie whose value is provided by the browser with each HTTP request (*session cookie*). Internally, Edge Service maintains a mapping between the session identifier and user identifier, as shown in figure [11.1](#).

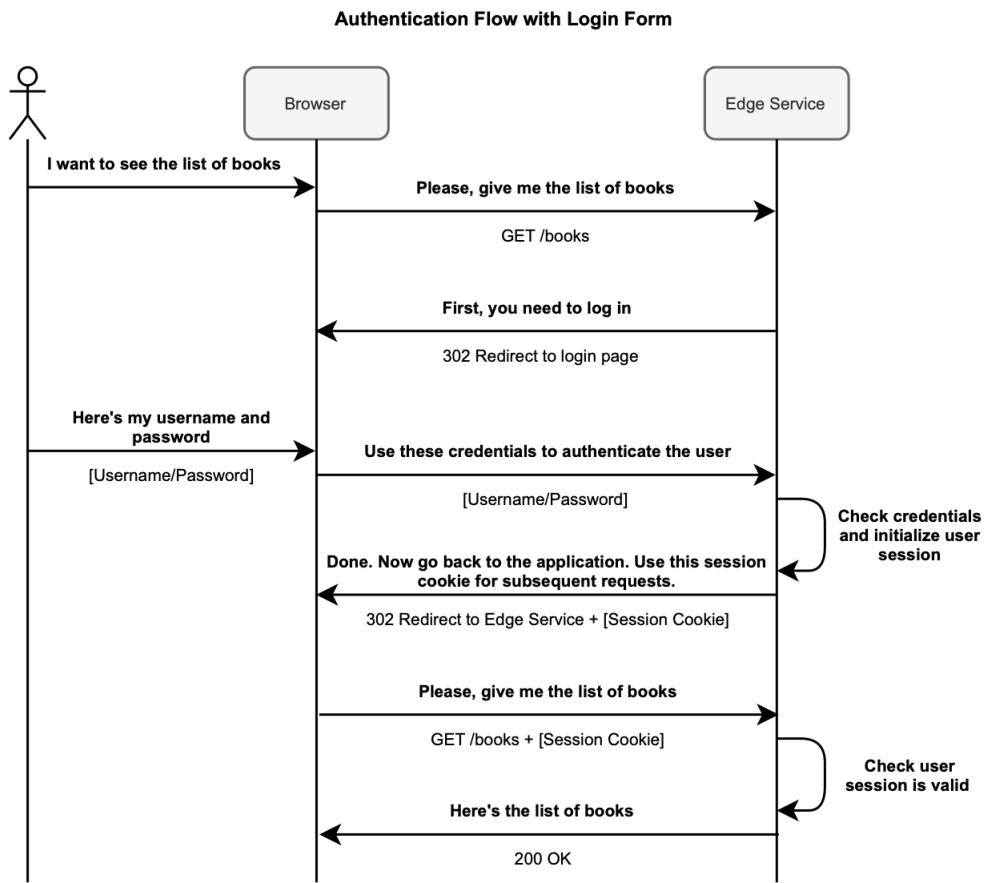


Figure 11.1 After the login step, the user session is kept alive through a session cookie.

When you are done testing the application, terminate the process with `ctrl+c`. Then, navigate to the folder where you keep your Docker Compose file (`polar-deployment/docker/docker-compose.yml`), and run the following command to stop the Redis container.

```
$ docker-compose down
```

There are a few problems with the previous approach when applied to a cloud native system. In the rest of the chapters, we'll analyze those problems, identify viable solutions for cloud native applications, and use them on top of what we have just implemented.

11.2 Managing user accounts with Keycloak

In the previous section, we added user authentication to Edge Service based on a login form. You tried logging in via a user account auto-generated in memory at startup time. That's fine for experimenting with Spring Security the first time, but it's not something you want to do in production.

As a minimum requirement, we need persistent storage for user accounts and an option to register new users. There should be a particular focus on storing passwords using a robust

encryption algorithm and preventing unauthorized access to the database. Given the criticality of such a feature, it makes sense to delegate it to a dedicated application.

Keycloak (<https://www.keycloak.org>) is an open-source identity and access management solution developed and maintained by the RedHat community. It offers a broad set of features, including single sign-on (SSO), social login, user federation, multi-factor authentication, and centralized user management. Keycloak relies on standards like OAuth2, OpenID Connect, and SAML 2.0. For now, we'll use Keycloak to manage user accounts in Polar Bookshop. Later, I'll show you how to use its OpenID Connect and OAuth2 features.

NOTE

Spring Security provides all the necessary features to implement a user management service. If you'd like to learn more about this subject, you can refer to chapters 3 and 4 from "Spring Security in Action" by Laureniu Spilc (<https://livebook.manning.com/book/spring-security-in-action/chapter-3>).

You can run Keycloak locally as a standalone Java application or a container. For production, there are a few solutions to run Keycloak on Kubernetes. Keycloak also needs a relational database for persistence. It comes built-in with an embedded H2 database, but you want to replace it with an external one in production.

For Polar Bookshop, we'll run Keycloak locally as a Docker container relying on the embedded H2 database. In production, we'll use PostgreSQL. It might seem in contradiction with the environment parity principle. Since it's a third-party application, it's not your responsibility to test its interaction with a data source.

This section will guide you step-by-step through the Keycloak configuration for the specific Polar Bookshop use case. First, open your `polar-deployment` repository. Then, define a new `polar-keycloak` container in `docker/docker-compose.yml`.

Listing 11.3 Define a Keycloak container in Docker Compose (docker-compose.yml)

```
version: "3.8"
services:
  ...
  polar-keycloak: ①
    image: quay.io/keycloak/keycloak:18.0
    container_name: "polar-keycloak"
    command: start-dev ②
    environment: ③
      - KEYCLOAK_ADMIN=user
      - KEYCLOAK_ADMIN_PASSWORD=password
    ports:
      - 8080:8080
```

- ① The section describing the Keycloak container.
- ② Starts Keycloak in development mode (using an embedded database).

- ③ Defines admin credentials as environment variables.

NOTE

Later, I will provide you with a JSON file you can use to load the entire configuration when starting up the Keycloak container without being concerned with its persistence.

You can start a Keycloak container by opening a Terminal window, navigating to the folder where you keep the `docker-compose.yml` file, and running this command.

```
$ docker-compose up -d polar-keycloak
```

Before we can start managing user accounts, we need to define a security realm.

11.2.1 Defining a security realm

In Keycloak, any security aspect of an application or a system is defined in the context of a *realm*, a logical domain in which we apply specific security policies. By default, Keycloak comes pre-configured with a *Master* realm, but you probably want to create a dedicated one for each product you build. Let's create a new *PolarBookshop* realm to host any security-related aspect of the Polar Bookshop system.

Make sure the Keycloak container you started earlier is still running. Then, open a Terminal window and enter a bash console inside the Keycloak container.

```
$ docker exec -it polar-keycloak bash
```

TIP

Keycloak might take several seconds to start up. If you try to access it right after starting the container, you might get an error because it's not ready yet to accept connections. If that happens, wait a few seconds and try again. You can check the Keycloak logs with `docker logs -f polar-keycloak`. After the message "Running the server in development mode" is printed out, Keycloak is ready to be used.

We'll configure Keycloak through its Admin CLI, but you could achieve the same result using the GUI available at <http://localhost:8080>. First, navigate to the folder where the Keycloak Admin CLI scripts are located.

```
$ cd /opt/keycloak/bin
```

The Admin CLI is protected by the username and password we defined in Docker Compose for the Keycloak container. We need to start an authenticated session before running any other command.

```
$ ./kcadm.sh config credentials \
--server http://localhost:8080 \ ❶
--realm master \ ❷
--user user \ ❸
--password password ❹
```

- ❶ Keycloak runs on port 8080 inside the container.
- ❷ The default realm configured in Keycloak.
- ❸ The username we defined in Docker Compose.
- ❹ The password we defined in Docker Compose.

TIP

You should keep the current Terminal window open until we're done with configuring Keycloak. If at any point the authenticated session expires, you can always start a new one by running the previous command.

At this point, you can go ahead and create a new security realm where all the policies associated with Polar Bookshop will be stored.

```
$ ./kcadm.sh create realms -s realm=PolarBookshop -s enabled=true
```

11.2.2 Managing users and roles

We'll need some users to test different authentication scenarios. As anticipated in chapter 2, Polar Bookshop has two types of users: customers and employees.

- *Customers* can browse books and purchase them,
- *Employees* can also add new books to the catalog, modify the existing ones, and delete them.

To manage the different permissions associated with each type of user, let's create two roles: *customer* and *employee*. Later, you'll protect application endpoints based on those roles. It's an authorization strategy called *Role-Based Access Control* (RBAC).

First, create the two roles in the Polar Bookshop realm from the Keycloak Admin CLI console you have used so far.

```
$ ./kcadm.sh create roles -r PolarBookshop -s name=employee
$ ./kcadm.sh create roles -r PolarBookshop -s name=customer
```

Then, create two users. *Isabelle Dahl* will be both an employee and a customer of the bookshop (username: *isabelle*). You can create an account for her as follows.

```
$ ./kcadm.sh create users -r PolarBookshop \
-s username=isabelle \①
-s firstName=Isabelle \
-s lastName=Dahl \
-s enabled=true ②

$ ./kcadm.sh add-roles -r PolarBookshop \
--username isabelle \③
--rolename employee \
--rolename customer
```

- ①** The username for the new user. It will be used for logging in.
- ②** The user should be active.
- ③** Isabelle is both an employee and a customer.

Then, do the same for *Bjorn Vinterberg* (username: bjorn), a customer of the bookshop.

```
$ ./kcadm.sh create users -r PolarBookshop \
-s username=bjorn \①
-s firstName=Bjorn \
-s lastName=Vinterberg\
-s enabled=true ②

$ ./kcadm.sh add-roles -r PolarBookshop \
--username bjorn \③
--rolename customer
```

- ①** The username for the new user. It will be used for logging in.
- ②** The user should be active.
- ③** Bjorn is a customer.

In a real scenario, users would choose a password themselves and preferably enable Two-Factor Authentication. Isabelle and Bjorn are test users, so assigning an explicit password (`password`) is ok. You can do that from the Keycloak Admin CLI as follows.

```
$ ./kcadm.sh set-password -r PolarBookshop \
--username isabelle --new-password password
$ ./kcadm.sh set-password -r PolarBookshop \
--username bjorn --new-password password
```

That's it for user management. You can get out of the bash console inside the Keycloak container with the `exit` command, but keep Keycloak running.

Next, let's explore how we can improve the authentication strategy in Edge Service.

11.3 Authentication with OpenID Connect, JWT, and Keycloak

At the moment, users must log in via a browser with a username and password. Since Keycloak now manages user accounts, we could go ahead and update Edge Service to check the user credentials with Keycloak itself rather than using its internal storage. What happens if we introduce different clients to the Polar Bookshop system, such as mobile applications and IoT devices? How should the users authenticate then? What if the bookshop employees are already registered in the company Active Directory (AD) and want to log in via SAML? Can we provide a Single-Sign-On (SSO) experience across different applications? Will the users be able to log in via their GitHub or Twitter accounts (*social login*)?

We could think of supporting all those authentication strategies in Edge Service as we get new requirements. However, that is not a scalable approach. A better solution is delegating a dedicated **identity provider** to authenticate users following any supported strategy. Edge Service would then use that service to verify the identity of a user without being concerned with performing the actual authentication step. The dedicated service could let users authenticate in various ways, for example, using the credentials registered in the system, through social login, or via SAML to rely on the identity defined in the company AD.

Using a dedicated service to authenticate users leads to two aspects we need to address to make the system work. First, we need to establish a protocol for Edge Service to delegate user authentication to the identity provider and for the latter to provide information about the authentication result. Second, we need to define a data format that the identity provider can use to inform Edge Service in a secure way about the identity of users after they have been successfully authenticated. This section will address both issues using OpenID Connect (OIDC) and JSON Web Token (JWT).

11.3.1 Authenticating users with OpenID Connect

OpenID Connect (OIDC) is a protocol that enables an application (called *Client*) to verify the identity of a user based on the authentication performed by a trusted party (called an *Authorization Server*) and retrieve the user profile information. The Authorization Server informs the Client application about the result of the authentication step via an *ID Token*.

OIDC is an identity layer on top of OAuth2, an authorization framework that solves the problem of delegating access using tokens for authorization but doesn't deal with authentication. As you know, authorization can only happen after authentication. That's why I decided to cover OIDC first, whereas OAuth2 will be further explored in the next chapter. It's not the typical way of covering those subjects, but I think it makes sense when designing an access control system like we're doing for Polar Bookshop.

NOTE

This book will only cover some essential aspects of OAuth2 and OIDC. If you're interested in learning more about them, Manning has a couple of books in its catalog on the subject: "OAuth 2 in Action" by Justin Richer and Antonio Sanso (<https://www.manning.com/books/oauth-2-in-action>) and "OpenID Connect in Action" by Prabath Siriwardena (<https://www.manning.com/books/openid-connect-in-action>).

When it comes to handling user authentication, we can identify three main actors in the OAuth2 framework, which are used by the OIDC protocol.

- **Authorization Server.** The entity responsible for authenticating users and issuing tokens. In Polar Bookshop, this will be Keycloak.
- **User.** Also called *Resource Owner*. It's the human logging in with the Authorization Server to get authenticated access to the Client application. In Polar Bookshop, it's either a customer or an employee.
- **Client.** The application requiring the user to be authenticated. It can be a mobile application, a browser-based application, a server-side application, or even a smart TV application. In Polar Bookshop, it's Edge Service.

Figure 11.2 shows how the three actors are mapped into the Polar Bookshop architecture.

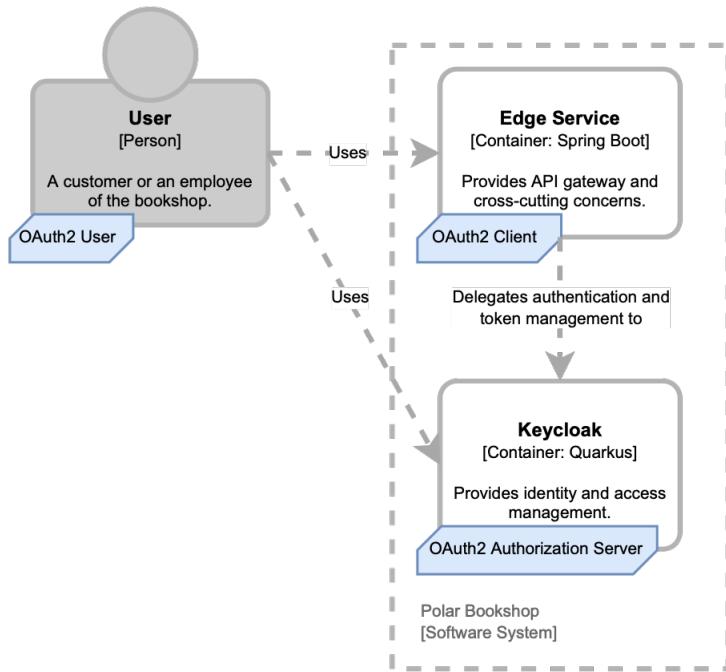


Figure 11.2 How the OIDC/OAuth2 roles are assigned to the entities in the Polar Bookshop architecture for user authentication.

NOTE

The roles defined by the OAuth2 framework are also known with different names when used in the context of OpenID Connect. The OAuth2 Authorization Server is also called OIDC Provider. The OAuth2 Client, which relies on the Authorization Server for authentication and token issuing, is also called Relying Party (RP). The OAuth2 User is also called End-User. We'll stick to the OAuth2 naming for consistency, but it's helpful to know the alternative terminology used in OIDC.

In Polar Bookshop, Edge Service will initiate the user login flow, but then it will delegate the actual authentication step to Keycloak via the OIDC protocol (supported out-of-the-box by Spring Security). Keycloak provides several authentication strategies, including traditional login forms, social logins via providers like GitHub or Twitter, and SAML. It also supports Two-Factor Authentication (2FA). In the following, we'll use the login form strategy as an example. Since users will interact directly with Keycloak to log in, their credentials are never exposed to any component of the system except Keycloak, which is one of the benefits of adopting such a solution.

When an unauthenticated user calls a secure endpoint exposed by Edge Service, the following happens:

1. Edge Service (Client) redirects the browser to Keycloak (Authorization Server) for authentication.
2. Keycloak authenticates the user (for example, by asking for a username and password via a login form) and then redirects the browser back to Edge Service together with an *Authorization Code*.
3. Edge Service calls Keycloak to exchange the Authorization Code with an ID Token, containing information about the authenticated user.
4. Edge Service initializes an authenticated user session with the browser based on a session cookie. Internally, Edge Service maintains a mapping between the session identifier and ID Token (the user identity).

NOTE

The authentication flow supported by OIDC is based on the OAuth2 Authorization Code Flow. The second step might seem redundant, but the Authorization Code is essential for ensuring that only the legitimate Client can exchange it with tokens.

Figure 11.3 describes the essential parts of the authentication flow supported by the OIDC protocol. Even if Spring Security supports that out-of-the-box and you won't need to implement any of that yourself, it's still beneficial to have an overview of the flow in mind.

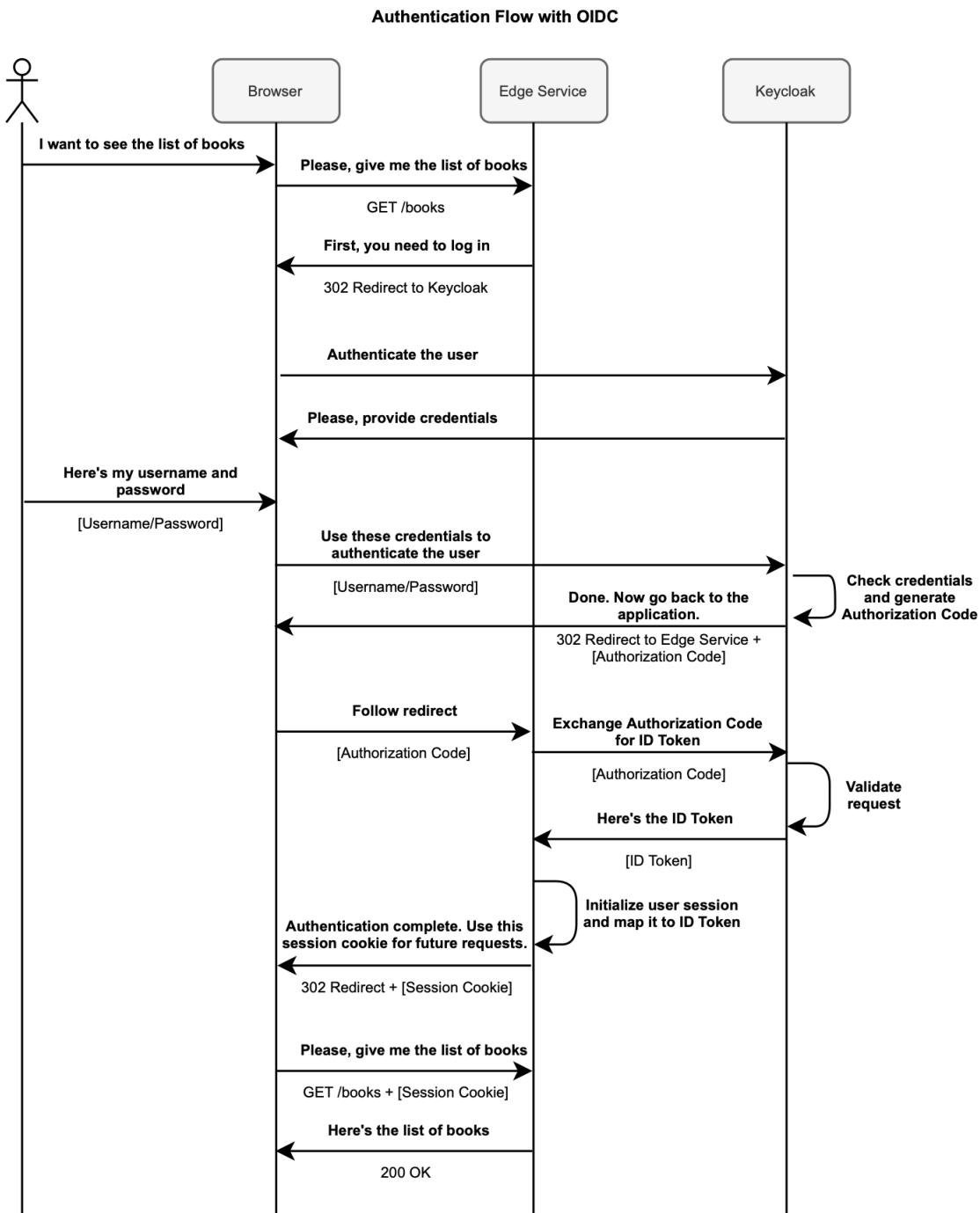


Figure 11.3 The authentication flow supported by the OIDC protocol.

When adopting the authentication flow illustrated in figure 11.3, Edge Service is not affected by the specific authentication strategy. We could configure Keycloak to use an Active Directory or perform a social login via GitHub. Yet, Edge Service would not require any change. It only needs to support OIDC to verify the authentication happened correctly and get information about the user via an ID Token. What's an ID Token? It's a *JSON Web Token* (JWT) containing information about the user authentication event. Let's have a closer look at JWTs in the next section.

NOTE Whenever I mention OIDC, I refer to the OpenID Connect Core 1.0 specification (https://openid.net/specs/openid-connect-core-1_0.html). Whenever I mention OAuth2, unless specified otherwise, I refer to the OAuth 2.1 specification currently under standardization and meant to replace the OAuth 2.0 standard described in the RFC 6749 (<https://tools.ietf.org/html/rfc6749>).

11.3.2 Exchanging user information with JWT

In distributed systems, including microservices and cloud native applications, the most-used strategy for exchanging information about an authenticated user and their authorization is through tokens.

JSON Web Token (JWT) is an industry-standard for representing *claims* to be transferred between two parties. It's a widely used format to propagate information about an authenticated user and their permissions securely among different parties in a distributed system. A JWT is not used by itself, but it's included in a larger structure, the JSON Web Signature (JWS), which ensures the integrity of the claims by digitally signing the JWT object.

A digitally signed JWT (JWS) is a string composed of three parts encoded in Base64 and separated by a . character (a dot). For debugging purposes, you can use the tool available on <https://jwt.io> to encode and decode tokens.

```
<header>.<payload>.<signature>
```

- **Header.** A JSON object containing information about the cryptographic operations performed on the payload (*JOSE Header*). The operations follow the standards from the Javascript Object Signing and Encryption framework (JOSE). A decoded header looks like the following.

```
{
  "alg": "HS256", ①
  "typ": "JWT" ②
}
```

- ① The algorithm used to sign the token digitally.
- ② The type of token.

- **Payload.** A JSON object containing the claims conveyed by the token (*Claims Set*). The JWT specification defines some standard claim names, but you can also define your own. A decoded payload looks as follows.

```
{
  "iss": "https://sso.polarbookshop.com", ①
  "sub": "isabelle", ②
  "exp": 1626439022 ③
}
```

- ① The entity that issued the JWT (issuer).
 - ② The entity that is the subject of the JWT (the end-user).
 - ③ When the JWT expires (timestamp).
- **Signature.** The signature of the JWT, ensuring the claims have not been tampered with. A prerequisite of using a JWS structure is that we trust the entity issuing the token (the *issuer*), and we have a way to check its validity.

When a JWT requires integrity and confidentiality, it's first signed as a JWS and then encrypted with JSON Web Encryption (JWE). In this book, we'll use only JWS.

NOTE

If you're interested in learning more about JWT and its related aspects, you can refer to the IETF standard specifications. JSON Web Token (JWT) is documented in RFC 7519 (<https://tools.ietf.org/html/rfc7519>), JSON Web Signature (JWS) is described in RFC 7515 (<https://tools.ietf.org/html/rfc7515>), JSON Web Encryption (JWE) is presented in RFC 7516 (<https://tools.ietf.org/html/rfc7516>). You might also be interested in JSON Web Algorithms (JWA), which defines the available cryptographic operations for JWTs and is detailed in RFC 7518 (<https://tools.ietf.org/html/rfc7518>).

In the case of Polar Bookshop, Edge Service can delegate the authentication step to Keycloak. After authenticating the user successfully, Keycloak would send a JWT to Edge Service with information about the newly authenticated user (ID Token). Edge Service would validate the JWT through its signature and inspect it to retrieve data (claims) about the user. Finally, it would establish an authenticated session with the user browser based on a session cookie, whose identifier is mapped to the JWT.

To delegate authentication and retrieve tokens securely, Edge Service must be registered as an OAuth2 Client in Keycloak. Let's see how.

11.3.3 Registering an application in Keycloak

As you learned in the previous sections, an OAuth2 Client is an application that can request user authentication and ultimately receive tokens from an Authorization Server. In the Polar Bookshop architecture, this role is played by Edge Service. When using OIDC/OAuth2, you need to register each OAuth2 Client with the Authorization Server before using it for authenticating users.

Clients can be *public* or *confidential*. We register an application as a public Client if it can't keep a secret. For example, mobile applications would be registered as public Clients. On the other hand, confidential Clients are those that can keep a secret, and they are usually backend applications like Edge Service. The registration process is similar either way. The main

difference is that confidential Clients are required to authenticate themselves with the Authorization Server, for example, by relying on a shared secret. It's an additional protection layer we can't use for public Clients since they have no way to store the shared secret securely.

SIDE BAR

The Client dilemma in OAuth2

The Client role can be assigned either to a frontend or a backend application. The main difference is in the level of security of the solution. The Client is the entity that will receive tokens from the Authorization Server and will have to store them somewhere to be used in subsequent requests from the same user. Tokens are sensitive data that should be secured, and there's no better place than a backend application to do that. But it's not always possible.

My rule of thumb is the following. If the frontend is a mobile or desktop application like iOS or Android, that will be the OAuth2 Client. In this case, it would be categorized as a public Client. You can use libraries like AppAuth (<https://appauth.io>) to add support for OIDC/OAuth2 and store the tokens as securely as possible on-device.

If the frontend is a web application (like in Polar Bookshop), then a backend service should be the Client. In this case, it would be categorized as a confidential Client. The reason is that no matter how much you try to hide the OIDC/OAuth2 tokens in the browser (cookies, local storage, session storage), they will always be at risk of being exposed and misused.

"From a security perspective, it is virtually impossible to secure tokens in a frontend web application." That's what the application security expert Philippe De Ryck writes, recommending engineers rely on the backend-for-frontend pattern to have a backend application deal with tokens instead. (<http://mng.bz/QWG6>)

I recommend keeping the interaction between browser and backend based on a session cookie (like you'd do for monoliths) and having the backend application responsible for controlling the authentication flow and using the tokens issued by the Authorization Server, even in case of SPAs. That is the current best practice recommended by security experts.

Since Edge Service will be the OAuth2 Client in the Polar Bookshop system, let's register it with Keycloak. We can rely on the Keycloak Admin CLI one more time.

Make sure the Keycloak container you started earlier is still running. Then, open a Terminal window and enter a bash console inside the Keycloak container.

```
$ docker exec -it polar-keycloak bash
```

Next, navigate to the folder where the Keycloak Admin CLI scripts are located.

```
$ cd /opt/keycloak/bin
```

As you learned earlier, the Admin CLI is protected by the username and password we defined in Docker Compose for the Keycloak container, so we need to start an authenticated session before running any other command.

```
$ ./kcadm.sh config credentials --server http://localhost:8080 \
--realm master --user user --password password
```

Finally, register Edge Service as an OAuth2 Client in the PolarBookshop realm.

```
$ ./kcadm.sh create clients -r PolarBookshop \
-s clientId=edge-service \ ①
-s enabled=true \ ②
-s publicClient=false \ ③
-s secret=polar-keycloak-secret \ ④
-s 'redirectUris=["http://localhost:9000",
➥ "http://localhost:9000/login/oauth2/code/*"]' ⑤
```

- ① The OAuth2 Client identifier.
- ② It must be enabled.
- ③ Edge Service is a confidential client, not public.
- ④ Since it's a confidential client, it needs a secret to authenticate with Keycloak.
- ⑤ The application URLs to which Keycloak is authorized to redirect a request after a user login or logout.

The valid redirect URLs are the endpoints exposed by the OAuth2 Client application (Edge Service), where Keycloak will redirect authentication requests. Since Keycloak can include sensitive information in a redirect request, we want to limit which applications and endpoints are authorized to receive such information. As you'll learn later, the redirect URL for authentication requests will be http://localhost:9000/login/oauth2/code/*, following the default format provided by Spring Security. To support redirects after logout operations, we also need to add <http://localhost:9000> as a valid redirect URL.

That's it for this section. In the source code repository accompanying the book, I included a JSON file you can use to load the entire configuration when starting up the Keycloak container in the future (Chapter11/11-end/polar-deployment/docker/keycloak/realm-config.json). After familiarizing yourself with Keycloak, you can now update the container definition to ensure you always have the needed configuration at startup. Copy the JSON file over the same path in your own project and update the polar-keycloak service as follows.

Listing 11.4 Import realm configuration in Keycloak container (docker-compose.yml)

```
version: "3.8"
services:
  ...
  polar-keycloak:
    image: quay.io/keycloak/keycloak:18.0
    container_name: "polar-keycloak"
    command: start-dev --import-realm ①
    volumes: ②
      - ./keycloak:/opt/keycloak/data/import
    environment:
      - KEYCLOAK_ADMIN=user
      - KEYCLOAK_ADMIN_PASSWORD=password
    ports:
      - 8080:8080
```

- ① Imports the provided configuration at startup time.
- ② Configures a volume to load the configuration file into the container.

SIDE BAR

Why Keycloak

I decided to use Keycloak since it's a mature, open-source solution to run an Authorization Server yourself. After increasing demands from the community, Spring started a new Spring Authorization Server project (<https://github.com/spring-projects/spring-authorization-server>). Since version 0.2.0, it's a production-ready solution to set up an OAuth2 Authorization Server. At the time of writing, the project provides an implementation for the most common OAuth2 features and is currently working on extending support for OIDC-specific features. You can follow the progress and contribute to the project on GitHub.

Another option would be to use a SaaS solution like Okta (<https://www.okta.com>) or Auth0 (<https://auth0.com>). They are both excellent solutions to get OIDC/OAuth2 as a managed service, and I encourage you to try them out. For this book, I wanted to use a solution that you could run and reliably reproduce in your local environment without depending on other services that could change over time, with the risk of providing instructions that will not be valid anymore at some point.

Before moving on, let's stop any running container. Open a Terminal window, navigate to the folder where you keep your Docker Compose file (polar-deployment/docker/docker-compose.yml), and run the following command.

```
$ docker-compose down
```

We now have all the pieces to refactor Edge Service to use an authentication strategy relying on OIDC/OAuth2, JWT, and Keycloak. The best part is that it's based on standards and supported by all major languages and frameworks (frontend, backend, mobile, IoT), including Spring

Security.

11.4 Authenticating users with Spring Security and OpenID Connect

As mentioned earlier, Spring Security supports several authentication strategies. The current security setup for Edge Service handles user accounts and authentication via a login form provided by the application itself. After learning about OpenID Connect, we can refactor the application to delegate user authentication to Keycloak via the OIDC protocol.

Support for OAuth2 used to be in a separate project called *Spring Security OAuth*, which you would use as part of *Spring Cloud Security* to adopt OAuth2 in a cloud native application. Both those projects are now deprecated in favor of the native, more comprehensive support for OAuth2 and OpenID Connect introduced in the main *Spring Security* project starting from version 5. This chapter focuses on using the new OIDC/OAuth2 support added in Spring Security 5 to authenticate the users of Polar Bookshop.

NOTE

In case you find yourself working on a project using the deprecated Spring Security OAuth and Spring Cloud Security projects, you might want to check out chapters 12 through 15 from "Spring Security in Action" by Laureniu Spilc (<https://livebook.manning.com/book/spring-security-in-action/chapter-12>), where they are explained in great detail.

Using Spring Security and its OAuth2/OIDC support, this section will show you how to do the following for Edge Service:

- use OpenID Connect to authenticate users;
- configure user logout;
- extract information about the authenticated user.

Let's get started!

11.4.1 Adding the new dependencies

First of all, we need to update the dependencies for Edge Service. We can replace the existing Spring Security starter dependency with the more specific OAuth2 Client one, which adds support for OIDC/OAuth2 client features. Furthermore, we can add the Spring Security Test dependency, which provides additional support for testing security scenarios in Spring.

Open the `build.gradle` file for the Edge Service project (`edge-service`) and add the new dependencies. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 11.5 Add dependencies for Spring Security OAuth2 Client in Edge Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-client'
    testImplementation 'org.springframework.security:spring-security-test'
}
```

SIDE BAR **Spring integration with Keycloak**

When choosing Keycloak as the Authorization Server, an alternative to the native OpenID Connect/OAuth2 support provided by Spring Security was the Keycloak Spring Adapter. It was a library supplied by the Keycloak project itself to integrate with Spring Boot and Spring Security, but it's been retired after the release of Keycloak 17.

In case you find yourself working on a project using the Keycloak Spring Adapter, you might want to check out my articles on the subject (<https://www.thomasvitale.com/tag/keycloak>) or chapter 9 from "Spring Microservices in Action" by John Carnell and Illary Huaylupo Sánchez (<https://livebook.manning.com/book/spring-microservices-in-action-second-edition/chapter-9>).

11.4.2 Configuring the integration between Spring Security and Keycloak

After adding the relevant dependencies on Spring Security, we need to configure the integration with Keycloak. In the previous section, we registered Edge Service in Keycloak as an OAuth2 Client, defining both a client identifier (edge-service) and a shared secret (polar-keycloak-secret). Now, we'll use that information to tell Spring Security how to interact with Keycloak.

Open the `application.yml` file in your Edge Service project and add the following configuration.

Listing 11.6 Configure Edge Service as an OAuth2 Client with Spring Security (application.yml)

```

spring:
  security:
    oauth2:
      client:
        registration:
          keycloak: ①
            client-id: edge-service ②
            client-secret: polar-keycloak-secret ③
            scope: openid ④
        provider:
          keycloak: ⑤
            issuer-uri:
              ↵http://localhost:8080/realms/PolarBookshop ⑥

```

- ① The name identifying the Client registration in Spring Security (called "registrationId"). It can be any string.
- ② The OAuth2 Client identifier as defined in Keycloak.
- ③ The shared secret used by the Client to authenticate with Keycloak.
- ④ The list of scopes the Client would like to have access to. The "openid" scope triggers OIDC authentication on top of OAuth2.
- ⑤ The same name used as "registrationId" a few lines above.
- ⑥ The Keycloak URL providing information about all the relevant OAuth2 and OIDC endpoints for the specific realm.

Each Client registration in Spring Security must have an identifier (*registrationId*). In the example, it's `keycloak`. The registration identifier is used to build the URL where Spring Security receives the Authorization Code by Keycloak. The default URL template is `/login/oauth2/code/{registrationId}`. For Edge Service, the full URL is <http://localhost:9000/login/oauth2/code/keycloak> that we have already configured in Keycloak as a valid redirect URL.

Scopes are an OAuth2 concept to limit the access of an application to user resources. You can think of them as roles, but for applications instead of users. When we use the OpenID Connect extension on top of OAuth2 to verify the user identity, we need to include the `openid` scope to inform the Authorization Server and receive an ID Token containing data about the user authentication. The next chapter will tell you more about scopes in the context of authorization.

After defining the integration with Keycloak, let's see how to configure Spring Security to apply the desired security policies.

11.4.3 Basic Spring Security configuration

The central place for defining and configuring security policies in Spring Security is a `SecurityWebFilterChain`. Edge Service is currently configured to require user authentication for all endpoints and uses an authentication strategy based on a login form. Let's change that to use OIDC authentication instead.

The `ServerHttpSecurity` object provides two ways of configuring an OAuth2 Client in Spring Security. With `oauth2Login()`, you can configure an application to act as an OAuth2 Client and also authenticate users through OpenID Connect. With `oauth2Client()`, the application will not authenticate users, so it's up to you to define another authentication mechanism. We want to use OIDC authentication, so we'll use `oauth2Login()` and the default configuration.

Listing 11.7 Require authentication for all endpoints through OIDC (SecurityConfig.java)

```
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http
    ) {
        return http
            .authorizeExchange(exchange ->
                exchange.anyExchange().authenticated())
            .oauth2Login(Customizer.withDefaults()) ①
            .build();
    }
}
```

- ① Enables user authentication with OAuth2/OpenID Connect.

Let's verify that it works correctly. First, start the Redis and Keycloak containers. Open a Terminal window, navigate to the folder where you keep your Docker Compose file (`polar-deployment/docker/docker-compose.yml`), and run the following command.

```
$ docker-compose up -d polar-redis polar-keycloak
```

Then, run the Edge Service application (`./gradlew bootRun`), open a browser window, and head to <http://localhost:9000>. You should get redirected to a login page served by Keycloak, where you can authenticate as one of the users we created previously (figure 11.4).

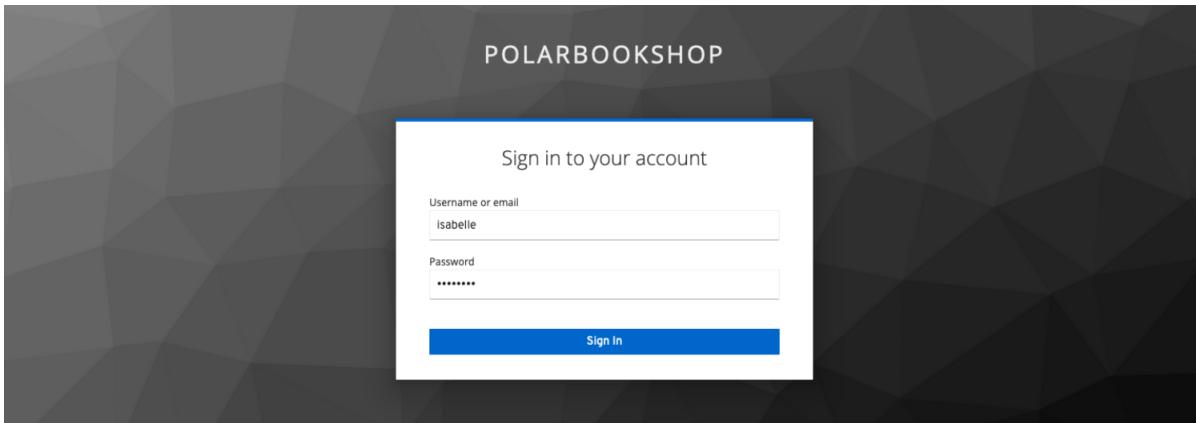


Figure 11.4 The Keycloak login page for the Polar Bookshop realm, shown after Edge Service triggered the OIDC authentication flow.

For example, log in as Isabelle (isabelle/password) and pay attention to how Keycloak redirects you back to Edge Service after validating the provided credentials. Since Edge Service doesn't expose any content through the root endpoint, you'll see an error message ("Whitelabel Error Page"). But don't worry! That's where we'll integrate an Angular frontend later. The crucial point of this test is that Edge Service required you to be authenticated before accessing any of its endpoints and triggered the OIDC authentication flow.

When you're done trying out the OIDC authentication flow, stop the application with `Ctrl+C`.

If the authentication is successful, Spring Security starts an authenticated session with the browser and saves information about the user. In the next section, you'll see how we can retrieve and use that information.

11.4.4 Inspecting the authenticated user context

As part of the authentication process, Spring Security defines a context to hold information about the user and map a user session to an ID Token. In this section, you'll learn more about such a context, what classes are involved, how to retrieve the data and expose it through a new `/user` endpoint in Edge Service.

First, let's define a `User` model to collect the username, first name, last name, and roles of an authenticated user. That is the same information we provided when registering the two users in Keycloak, and that is returned in the ID Token. In a new `com.polarbookshop.edgeservice.user` package, create a `User` record as follows.

Listing 11.8 The User record collects basic information about an authenticated user (User.java)

```
package com.polarbookshop.edgeservice.user;

import java.util.List;

public record User( ①
    String username,
    String firstName,
    String lastName,
    List<String> roles
){}
```

- ① Immutable data class holding user data.

Independently from the authentication strategy adopted (e.g., username/password, OpenID Connect/OAuth2, or SAML2), Spring Security keeps the information about an authenticated user (also called *principal*) in an `Authentication` object. In the case of OIDC, the *principal* object is of type `OidcUser` and it's where Spring Security stores the ID Token. In turn, `Authentication` is saved in a `SecurityContext` object.

One way to access the `Authentication` object for the currently logged-in user is extracting it from the related `SecurityContext` retrieved from the `ReactiveSecurityContextHolder` (or `SecurityContextHolder` for imperative applications). Figure 11.5 illustrates how all the mentioned objects are related with each other.

Security context structure for OIDC authentication in Spring Security

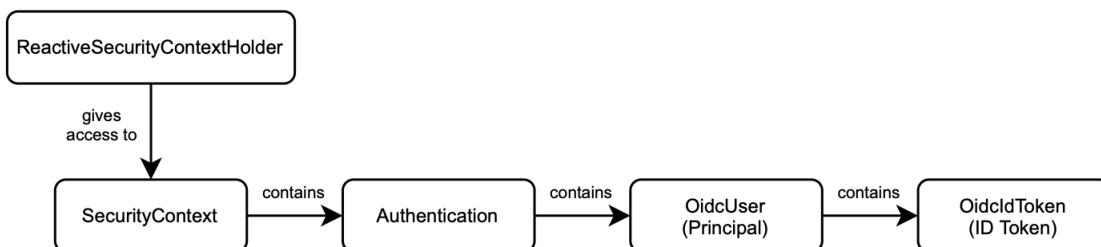


Figure 11.5 The main classes used to store information about the currently authenticated user.

Let's see how that works.

1. Create a `UserController` class annotated with `@RestController` in the `com.polarbookshop.edgeservice.user` package.
2. Define a method to handle GET requests to a new `/user` endpoint.
3. Return a `User` object for the currently authenticated user, retrieving the necessary information from `OidcUser`. To get the correct data, we can use the call hierarchy shown in figure 11.5.

```

@GetMapping("user")
public Mono<User> getUser() {
    return ReactiveSecurityContextHolder.getContext() ①
        .map(SecurityContext::getAuthentication) ②
        .map(authentication ->
            (OidcUser) authentication.getPrincipal()) ③
        .map(oidcUser -> ④
            new User(
                oidcUser.getPreferredUsername(),
                oidcUser.getGivenName(),
                oidcUser.getFamilyName(),
                List.of("employee", "customer"))
        )
    );
}

```

- ① Gets SecurityContext for the currently authenticated user from ReactiveSecurityContextHolder.
- ② Gets Authentication from SecurityContext.
- ③ Gets the principal from Authentication. For OIDC, it's of type OidcUser.
- ④ Builds a User object using data from OidcUser (extracted from the ID Token).

In the next chapter, focused on authorization strategies, we'll configure Keycloak to include a custom `roles` claim in the ID Token and use that value to build the `User` object in the `UserController` class. Until then, we'll use a fixed list of values.

For Spring Web MVC and WebFlux controllers, besides using `ReactiveSecurityContextHolder` directly, we have the option to use the annotations `@CurrentSecurityContext` and `@AuthenticationPrincipal` to inject the `SecurityContext` and the principal (in this case, `OidcUser`) respectively.

Let's simplify the implementation of the `getUser()` method by injecting the `OidcUser` object directly as an argument. The final result is shown in listing 11.9.

Listing 11.9 Defines a REST endpoint to return information about the currently authenticated user (UserController.java)

```
package com.polarbookshop.edgeservice.user;

import java.util.List;
import reactor.core.publisher.Mono;
import org.springframework.security.core.annotation.
➡AuthenticationPrincipal;
import org.springframework.security.oauth2.core.oidc.user.OidcUser;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @GetMapping("user")
    public Mono<User> getUser(
        @AuthenticationPrincipal OidcUser oidcUser ①
    ) {
        var user = new User( ②
            oidcUser.getPreferredUsername(),
            oidcUser.getGivenName(),
            oidcUser.getFamilyName(),
            List.of("employee", "customer")
        );
        return Mono.just(user); ③
    }
}
```

- ① Injects an "OidcUser" object containing info about the currently authenticated user.
- ② Builds a "User" object from relevant claims contained in "OidcUser".
- ③ Wraps the "User" object in a reactive publisher since Edge Service is a reactive application.

Ensure Keycloak and Redis are still running from the previous section, run the Edge Service application (`./gradlew bootRun`), open an incognito browser window, and navigate to <http://localhost:9000/user>. Spring Security will redirect you to Keycloak, which will prompt you to log in with a username and password. For example, authenticate as Bjorn (`bjorn/password`). After successfully authenticating, you will be redirected back to the `/user` endpoint. The result is the following.

```
{
    "username": "bjorn",
    "firstName": "Bjorn",
    "lastName": "Vinterberg",
    "roles": [
        "employee",
        "customer"
    ]
}
```

NOTE The `roles` list includes hard-coded values. In the next chapter, we'll fix it to return the actual roles assigned to each user in Keycloak.

When you're done trying out the new endpoint, stop the application with `ctrl+c` and the containers with `docker-compose down`.

Consider what happened when you tried to access the `/user` endpoint and got redirected to Keycloak. After successfully validating the user credentials, Keycloak called Edge Service back and sent the ID Token for the newly authenticated user. Then, Edge Service stored the token and redirected the browser to the required endpoint together with a session cookie. From that point on, any communication between the browser and Edge Service would use that session cookie to identify the authenticated context for that user. No token is exposed to the browser.

The ID Token is stored in `OidcUser`, part of `Authentication` and ultimately included in `SecurityContext`. In chapter 9, we used the Spring Session project to make Edge Service store session data in an external data service (Redis) to remain stateless and be able to scale out. `SecurityContext` objects are included in the session data and are therefore stored in Redis automatically, making it possible for Edge Service to scale out without any problem.

Another option for retrieving the currently authenticated user (called *principal*) is from the context associated with a specific HTTP request (called *exchange*). We'll use that option to update the rate limiter configuration. In chapter 9, we implemented rate-limiting with Spring Cloud Gateway and Redis. Currently, the rate-limiting is computed based on the total number of requests received every second. We should update it to apply the rate limits to each user independently.

Open the `RateLimiterConfig` class and configure how to extract the username of the currently authenticated principal from the request. If no user is defined (that is, the request is unauthenticated, *anonymous*), we use a default key to apply rate-limiting to all unauthenticated requests as a whole.

Listing 11.10 Configure rate limiting for each user (`RateLimiterConfig.java`)

```
@Configuration
public class RateLimiterConfig {

    @Bean
    KeyResolver keyResolver() {
        return exchange -> exchange.getPrincipal()          ①
            .map(Principal::getName)                      ②
            .defaultIfEmpty("anonymous");                 ③
    }
}
```

- ① Gets the currently authenticated user (the principal) from the current request (the exchange).
- ② Extracts the username from the principal.
- ③ If the request is unauthenticated, it uses "anonymous" as the default key to apply rate-limiting.

That concludes the basic configuration for authenticating the users of Polar Bookshop using OpenID Connect. The following section will cover how logout works in Spring Security and how to customize it for the OAuth2/OIDC scenario.

11.4.5 Configuring user logout in Spring Security and Keycloak

So far, we have addressed the challenges and solutions for authenticating users in a distributed system. Still, we should also consider what happens when a user logs out.

In Spring Security, logging out results in all the session data associated with the user getting deleted. When OpenID Connect/OAuth2 is used, the tokens stored by Spring Security for that user are also deleted. However, the user will keep having an active session in Keycloak. Just like the authentication process involves both Keycloak and Edge Service, completely logging a user out requires propagating the logout request to both components.

By default, a logout performed against an application protected by Spring Security will not affect Keycloak. Fortunately, Spring Security provides an implementation of the *OpenID Connect - RP-Initiated Logout* specification, which defines how to propagate a logout request from an OAuth2 Client (i.e., the Relying Party) to the Authorization Server. You'll see how to configure it for Edge Service soon.

NOTE	The OpenID Connect specification includes a few different scenarios for session management and logout. If you'd like to learn more, I recommend you check the official documents for OIDC Session Management (https://openid.net/specs/openid-connect-session-1_0.html), OIDC Front-Channel Logout (https://openid.net/specs/openid-connect-frontchannel-1_0.html), OIDC Back-Channel Logout (https://openid.net/specs/openid-connect-backchannel-1_0.html), and OIDC RP-Initiated Logout (https://openid.net/specs/openid-connect-rpinitiated-1_0.html).
-------------	--

Spring Security supports logging out by sending a POST request to the `/logout` endpoint implemented and exposed by the framework by default. We want to enable the RP-Initiated Logout scenario so that when a user logs out of the application, they are also logged out of the Authorization Server. Spring Security has full support for this scenario and provides an `OidcClientInitiatedServerLogoutSuccessHandler` object you can use to configure how to propagate a logout request to Keycloak.

When you enable the RP-Initiated Logout, after a user has been successfully logged out of Spring Security, Edge Service will send a logout request to Keycloak through the browser (using a redirect). Next, you probably want the user to be redirected back to the application after the logout operation has been performed on the Authorization Server as well.

You can configure where to redirect the user after a logout through the `setPostLogoutRedirectUri()` method exposed by the `OidcClientInitiatedServerLogoutSuccessHandler` class. You might specify a direct URL, but that will not work well in a cloud environment due to many variables in terms of hostnames, service names, and protocols (`http` vs. `https`). The Spring Security team knew that and added support for placeholders that get resolved at runtime dynamically. Instead of hard-coding a URL value, you can use the `{baseUrl}` placeholder. When you run Edge Service locally, the placeholder will get resolved to `http://localhost:9000`. If you ran it in the cloud behind a proxy with TLS termination and accessible through the DNS name `polarbookshop.com`, it would be automatically replaced with `https://polarbookshop.com`.

However, the Client configuration in Keycloak requires an exact URL. That's why we added `http://localhost:9000` to the list of valid redirect URLs when we registered Edge Service in Keycloak. In production, you'll have to update the list of valid redirect URLs in Keycloak to match the actual URLs used there.

Figure [11.6](#) illustrates the logout scenario I've just described.

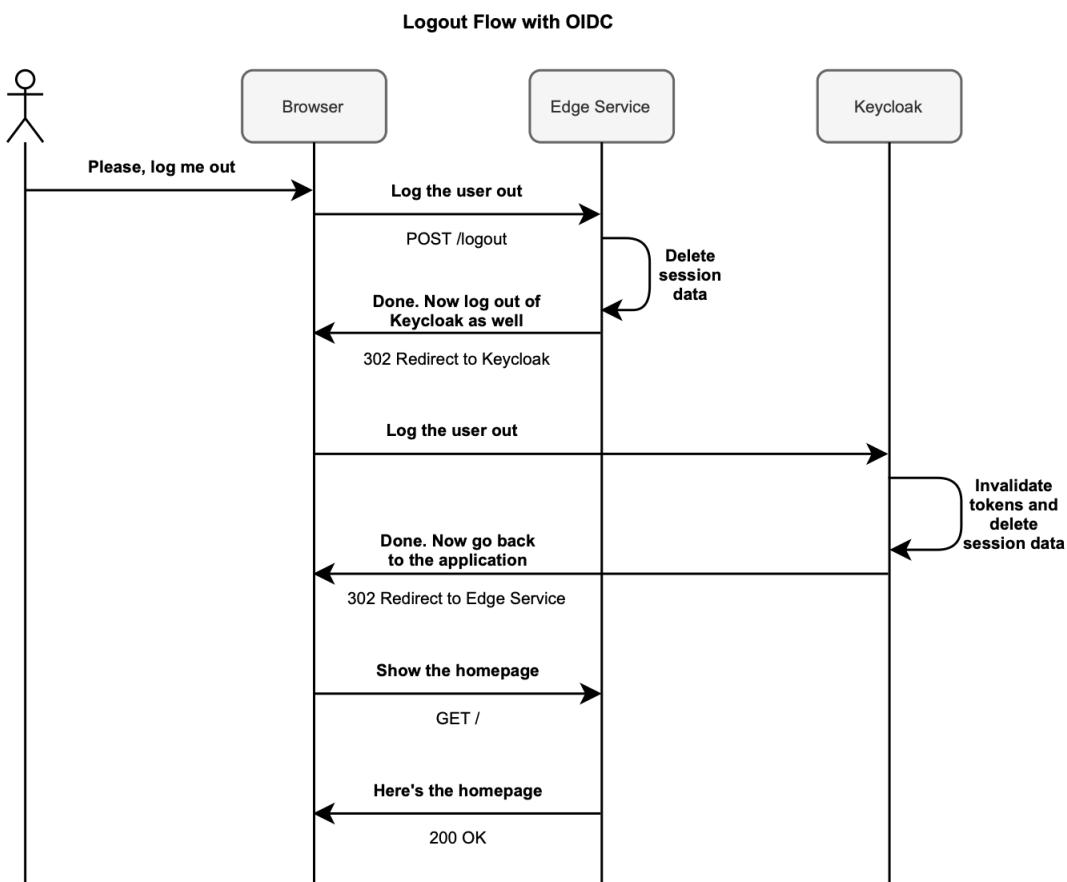


Figure 11.6 When a user logs out, the request is processed by Spring Security first, then forwarded to Keycloak, and the user is finally redirected to the application.

Since the application logout functionality is already provided by default in Spring Security, you only need to enable and configure the RP-Initiated Logout for Edge Service.

1. In the `SecurityConfig` class, define an `oidcLogoutSuccessHandler()` method to build an `OidcClientInitiatedServerLogoutSuccessHandler` object.
2. Configure the post-logout redirect URL using the `setPostLogoutRedirectUri()` method.
3. Call the `oidcLogoutSuccessHandler()` method from the `logout()` configuration defined in the `SecurityWebFilterChain` bean.

Listing 11.11 Configure RP-Initiated Logout and redirect to the application base URL (SecurityConfig.java)

```
package com.polarbookshop.edgeservice.config;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.reactive.
➥EnableWebFluxSecurity;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.oauth2.client.oidc.web.server.logout.
OidcClientInitiatedServerLogoutSuccessHandler;
import org.springframework.security.oauth2.client.registration.
ReactiveClientRegistrationRepository;
import org.springframework.security.web.server.SecurityWebFilterChain;
import org.springframework.security.web.server.authentication.logout.
ServerLogoutSuccessHandler;

@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationRepository
    ) {
        return http
            .authorizeExchange(exchange ->
                exchange.anyExchange().authenticated()
            )
            .oauth2Login(Customizer.withDefaults())
            .logout(logout -> logout.logoutSuccessHandler( ①
                oidcLogoutSuccessHandler(clientRegistrationRepository)))
            .build();
    }

    private ServerLogoutSuccessHandler oidcLogoutSuccessHandler(
        ReactiveClientRegistrationRepository clientRegistrationRepository
    ) {
        var oidcLogoutSuccessHandler =
            new OidcClientInitiatedServerLogoutSuccessHandler(
                clientRegistrationRepository);
        oidcLogoutSuccessHandler
            .setPostLogoutRedirectUri("{baseUrl}"); ②
        return oidcLogoutSuccessHandler;
    }
}
```

- ① Defines a custom handler for the scenario where a logout operation is completed successfully.
- ② After logging out from the OIDC Provider, it redirects to the application base URL computed dynamically from Spring (locally, it's <http://localhost:9000>).

NOTE

The `ReactiveClientRegistrationRepository` is a bean automatically configured by Spring Boot for storing the information about the clients registered with Keycloak and used by Spring Security for authentication/authorization purposes. In our example, there's only a client: the one we configured earlier in the `application.yml` file.

I won't ask you to test the logout functionality just yet. The reason will be apparent after introducing the Angular frontend to the Polar Bookshop system.

The user authentication feature based on OpenID Connect/OAuth2 is now complete, including logout and scalability concerns. If Edge Service used a template engine like Thymeleaf to build the frontend, then the work done so far would be enough. However, when integrating a secured backend application with an SPA like Angular, there are a few more aspects to consider. That will be the focus of the next section.

11.5 Integrating Spring Security with SPAs

The web frontend part of microservices architectures and other distributed systems is often built as one or more single-page applications using frameworks like Angular, React, or Vue. Analyzing how SPAs are created is not in the scope of this book, but it's essential to look at what changes are needed to support such frontend clients.

So far, you have interacted with the services composing the Polar Bookshop system through a Terminal window. In this section, we'll add an Angular application that will be the frontend of the system. It will be served by an NGINX container and accessible via the gateway provided by Edge Service. Supporting an SPA will require some additional configuration in Spring Security to address concerns like Cross-Origin Request Sharing (CORS) and Cross-Site Request Forgery (CSRF). This section shows how to do that.

11.5.1 Running an Angular application

The Polar Bookshop system will have an Angular application as the frontend. Since this book doesn't cover frontend technologies and patterns, I have prepared one already. How can we include it in the Polar Bookshop system?

One option is to let Edge Service serve the SPA static resources. Spring Boot applications serving a frontend usually host the source code in `src/main/resources`. That's a convenient strategy when using template engines like Thymeleaf, but for SPAs like Angular, I prefer keeping the code in a separate module. SPAs have their own development, build, and release

tools, so having a dedicated folder is cleaner and more maintainable. Then, you can configure Spring Boot to process the SPA static resources at build time and include them in the final release.

Another option is having a dedicated service taking care of serving the Angular static resources. That's the strategy we'll use for Polar Bookshop. I have already packaged the Angular application in an NGINX container. NGINX (<https://nginx.org>) provides HTTP server features, and it's very convenient for serving static resources such as the HTML, CSS, and JavaScript files composing the Angular application.

Let's go ahead and run the Polar Bookshop frontend (`polar-ui`) in Docker. First, go to your `polar-deployment` repository and open your Docker Compose file (`docker/docker-compose.yml`). Then, add the configuration to run `polar-ui` and expose it through port 9004.

**Listing 11.12 Configuration for running the Angular application as a container
(`docker-compose.yml`)**

```
version: "3.8"
services:
  ...
  polar-ui:
    image: "ghcr.io/polarbookshop/polar-ui:latest" ①
    container_name: "polar-ui"
    ports:
      - 9004:9004 ②
    environment:
      - PORT=9004 ③
```

- ① The container image I built to package the Angular application.
- ② NGINX will serve the SPA on port 9004.
- ③ Configures the NGINX server port.

Similar to the other applications in the Polar Bookshop system, we don't want the Angular application to be accessible directly from the outside. Instead, we want to make it accessible via the gateway provided by Edge Service. We can do that by adding a new route for Spring Cloud Gateway to forward any request for static resources to the Polar UI application.

Go to your Edge Service project (`edge-service`), open the `application.yml` file, and configure the new route as follows.

Listing 11.13 Configure a new gateway route for the SPA static resources (application.yml)

```
spring:
  gateway:
    routes:
      - id: spa-route ①
        uri: ${SPA_URL:http://localhost:9004} ②
        predicates:
          - Path=/,/*.css,/*.js,/favicon.ico
```

- ① The route ID.
- ② The URI value comes from an environment variable, or else the specified default.
- ③ The predicate is a list of paths matching the root endpoint and the SPA static resources.

The URI for the Polar UI application is computed using the value from an environment variable (`SPA_URL`). If it's not defined, the default value written after the first `:` symbol will be used.

NOTE

When running Edge Service as a container, remember to configure the `SPA_URL` environment variable. On Docker, you can use the container name and port as the value, resulting in <http://polar-ui:9004>.

Let's test it out. First, run the Polar UI container together with Redis and Keycloak. Open a Terminal window, navigate to the folder where you keep your Docker Compose file (`polar-deployment/docker/docker-compose.yml`), and run the following command.

```
$ docker-compose up -d polar-ui polar-redis polar-keycloak
```

Then, build the Edge Service project again and run the application (`./gradlew bootRun`). Finally, open an incognito browser window and navigate to <http://localhost:9000>.

Spring Security is configured to protect all endpoints and resources, so you automatically get redirected to the Keycloak login page. After you authenticate as either Isabelle or Bjorn, you get redirected back to the Edge Service root endpoint from which the Angular frontend is served.

For now, there's not much you can do. The authentication flow is triggered by Spring Security when receiving an unauthenticated request, but it doesn't work if it's an AJAX request due to CORS issues. Furthermore, POST requests (including the logout operation) will fail due to the CSRF protection enabled by Spring Security. In the following sections, I'll show you how to update the Spring Security configuration to overcome those issues. Before moving on, stop the application with `ctrl+C` (but keep the containers running, you'll need them!).

11.5.2 Controlling the authentication flow

In the previous section, you tried accessing the Edge Service homepage and experienced being automatically redirected to Keycloak to provide a username and password. When the frontend consists of server-rendered pages (for example, when using Thymeleaf), that behavior works fine, and it's convenient since it doesn't require any extra configuration. If you're not authenticated yet, or your session has expired, Spring Security will automatically trigger the authentication flow and redirect your browser to Keycloak.

With a single-page application, things work a bit differently. The Angular application is returned by the backend when accessing the root endpoint through a standard HTTP GET request performed by the browser. After that first step, the SPA interacts with the backend through AJAX requests. When the SPA sends an unauthenticated AJAX request to a protected endpoint, you don't want Spring Security to reply with an HTTP 302 response redirecting to Keycloak. Instead, you want it to return a response with an error status like HTTP 401 Unauthorized.

The main reason is that you would run into Cross-Origin Request Sharing (CORS) issues. Consider the scenario where an SPA is served from <https://client.polarbookshop.com> and makes HTTP calls through AJAX to a backend at <https://server.polarbookshop.com>. The communication is blocked because the two URLs don't have the same origin (i.e., same protocol, domain, and port). That's the standard same-origin policy enforced by all web browsers.

CORS is a mechanism to allow a server to accept HTTP calls through AJAX from a browser-based client like an SPA, even if the two have different origins. In Polar Bookshop, we serve the Angular frontend via the gateway implemented in Edge Service (same origin). Therefore, there won't be any CORS issue between these two components. However, suppose Spring Security is configured to reply to an unauthenticated AJAX call with a redirect to Keycloak (having a different origin). In that case, the request will be blocked because redirects to different origins are not permitted during AJAX requests.

NOTE

To learn more about CORS in Spring Security, you can check out chapter 10 from "Spring Security in Action" by Laureniu Spilc (<https://livebook.manning.com/book/spring-security-in-action/chapter-10/>) where the subject is explained in great detail. For a comprehensive explanation of CORS, Manning has a book in its catalog on the topic: "CORS in Action" by Monsur Hossain (<https://livebook.manning.com/book/cors-in-action>).

When changing the Spring Security configuration to reply with an HTTP 401 response to unauthenticated requests, it's up to the SPA to handle the error and call the backend to initiate

the authentication flow. Redirects are only a problem during AJAX requests. The crucial part here is that the call to the backend for starting the user authentication is not an AJAX request sent by Angular. Instead, it's a standard HTTP call sent from the browser, like the following.

```
login(): void {
  window.open('/oauth2/authorization/keycloak', '_self');
}
```

I'd like to stress how the login call is not an AJAX request sent from the Angular `HttpClient`. Instead, it instructs the browser to call the login URL. Spring Security exposes an `/oauth2/authorization/{registrationId}` endpoint that you can use to start the authentication flow based on OAuth2/OIDC. Since the client registration identifier for Edge Service is 'keycloak', the login endpoint will be `/oauth2/authorization/keycloak`.

To make that possible, we need to define a custom `AuthenticationEntryPoint` to instruct Spring Security to reply with an HTTP 401 status when an unauthenticated request is received for a protected resource. The framework already provides an `HttpStatusServerEntryPoint` implementation that perfectly suits this scenario since it lets you specify which HTTP status to return when a user is required to authenticate.

Listing 11.14 Define an authentication entry point to return 401 when a request requires authentication (`SecurityConfig.java`)

```
@EnableWebFluxSecurity
public class SecurityConfig {
  ...

  @Bean
  SecurityWebFilterChain springSecurityFilterChain(
    ServerHttpSecurity http,
    ReactiveClientRegistrationRepository clientRegistrationRepository
  ) {
    return http
      .authorizeExchange(exchange -> exchange.anyExchange().authenticated())
      .exceptionHandling(exceptionHandling ->
        exceptionHandling.authenticationEntryPoint( ①
          new HttpStatusServerEntryPoint(HttpStatus.UNAUTHORIZED)))
      .oauth2Login(Customizer.withDefaults())
      .logout(logout -> logout.logoutSuccessHandler(
        oidcLogoutSuccessHandler(clientRegistrationRepository)))
      .build();
  }
}
```

- ① When an exception is thrown because a user is not authenticated, it replies with an HTTP 401 response.

At this point, the Angular application can explicitly intercept HTTP 401 responses and trigger the authentication flow. However, since the SPA is now in charge of starting the flow, we need to permit unauthenticated access to its static resources. We would also like to retrieve the books

in the catalog without being authenticated, so let's permit GET requests to the `/books/**` endpoints as well. Go ahead and update the `SecurityWebFilterChain` bean in the `SecurityConfig` class as follows.

Listing 11.15 Allow unauthenticated GET requests to the SPA static resources and the book catalog (SecurityConfig.java)

```
@EnableWebFluxSecurity
public class SecurityConfig {
    ...
    @Bean
    SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationRepository
    ) {
        return http
            .authorizeExchange(exchange -> exchange
                .pathMatchers("/", "/*.css", "/*.js", "/favicon.ico")
                .permitAll() ①
                .pathMatchers(HttpMethod.GET, "/books/**")
                .permitAll() ②
                .anyExchange().authenticated() ③
            )
            .exceptionHandling(exceptionHandling -> exceptionHandling
                .authenticationEntryPoint(
                    new HttpStatusServerEntryPoint(HttpStatus.UNAUTHORIZED)))
            .oauth2Login(Customizer.withDefaults())
            .logout(logout -> logout.logoutSuccessHandler(
                oidcLogoutSuccessHandler(clientRegistrationRepository)))
            .build();
    }
}
```

- ①** Allows unauthenticated access to the SPA static resources.
- ②** Allows unauthenticated read access to the books in the catalog.
- ③** Any other request requires user authentication.

Let's test how Edge Service works now. Ensure that the Polar UI, Redis, and Keycloak containers are still running. Next, build and run the Edge Service application (`./gradlew bootRun`), then go to <http://localhost:9000> from an incognito browser window. The first thing to notice is that you don't get redirected to the login page, but are presented with the Angular frontend application immediately. You can start the authentication flow by clicking the "Login" button from the upper right menu.

After logging in, the upper-right menu contains a "Logout" button, only shown if the current user is successfully authenticated. Click the button to log out. It should trigger the logout flow, but it won't work due to CSRF issues. In the next section, you'll learn how to fix that. In the meantime, stop the application with `Ctrl+C`.

11.5.3 Protecting against Cross-Site Request Forgery

The interaction between the frontend and backend is based on a session cookie. After the user is successfully authenticated with the OIDC/OAuth2 strategy, Spring will generate a session identifier to match the authenticated context and send it to the browser as a cookie. Any subsequent request to the backend will have to contain the session cookie, from which Spring Security can retrieve the tokens associated with the specific user and validate the request.

However, session cookies are not enough to validate requests, which are vulnerable to Cross-Site Request Forgery (CSRF) attacks. CSRF affects *modifying* HTTP requests like POST, PUT, and DELETE. An attacker could induce users to perform requests they didn't intend to by forging a request meant to cause harm. Examples of forged requests could be transferring money from your bank account or compromising critical data.

WARNING Many online tutorials and guides show how to disable the CSRF protection as the first thing when configuring Spring Security. That's dangerous to do without explaining the reasoning or considering the consequences. I recommend keeping the protection enabled unless there's a good reason not to (we'll see one good reason in chapter 12). As a general guideline, browser-faced applications like Edge Service should be protected against CSRF attacks.

Fortunately, Spring Security has built-in protection against such attacks. The protection is based on a so-called CSRF token generated by the framework, provided to the client at the beginning of the session, and required to be sent along with any state-changing requests.

NOTE To learn more about CSRF protection in Spring Security, you can check out chapter 10 from "Spring Security in Action" by Laurentiu Spilca (<https://livebook.manning.com/book/spring-security-in-action/chapter-10>) where the subject is explained in great detail.

In the previous section, you tried to log out, but the request failed. Since the logout operation is available through a POST request to the `/logout` endpoint, it expects to receive the CSRF token generated by Spring Security for that user session. By default, the generated CSRF token is sent to the browser as an HTTP header. However, Angular applications cannot work with that and expect to receive the token value as a cookie. Spring Security supports this specific requirement, but it's not enabled by default.

You can instruct Spring Security to provide CSRF tokens as cookies through the `csrf()` DSL exposed by `ServerHttpSecurity` and the `CookieServerCsrfTokenRepository` class. For imperative applications, that would be enough. On the other hand, for reactive applications like Edge Service, you need an extra step to ensure the `CsrfToken` value is actually provided.

In chapter 8, you learned that reactive streams need to be subscribed to in order to activate them. At the moment, `CookieServerCsrfTokenRepository` doesn't ensure a subscription to `CsrfToken`, so you must explicitly provide a workaround in a `WebFilter` bean. This problem should be solved in future versions of Spring Security (<https://mng.bz/XW89>). For now, update the `SecurityConfig` class as follows.

**Listing 11.16 Configures CSRF to support a cookie-based strategy for SPAs
(`SecurityConfig.java`)**

```
@EnableWebFluxSecurity
public class SecurityConfig {
    ...
    @Bean
    SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationRepository
    ) {
        return http
            ...
            .csrf(csrf -> csrf.csrfTokenRepository( ①
                CookieServerCsrfTokenRepository.withHttpOnlyFalse()))
            .build();
    }

    @Bean
    WebFilter csrfWebFilter() { ②
        return (exchange, chain) -> {
            exchange.getResponse().beforeCommit(() -> Mono.defer(() -> {
                Mono<CsrfToken> csrfToken =
                    exchange.getAttribute(CsrfToken.class.getName());
                return csrfToken != null ? csrfToken.then() : Mono.empty();
            }));
            return chain.filter(exchange);
        };
    }
}
```

- ① Uses a cookie-based strategy for exchanging CSRF tokens with the Angular frontend.
- ② A filter with the only purpose of subscribing to the `CsrfToken` reactive stream and ensuring its value is extracted correctly.

Let's verify if the logout flow works now. Ensure that the Polar UI, Redis, and Keycloak containers are still up and running. Next, build and run the application (`./gradlew bootRun`), then go to <http://localhost:9000> from an incognito browser window. Start the authentication flow by clicking the "Login" button from the upper right menu. Then, click the "Logout" button. Under the hood, Spring Security will now accept your logout request (Angular adds the CSRF token value from the cookie as an HTTP header), terminate your web session, propagate the request to Keycloak, and finally redirect you to the homepage, unauthenticated.

Thanks to this change, you can also perform any POST, PUT, and DELETE requests without receiving any CSRF error. Feel free to explore the Angular application. If you start up Catalog

Service and Order Service, you can try out adding new books to the catalog, modifying them, or placing an order.

Both Isabelle and Bjorn can perform any action at the moment, which is not what we want since customers (like Bjorn) shouldn't be allowed to manage the book catalog. The next chapter will cover authorization, and you'll see how to protect each endpoint with different access policies. Before addressing authorization, though, we need to write autotests to cover the new functionality. That's coming up in the next section.

Before moving on, stop the application with `ctrl+c` and all the containers with `docker-compose down` (from `polar-deployment/docker`).

11.6 Testing Spring Security and OpenID Connect

The importance of writing autotests is usually apparent to developers. Still, things can get challenging when it comes to security, and the risk is that it ends up not being covered by automated tests because of its complexity. Fortunately, Spring Security provides several utilities to help you include security in your sliced and integration tests in a simple way.

In this section, you'll learn how to use the `WebTestClient` support for Spring Security to test OIDC authentication and CSRF protection. Let's get started.

11.6.1 Testing OIDC authentication

In chapter 8, we tested REST controllers exposed by Spring WebFlux relying on the `@SpringWebFlux` annotation and `WebTestClient`. Since in this chapter we added a new controller (`UserController`), let's write some autotests for it with different security setups.

First of all, open your Edge Service project, create a `UserControllerTests` class annotated with `@WebFluxTest(UserController.class)` in `src/test/java`, and autowire a `webTestClient` bean. So far, the setup is similar to what we used in chapter 8: a sliced test for the web layer. We need some extra setup to cover security scenarios, as shown in the following listing.

Listing 11.17 Define a class for testing the security policies for the UserController (UserControllerTests.java)

```
@WebFluxTest(UserController.class)
@Import(SecurityConfig.class) ①
class UserControllerTests {

    @Autowired
    WebTestClient webClient;

    @MockBean ②
    ReactiveClientRegistrationRepository clientRegistrationRepository;
}
```

- ① Imports the application security configuration.
- ② A mock bean to skip the interaction with Keycloak when retrieving information about the Client registration.

Since we configured Edge Service to return an HTTP 401 response when a request is unauthenticated, let's verify that's true when calling the `/user` endpoint without authenticating first.

```
@Test
void whenNotAuthenticatedThen401() {
    webClient
        .get()
        .uri("/user")
        .exchange()
        .expectStatus().isUnauthorized();
}
```

To test the scenario where a user is authenticated, we can use `mockOidcLogin()`, a configuration object supplied by `SecurityMockServerConfigurers` to mock an OIDC login, synthesize an ID Token, and mutate the request context in `WebTestClient` accordingly.

The `/user` endpoint reads claims from the ID Token through the `OidcUser` object, so we need to build an ID Token with username, first name, and last name (the roles are hard-coded in the controller, for now). The following snippet shows how to do that.

```
@Test
void whenAuthenticatedThenReturnUser() {
    var expectedUser = new User("jon.snow", "Jon", "Snow",
        List.of("employee", "customer")); ①

    webClient
        .mutateWith(configureMockOidcLogin(expectedUser)) ②
        .get()
        .uri("/user")
        .exchange()
        .expectStatus().is2xxSuccessful() ③
        .expectBody(User.class)
        .value(user -> assertThat(user).isEqualTo(expectedUser));
}

private SecurityMockServerConfigurers.OidcLoginMutator
➥ configureMockOidcLogin(User expectedUser) {
    return SecurityMockServerConfigurers.mockOidcLogin().idToken(
        builder -> { ④
            builder.claim(StandardClaimNames.PREFERRED_USERNAME,
                expectedUser.username());
            builder.claim(StandardClaimNames.GIVEN_NAME,
                expectedUser.firstName());
            builder.claim(StandardClaimNames.FAMILY_NAME,
                expectedUser.lastName());
        });
}
```

- ① The expected authenticated user.
- ② Defines an authentication context based on OIDC and uses the expected user.

- ③ Expects a User object with the same information of the currently authenticated user.
- ④ Builds a mock ID Token.

Finally, run the tests as follows.

```
$ ./gradlew test --tests UserControllerTests
```

The testing utilities supplied by Spring Security cover a wide array of scenarios and integrate well with `WebTestClient`. In the next section, you'll see how to test CSRF protection using a similar approach.

11.6.2 Testing CSRF

In Spring Security, CSRF protection applies to all mutating HTTP requests by default (e.g., POST, PUT, DELETE). As you saw in previous sections, Edge Service accepts POST requests to the `/logout` endpoint to initiate the logout flow, and such requests require a valid CSRF token to be executed. Furthermore, we configured the RP-Initiated Logout feature from OIDC, so a POST request to `/logout` will actually result in an HTTP 302 response redirecting the browser to Keycloak to also log the user out of there.

Create a new `SecurityConfigTests` class and use the same strategy you learned in the previous section to set up a Spring WebFlux test with security support, as shown in the following snippet.

Listing 11.18 Define a class for testing the authentication flow (`SecurityConfigTests.java`)

```
@WebFluxTest
@Import(SecurityConfig.class) ①
class SecurityConfigTests {

    @Autowired
    WebTestClient webClient;

    @MockBean ②
    ReactiveClientRegistrationRepository clientRegistrationRepository;
}
```

- ① Imports the application security configuration.
- ② A mock bean to skip the interaction with Keycloak when retrieving information about the Client registration.

Then, add a test case to check if the application returns an HTTP 302 response after sending an HTTP POST request to the `/logout` with the correct OIDC login and CSRF context.

```

@Test
void whenLogoutAuthenticatedAndWithCsrfTokenThen302() {
    when(clientRegistrationRepository.findByRegistrationId("test"))
        .thenReturn(Mono.just(testClientRegistration()));

    webClient
        .mutateWith(
            SecurityMockServerConfigurers.mockOidcLogin()) ①
        .mutateWith(SecurityMockServerConfigurers.csrf()) ②
        .post()
        .uri("/logout")
        .exchange()
        .expectStatus().isFound(); ③
}

private ClientRegistration testClientRegistration() { ④
    return ClientRegistration.withRegistrationId("test")
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .clientId("test")
        .authorizationUri("https://sso.polarbookshop.com/auth")
        .tokenUri("https://sso.polarbookshop.com/token")
        .redirectUri("https://polarbookshop.com")
        .build();
}

```

- ① Uses a mock ID Token for authenticating the user.
- ② Enhances the request to provide the required CSRF token.
- ③ The response is a redirect to Keycloak to propagate the logout operation.
- ④ A mock ClientRegistration used by Spring Security to get the URLs to contact Keycloak.

Finally, run the tests as follows.

```
$ ./gradlew test --tests SecurityConfigTests
```

As always, you can find more test examples in the source code repository accompanying this book. When it comes to security, unit and integration tests are critical to ensure the correctness of an application, but they are not enough. Those tests cover the default security configuration, which might be different in production. That's why we also need security-oriented autotests in the acceptance stage of the deployment pipeline (as explained in chapter 3), testing applications deployed in a production-like environment.

SIDEBAR

Polar Labs

Until now, the only application supposed to be accessed by users directly was Edge Service. All the other Spring Boot applications interact with each other from within the environment where they are deployed.

Service-to-service interactions within the same Docker network or Kubernetes cluster can be configured using the container name or the Service name, respectively. For example, Edge Service forwards requests to Polar UI via the `http://polar-ui:9004` URL on Docker (`<container-name>:<container-port>`) and via the `http://polar-ui` URL on Kubernetes (Service name).

Keycloak is different because it's involved in service-to-service interactions (for now, it's just with Edge Service) and also interactions with end users via the web browser. In production, Keycloak will be accessible via a public URL that both applications and users will use, so there will be no problem. How about local environments?

Since we don't deal with public URLs when working locally, we need to configure things differently. On Docker, we can solve the problem by using the `http://host.docker.internal` special URL configured automatically when installing the software. It resolves to your localhost IP address and can be used both within a Docker network and outside.

On Kubernetes, we don't have a generic URL to let Pods within a cluster access your local host. That means Edge Service will interact with Keycloak via its Service name (`http://polar-keycloak`). When Spring Security redirects a user to Keycloak to log in, the browser will return an error because the `http://polar-keycloak` URL cannot be resolved outside the cluster. To make that possible, we can update the local DNS configuration to resolve the `polar-keycloak` hostname to the cluster IP address. Then, a dedicated Ingress will make it possible to access Keycloak when requests are directed to the `polar-keycloak` hostname.

If you're on Linux or macOS, you can map the `polar-keycloak` hostname to the `minikube` local IP address in the `/etc/hosts` file. On Linux, the IP address is the one returned by the `minikube ip --profile polar` command (as explained in chapter 9). On macOS, it's going to be `127.0.0.1`. Open a Terminal window, and run the following command. Make sure you replace the `<ip-address>` placeholder with the cluster IP address, depending on your operating system.

```
$ echo "<ip-address> polar-keycloak" | sudo tee -a /etc/hosts
```

On Windows, you must map the `polar-keycloak` hostname to `127.0.0.1` in the `hosts` file. Open a PowerShell window as an administrator, and run the following command.

```
$ Add-Content C:\Windows\System32\drivers\etc\hosts "127.0.0.1 polar-keycloak"
```

I have updated the scripts for deploying all backing services for Polar Bookshop, including Keycloak and Polar UI. You can get them from the /Chapter11/11-end/polar-deployment/kubernetes/platform/development folder in the code repository accompanying the book (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and copy them into the same path in your polar-deployment repository. The deployment also includes the configuration of a dedicated Ingress for Keycloak, accepting requests directed to the polar-keycloak hostname.

At this point, you can run the `./create-cluster.sh` script (polar-deployment/kubernetes/platform/development) to start a minikube cluster and deploy all the backing services for Polar Bookshop. If you're on Linux, you'll be able to access Keycloak directly. If you're on macOS or Windows, remember to run the `minikube tunnel --profile polar` command first. Either way, you can open a browser window and access Keycloak at `polar-keycloak/` (include the final slash).

Finally, try running the entire system on Kubernetes after updating the deployment scripts for Edge Service to configure the URLs for Polar UI and Keycloak. You can refer to the Chapter11/11-end folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

The next chapter will expand on the security subject and cover how to propagate the authentication context from Edge Service to the downstream applications, and how to configure authorization.

11.7 Summary

- Access control systems require identification (who are you?), authentication (can you prove it's really you?), and authorization (what are you allowed to do?).
- A common strategy to implement authentication and authorization in cloud native applications is based on JWT as the data format, OAuth2 as the authorization framework, and OpenID Connect as the authentication protocol.
- When using OIDC authentication, a Client application initiates the flow and delegates an Authorization Server for the actual authentication. Then, the Authorization Server issues an ID Token to the Client.
- The ID Token includes information about the user authentication.
- Keycloak is an identity and access management solution that supports OAuth2 and OpenID Connect and can be used as an Authorization Server.
- Spring Security provides native support for OAuth2 and OpenID Connect, and you can use it to turn Spring Boot applications into OAuth2 Clients.
- In Spring Security, you can configure both authentication and authorization in a `SecurityWebFilterChain` bean. To enable the OIDC authentication flow, you can use the `oauth2Login()` DSL.
- By default, Spring Security exposes a `/logout` endpoint for logging a user out.
- In an OIDC/OAuth2 context, we also need to propagate the logout request to the Authorization Server (such as Keycloak) to log the user out of there. We can do that via the RP-Initiated Logout flow supported by Spring Security via the `OidcClientInitiatedServerLogoutSuccessHandler` class.
- When a secure Spring Boot application is the backend for an SPA, we need to configure CSRF protection through cookies and implement an authentication entry point that returns an HTTP 401 response when a request is not authenticated (as opposed to the default HTTP 302 response redirecting to the Authorization Server automatically).
- The Spring Security Test dependency supplies several convenient utilities to test security.
- The `WebTestClient` bean can be enhanced by mutating its request context through a particular configuration for OIDC login and CSRF protection.

12

Security: Authorization and auditing

This chapter covers

- Authorization and roles with Spring Cloud Gateway and OAuth2
- Protecting APIs with Spring Security and OAuth2 (imperative)
- Protecting APIs with Spring Security and OAuth2 (reactive)
- Protecting and auditing data with Spring Security and Spring Data

In the previous chapter, I introduced access control systems for cloud native applications. You saw how to add authentication to Edge Service with Spring Security and OpenID Connect, manage the user session lifecycle, and address CORS and CSRF concerns when integrating an Angular frontend with Spring Boot.

By delegating the authentication step to Keycloak, Edge Service is not affected by the specific authentication strategy. As an example, we used the login form feature offered by Keycloak. Still, we could also enable social login via GitHub or rely on an existing Active Directory to authenticate users. Edge Service only needs to support OIDC to verify the authentication happened correctly and get information about the user via an ID Token.

There are still a few issues we haven't addressed. Polar Bookshop is a distributed system. After a user authenticates successfully with Keycloak, Edge Service is supposed to interact with Catalog Service and Order Service on behalf of the user. How can we securely propagate the authentication context to the other system applications? This chapter will help you solve that problem using OAuth2 and Access Tokens.

After dealing with authentication, we should address the authorization step. Right now, both customers and employees of Polar Bookshop can perform any action on the system. This chapter will walk you through a few authorization scenarios handled with OAuth2, Spring Security, and

Spring Data:

- using a Role-Based Access Control strategy (RBAC) to protect the REST endpoints exposed by Spring Boot depending on whether the user is a customer or an employee of the bookshop;
- configuring data auditing to keep track of which user made what changes;
- enforcing protection rules for data so that only its owner can access it.

Finally, you'll explore how to test those changes using Spring Boot, Spring Security, and Testcontainers.

NOTE

The source code for the examples in this chapter is available in the Chapter12/12-begin and Chapter12/12-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

12.1 Authorization and roles with Spring Cloud Gateway and OAuth2

In the previous chapter, we added user authentication features to Polar Bookshop. Edge Service is the access point to the system. Therefore, it's an excellent candidate for addressing cross-cutting concerns like security. For that reason, we made it responsible for authenticating the users. Edge Service initiates the authentication flow but delegates the actual authentication step to Keycloak using the OpenID Connect protocol.

Once a user authenticates successfully with Keycloak, Edge Service receives an ID Token from Keycloak with information about the authentication event and initiates an authenticated session with the user browser. At the same time, Keycloak also issues an **Access Token**, which is used to grant Edge Service access to downstream applications on behalf of the user as per OAuth2.

OAuth2 is an authorization framework that enables an application (called *Client*) to obtain limited access to a protected resource provided by another application (called *Resource Server*) on behalf of a user. When a user authenticates with Edge Service and asks to access their book orders, OAuth2 provides a solution for Edge Service to retrieve orders from Order Service on behalf of that user. Such a solution relies on a trusted party (called an *Authorization Server*) which issues an *Access Token* to Edge Service and grants access to the user's book orders from Order Service.

You might recognize some of the roles from the OIDC authentication flow we adopted in the previous chapter. As anticipated, OIDC is an identity layer built on top of OAuth2 and relies on the same basic concepts.

- **Authorization Server.** The entity responsible for authenticating users and issuing, refreshing, and revoking Access Tokens. In Polar Bookshop, this is Keycloak.

- **User.** Also called *Resource Owner*. It's the human logging in with the Authorization Server to get authenticated access to the Client application. It's also the human or service granting a Client access to the protected resources provided by a Resource Server. In Polar Bookshop, it's either a customer or an employee.
- **Client.** The application requiring the user to be authenticated and asking the user for authorization to access protected resources on their behalf. It can be a mobile application, a browser-based application, a server-side application, or even a smart TV application. In Polar Bookshop, it's Edge Service.
- **Resource Server.** It's the application hosting the protected resources a Client wants to access on the user's behalf. In Polar Bookshop, Catalog Service and Order Service are Resource Servers. Dispatcher Service is decoupled from the other applications and won't be accessed on behalf of the user. As a result, it won't participate in the OAuth2 setup.

Figure 12.1 shows how the four actors are mapped into the Polar Bookshop architecture.

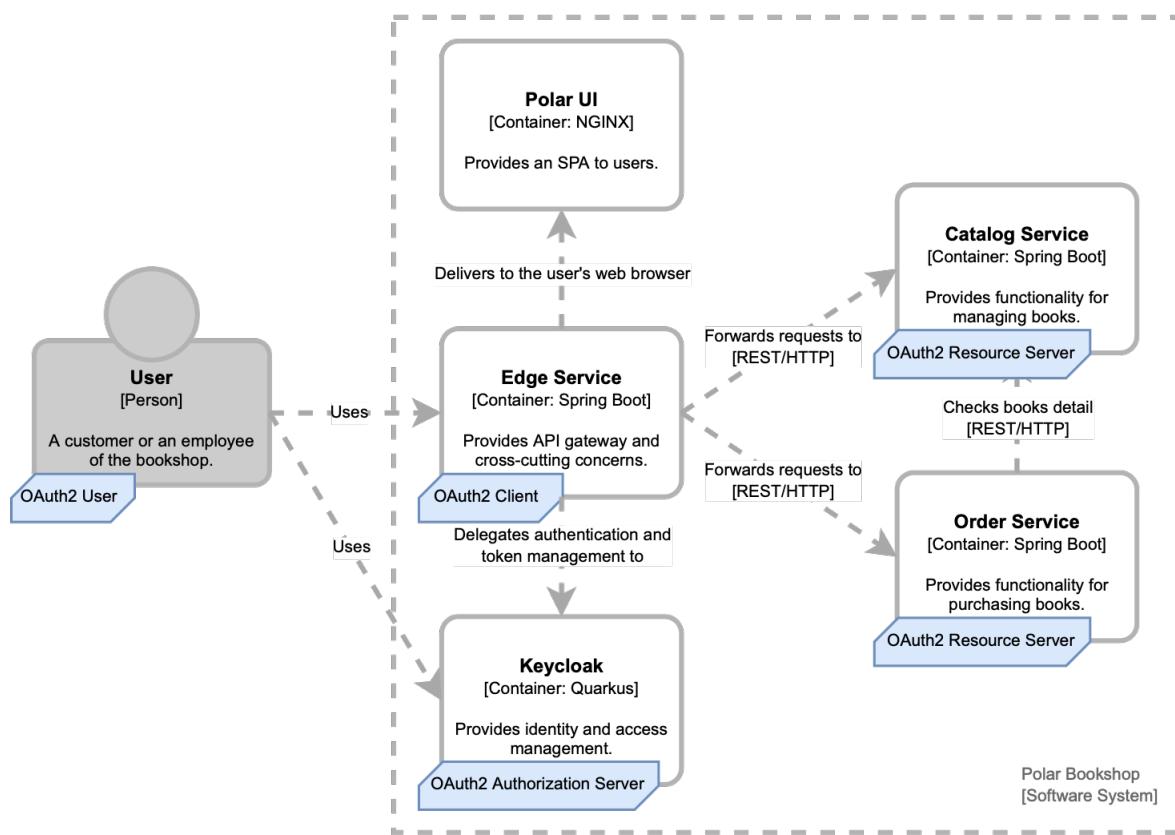


Figure 12.1 How the OIDC/OAuth2 roles are assigned to the entities in the Polar Bookshop architecture.

Edge Service can access downstream applications on behalf of the user through an Access Token issued by Keycloak during the OIDC authentication phase. In this section, you'll see how to configure Spring Cloud Gateway in Edge Service to use the Access Token whenever a request is routed to Catalog Service and Order Service.

In the previous chapter, we defined two users: Isabelle has both the *employee* and *customer* roles, while Bjorn has just the *customer* role. In this section, you'll also learn how to include that

information in both ID Tokens and Access Tokens for Spring Security to read it and set up a role-based access control (RBAC) mechanism.

NOTE

In Polar Bookshop, the OAuth2 Client (Edge Service) and the OAuth2 Resource Servers (Catalog Service and Order Service) belong to the same system, but the same framework can be used when the OAuth2 Client is a third-party application. In fact, that was the original use case of OAuth2 and why it became so popular. Using OAuth2, services like GitHub or Twitter let you give third-party applications limited access to your account. For example, you could authorize a scheduling application to publish tweets on your behalf without exposing your Twitter credentials.

12.1.1 Token Relay from Spring Cloud Gateway to other services

After a user successfully authenticates with Keycloak, Edge Service (the OAuth2 Client) receives an ID Token and an Access Token.

- **ID Token.** It represents a successful authentication event and includes information about the authenticated user.
- **Access Token.** It represents the authorization given to the OAuth2 Client to access protected data provided by an OAuth2 Resource Server on the user's behalf.

In Edge Service, Spring Security uses the ID Token to extract information about the authenticated user, set up a context for the current user session, and make the data available through the `oidcUser` object. That's what you saw in the previous chapter.

The Access Token grants Edge Service authorized access to Catalog Service and Order Service (the OAuth2 Resource Servers) on behalf of the user. After securing both applications, Edge Service will have to include the Access Token in all requests routed to them as an Authorization HTTP header. Unlike ID Tokens, Edge Service doesn't read the Access Token content because it's not the intended audience. It stores the Access Token received from Keycloak and then includes it as-is in any request to a protected endpoint downstream.

This pattern is called **Token Relay** and is supported by Spring Cloud Gateway as a built-in filter, so you don't need to implement anything yourself. When the filter is enabled, the Access Token is included automatically in all requests sent to one of the downstream applications. Figure 12.2 illustrates how the Token Relay pattern works.

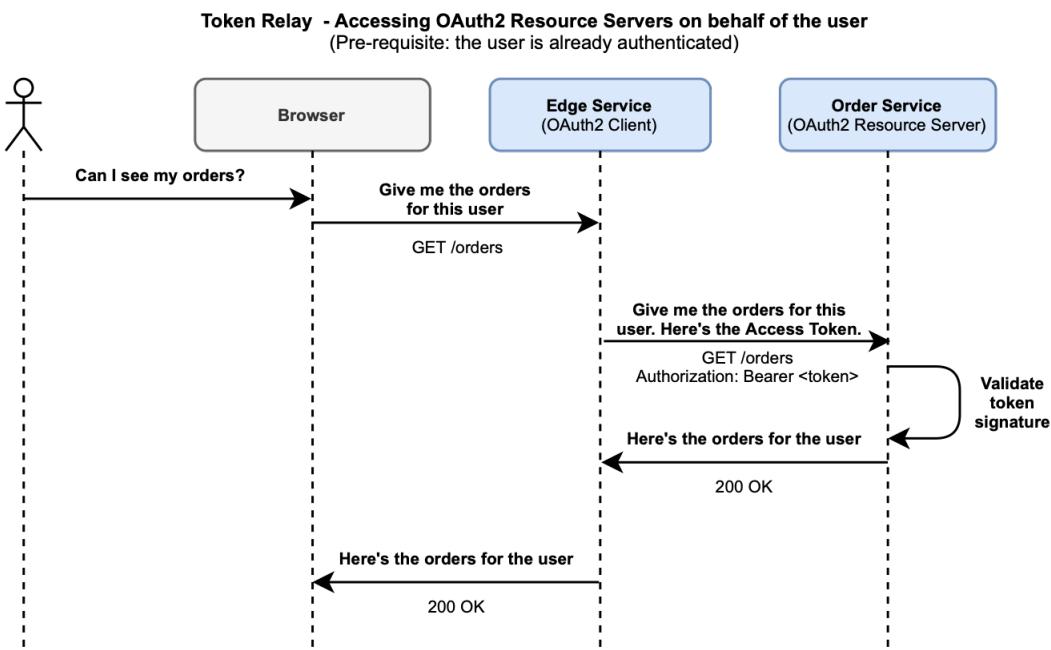


Figure 12.2 After a user is authenticated, Edge Service relays the Access Token to Order Service to call its protected endpoints on behalf of the user.

Let's see how to configure the Access Token relay in Edge Service.

NOTE

An Access Token has a validity period configured in Keycloak and should be as short as possible to reduce the exploitation time window if the token gets leaked. An acceptable length is 5 minutes. When the token expires, the OAuth2 Client can ask the Authorization Server for a new one using a third type of token called Refresh Token (which also has a validity period). The refresh mechanism is handled by Spring Security transparently, and I'll not describe it further.

ADOPTING THE TOKEN RELAY PATTERN IN SPRING CLOUD GATEWAY

Spring Cloud Gateway implements the Token Relay pattern as a filter. In the Edge Service project (`edge-service`), open the `application.yml` file and add `TokenRelay` as a default filter since we want it applied to all routes.

Listing 12.1 Enable the token relay pattern in Spring Cloud Gateway (`application.yml`)

```

spring:
  cloud:
    gateway:
      default-filters:
        - SaveSession
        - TokenRelay ①
  
```

- ① Enables the propagation of the Access Token when calling a downstream service.

With the filter enabled, Spring Cloud Gateway takes care of propagating the right Access Token as an `Authorization` header in all outgoing requests to Catalog Service and Order Service. For example:

```
GET /orders
Authorization: Bearer <access_token>
```

NOTE Unlike ID Tokens which are JWTs, the OAuth2 framework doesn't enforce a data format for Access Tokens. They can be of any String-based form. The most popular format is JWT, though, so that's how we'll parse Access Tokens on the consumer-side (Catalog Service and Order Service).

By default, Spring Security stores the Access Tokens for the currently authenticated users in memory. When you have multiple instances of Edge Service running (which is always true in a cloud production environment to ensure high availability), you will encounter issues due to the statefulness of the application. Cloud native applications should be stateless. Let's fix that.

STORING ACCESS TOKENS IN REDIS

Spring Security stores Access Tokens in an `OAuth2AuthorizedClient` object that is accessible through a `ServerOAuth2AuthorizedClientRepository`. The default implementation for that repository adopts an in-memory strategy for persistence. That's what makes Edge Service a stateful application. How can we keep it stateless and scalable?

A simple way to do that is to store `OAuth2AuthorizedClient` objects in the web session rather than in memory so that Spring Session will pick them up automatically and save them in Redis just like it does with ID Tokens. Fortunately, the framework already provides an implementation of the `ServerOAuth2AuthorizedClientRepository` interface to save data in the web session: `WebSessionServerOAuth2AuthorizedClientRepository`. Figure 12.3 illustrates how all the mentioned objects are related to each other.

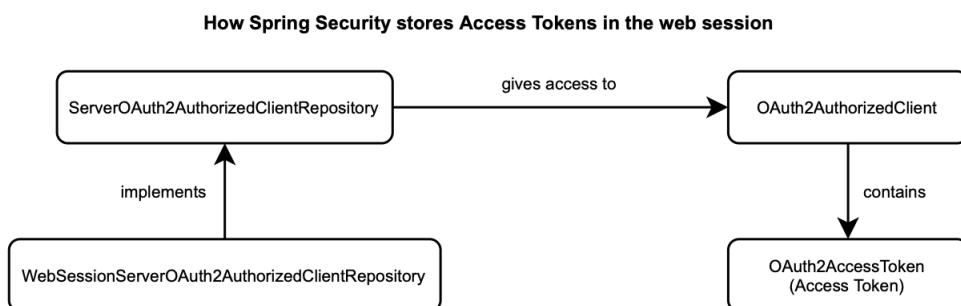


Figure 12.3 The main classes involved in storing the Access Token for the currently authenticated user in Spring Security.

In the Edge Service project, open the `SecurityConfig` class and define a bean of type

ServerOAuth2AuthorizedClientRepository using the implementation that stores Access Tokens in the web session.

Listing 12.2 Configure OAuth2AuthorizedClient objects to be saved in the web session (SecurityConfig.java)

```
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean ①
    ServerOAuth2AuthorizedClientRepository authorizedClientRepository() {
        return new WebSessionServerOAuth2AuthorizedClientRepository();
    }

    ...
}
```

- ① Defines a repository to store Access Tokens in the web session.

WARNING Access Tokens defined as JWTs should be handled with care. They are bearer tokens, meaning that any application can use them in an HTTP request and get access to an OAuth2 Resource Server. Handling the OIDC/OAuth2 flow in the backend rather than in an SPA provides better security because we don't expose any token to the browser. However, there might be other risks to manage, so carefully consider the trust boundaries of your system.

In the next section, you'll see how to enhance ID Token and Access Token to propagate information about user roles.

12.1.2 Customizing tokens and propagating user roles

Both ID Token and Access Token can include different information about the user, which is formatted as **claims** in the JWT. Claims are simple key/value pairs in JSON format. For example, OpenID Connect defines several standard claims to carry information about the user, like `given_name`, `family_name`, `preferred_username`, or `email`.

Access to such claims is controlled through **scopes**, a mechanism provided by OAuth2 to limit what data an OAuth2 Client can access. You can think of scopes as roles assigned to applications rather than users. In the previous chapter, we used Spring Security to make Edge Service an OAuth2 Client and configured it with the `openid` scope. That scope grants Edge Service access to the authenticated user's identity (provided in the `sub` claim).

Perhaps you tried logging in to a third-party website using GitHub or Google (social login based on OAuth2). If you did, you might have noticed that right after the authentication step, the service prompts you with a second request about what information from your GitHub or Google

account you consent the third party to access. That consent functionality is based on scopes, granting the third party (the OAuth2 Client) specific permissions depending on what scopes have been assigned.

Regarding Edge Service, we can decide upfront what scopes it should be granted. This section will show you how to configure a `roles` claim with the list of roles assigned to the authenticated user. Then, you'll use a `roles` scope for granting Edge Service access to that claim and instruct Keycloak to include it in both ID Tokens and Access Tokens.

Before moving on, we need a Keycloak container up and running. Open a Terminal window, navigate to the folder where you keep your Docker Compose file, and run the following command. If you haven't followed along, you can refer to `Chapter12/12-begin/polar-deployment/docker/docker-compose.yml` in the accompanying repository.

```
$ docker-compose up -d polar-keycloak
```

NOTE

Later, similar to what I did in the previous chapter, I will provide you with a JSON file you can use to load the entire configuration when starting up the Keycloak container without being concerned with its persistence. Should you follow this second option directly, I would still invite you to read through this section since it provides the essential information you'll need when moving to the Spring Security part.

CONFIGURING ACCESS TO USER ROLES IN KEYCLOAK

Keycloak comes pre-configured with a `roles` scope you can use to give an application access to the user roles contained in a `roles` claim. However, the default representation of the roles list is not very handy to use because it's defined as a nested object. Let's change that.

Once Keycloak is up and running, open a browser window, head to <http://localhost:8080>, log in to the administration console with the same credentials defined in the Docker Compose file (`user/password`), and choose the "PolarBookshop" realm. Then, select *Client Scopes* from the left menu. On the new page (figure 12.4), you can find a list of all the pre-configured scopes in Keycloak and have the possibility to create new ones. In our case, we want to customize the existing `roles` scope, so click on it to open its settings.

Name	Protocol	GUI order	Actions	
acr	openid-connect		Edit	Delete
address	openid-connect		Edit	Delete
email	openid-connect		Edit	Delete
micropoint-jwt	openid-connect		Edit	Delete
offline_access	openid-connect		Edit	Delete
phone	openid-connect		Edit	Delete
profile	openid-connect		Edit	Delete
role_list	saml		Edit	Delete
roles	openid-connect		Edit	Delete
web-origins	openid-connect		Edit	Delete

Figure 12.4 Client Scopes - Create and manage scopes.

In the `roles` scope page, open the *Mappers* tab. That is where you can define the set of claims to which the given scope provides access (i.e., *mappings*). By default, Keycloak already has some mappers defined to map claims to the `roles` scope. We’re interested in the *realm roles* mapper, which maps the user realm roles (including `employee` and `customer`) to a JWT claim. Select that mapper.

The settings page for the *realm roles* mapper provides a few options for customization. We want to change two things.

- The token claim name should be `roles` instead of `realm_access.roles` (so we remove the nested object).
- The `roles` claim should be included in both ID Token and Access Token, so we must ensure both options are enabled. We need both because Edge Service reads the claims from the ID Token while Catalog Service and Order Service read the claims from the Access Token. Edge Service is not the intended audience for the Access Token, which is forwarded to downstream applications as-is.

Figure 12.5 shows the final settings. When you’re done, click *Save*.

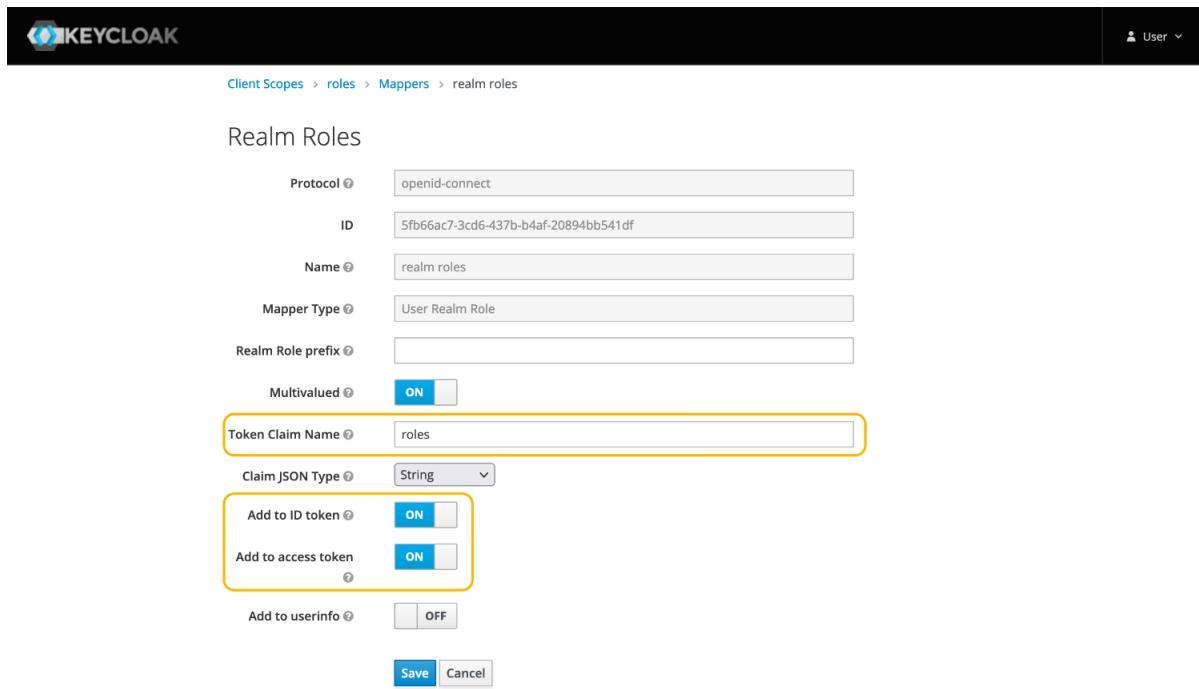


Figure 12.5 User Realm Roles - Configure the mapper to include user realm roles in a "roles" JWT claim.

NOTE

In the source code repository accompanying the book, I included a JSON file you can use to load the entire configuration when starting up the Keycloak container in the future, including the latest changes regarding roles (Chapter12/12-end/polar-deployment/docker/keycloak/full-realm-config.json). I recommend updating your polar-keycloak container definition in Docker Compose to use this new JSON file.

Before moving to the next section, stop any running container (`docker-compose down`).

CONFIGURING ACCESS TO USER ROLES IN SPRING SECURITY

Keycloak is now configured to return the authenticated user roles in a `roles` claim included in both ID Token and Access Token. However, the `roles` claim will only be returned if the OAuth2 Client (Edge Service) asks for the `roles` scope.

In the Edge Service project, open the `application.yml` file and update the Client Registration configuration to include the `roles` scope.

Listing 12.3 Assign the "roles" scope to Edge Service (application.yml)

```
spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: edge-service
            client-secret: polar-keycloak-secret
            scope: openid,roles ①
      provider:
        keycloak:
          issuer-uri: http://localhost:8080/realm/PolarBookshop
```

- ① Adds "roles" to the list of scopes so that Edge Service can get access to the user roles.

Next, you'll see how to extract the roles for the currently authenticated user from the ID Token.

EXTRACTING USER ROLES FROM THE ID TOKEN

In the previous chapter, we hard-coded the list of user roles in the `UserController` class of the Edge Service project because we didn't have them available in the ID Token yet. Now that we do, let's refactor the implementation to fetch the roles for the currently authenticated user from the `OidcUser` class, which gives us access to the claims in the ID Token, including the brand-new `roles` claim.

Listing 12.4 Extract the user roles list from the ID Token, accessible through OidcUser (UserController.java)

```
@RestController
public class UserController {

  @GetMapping("user")
  public Mono<User> getUser(@AuthenticationPrincipal OidcUser oidcUser) {
    var user = new User(
      oidcUser.getPreferredUsername(),
      oidcUser.getGivenName(),
      oidcUser.getFamilyName(),
      oidcUser.getClaimAsStringList("roles") ①
    );
    return Mono.just(user);
  }
}
```

- ① Gets the "roles" claim and extracts it as a list of strings.

Finally, remember to update the test setup in `UserControllerTests` so that the mock ID Token contains a `roles` claim.

Listing 12.5 Add roles list to mock ID Token (UserControllerTests.java)

```
@WebFluxTest(UserController.class)
@Import(SecurityConfig.class)
class UserControllerTests {

    ...

    private SecurityMockServerConfigurers.OidcLoginMutator
        configureMockOidcLogin(User expectedUser)
    {
        return mockOidcLogin().idToken(builder -> {
            builder.claim(StandardClaimNames.PREFERRED_USERNAME,
                expectedUser.username());
            builder.claim(StandardClaimNames.GIVEN_NAME,
                expectedUser.firstName());
            builder.claim(StandardClaimNames.FAMILY_NAME,
                expectedUser.lastName());
            builder.claim("roles", expectedUser.roles()); ①
        });
    }
}
```

- ① Adds a "roles" claim to the mocked ID Token.

You can verify the changes are correct by running the following command:

```
$ ./gradlew test --tests UserControllerTests
```

NOTE The roles claim configured in Keycloak will include our custom roles (`employee` and `customer`) together with a few extra roles managed and assigned by Keycloak itself.

So far, we configured Keycloak to include the user roles in the tokens and updated Edge Service to relay the Access Token to the applications downstream. We're now ready to start securing Catalog Service and Order Service with Spring Security and OAuth2.

12.2 Protecting APIs with Spring Security and OAuth2 (imperative)

When a user accesses the Polar Bookshop application, Edge Service initiates the OpenID Connect authentication flow through Keycloak and ultimately receives an Access Token granting it access to downstream services on behalf of that user.

In this section and the next one, you'll see how to secure Catalog Service and Order Service by requiring a valid Access Token to access their protected endpoints. In the OAuth2 authorization framework, they play the role of OAuth2 Resource Servers: the applications hosting protected data that a user can access through a third party (Edge Service, in our example).

OAuth2 Resource Servers don't deal with user authentication. They receive an Access Token in the `Authorization` header of each HTTP request. Then, they verify the signature and authorize

the request according to the content of the token. We have already configured Edge Service to send over an Access Token when routing a request downstream. Now, you'll see how to use that token on the receiver side. This section will guide you through securing Catalog Service, built on the imperative Spring stack. The next one will cover how to achieve the same result in Order Service, built on the reactive Spring stack.

12.2.1 Securing Spring Boot as an OAuth2 Resource Server

The first step for securing a Spring Boot application leveraging OAuth2 is adding a dependency on the dedicated Spring Boot starter that includes Spring Security and OAuth2 support for Resource Servers.

In the Catalog Service project (`catalog-service`), open the `build.gradle` file and add the new dependency. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 12.6 Add dependency for Spring Security OAuth2 Resource Server in Catalog Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
}
```

Next, let's configure the integration between Spring Security and Keycloak.

CONFIGURING THE INTEGRATION BETWEEN SPRING SECURITY AND KEYCLOAK

Spring Security supports protecting endpoints using two data formats for the Access Token: JWT and Opaque Token. We'll work with Access Tokens defined as JWTs, similar to what we did for ID Tokens. With Access Tokens, Keycloak grants Edge Service access to downstream applications on behalf of the user. When the Access Token is a JWT, we can also include relevant information as claims about the authenticated user and propagate such context to Catalog Service and Order Service with ease. On the other hand, Opaque Tokens would require the application downstream to contact Keycloak every time to fetch the information associated with the token.

Configuring Spring Security to integrate with Keycloak as an OAuth2 Resource Server is more straightforward than the OAuth2 Client scenario. When working with JWTs, the application will contact Keycloak mainly to fetch the public keys necessary to verify the token signature. Using the `issuer-uri` property, similar to what we did for Edge Service, we let the application auto-discover the Keycloak endpoint where to find the public keys.

The default behavior is for the application to fetch the public keys lazily upon the first received HTTP request rather than at startup, both for performance and coupling reasons (you don't need

Keycloak up and running when starting the application). OAuth2 Authorization Servers provide their public keys using the JSON Web Key (JWK) format. The collection of public keys is called a **JWK Set**. The endpoint where Keycloak exposes its public keys is called *JKW Set URI*. Spring Security will automatically rotate the public keys whenever Keycloak makes new ones available.

For each incoming request containing an Access Token in the `Authorization` header, Spring Security will automatically validate the token signature using the public keys provided by Keycloak and decode its claims via a `JwtDecoder` object which is autoconfigured behind the scenes.

In the Catalog Service project (`catalog-service`), open the `application.yml` file and add the following configuration.

Listing 12.7 Configure Catalog Service as an OAuth2 Resource Server (application.yml)

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt: ①
          issuer-uri:
            ↳ http://localhost:8080/realm/PolarBookshop ②
```

- ① OAuth2 doesn't enforce a data format for Access Tokens, so we must be explicit about our choice. In this case, we want to use JWT.
- ② The Keycloak URL providing information about all the relevant OAuth2 endpoints for the specific realm.

NOTE

Explaining the cryptographic algorithm used to sign the Access Token is out of scope for this book. If you'd like to learn more about cryptography, Manning has a book in its catalog on the subject: "Real-World Cryptography" by David Wong (<https://www.manning.com/books/real-world-cryptography>).

The integration between Catalog Service and Keycloak is now established. Next, you'll define some basic security policies to protect the application endpoints.

DEFINING SECURITY POLICIES FOR JWT AUTHENTICATION

For the Catalog Service application, we want to enforce the following security policies:

- GET requests to fetch books should be allowed without authentication;
- All other requests should require authentication;
- The application should be configured as an OAuth2 Resource Server and use JWT authentication;
- The flow for handling JWT authentication should be stateless.

Let's expand on the last policy. Edge Service triggers the user authentication flow and leverages

the web session to store data like ID Tokens and Access Tokens that would otherwise get lost at the end of each HTTP request, forcing a user to authenticate at each request. To make it possible for the application to scale, we used Spring Session to store the web session data in Redis and keep the application stateless.

Unlike Edge Service, Catalog Service needs only an Access Token to authenticate a request. Since the token is always provided in each HTTP request to a protected endpoint, Catalog Service doesn't need to store any data between requests. We call this strategy *Stateless Authentication* or *Token-Based Authentication*. We use JWTs as Access Tokens, so we can also refer to it as **JWT Authentication**.

Now on to the code. In the Catalog Service project, create a new `SecurityConfig` class in the `com.polarbookshop.catalogservice.config` package. Similar to what we did for Edge Service, we can use the DSL provided by `HttpSecurity` to build a `SecurityFilterChain` configured with the required security policies.

Listing 12.8 Configure security policies and JWT authentication for Catalog Service (`SecurityConfig.java`)

```
@EnableWebSecurity ①
public class SecurityConfig {

    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(authorize -> authorize
                .mvcMatchers(HttpMethod.GET, "/", "/books/**")
                .permitAll() ②
                .anyRequest().authenticated() ③
            )
            .oauth2ResourceServer(
                OAuth2ResourceServerConfigurer::jwt ④
            )
            .sessionManagement(sessionManagement -> ⑤
                sessionManagement
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(AbstractHttpConfigurer::disable) ⑥
            .build();
    }
}
```

- ① Enables Spring MVC support for Spring Security.
- ② Allows users to fetch greetings and books without being authenticated.
- ③ Any other request requires authentication.
- ④ Enables OAuth2 Resource Server support using the default configuration based on JWT (i.e., JWT authentication).
- ⑤ Each request must include an Access Token, so there's no need to keep a user session alive between requests. We want it to be stateless.
- ⑥ Since the authentication strategy is stateless and doesn't involve a browser-based client, we can safely disable the CSRF protection.

Let's check if it works. First, the Polar UI, Keycloak, Redis, and PostgreSQL containers should be started. Open a Terminal window, navigate to the folder where you keep your Docker Compose configuration (`polar-deployment/docker`) and run the following.

```
$ docker-compose up -d polar-ui polar-keycloak polar-redis polar-postgres
```

Then, run both Edge Service and Catalog Service (`./gradlew bootRun` from each project). Finally, open a browser window, and go to <http://localhost:9000>.

Ensure you can see the list of books in the catalog without being authenticated, but not add/update/delete them. Then, log in as Isabelle (`isabelle/password`). She is an employee of the bookshop, so she should be allowed to modify the books in the catalog. Next, log in as Bjorn (`bjorn/password`). He is a customer, so he shouldn't be able to change anything in the catalog.

Under the hood, the Angular application fetches the user roles from the `/user` endpoint exposed by Edge Service and uses them to block pieces of functionality. That improves the user experience, but it's not secure. The actual endpoints exposed by Catalog Service don't take roles into account. We need to enforce a role-based authorization. That's the topic of the next section.

12.2.2 Role-based access control with Spring Security and JWT

So far, when talking about authorization, we referred to granting an OAuth2 Client (Edge Service) access to an OAuth2 Resource Server (like Catalog Service) on behalf of the user. Now, we'll move from application authorization to user authorization. What can an authenticated user do in the system?

Spring Security associates each authenticated user with a list of `GrantedAuthority` objects modeling the authorities the user has been granted. Granted authorities can be used to represent fine-grained permissions, roles, or even scopes and come from different sources depending on the authentication strategy. The authorities are available through the `Authentication` object representing the authenticated user and stored in the `SecurityContext`.

Since Catalog Service is configured as an OAuth2 Resource Server and uses JWT Authentication, Spring Security extracts the list of scopes from the `scopes` claim of the Access Token and uses them as granted authorities for the given user automatically. Each `GrantedAuthority` object built in this way will be named with the `SCOPE_` prefix and the scope value.

The default behavior is acceptable in many scenarios where scopes are used to model permissions, but it doesn't fit our case where we rely on user roles to know which privileges each user has. We want to set up a role-based access control strategy (RBAC) using the user roles provided in the `roles` claim of the Access Token (figure 12.6). In this section, I'll show you how to define a custom converter for the Access Token to build a list of `GrantedAuthority` objects

using the values in the `roles` claim and the `ROLE_` prefix. Then, we'll use those authorities to define authorization rules for the endpoints of Catalog Service.

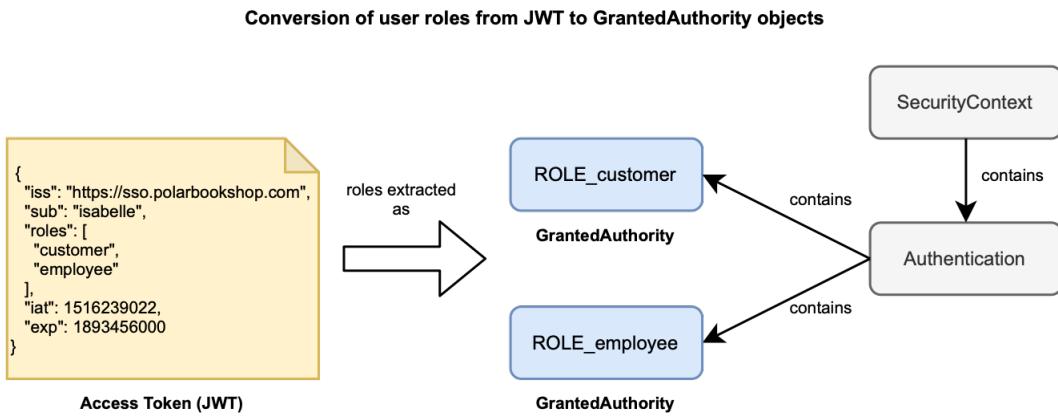


Figure 12.6 How the user roles listed in the Access Token (JWT) are converted into GrantedAuthority objects used by Spring Security for RBAC.

NOTE

You might be wondering why using the `SCOPE_` or `ROLE_` prefix. Since granted authorities can be used to represent different items (roles, scopes, permissions), Spring Security uses prefixes to group them. We rely on this default naming convention for the Polar Bookshop example, but it's possible to use different prefixes or even no prefixes at all. For more information, you can refer to the Spring Security documentation (<https://spring.io/projects/spring-security>).

EXTRACTING USER ROLES FROM THE ACCESS TOKEN

Spring Security provides a `JwtAuthenticationConverter` class we can use to define a custom strategy to extract information from a JWT. In our case, the JWT is an Access Token, and we want to configure how to build `GrantedAuthority` objects from the values in the `roles` claim. In the Catalog Service project (`catalog-service`), open the `SecurityConfig` class and define a new `JwtAuthenticationConverter` bean.

Listing 12.9 Configure converter to map roles from JWT to granted authorities (SecurityConfig.java)

```

@EnableWebSecurity
public class SecurityConfig {
    ...

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        var jwtGrantedAuthoritiesConverter =
            new JwtGrantedAuthoritiesConverter(); ①
        jwtGrantedAuthoritiesConverter
            .setAuthorityPrefix("ROLE_"); ②
        jwtGrantedAuthoritiesConverter
            .setAuthoritiesClaimName("roles"); ③

        var jwtAuthenticationConverter =
            new JwtAuthenticationConverter(); ④
        jwtAuthenticationConverter
            .setJwtGrantedAuthoritiesConverter(jwtGrantedAuthoritiesConverter);
        return jwtAuthenticationConverter;
    }
}

```

- ① Defines a converter to map claims to GrantedAuthority objects.
- ② Applies the "ROLE_" prefix to each user role.
- ③ Extracts the list of roles from the "roles" claim.
- ④ Defines a strategy to convert a JWT. We only customize how to build granted authorities out of it.

With this bean in place, Spring Security will associate a list of `GrantedAuthority` objects to each authenticated user, and we can use them to define authorization policies.

DEFINING AUTHORIZATION POLICIES BASED ON USER ROLES

The Catalog Service endpoints should be protected according to the following policies:

- All GET requests sent to the `/`, `/books` or `/books/{isbn}` endpoints should be allowed even without authentication.
- Any other request should require both user authentication and the `employee` role.

Spring Security provides an expression-based DSL to define authorization policies. The most generic one is `hasAuthority("ROLE_employee")` which you can use to check for any type of authority. In our case, authorities are roles, so we can use the most descriptive `hasRole("employee")` and drop the prefix (which is added by Spring Security under the hood).

Listing 12.10 Apply RBAC to limit write access to users with the "employee" role (SecurityConfig.java)

```

@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(authorize -> authorize
                .mvcMatchers(HttpMethod.GET, "/", "/books/**")
                .permitAll() ①
                .anyRequest().hasRole("employee") ②
            )
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt)
            .sessionManagement(sessionManagement -> sessionManagement
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(AbstractHttpConfigurer::disable)
            .build();
    }
    ...
}

```

- ① Allows users to fetch greetings and books without being authenticated.
- ② Any other request requires not only authentication but also the "employee" role (which is the same as the "ROLE_employee" authority).

Now, you can re-build and run Catalog Service (`./gradlew bootRun`) and go through the same flows as before. This time, Catalog Service will ensure that only the bookshop employees are allowed to add/update/delete books.

Finally, stop the running applications (`Ctrl+C`) and containers (`docker-compose down`).

NOTE

To learn more about the authorization architecture in Spring Security and the different strategies available for access control, you can refer to the chapters 7 and 8 from "Spring Security in Action" by Laureniu Spilc (<https://livebook.manning.com/book/spring-security-in-action/chapter-7>) where they are explained in great detail.

Next, I'll guide you through some techniques for testing security in an imperative Spring Boot application configured as an OAuth2 Resource Server.

12.2.3 Testing OAuth2 with Spring Security and Testcontainers

When it comes to security, writing autotests is usually challenging. Fortunately, Spring Security provides us with convenient utilities to verify the security setup in sliced tests.

This section will show you how to write slice tests for the web slice using mock Access Tokens and full integration tests relying on an actual Keycloak container run via Testcontainers.

Before starting, we need to add new dependencies on Spring Security Test and Testcontainers Keycloak. Open the `build.gradle` file for the Catalog Service project (`catalog-service`) and update it as follows. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 12.11 Add dependencies for Spring Security Test and Testcontainers Keycloak in Catalog Service (`build.gradle`)

```
ext {
    ...
    set('testKeycloakVersion', "2.1.2")      ①
}

dependencies {
    ...
    testImplementation 'org.springframework.security:spring-security-test'
    testImplementation 'org.testcontainers:junit-jupiter'
    testImplementation "com.github.dansniko:
    testcontainers-keycloak:${testKeycloakVersion}" ②
}
```

- ① The version of Testcontainers Keycloak.
- ② Provides Keycloak testing utilities on top of Testcontainers.

TESTING SECURED REST CONTROLLERS WITH @WEBMVCTEST AND SPRING SECURITY

First, let's update the `BookControllerMvcTests` class to cover the new scenarios depending on the user authentication and authorization. For example, we could write test cases for DELETE operations in these situations:

- the user is authenticated and has the `employee` role;
- the user is authenticated but doesn't have the `employee` role;
- the user is not authenticated.

Delete operations are only allowed to the bookshop employees, so only the first request will return a successful answer.

As part of the OAuth2 Access Token validation, Spring Security relies on the public keys provided by Keycloak to verify the JWT signature. Internally, the framework configures a `JwtDecoder` bean to decode and verify a JWT using those keys. In the context of a web slice test, we can provide a mock `JwtDecoder` bean so that Spring Security skips the interaction with Keycloak (which we'll validate later in a full integration test).

Listing 12.12 Verify security policies at the web layer with slice tests (BookControllerMvcTests.java)

```

@WebMvcTest(BookController.class)
@Import(SecurityConfig.class) ①
class BookControllerMvcTests {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    JwtDecoder jwtDecoder; ②

    ...

    @Test
    void whenDeleteBookWithEmployeeRoleThenShouldReturn204()
        throws Exception
    {
        var isbn = "7373731394";
        mockMvc ③
            .perform(MockMvcRequestBuilders.delete("/books/" + isbn)
                .with(SecurityMockMvcRequestPostProcessors.jwt()
                    .authorities(new SimpleGrantedAuthority("ROLE_employee"))))
                .andExpect(MockMvcResultMatchers.status().isNoContent());
    }

    @Test
    void whenDeleteBookWithCustomerRoleThenShouldReturn403()
        throws Exception
    {
        var isbn = "7373731394";
        mockMvc ④
            .perform(MockMvcRequestBuilders.delete("/books/" + isbn)
                .with(SecurityMockMvcRequestPostProcessors.jwt()
                    .authorities(new SimpleGrantedAuthority("ROLE_customer"))))
                .andExpect(MockMvcResultMatchers.status().isForbidden());
    }

    @Test
    void whenDeleteBookNotAuthenticatedThenShouldReturn401()
        throws Exception
    {
        var isbn = "7373731394";
        mockMvc
            .perform(MockMvcRequestBuilders.delete("/books/" + isbn))
            .andExpect(MockMvcResultMatchers.status().isUnauthorized());
    }
}

```

- ① Imports the application security configuration.
- ② Mocks the "JwtDecoder" so that the application doesn't try to call Keycloak and get the public keys for decoding the Access Token.
- ③ Mutates the HTTP request with a mock, JWT-formatted Access Token for a user with the "employee" role.
- ④ Mutates the HTTP request with a mock, JWT-formatted Access Token for a user with the "customer" role.

Open a Terminal window, navigate to the Catalog Service root folder, and run the newly added

tests as follows.

```
$ ./gradlew test --tests BookControllerMvcTests
```

Feel free to add more web slice autotests also to cover GET, POST, and PUT requests. For inspiration, you can refer to the source code accompanying the book (Chapter12/12-end/catalog-service).

INTEGRATION TESTS WITH @SPRINGBOOTTEST, SPRING SECURITY, AND TESTCONTAINERS

Currently, the integration tests we wrote in the previous chapters will not work anymore for two reasons. First, all POST, PUT, and DELETE requests will fail because we are not providing any valid OAuth2 Access Token. Even if we were, there is no Keycloak up and running, required by Spring Security to fetch the public keys used to validate the Access Tokens.

You can verify the failure by running the following command from the Catalog Service root folder.

```
$ ./gradlew test --tests CatalogServiceApplicationTests
```

We have already seen how to use Testcontainers to write integration tests against data services like a PostgreSQL database, making our tests more reliable and ensuring environment parity. In this section, we'll do the same for Keycloak.

Let's start by configuring a Keycloak container via Testcontainers. Open the CatalogServiceApplicationTests class and add the following setup.

Listing 12.13 Setup for a Keycloak test container (CatalogServiceApplicationTests.java)

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("integration")
@Testcontainers ①
class CatalogServiceApplicationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Container ②
    private static final KeycloakContainer keycloakContainer =
        new KeycloakContainer("quay.io/keycloak/keycloak:18.0")
            .withRealmImportFile("test-realm-config.json");

    @DynamicPropertySource ③
    static void dynamicProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.security.oauth2.resourceserver.jwt.issuer-uri",
            () -> keycloakContainer.getAuthServerUrl() + "realms/PolarBookshop");
    }

    ...
}
```

- ① Activates automatic startup and cleanup of test containers.

- ② Defines a Keycloak container for testing.
- ③ Overwrites the Keycloak Issuer URI configuration to point to the test Keycloak instance.

The Keycloak test container is initialized via a configuration file I included in the code repository `a c c o m p a n y i n g t h i s b o o k (Chapter12/12-end/catalog-service/src/test/resources/test-realm-config.json)`. Go ahead and copy it over to the `src/test/resources` folder of your Catalog Service project (`catalog-service`).

In production, we would call Catalog Service via Edge Service, which is responsible for authenticating users and relaying Access Tokens to the downstream applications. We now want to test Catalog Service in isolation and verify different authorization scenarios. Therefore, we need to generate some Access Tokens first so that we can use them to call the Catalog Service endpoints under testing.

The Keycloak configuration I provided in the JSON file includes the definition of a test Client (`polar-test`) we can use to authenticate users via a username and password directly instead of going through the browser-based flow we implemented in Edge Service. In OAuth2, such a flow is called *Password Grant*, and it's not recommended for production usage. In the following, we'll use it just for testing purposes.

Let's set up `CatalogServiceApplicationTests` to authenticate with Keycloak as Isabelle and Bjorn so that we can obtain the Access Tokens we need to call the Catalog Service protected endpoints. Keep in mind that Isabelle is both a customer and employee, whereas Bjorn is only a customer.

Listing 12.14 Setup for obtaining test Access Tokens (CatalogServiceApplicationTests.java)

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("integration")
@Testcontainers
class CatalogServiceApplicationTests {
    private static KeycloakToken bjornTokens;
    private static KeycloakToken isabelleTokens;
    ...

    @BeforeAll
    static void generateAccessTokens() {
        WebClient webClient = WebClient.builder() ①
            .baseUrl(keycloakContainer.getAuthServerUrl()
                + "realms/PolarBookshop/protocol/openid-connect/token")
            .defaultHeader(HttpHeaders.CONTENT_TYPE,
                MediaType.APPLICATION_FORM_URLENCODED_VALUE)
            .build();

        isabelleTokens = authenticateWith( ②
            "isabelle", "password", webClient);
        bjornTokens = authenticateWith( ③
            "bjorn", "password", webClient);
    }

    private static KeycloakToken authenticateWith(
        String username, String password, WebClient webClient
    ) {
        return webClient
            .post()
            .body( ④
                BodyInserters.fromFormData("grant_type", "password")
                    .with("client_id", "polar-test")
                    .with("username", username)
                    .with("password", password)
            )
            .retrieve()
            .bodyToMono(KeycloakToken.class)
            .block(); ⑤
    }

    private record KeycloakToken(String accessToken) {
        @JsonCreator ⑥
        private KeycloakToken(
            @JsonProperty("access_token") final String accessToken
        ) {
            this.accessToken = accessToken;
        }
    }
}

```

- ① A WebClient used to call Keycloak.
- ② Authenticates as Isabelle and obtain an Access Token.
- ③ Authenticates as Bjorn and obtain an Access Token.
- ④ Uses the Password Grant flow to authenticate with Keycloak directly.
- ⑤ Blocks until a result is available. It's how we use WebClient imperatively rather than reactively.
- ⑥ Instructs Jackson to use this constructor when deserializing JSON into KeycloakToken objects.

Finally, we can update the test cases in `CatalogServiceApplicationTests` to cover several authentication/authorization scenarios. For example, we could write test cases for POST operations in these situations:

- the user is authenticated and has the `employee` role (extending the existing test case);
- the user is authenticated but doesn't have the `employee` role (new test case);
- the user is not authenticated (new test case).

NOTE

In the context of an OAuth2 Resource Server, authentication means token authentication. In this case, it happens by providing an Access Token in the Authorization header of each HTTP request.

Create operations are only allowed to the bookshop employees, so only the first request will return a successful answer.

Listing 12.15 Verify security scenarios in integration tests (CatalogServiceApplicationTests.java)

```

@Test
void whenPostRequestThenBookCreated() {
    var expectedBook = Book.of("1231231231", "Title", "Author",
        9.90, "Polarsophia");

    webTestClient.post().uri("/books")
        .headers(headers -> ①
            headers.setBearerAuth(isabelleTokens.accessToken()))
        .bodyValue(expectedBook)
        .exchange()
        .expectStatus().isCreated() ②
        .expectBody(Book.class).value(actualBook -> {
            assertThat(actualBook).isNotNull();
            assertThat(actualBook.isbn()).isEqualTo(expectedBook.isbn());
        });
}

@Test
void whenPostRequestUnauthorizedThen403() {
    var expectedBook = Book.of("1231231231", "Title", "Author",
        9.90, "Polarsophia");

    webTestClient.post().uri("/books")
        .headers(headers -> ③
            headers.setBearerAuth(bjornTokens.accessToken()))
        .bodyValue(expectedBook)
        .exchange()
        .expectStatus().isForbidden(); ④
}

@Test
void whenPostRequestUnauthenticatedThen401() {
    var expectedBook = Book.of("1231231231", "Title", "Author",
        9.90, "Polarsophia");

    webTestClient.post().uri("/books") ⑤
        .bodyValue(expectedBook)
        .exchange()
        .expectStatus().isUnauthorized(); ⑥
}

```

- ① Sends a request to add a book to the catalog as an authenticated employee user (Isabelle).
- ② The book has been successfully created (201).
- ③ Sends a request to add a book to the catalog as an authenticated customer user (Bjorn).
- ④ The book has not been created because the user doesn't have the correct authorization, no "employee" role (403).
- ⑤ Sends a request to add a book to the catalog as an unauthenticated user.
- ⑥ The book has not been created because the user is not authenticated (401).

Open a Terminal window, navigate to the Catalog Service root folder, and run the newly added tests as follows.

```
$ ./gradlew test --tests CatalogServiceApplicationTests
```

There are still tests failing, though. Go ahead and update them by including the right Access Token (Isabelle's or Bjorn's) in any POST, PUT, and DELETE request, as you learned in the previous example. When you're done, rerun the tests and verify they are all successful. For inspiration, you can refer to the source code accompanying this book (Chapter12/12-end/catalog-service).

12.3 Protecting APIs with Spring Security and OAuth2 (reactive)

Securing reactive Spring Boot applications like Order Service is similar to what we did for Catalog Service. Spring Security provides intuitive and consistent abstractions across the two stacks, which makes it easy to move from one stack to the other.

In this section, I'll guide you through configuring Order Service as an OAuth2 Resource Server, enabling JWT Authentication, and defining security policies for the web endpoints.

12.3.1 Securing Spring Boot as an OAuth2 Resource Server

The Spring Boot starter dependency to include Spring Security and OAuth2 support for Resource Servers is the same for imperative and reactive applications. In the Order Service project (order-service), open the `build.gradle` file and add the new dependency. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 12.16 Add dependency for Spring Security OAuth2 Resource Server in Order Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
}
```

Next, let's configure the integration between Spring Security and Keycloak.

CONFIGURING THE INTEGRATION BETWEEN SPRING SECURITY AND KEYCLOAK

The strategy for integrating Spring Security with Keycloak will be similar to what we did in Catalog Service. Open the Order Service project (`order-service`), and update the `application.yml` file with the following configuration.

Listing 12.17 Configure Order Service as an OAuth2 Resource Server (application.yml)

```
spring:
  security:
    oauth2:
      resourceServer:
        jwt:
          ①
          issuer-uri:
            ↪http://localhost:8080/realms/PolarBookshop ②
```

- ① OAuth2 doesn't enforce a data format for Access Tokens, so we must be explicit about our choice. In this case, we want to use JWT.
- ② The Keycloak URL providing information about all the relevant OAuth2 endpoints for the specific realm.

The integration between Order Service and Keycloak is now established. Next, we'll define the necessary security policies to protect the application endpoints.

DEFINING SECURITY POLICIES FOR JWT AUTHENTICATION

For the Order Service application, we want to enforce the following security policies:

- All requests should require authentication;
- The application should be configured as an OAuth2 Resource Server and use JWT Authentication;
- The flow for handling JWT Authentication should be stateless.

There are two main differences from what we did in Catalog Service:

- The reactive syntax is slightly different from its imperative counterpart, especially the part to enforce JWT Authentication (stateless).
- We're not extracting user roles from the Access Token because the endpoints don't have

special requirements depending on user roles.

In the Order Service project, create a `SecurityConfig` class in a new `com.polarbookshop.orderservice.config` package. Then, use the DSL provided by `ServerHttpSecurity` to build a `SecurityWebFilterChain` configured with the required security policies.

Listing 12.18 Configure security policies and JWT authentication for Order Service (`SecurityConfig.java`)

```
@EnableWebFluxSecurity ①
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain filterChain(ServerHttpSecurity http) {
        return http
            .authorizeExchange(exchange -> exchange
                .anyExchange().authenticated() ②
            )
            .oauth2ResourceServer( ③
                ServerHttpSecurity.OAuth2ResourceServerSpec::jwt)
            .requestCache(requestCacheSpec -> ④
                requestCacheSpec.requestCache(NoOpServerRequestCache.getInstance())
            )
            .csrf(ServerHttpSecurity.CsrfSpec::disable) ⑤
            .build();
    }
}
```

- ① Enables Spring WebFlux support for Spring Security.
- ② All requests require authentication.
- ③ Enables OAuth2 Resource Server support using the default configuration based on JWT (i.e., JWT authentication).
- ④ Each request must include an Access Token, so there's no need for keeping a session cache alive between requests. We want it to be stateless.
- ⑤ Since the authentication strategy is stateless and doesn't involve a browser-based client, we can safely disable the CSRF protection.

Let's check if it works. First, we need to run the backing services (Polar UI, Keycloak, Redis, RabbitMQ, and PostgreSQL). Open a Terminal window, navigate to the folder where you keep your Docker Compose configuration (`polar-deployment/docker`) and run the following.

```
$ docker-compose up -d polar-ui polar-keycloak polar-redis \
    polar-rabbitmq polar-postgres
```

Then, run Edge Service, Catalog Service, and Order Service on the JVM (`./gradlew bootRun` from each project). Finally, open a browser window, and go to <http://localhost:9000>.

Since Order Service doesn't have particular requirements depending on the user role, you can log in with either Isabelle (`isabelle/password`) or Bjorn (`bjorn/password`). Then, pick a book from the catalog and submit an order for it. Since you're authenticated, you're allowed to create

an order. When you're done, you can visit the *Orders* page to check all the submitted orders.

"Wait a second! What do you mean all the submitted orders?". I'm glad you asked. At the moment, each person can see the orders submitted by all users. Don't worry! Later in the chapter, we'll fix that.

Before going there, we need to discuss how to test the new Order Service security policies. Stop the running applications (`Ctrl+C`) and containers (`docker-compose down`), and move to the next section to see how to test security in a reactive application.

12.3.2 Testing OAuth2 with Spring Security and Testcontainers

Testing secured reactive Spring Boot applications is similar to what we do in imperative applications. Before starting, we need to add new dependencies on Spring Security Test and Testcontainers Keycloak. The dependency adding JUnit5 support for Testcontainers is already there. Open the `build.gradle` file and update it as follows. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 12.19 Add dependencies for Spring Security Test and Testcontainers Keycloak in Order Service (build.gradle)

```
ext {
    ...
    set('testKeycloakVersion', "2.1.2") ①
}

dependencies {
    ...
    testImplementation 'org.springframework.security:spring-security-test'
    testImplementation 'org.testcontainers:junit-jupiter'
    testImplementation "com.github.dasniko:
    testcontainers-keycloak:${testKeycloakVersion}" ②
}
```

- ① The version of Testcontainers Keycloak.
- ② Provides Keycloak testing utilities on top of Testcontainers.

We can implement full integration tests using `@SpringBootTest` and Testcontainers Keycloak. Since the setup is the same as Catalog Service, I won't cover those tests here, but you can find them in the repository accompanying the book (Chapter12/12-end/order-service/src/test). Make sure you update those integration tests, or the application build will fail.

In this section, I'll show you how to test the web slice of a reactive application when the endpoints are secured, following a similar approach we used for Catalog Service.

TESTING SECURED REST CONTROLLERS WITH @WEBFLUXTEST AND SPRING SECURITY

We have already written autotests in `OrderControllerWebFluxTests` for the web slice using `@WebFluxTest`. Let's see now how to update them to take security into account.

As part of the OAuth2 Access Token validation, Spring Security relies on the public keys provided by Keycloak to verify the JWT signature. Internally, the framework configures a `ReactiveJwtDecoder` bean to decode and verify a JWT using those keys. In the context of a web slice test, we can provide a mock `ReactiveJwtDecoder` bean so that Spring Security skips the interaction with Keycloak (validated by full integration tests).

Listing 12.20 Verify security policies at the web layer with slice tests (`OrderControllerWebFluxTests.java`)

```
@WebFluxTest(OrderController.class)
@Import(SecurityConfig.class) ①
class OrderControllerWebFluxTests {

    @Autowired
    WebTestClient webClient;

    @MockBean
    OrderService orderService;

    @MockBean
    ReactiveJwtDecoder reactiveJwtDecoder; ②

    @Test
    void whenBookNotAvailableThenRejectOrder() {
        var orderRequest = new OrderRequest("1234567890", 3);
        var expectedOrder = OrderService.buildRejectedOrder(
            orderRequest.isbn(), orderRequest.quantity());
        given(orderService.submitOrder(
            orderRequest.isbn(), orderRequest.quantity()))
            .willReturn(Mono.just(expectedOrder));

        webClient
            .mutateWith(SecurityMockServerConfigurers
                .mockJwt() ③
                .authorities(new SimpleGrantedAuthority("ROLE_customer")))
            .post()
            .uri("/orders/")
            .bodyValue(orderRequest)
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(Order.class).value(actualOrder -> {
                assertThat(actualOrder).isNotNull();
                assertThat(actualOrder.status()).isEqualTo(OrderStatus.REJECTED);
            });
    }
}
```

- ① Imports the application security configuration.
- ② Mocks the "ReactiveJwtDecoder" so that the application doesn't try to call Keycloak and get the public key for decoding the Access Token.
- ③ Mutates the HTTP request with a mock, JWT-formatted Access Token for a user with the "customer" role.

Open a Terminal window, navigate to the Order Service root folder, and run the newly added tests as follows.

```
$ ./gradlew test --tests OrderControllerWebFluxTests
```

As always, you can find more test examples in the source code repository accompanying this book ([Chapter12/12-end/order-service](#)).

12.4 Protecting and auditing data with Spring Security and Spring Data

So far, we have looked at securing the APIs exposed by Spring Boot applications and handling concerns like authentication and authorization. What about data? Once you have Spring Security in place, you can also secure the business and the data layers.

Regarding the business logic, you can enable the method security feature to check for user authentication or authorization directly on business methods, leveraging annotations like `@PreAuthorize`. In the Polar Bookshop system, the business layer is not complex enough to require additional security policies, so I'll not describe that.

NOTE

To learn more about how to use method authentication and authorization, you can refer to chapter 8 from "Spring Security in Action" by Laurentiu Spilca (<https://livebook.manning.com/book/spring-security-in-action/chapter-8>) where they are explained in great detail.

On the other hand, the data layer requires some extra work to address two main concerns.

- How can we tell which users created what data? Who changed it last?
- How can we ensure that each user can only access their own book orders?

This section will address both concerns. First, I'll explain how to enable auditing for users' actions on data in both Catalog Service and Order Service. Then, I'll walk you through the changes required by Order Service to keep the data private.

12.4.1 Auditing data with Spring Security and Spring Data JDBC

Let's start by considering Catalog Service, where the data layer is implemented with Spring Data JDBC. In chapter 5, you learned how to enable JDBC data auditing and configured it to save both the creation date and last modified date for each data entity. Building on top of that, we can now extend the audit scope to include the username of who created the entity and the person who modified it last.

First, we need to tell Spring Data where to get the information about the currently authenticated

user. In the previous chapter, you learned that Spring Security stores information about authenticated users in an `Authentication` object, which is stored in a `SecurityContext` object available through the `SecurityContextHolder`. We can use that object hierarchy to configure how to extract the principal for Spring Data.

DEFINING AN AUDITOR TO CAPTURE WHO CREATED/UPDATED A JDBC DATA ENTITY

In the Catalog Service project (`catalog-service`), open the `DataConfig` class. That's where we used the `@EnableJdbcAuditing` annotation to enable data auditing. Now, we'll also define an `AuditorAware` bean that should return the principal, that is, the currently authenticated user.

Listing 12.21 Configure user auditing in Spring Data JDBC (DataConfig.java)

```
@Configuration
@EnableJdbcAuditing ①
public class DataConfig {

    @Bean
    AuditorAware<String> auditorAware() { ②
        return () -> Optional
            .ofNullable(SecurityContextHolder.getContext()) ③
            .map(SecurityContext::getAuthentication) ④
            .filter(Authentication::isAuthenticated) ⑤
            .map(Authentication::getUsername); ⑥
    }
}
```

- ① Enables entity auditing in Spring Data JDBC.
- ② Returns the currently authenticated user for auditing purposes.
- ③ Extracts the `SecurityContext` object for the currently authenticated user from `SecurityContextHolder`.
- ④ Extracts the `Authentication` object for the currently authenticated user from `SecurityContext`.
- ⑤ Handles the case where a user is not authenticated, but it's manipulating data. Since we protected all the endpoints, this case should never happen, but we include it for completeness.
- ⑥ Extracts the username for the currently authenticated user from the `Authentication` object.

ADDING AUDIT METADATA FOR THE USERS WHO CREATED/UPDATED A JDBC DATA ENTITY

When an `AuditorAware` bean is defined and auditing is enabled, Spring Data will use it to extract the principal. In our case, it's the username of the currently authenticated user, represented as a `String`. Then, we can use `@CreatedBy` and `@LastModifiedBy` to annotate two new fields in the `Book` record. They will be populated by Spring Data automatically whenever a create or update operation is performed on the entity.

Listing 12.22 Define fields to capture user audit metadata in a JDBC data entity (Book.java)

```
public record Book (
    ...
    @CreatedBy ①
    String createdBy,
    @LastModifiedBy ②
    String lastModifiedBy,
) {
    public static Book of(String isbn, String title, String author,
        Double price, String publisher
    ) {
        return new Book(null, isbn, title, author, price, publisher,
            null, null, null, null, 0);
    }
}
```

- ① Who created the entity.
- ② Who modified the entity the last time.

After adding the new fields, we need to update a few classes using the `Book` all-args constructor, which now requires passing values for `createdBy` and `lastModifiedBy`.

The `BookService` class contains the logic for updating books. Open it and change the `editBookDetails()` method to ensure the audit metadata are correctly passed along when calling the data layer.

Listing 12.23 Include the existing audit metadata when updating a book (BookService.java)

```
@Service
public class BookService {
    ...
    public Book editBookDetails(String isbn, Book book) {
        return bookRepository.findById(isbn)
            .map(existingBook -> {
                var bookToUpdate = new Book(
                    existingBook.id(),
                    existingBook.isbn(),
                    book.title(),
                    book.author(),
                    book.price(),
                    book.publisher(),
                    existingBook.createdDate(),
                    existingBook.lastModifiedDate(),
                    existingBook.createdBy(), ①
                    existingBook.lastModifiedBy(), ②
                    existingBook.version());
                return bookRepository.save(bookToUpdate);
            })
            .orElseGet(() -> addBookToCatalog(book));
    }
}
```

- ① Who created the entity.
- ② Who updated the entity last.

I'll leave it to you to update the autotests in a similar way. You can also extend the tests in `BookJsonTests` to verify the serialization and deserialization of the new fields. As a reference, you can check `Chapter12/12-end/catalog-service` from the code repository accompanying this book. Make sure you update the tests that use the `Book()` constructor, or else the application build will fail.

WRITING A FLYWAY MIGRATION TO ADD THE NEW AUDIT METADATA TO THE SCHEMA

Since we changed the entity model, we need to update the database schema accordingly. Let's assume that Catalog Service is already in production, so we need a Flyway migration to update the schema in the next release. In chapter 5, we introduced Flyway for adding version control to our databases. Each change to a schema must be registered as a migration, ensuring a robust schema evolution and reproducibility.

Any change to a database schema should also be backward compatible to support common deployment strategies for cloud native applications, like rolling upgrades, blu/green deployments, or canary releases (a subject covered in chapter 15). In this case, we need to add new columns to the `book` table. As long as we don't make them mandatory, the change will be backward compatible. After changing the schema, any running instance of the previous release of Catalog Service will continue to work without errors, simply ignoring the new columns.

In the `src/main/resources/db/migration` folder for the Catalog Service project, create a new `v3__Add_user_audit.sql` migration script to add two new columns to the `book` table. Ensure you type two underscores after the version number.

Listing 12.24 Add new audit metadata to the "book" table (V3__Add_user_audit.sql)

```
ALTER TABLE book
  ADD COLUMN created_by varchar(255); ①
ALTER TABLE book
  ADD COLUMN last_modified_by varchar(255); ②
```

- ① Add a column to hold the username of who created the row.
- ② Add a column to hold the username of who updated the row last.

During the application startup, Flyway will automatically go through all the migration scripts and apply those not yet applied.

The tradeoff of enforcing backward-compatible changes is that we now have to treat as optional two fields that we need to have always filled in and possibly fail validation if they're not. That is

a common problem that can be solved over two subsequent releases of the application.

- In the first release, you add the new columns as optional and implement a data migration to fill in the new columns for all the existing data. For Catalog Service, you could use a conventional value to represent we don't know who created/updated the entity, such as unknown or anonymous.
- In the second release, we can create a new migration to update the schema safely and make the new columns required.

I'll leave that up to you if you'd like to do that. If you're interested in implementing data migrations, I recommend you check out the Flyway official documentation (<https://flywaydb.org>).

In the next section, you'll see how to test user-related auditing in Spring Data JDBC.

12.4.2 Testing data auditing with Spring Data and @WithMockUser

When we test security at the data layer, we're not interested in which authentication strategy has been adopted. The only thing we need is knowing whether the operation is performed in the context of an authenticated request.

The Spring Security Test project provides us with a handy `@WithMockUser` annotation we can use on test cases to make them run in an authenticated context. You can also add information about the mock user. Since we're testing auditing, we want to define at least a username to use as the principal.

Let's extend the `BookRepositoryJdbcTests` class with new test cases covering data auditing for users.

**Listing 12.25 Test data auditing when users are authenticated or not
(BookRepositoryJdbcTests.java)**

```
@DataJdbcTest
@Import(DataConfig.class)
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("integration")
class BookRepositoryJdbcTests {

    ...

    @Test ①
    void whenCreateBookNotAuthenticatedThenNoAuditMetadata() {
        var bookToCreate = Book.of("1232343456", "Title",
            "Author", 12.90, "Polarsophia");
        var createdBook = bookRepository.save(bookToCreate);

        assertThat(createdBook.createdBy()).isNull(); ②
        assertThat(createdBook.lastModifiedBy()).isNull();
    }

    @Test
    @WithMockUser("john") ③
    void whenCreateBookAuthenticatedThenAuditMetadata() {
        var bookToCreate = Book.of("1232343457", "Title",
            "Author", 12.90, "Polarsophia");
        var createdBook = bookRepository.save(bookToCreate);

        assertThat(createdBook.createdBy())
            .isEqualTo("john"); ④
        assertThat(createdBook.lastModifiedBy())
            .isEqualTo("john");
    }
}
```

- ① This test case is executed in an unauthenticated context.
- ② No audit data when there is no authenticated user.
- ③ This test case is executed in an authenticated context for the user "john".
- ④ Audit data when authenticated user.

Open a Terminal window, navigate to the Catalog Service root folder, and run the newly added tests as follows.

```
$ ./gradlew test --tests BookRepositoryJdbcTests
```

If you experience any failure, it might be because you haven't updated the test cases where the `Book()` constructor is used. We have added new fields to the domain model, so remember to update those test cases as well.

12.4.3 Protecting user data with Spring Security and Spring Data R2DBC

Similar to what we did in Catalog Service, this section will show you how to add data auditing for users in Order Service. Thanks to the abstractions provided by Spring Data and Spring Security, the implementation will not be much different, even if we're using Spring Data R2DBC and reactive Spring.

Besides data auditing, Order Service has one additional critical requirement. Users should be able to access only their own orders. We need to ensure the privacy of all those data. This section will also walk you through the changes necessary to accomplish that result.

DEFINING AN AUDITOR TO CAPTURE WHO CREATED/UPDATED AN R2DBC DATA ENTITY

Even in this case, we need to tell Spring Data where to get the information about the currently authenticated user. Since it's a reactive application, this time we'll get the `SecurityContext` object for the principal from the `ReactiveSecurityContextHolder`.

In the Order Service project (`order-service`), open the `DataConfig` class and add a `ReactiveAuditorAware` bean to return the username of the currently authenticated user.

Listing 12.26 Configure user auditing in Spring Data R2DBC (DataConfig.java)

```
@Configuration
@EnableR2dbcAuditing ①
public class DataConfig {

    @Bean
    ReactiveAuditorAware<String> auditorAware() { ②
        return () ->
            ReactiveSecurityContextHolder.getContext() ③
                .map(SecurityContext::getAuthentication) ④
                .filter(Authentication::isAuthenticated) ⑤
                .map(Authentication::getUsername); ⑥
    }
}
```

- ① Enables entity auditing in Spring Data R2DBC.
- ② Returns the currently authenticated user for auditing purposes.
- ③ Extracts the `SecurityContext` object for the currently authenticated user from `ReactiveSecurityContextHolder`.
- ④ Extracts the `Authentication` object for the currently authenticated user from `SecurityContext`.
- ⑤ Handles the case where a user is not authenticated, but it's manipulating data. Since we protected all the endpoints, this case should never happen, but we include it for completeness.
- ⑥ Extracts the username for the currently authenticated user from the `Authentication` object.

ADDING AUDIT METADATA FOR THE USERS WHO CREATED/UPDATED AN R2DBC DATA ENTITY

When a `ReactiveAuditorAware` bean is defined and auditing is enabled, Spring Data will use it to extract the username of the currently authenticated user represented as a `String`. Even in this case, we can use `@CreatedBy` and `@LastModifiedBy` to annotate two new fields in the `Order` record. They will be populated by Spring Data automatically whenever a create or update operation is performed on the entity.

Listing 12.27 Define fields to capture user audit metadata in an R2DBC data entity (Order.java)

```
@Table("orders")
public record Order (
    ...
    @CreatedBy ①
    String createdBy,
    @LastModifiedBy ②
    String lastModifiedBy,
) {
    public static Order of(String bookIsbn, String bookName,
        Double bookPrice, Integer quantity, OrderStatus status
    ) {
        return new Order(null, bookIsbn, bookName, bookPrice, quantity, status,
            null, null, null, null, 0);
    }
}
```

- ① Who created the entity.
- ② Who modified the entity the last time.

After adding the new fields, we need to update a few classes using the `Order` all-args constructor, which now requires passing values for `createdBy` and `lastModifiedBy`.

The `OrderService` class contains the logic for updating dispatched orders. Open it and change the `buildDispatchedOrder()` method to ensure the audit metadata are correctly passed along when calling the data layer.

Listing 12.28 Include the existing audit metadata when updating an order (OrderService.java)

```

@Service
public class OrderService {
    ...

    private Order buildDispatchedOrder(Order existingOrder) {
        return new Order(
            existingOrder.id(),
            existingOrder.bookIsbn(),
            existingOrder.bookName(),
            existingOrder.bookPrice(),
            existingOrder.quantity(),
            OrderStatus.DISPATCHED,
            existingOrder.createdDate(),
            existingOrder.lastModifiedDate(),
            existingOrder.createdBy(), ①
            existingOrder.lastModifiedBy(), ②
            existingOrder.version()
        );
    }
}

```

- ① Who created the entity.
- ② Who updated the entity last.

I'll leave it to you to update the autotests in a similar way. You can also extend the tests in `OrderJsonTests` to verify the serialization of the new fields. As a reference, you can check `Chapter12/12-end/order-service` from the code repository accompanying this book. Ensure you update the tests using the `Order()` constructor, or the application build will fail.

WRITING A FLYWAY MIGRATION TO ADD THE NEW AUDIT METADATA TO THE SCHEMA

Similar to what we did for Catalog Service, we need to write a migration to update the database schema according to the new two fields hosting the usernames of who created an entity and who modified it last.

In the `src/main/resources/db/migration` folder for the Order Service project, create a new `v2__Add_user_audit.sql` migration script to add two new columns to the `orders` table. Ensure you type two underscores after the version number.

Listing 12.29 Add new audit metadata to the "orders" table (V2__Add_user_audit.sql)

```

ALTER TABLE orders
    ADD COLUMN created_by varchar(255); ①
ALTER TABLE orders
    ADD COLUMN last_modified_by varchar(255); ②

```

- ① Add a column to hold the username of who created the row.
- ② Add a column to hold the username of who updated the row last.

ENSURING USER DATA PRIVACY

There's one last requirement we haven't covered yet: ensuring that orders data are only accessed by the users who created them. No user should be able to see the orders from another person.

There are a few different solutions to implement this requirement in Spring. We'll follow these steps:

1. Adding a custom query to `OrderRepository` to filter orders based on the user who created them.
2. Updating `OrderService` to use the new query instead of the default `findAll()`.
3. Updating `OrderController` to extract the username of the currently authenticated user from the security context and pass it over to `OrderService` when asking for orders.

WARNING We'll rely on a specific solution that ensures each user can only access their own orders via the `/orders` endpoint. However, it won't prevent developers from using the other methods exposed by `OrderRepository` in the future and leaking private data. If you'd like to know more about how to improve this solution, you can refer to chapter 17 of "Spring Security in Action" by Laurentiu Spilca (<https://livebook.manning.com/book/spring-security-in-action/chapter-17>).

Let's start with `OrderRepository`. Using the conventions you learned in chapter 5, define a method to find all orders created by the specified user. Spring Data will generate an implementation for it at runtime.

Listing 12.30 Define a method returning orders per user (`OrderRepository.java`)

```
public interface OrderRepository
    extends ReactiveCrudRepository<Order, Long>
{
    Flux<Order> findAllByCreatedBy(String userId); ①
}
```

- ① Custom method to query only the orders created by the given user.

Then, we need to update the `getAllOrders()` method in `OrderService` to accept a username as input and use the new query method provided by `OrderRepository`.

Listing 12.31 Return orders only per user (OrderService.java)

```
@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public Flux<Order> getAllOrders(String userId) { ①
        return orderRepository.findAllByCreatedBy(userId);
    }

    ...
}
```

- ① When requesting all orders, the response includes only those belonging to the given user.

Finally, let's update the `getAllOrders()` method in `OrderController`. As you learned in the previous chapter, you can autowire an object representing the currently authenticated user through the `@AuthenticationPrincipal` annotation. In Edge Service, the object was of type `OidcUser` because it's based on OpenID Connect authentication. Since Order Service is configured with JWT Authentication, the principal will be of type `Jwt`. We can use the JWT (an Access Token) to read the `sub` claim containing the username for which the Access Token was generated (i.e., the subject).

Listing 12.32 Get the username from the authentication principal and return only the orders created by them (OrderController.java)

```
@RestController
@RequestMapping("orders")
public class OrderController {
    private final OrderService orderService;

    @GetMapping
    public Flux<Order> getAllOrders(
        @AuthenticationPrincipal Jwt jwt ①
    ) {
        return orderService.getAllOrders(jwt.getSubject()); ②
    }

    ...
}
```

- ① Autowires the JWT representing the currently authenticated user.
- ② Extracts the subject of the JWT and uses it as the user identifier.

That's it for Order Service. In the next section, you'll write some autotests to verify the data auditing and protection requirements.

12.4.4 Testing data auditing and protection with @WithMockUser and Spring Data R2DBC

In the previous section, we configured data auditing for users and enforced a policy to return only the orders for the currently authenticated user. This section will show you how to test the data auditing part as a slice test. To verify the data protection requirement, you can refer to the repository accompanying the book and check how it's been covered by the integration tests in the `OrderServiceApplicationTests` class (Chapter12/12-end/order-service/src/test/java),

Data auditing is applied at the repository level. We can extend the `OrderRepositoryR2dbcTests` class with extra test cases covering the scenario where the user is authenticated and when it is not.

Similar to what we did in Catalog Service, we can use the `@WithMockUser` annotation from Spring Security to execute a test method in an authenticated context, relying on a mock user representation.

Listing 12.33 Test data auditing when users are authenticated or not (`OrderRepositoryR2dbcTests.java`)

```
@DataR2dbcTest
@Import(DataConfig.class)
@Testcontainers
class OrderRepositoryR2dbcTests {
    ...

    @Test
    void whenCreateOrderNotAuthenticatedThenNoAuditMetadata() {
        var rejectedOrder = OrderService.buildRejectedOrder("1234567890", 3);
        StepVerifier.create(orderRepository.save(rejectedOrder))
            .expectNextMatches(order -> Objects.isNull(order.createdBy()) &&
                Objects.isNull(order.lastModifiedBy())) ①
            .verifyComplete();
    }

    @Test
    @WithMockUser("marlena")
    void whenCreateOrderAuthenticatedThenAuditMetadata() {
        var rejectedOrder = OrderService.buildRejectedOrder("1234567890", 3);
        StepVerifier.create(orderRepository.save(rejectedOrder))
            .expectNextMatches(order -> order.createdBy().equals("marlena") &&
                order.lastModifiedBy().equals("marlena")) ②
            .verifyComplete();
    }
}
```

- ① When the user is not authenticated, no audit metadata is saved.
- ② When the user is authenticated, the information about who created/updated the entity is correctly included in the data.

Open a Terminal window, navigate to the Catalog Service root folder, and run the newly added

tests as follows.

```
$ ./gradlew test --tests OrderRepositoryR2dbcTests
```

If you experience any failure, it might be because you haven't updated the test cases where the `Order()` constructor is used. We have added new fields to the domain model, so remember to update those test cases as well.

That concludes the discussion around authentication, authorization, and auditing for both imperative and reactive cloud native applications using Spring Boot, Spring Security, Spring Data, and Keycloak.

SIDE BAR

Polar Labs

Feel free to apply what you learned in the previous chapters and update Catalog Service and Order Service for deployment.

1. **Update the Docker Compose definitions for both applications to configure the Keycloak URL. You can use the container name (`polar-keycloak:8080`), which gets resolved by the built-in Docker DNS.**
2. **Update the Kubernetes manifests for both applications to configure the Keycloak URL. You can use the Keycloak Service name (`polar-keycloak`) as the URL since all interactions happen within the cluster.**

You can refer to the `Chapter12/12-end` folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and deploy the backing services from the manifests available in the `Chapter12/12-end/polar-deployment/kubernetes/platform/development` folder with `kubectl apply -f services` or the entire cluster with `./create-cluster.sh`.

12.5 Summary

- In an OIDC/OAuth2 setup, the Client (Edge Service) is granted access to a Resource Server (Catalog Service and Order Service) on behalf of the user through an Access Token.
- Spring Cloud Gateway provides a `TokenRelay` filter to add the Access Token to any request routed downstream automatically.
- Following the JWT format, ID Tokens and Access Tokens can propagate relevant information as claims about the authenticated user. For example, you can add a `roles` claim and configure Spring Security with authorization policies depending on the user role.
- Spring Boot applications can be configured as OAuth2 Resource Servers using Spring Security.
- In an OAuth2 Resource Server, the strategy for authenticating users is entirely based on a valid Access Token provided in the `Authorization` header of each request. We call it JWT Authentication.
- In an OAuth2 Resource Server, security policies are still enforced through a `SecurityFilterChain` (imperative) or `SecurityWebFilterChain` (reactive) bean.
- Spring Security represents permissions, roles, and scopes as `GrantedAuthority` objects.
- You can provide a custom `JwtAuthenticationConverter` bean to define how to extract granted authorities from a JWT, for example, the `roles` claim.
- Granted authorities can be used to adopt an RBAC strategy and protect endpoints depending on the user role.
- Spring Data projects support auditing to track who created an entity and who updated it last. You can enable this feature in both Spring Data JDBC and Spring Data R2DBC by configuring an `AuditorAware` (or `ReactiveAuditorAware`) bean to return the username of the currently authenticated user.
- When data auditing is enabled, you can use the `@CreatedBy` and `@LastModifiedBy` annotations for automatically injecting the right values when a create or update operation occurs.
- Testing security is challenging, but Spring Security provides convenient utilities to make that easier, including expressions to mutate HTTP requests to include a JWT Access Token (`.with(jwt())` or `.mutateWith(mockJwt())`) or run a test case in a specific security context for a given user (`@WithMockUser`).
- Testcontainers can help write full integration tests by using an actual Keycloak container to verify the interactions with Spring Security.

13

Observability and monitoring

This chapter covers

- Logging with Spring Boot, Loki, and Fluent Bit
- Using health probes with Spring Boot Actuator and Kubernetes
- Producing metrics with Spring Boot Actuator, Prometheus, and Grafana
- Configuring distributed tracing with OpenTelemetry and Tempo
- Managing applications with Spring Boot Actuator

In the previous chapters, you learned several patterns and technologies to build secure, scalable, and resilient applications. However, we still lack visibility into the Polar Bookshop system, especially when something goes wrong. Before going to production, we should ensure our applications are observable and that the deployment platform provides all the tools needed to monitor and gain insights into the system.

Monitoring is about checking the telemetry available for the application and defining alerts for known failure states. Observability goes beyond that and aims at reaching a state where we can ask arbitrary questions about the system without knowing the question in advance. The product team should ensure their applications expose relevant information. The platform team should provide an infrastructure to consume that information and ask questions about their operations.

As you remember from chapter 1, *observability* is one of the properties of cloud native applications. Observability is a measure of how well we can infer the internal state of an application from its outputs. In chapter 2, you learned about the 15-Factors methodology, which contains two factors that help build observable applications. Factor XIV suggests treating your applications as space probes, therefore reasoning about what kind of telemetry you'd need to monitor and control your applications remotely, such as logs, metrics, and traces. Factor VI recommends treating logs as a stream of events rather than dealing with log files.

In this chapter, you'll learn how to ensure your Spring Boot applications expose relevant information to infer their internal states, such as logs, health probes, metrics, traces, and additional valuable data regarding schema migrations and builds. I'll also show you how to use the Grafana open-source observability stack to validate the changes you'll make to your applications. However, I won't go into too many details because that's something the platform team deploys and operates.

NOTE

The source code for the examples in this chapter is available in the Chapter13/13-begin and Chapter13/13-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

13.1 Logging with Spring Boot, Loki, and Fluent Bit

Logs (or *event logs*) are discrete records of something that happened over time in a software application. They are composed of a timestamp necessary to answer the question "when did the event happen?" and some information providing details about the event and its context, which lets us answer questions like "what happened at this time?", "which thread was processing the event?" or "which user/tenant was in the context?".

During troubleshooting and debugging tasks, logs are among the essential tools we can use to reconstruct what happened at a specific point in time in a single application instance. They're usually categorized depending on the type or severity of the event, such as *trace*, *debug*, *info*, *warn*, and *error*. It's a flexible mechanism that lets us log only the most severe events in production while still giving us the chance to change the log level temporarily during debugging.

The format of a log record can vary, going from simple plain text to a more organized collection of key/value pairs to fully structured records produced in a JSON format.

Traditionally, we've been configuring logs to be printed out in files located on the host machine, resulting in applications dealing with aspects like filename conventions, file rotation, and file sizes. In the cloud, we follow the 15-Factory methodology, which recommends treating logs as events streamed to the standard output. Cloud native applications stream logs and are not concerned with how they are processed or stored.

This section will teach you how to add and configure logs in Spring Boot applications. Then, I'll cover how collecting and aggregating logs works in a cloud native infrastructure. Finally, you'll run Fluent Bit for log collection, Loki for log aggregation, and see how to use Grafana to query the logs produced by your Spring Boot applications.

13.1.1 Logging with Spring Boot

Spring Boot comes with built-in support and auto-configuration for the most common logging frameworks, including Logback, Log4J2, Commons Logging, and Java Util Logging. By default, Logback is used (<https://logback.qos.ch>), but you can easily replace it with another library thanks to the abstraction provided by the Simple Logging Facade for Java (SLF4J).

Using the interfaces from SLF4J (<https://www.slf4j.org>), you get the freedom to change the logging library without changing your Java code. Furthermore, cloud native applications should treat logs as events and stream them to the standard out. That's precisely what Spring Boot does out of the box. Convenient, right?

CONFIGURING LOGGING IN SPRING BOOT

Event logs are categorized by level with decreasing details and increasing importance: trace, debug, info, warn, error. By default, Spring Boot logs everything from the *info* level up. A *logger* is a class producing log events. You can set logger levels through configuration properties, with options to apply global configurations or target specific packages or classes. For example, in chapter 9, we set a *debug* logger to get more details about the circuit breakers implemented with Resilience4J (application.yml file in the Edge Service project).

```
logging:
  level:
    io.github.resilience4j: debug ①
```

- ① Sets a debug logger for the Resilience4J library

You might need to configure multiple loggers at the same time. In that case, you can collect them in a *log group* and apply the configuration to the group directly. Spring Boot provides two pre-defined log groups, `web` and `sql`, but you can also define your own. For example, to better analyze the behavior of the circuit breakers defined in the Edge Service application, you could define a log group and configure a log level for both Resilience4J and Spring Cloud Circuit Breaker. In the Edge Service project (`edge-service`), you can configure the new log group in the `application.yml` file as follows.

Listing 13.1 Configure a group to control the log level for the circuit breakers logic (application.yml)

```
logging:
  group:
    circuitbreaker: io.github.resilience4j,
  ↵org.springframework.cloud.circuitbreaker ①
    level:
      circuitbreaker: info ②
```

- ① Collects multiple loggers into a group to apply the same configuration.

- ② Sets an "info" logger for both Resilience4J and Spring Cloud Circuit Breaker, easy to change if you need to debug the circuit breakers.

By default, each event log provides essential information, including the date and time of the occurrence, the log level, the process identifier (PID), the name of the thread from which the event was triggered, the logger name, and the log message. If you check the application logs from a Terminal that supports ANSI, the log messages are also colored to improve readability (figure 13.1). The logging format can be customized using the `logging.pattern` configuration property group.

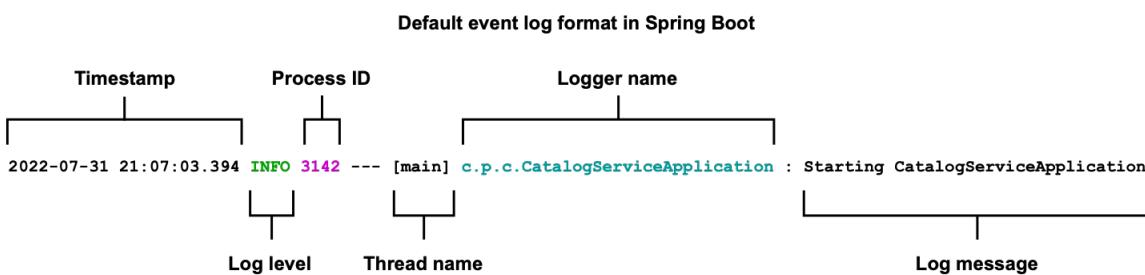


Figure 13.1 Event logs include a timestamp, contextual information, and a message about what happened.

NOTE

Spring Boot provides extensive options to configure logging to files. Since it's not useful for cloud native applications, I won't cover it in this book. If you're interested in the subject, feel free to check the official documentation to learn more about log files (<http://spring.io/projects/spring-boot>).

ADDING LOGS TO SPRING BOOT APPLICATIONS

Besides configuring loggers for the frameworks and libraries used in your project, you should define event logs in your code whenever applicable. How much logging is enough? It depends on the context. In general, I reckon that it's better to have too much logging than too little. I've seen many deployments containing just changes to add more logging, while it's pretty rare to see the opposite.

Thanks to the SLF4J facade, the syntax for defining new event logs in Java is the same no matter which logging library you use: a `Logger` instance created from a `LoggerFactory`. Let's see how it works by adding new log messages to the web controller in Catalog Service.

In the Catalog Service project (`catalog-service`), go to the `BookController` class, define a `Logger` instance from SLF4J, and add messages to be printed out whenever a client calls the application REST API.

Listing 13.2 Define log events using SL4FJ (BookController.java)

```

package com.polarbookshop.catalogservice.web;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...

@RestController
@RequestMapping("books")
public class BookController {
    private static final Logger log =
        LoggerFactory.getLogger(BookController.class); ①
    private final BookService bookService;

    @GetMapping
    public Iterable<Book> get() {
        log.info(②
            "Fetching the list of books in the catalog"
        );
        return bookService.viewBookList();
    }

    ...
}

```

- ① Defines a logger for the BookController class.
- ② Logs the given message at the "info" level.

NOTE

Go ahead and define new loggers and log events to all the applications composing the Polar Bookshop system whenever it makes sense. As a reference, you can look at the source code repository accompanying this book (Chapter13/13-end).

SIDE BAR

The Mapped Diagnostic Context (MDC)

You likely need to add common information to your log messages, for example, the identifier of the currently authenticated user, the tenant for the current context, or the request URI. You could directly add that information to your log message, as you did in the previous listing. It would work, but the data would not be structured. Instead, we prefer working with structured data.

SLF4J and common logging libraries like Logback and Log4J2 support adding structured information depending on the request context (authentication, tenant, thread) through a tool named Mapped Diagnostic Context (MDC). If you'd like to know more about MDC, I recommend you check the official documentation for the specific logging library you're using.

Now that applications log messages as an event stream, we need to collect and store them in a central place that we can query. The following section will provide a solution to accomplish all of that.

13.1.2 Managing logs with Loki, Fluent Bit, and Grafana

When moving to distributed systems like microservices and complex environments like the cloud, managing logs became challenging and required a different solution than more traditional applications. If something goes wrong, where can we find data about the failure? Traditional applications would rely on log files stored on the host machine. Cloud native applications are deployed in dynamic environments, replicated, and with different life spans. We need to collect logs from all applications running in the environment and send them to a central component where logs are aggregated, stored, and searchable.

There are plenty of options for managing logs in the cloud. Cloud providers have their own offerings, like Azure Monitor Logs or Google Cloud Logging. And many enterprise solutions are available on the market, such as Honeycomb, Humio, NewRelic, DataDog, and Elastic.

For Polar Bookshop, we'll use a solution based on the Grafana observability stack (<https://grafana.com>). It's composed of open-source technologies, and you can run it yourself in any environment. It's also available as a managed service (Grafana Cloud) offered by the Grafana Labs.

The components of the Grafana stack we'll use for managing logs are Loki for log storage/search, Fluent Bit for log collection/aggregation, and Grafana for log data visualization and querying.

NOTE

Which technology to use for managing logs is a platform choice and shouldn't impact applications at all. For example, you should be able to replace the Grafana stack with Humio without any change to the Polar Bookshop applications.

We need a **log collector** to fetch log messages from the standard output of all the running applications. Using the Grafana stack, you're free to choose a log collector among several options. For the Polar Bookshop system, we'll use Fluent Bit, an open-source and CNCF-graduated project that "allows you to collect any data like metrics and logs from different sources, enrich them with filters and send them to multiple destinations" (<https://fluentbit.io>). Fluent Bit is a subproject of Fluentd, "an open-source data collector for unified logging layer" (<https://www.fluentd.org>).

Fluent Bit will collect logs from all running containers and forward them to Loki, which will store them and make them searchable. Loki (<https://grafana.com/oss/loki>) is "a log aggregation system designed to store and query logs from all your applications and infrastructure".

Finally, Grafana will use Loki as a data source and provide log visualization features. Grafana (<https://grafana.com/oss/grafana>) "allows you to query, visualize, alert on and understand" your

telemetry no matter where they are stored. Figure 13.2 illustrates the logging architecture.

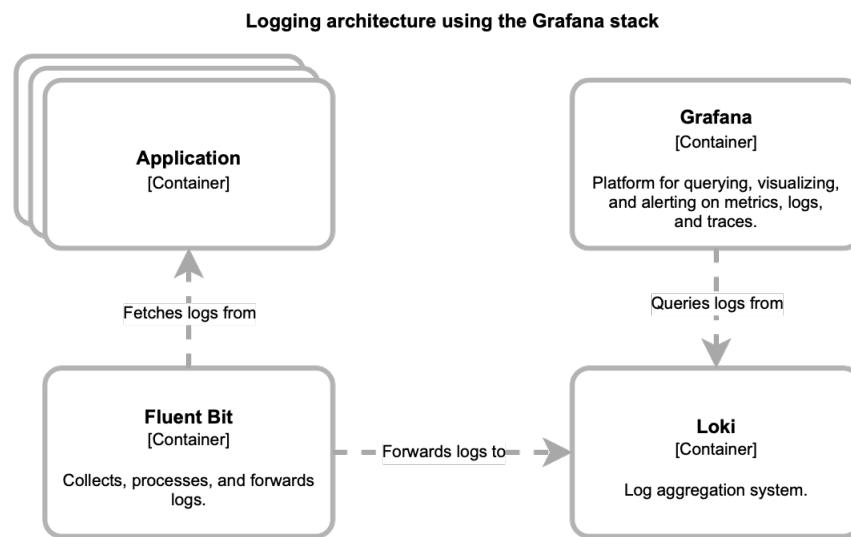


Figure 13.2 Logging architecture for cloud native applications based on the Grafana stack.

Let's start by running Grafana, Loki, and Fluent Bit as containers. In your Polar Deployment project (`polar-deployment`), update the Docker Compose configuration (`docker/docker-compose.yml`) to include the new services. They are configured through files I have included in the source code repository accompanying this book (`Chapter13/13-end/polar-deployment/docker/observability`). Copy the `observability` folder over the same path in your own project.

Listing 13.3 Define containers for Grafana, Loki, and Fluent Bit (docker-compose.yml)

```

version: "3.8"
services:
  ...
  grafana:
    image: grafana/grafana:8.4.4
    container_name: grafana
    depends_on:
      - loki
    ports:
      - "3000:3000"
    environment: ①
      - GF_SECURITY_ADMIN_USER=user
      - GF_SECURITY_ADMIN_PASSWORD=password
    volumes: ②
      - ./observability/grafana/datasource.yml:/etc/grafana/provisioning/
  ↵datasources/datasource.yml
      - ./observability/grafana/dashboards:/etc/grafana/provisioning/
  ↵dashboards
      - ./observability/grafana/grafana.ini:/etc/grafana/grafana.ini

  loki:
    image: grafana/loki:2.4.2
    container_name: loki
    depends_on:
      - fluent-bit
    ports:
      - "3100:3100"

  fluent-bit:
    image: grafana/fluent-bit-plugin-loki:2.4.2-amd64
    container_name: fluent-bit
    ports:
      - "24224:24224"
    environment:
      - LOKI_URL=http://loki:3100/loki/api/v1/push ③
    volumes: ④
      - ./observability/fluent-bit/fluent-bit.conf:/fluent-bit/etc/
  ↵fluent-bit.conf

```

- ① Username and password to access Grafana.
- ② Volumes are used to load configuration for data sources and dashboards.
- ③ Defines the Loki URL used to forward log messages.
- ④ Volumes are used to load configuration for collecting and delivering logs.

Next, start all three containers with the following command. Thanks to the dependencies defined in Docker Compose between containers, it's enough to start Grafana to run also Loki and Fluent Bit.

```
$ docker-compose up -d grafana
```

Fluent Bit can be configured to collect logs from different sources. For Polar Bookshop, we'll rely on the Fluentd driver available in Docker to collect logs automatically from running containers. The Docker platform itself listens to the log events from each container and routes

them to the specified service. In Docker, a logging driver can be configured directly on a container. For example, update the Catalog Service configuration in Docker Compose to use the Fluentd logging driver, which will send the logs over to the Fluent Bit container.

Listing 13.4 Use Fluentd driver to route container logs to Fluent Bit (docker-compose.yml)

```
version: "3.8"
services:
  ...
  catalog-service:
    depends_on:
      - fluent-bit ①
      - polar-keycloak
      - polar-postgres
    image: "catalog-service"
    container_name: "catalog-service"
    ports:
      - 9001:9001
      - 8001:8001
    environment:
      - BPL_JVM_THREAD_COUNT=50
      - BPL_DEBUG_ENABLED=true
      - BPL_DEBUG_PORT=8001
      - SPRING_CLOUD_CONFIG_URI=http://config-service:8888
      - SPRING_DATASOURCE_URL=
    ↪ jdbc:postgresql://polar-postgres:5432/polardb_catalog
      - SPRING_PROFILES_ACTIVE=testdata
      - SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUER_URI=
    ↪ http://host.docker.internal:8080/realms/PolarBookshop
      logging: ②
      driver: fluentd ③
      options:
        fluentd-address: 127.0.0.1:24224 ④
```

- ① Ensures the Fluent Bit container is started before Catalog Service.
- ② Section to configure the container logging driver.
- ③ Which logging driver to use.
- ④ The address of the Fluent Bit instance where to route the logs.

Next, package Catalog Service as a container image (`./gradlew bootBuildImage`) and run the application container as follows. Thanks to the dependencies defined in Docker Compose between containers, Keycloak and PostgreSQL are automatically started as well.

```
$ docker-compose up -d catalog-service
```

Now, we're ready to test the logging setup. First, send a few requests to Catalog Service to trigger the generation of some log messages.

```
$ http :9001/books
```

Next, open a browser window, head to Grafana (<http://localhost:3000>), and use the credentials configured in Docker Compose to log in (user/password). Then, select the "Explore" page from the left menu, choose "Loki" as the data source, choose "Last 1 hour" from the time drop-down

menu, and run the following query to search for all the logs produced by the catalog-service container.

```
{container_name="/catalog-service"}
```

The result should be as in figure 13.3, showing the logs from the application startup as well as the custom log messages you added to the BookController class.

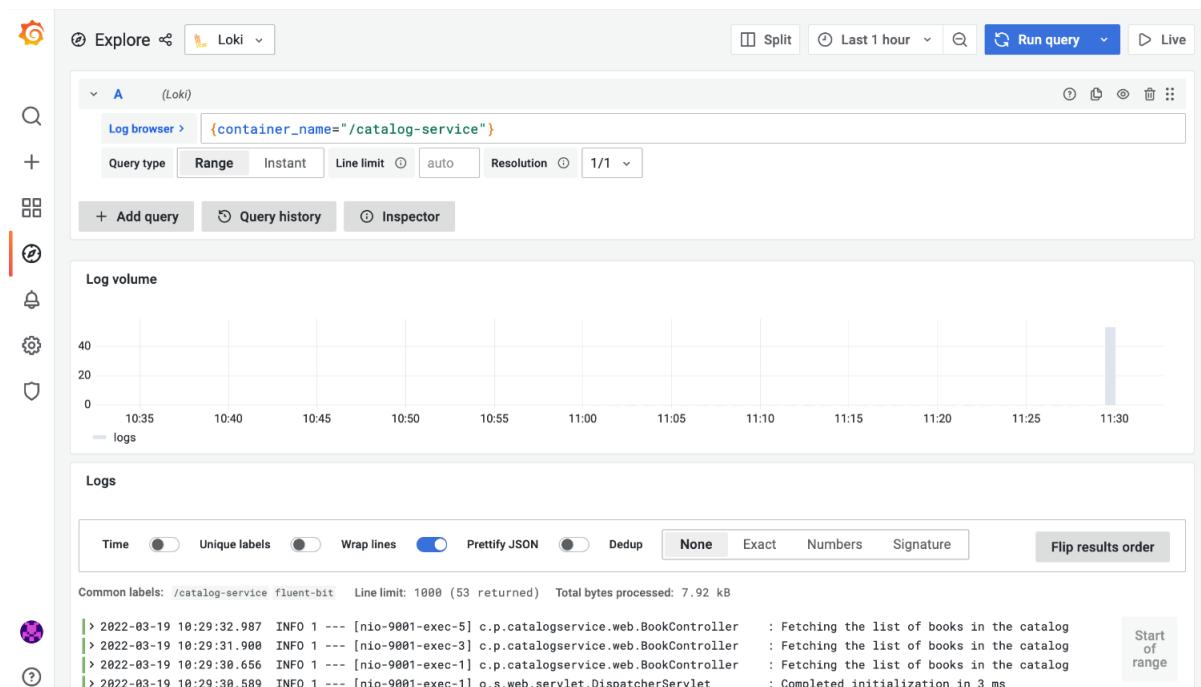


Figure 13.3 In Grafana, you can browse and search log messages aggregated and stored by Loki.

When you’re done testing the logging setup, stop all containers with docker-compose down.

NOTE

Following the same approach, update the Docker Compose configuration for all the other Spring Boot applications in the Polar Bookshop system to use the Fluentd logging driver and rely on Fluent Bit for collecting logs. As a reference, you can look at the source code repository accompanying this book (Chapter13/13-end/polar-deployment/docker).

Logs provide some information about how an application behaves, but they’re not enough to infer its internal state. The next section will cover how to make applications expose more data about their health status.

13.2 Health probes with Spring Boot Actuator and Kubernetes

Once an application is deployed, how can we tell if it's healthy? Is it capable of handling new requests? Did it enter a faulty state? Cloud native applications should provide information about their health so that monitoring tools and deployment platforms can detect when there's something wrong and act accordingly. We need dedicated health endpoints to check the status of the application and any components or services it might use.

The deployment platform can periodically invoke health endpoints exposed by applications. A monitoring tool could trigger an alert or a notification when an application instance is unhealthy. In the case of Kubernetes, the platform will check the health endpoints and automatically replace the faulty instance or temporarily stop sending traffic to it until it's ready to handle new requests again.

For Spring Boot applications, you can leverage the Actuator library to expose information about their health through a `/actuator/health` HTTP endpoint, including details about the application status and the components in use like databases, event brokers, and config servers.

Spring Boot Actuator is a useful library providing many endpoints to monitor and manage Spring Boot applications. Such endpoints can be exposed through HTTP or JMX. Either way, we must protect them from unauthorized access. We'll limit ourselves to using the HTTP endpoints, so we can use Spring Security to define access policies like for any other endpoint we've worked with so far.

This section will cover how to configure health endpoints in Spring Boot applications using Actuator. You'll then see how to define liveness and readiness probes for Kubernetes to accomplish its self-healing functionality.

13.2.1 Defining health probes for Spring Boot applications using Actuator

First of all, open the `build.gradle` file in the Catalog Service project (`catalog-service`) and ensure it contains a dependency on Spring Boot Actuator (we used it in chapter 4 for refreshing configuration at runtime).

Listing 13.5 Add dependency for Spring Boot Actuator in Catalog Service (build.gradle)

```
dependencies {
    ...
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
}
```

There are a few viable solutions to protect the Spring Boot Actuator endpoints. For example, you could enable HTTP Basic authentication just for the Actuator endpoints, while all the others keep

using OpenID Connect and OAuth2. For simplicity, in the Polar Bookshop system, we'll keep the Actuator endpoints unauthenticated from inside the Kubernetes cluster and block any access to them from the outside (as you'll see in chapter 15).

WARNING In a real production scenario, I would recommend protecting access to the Actuator endpoints even from within the cluster.

Go to the `SecurityConfig` class and update the Spring Security configuration to allow unauthenticated access to the Spring Boot Actuator endpoints.

Listing 13.6 Allow unauthenticated access to the Actuator endpoints in Catalog Service (`SecurityConfig.java`)

```
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(authorize -> authorize
                .mvcMatchers("/actuator/**").permitAll() ①
                .mvcMatchers(HttpMethod.GET, "/", "/books/**").permitAll()
                .anyRequest().hasRole("employee"))
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt)
            .sessionManagement(sessionManagement -> sessionManagement
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(AbstractHttpConfigurer::disable)
            .build();
    }
}
```

- ① Allows unauthenticated access to any Spring Boot Actuator endpoint.

Finally, open the `application.yml` file in your Catalog Service project (`catalog-service`) and configure Actuator to expose the health HTTP endpoint. If you followed the examples in chapter 4, you might have an existing configuration for the `refresh` endpoint. In that case, go ahead and replace it with the `health` endpoint.

Listing 13.7 Expose the "health" Actuator endpoint (`application.yml`)

```
management:
  endpoints:
    web:
      exposure:
        include: health ①
```

- ① Exposes the `/actuator/health` endpoint via HTTP.

Let's check the result. First, we need to run all the backing services used by Catalog Service: Config Service, Keycloak, and PostgreSQL. We'll run them as containers. Package Config

Service as a container image (`./gradlew bootBuildImage`). Then, open a Terminal window, navigate to the folder where you keep your Docker Compose file (`polar-deployment/docker`), and run the following command.

```
$ docker-compose up -d config-service polar-postgres polar-keycloak
```

After ensuring that all containers are ready, run Catalog Service on the JVM (`./gradlew bootRun`), open a Terminal window, and send an HTTP GET request to the health endpoint.

```
$ http :9001/actuator/health
```

The endpoint will return the overall health status for the Catalog Service application, which can be one of UP, OUT_OF_SERVICE, DOWN, or UNKNOWN. When the health status is UP, the endpoint returns a *200 OK* response. If it's not, it produces a *503 Service Unavailable* response.

```
{
  "status": "UP"
}
```

By default, Spring Boot Actuator only returns the overall health status. Through application properties, you can make it provide specific information regarding several components used by the application. To better protect access to this kind of information, you can enable showing health details and components always (always) or only when the request is authorized (when_authorized). Since we're not protecting the Actuator endpoints at the application level, let's make the extra information always available.

Listing 13.8 Configure the health endpoint to expose more information (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: health
  endpoint:
    health:
      show-details: always ①
      show-components: always ②
```

- ① Always shows details about the application's health.
- ② Always shows information about the components used by the application.

Once again, rerun Catalog Service (`./gradlew bootRun`) and send an HTTP GET request to <http://localhost:9001/actuator/health>. This time, the resulting JSON object contains more detailed information about the application's health. Here's a partial result as an example.

```
{
  "components": { ①
    "clientConfigServer": {
      "details": {
        "propertySources": [
          "configserver:https://github.com/PolarBookshop/
        ↗config-repo/catalog-service.yml",
          "configClient"
        ]
      },
      "status": "UP"
    },
    "db": {
      "details": {
        "database": "PostgreSQL",
        "validationQuery": "isValid()"
      },
      "status": "UP"
    },
    ...
  },
  "status": "UP" ②
}
}
```

- ① Detailed health information about components and features used by the application.
- ② Overall application health status.

The generic health endpoint provided by Spring Boot Actuator is useful for monitoring and configuring alerts or notifications since it contains details regarding both the application and the integration to its backing services. In the next section, you'll see how to expose more specific information used by a deployment platform like Kubernetes to manage containers. Before moving on, stop the application process (`Ctrl+C`), but keep all the current containers running. You'll need them soon!

13.2.2 Configuring health probes in Spring Boot and Kubernetes

Besides showing detailed information about the application health, Spring Boot Actuator detects automatically when the application runs on a Kubernetes environment and enables the *health probes* to return liveness (`/actuator/health/liveness`) and readiness (`/actuator/health/readiness`) states, as described in figure [13.4](#).

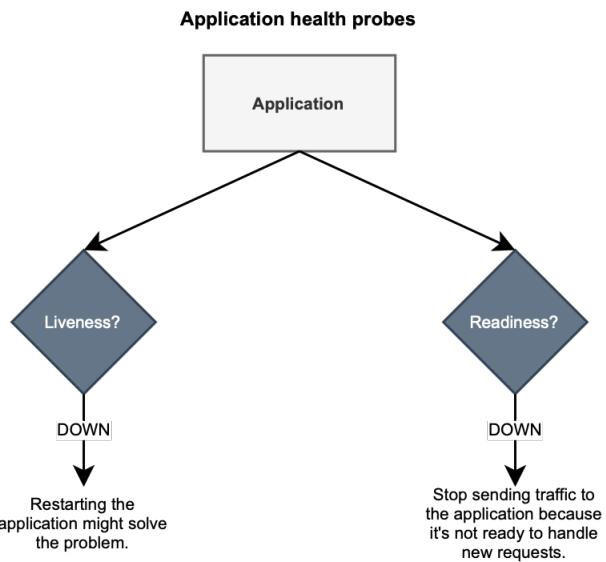


Figure 13.4 Kubernetes uses liveness and readiness probes to accomplish its self-healing features in case of failures.

- **Liveness state.** When an application is not live, it means it has entered a faulty internal state from which it won't recover. By default, Kubernetes will try restarting it to fix the problem.
- **Readiness state.** When an application is not ready, it means it can't process new requests, either because it's still initializing all its components (during the startup phase) or because it's overloaded. Kubernetes will stop forwarding requests to that instance until it's ready to accept new requests again.

CUSTOMIZING LIVENESS AND READINESS PROBES

To extend support for the health probes in any environment, you can configure Spring Boot Actuator through the dedicated properties. Open the Catalog Service project (`catalog-service`) and update the `application.yml` file as follows.

Listing 13.9 Enable liveness and readiness probes in any environment (application.yml)

```

management:
  endpoints:
    web:
      exposure:
        include: health
  endpoint:
    health:
      show-details: always
      show-components: always
  probes:
    enabled: true ①
  
```

- ① Enables support for the health probes.

Let's check the result. All the backing services for Catalog Service should be up and running on Docker from the previous section. If not, go back and follow the instructions to start them all (

`docker-compose up -d config-service polar-postgres polar-keycloak`). Then, run Catalog Service on the JVM (`./gradlew bootRun`) and invoke the endpoint for the liveness probe.

```
$ http :9001/actuator/health/liveness
{
  "status": "UP"
}
```

The liveness state of a Spring Boot application tells whether it's in a correct or broken internal state. If the Spring application context has started successfully, the internal state is valid. It doesn't depend on any external components. Otherwise, it would cause cascading failures since Kubernetes would try to restart the broken instances.

Finally, check the result for the readiness probe endpoint.

```
$ http :9001/actuator/health/readiness
{
  "status": "UP"
}
```

The readiness state of a Spring Boot application tells whether it's ready to accept traffic and process new requests. During the startup phase or graceful shutdown, the application is not ready and will refuse any request. It might also become temporarily not ready if, at some point, it's overloaded. Whenever not ready, Kubernetes will not send any traffic to the application instance.

When you're done testing the health endpoints, stop the application (`Ctrl+C`) and the containers (`docker-compose down`).

NOTE

Go ahead and add Spring Boot Actuator to all the applications composing the Polar Bookshop system. In Order Service and Edge Service, remember to configure unauthenticated access to the Actuator endpoints in the `SecurityConfig` class, similar to what we did for Catalog Service. In Dispatcher Service, you also need to add a dependency on Spring WebFlux (`org.springframework.boot:spring-boot-starter-webflux`) because Actuator needs a web server configured to serve its endpoints over HTTP. Then, configure the health endpoints for all the applications, as you learned in this section. As a reference, you can look at the source code repository accompanying this book (Chapter13/13-end).

By default, the readiness probe in Spring Boot doesn't depend on any external component. It's up to you to decide whether any external system should be included in the readiness probe.

For example, Catalog Service is an external system for Order Service. Should you include it in the readiness probe? Since Order Service adopts resilience patterns to deal with the scenario where Catalog Service is unavailable, you should keep Catalog Service out of the readiness

probe. When not available, Order Service will keep working correctly, but with graceful functionality degradation.

Let's consider another example. Edge Service depends on Redis for storing and retrieving web session data. Should you include it in the readiness probe? Since Edge Service can't process any new request without accessing Redis, including Redis in the readiness probe might be a good idea. Spring Boot Actuator will consider both the internal state of the application and the integration with Redis to determine whether the application is ready to accept new requests.

In the Edge Service project (`edge-service`), open the `application.yml` file and define what indicators to use in the readiness probe: the application standard readiness state and the Redis health status. I assume you have already added Spring Boot Actuator to Edge Service and configured the health endpoints as described earlier.

Listing 13.10 Include Redis in the computation of the readiness state (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: health
  endpoint:
    health:
      show-details: always
      show-components: always
    probes:
      enabled: true
  group:
    readiness:
      include: readinessState,redis ①
```

- ① The readiness probe will combine the application readiness state and the Redis availability.

CONFIGURING LIVENESS AND READINESS PROBES IN KUBERNETES

Kubernetes relies on the health probes (liveness and readiness) to accomplish its tasks as a container orchestrator. For example, when the desired state of an application is with three replicas, Kubernetes ensures there are always three application instances running. If any of them doesn't return a 200 response from the liveness probe, Kubernetes will restart it. When starting or upgrading an application instance, we'd like the process to happen without downtime for the user. Therefore, Kubernetes will not enable an instance in the load balancer until it's ready to accept new requests (i.e., a 200 response from the readiness probe).

Since liveness and readiness information is application-specific, Kubernetes needs the application itself to declare how to retrieve it. Relying on Actuator, Spring Boot applications provide liveness and readiness probes as HTTP endpoints. Let's see how to configure Kubernetes to use those endpoints for the health probes.

In your Catalog Service project (`catalog-service`), open the Deployment manifest (`k8s/deployment.yaml`) and update it with configuration for liveness and readiness probes as follows.

Listing 13.11 Configure liveness and readiness probes for the Catalog Service Deployment (deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  ...
spec:
  ...
  template:
    ...
    spec:
      containers:
        - name: catalog-service
          image: catalog-service
          ...
          livenessProbe: ①
            httpGet: ②
              path: /actuator/health/liveness ③
              port: 9001 ④
            initialDelaySeconds: 10 ⑤
            periodSeconds: 5 ⑥
          readinessProbe: ⑦
            httpGet:
              path: /actuator/health/readiness
              port: 9001
            initialDelaySeconds: 5
            periodSeconds: 15
```

- ① Configuration for the liveness probe.
- ② Uses an HTTP GET request to get the liveness state.
- ③ The endpoint to call for the liveness state.
- ④ The port to use to fetch the liveness state.
- ⑤ An initial delay before starting checking the liveness state.
- ⑥ The frequency for checking the liveness state.
- ⑦ Configuration for the readiness probe.

Both probes can be configured so that Kubernetes will start using them after an initial delay (`initialDelaySeconds`), and you can also define the frequency with which to invoke them (`periodSeconds`). The initial delay should consider that the application will take a few seconds to start, and it depends on the available computational resources. The polling period should not be too long to reduce the time between the application instance entering a faulty state and the platform taking action to self-heal.

WARNING If you run the examples on resource-constraint environments, you might need to adjust the initial delay and the polling frequency to allow the application more time to start and be ready to accept requests. You might need to do the same also when running the examples on Apple Silicon computers as long as AMD64 support is not part of Paketo Buildpacks (you can follow the updates here: <https://github.com/paketo-buildpacks/stacks/issues/51>). That is because ARM64 container images are run on the Apple Silicon computers (AMD64) through a compatibility layer based on Rosetta, which impacts the application startup time.

Go ahead and configure the liveness and readiness probes in the Deployment manifests for all the applications composing the Polar Bookshop system. As a reference, you can look at the source code repository accompanying this book (Chapter13/13-end).

On top of event logs, health information improves the knowledge we can infer about the application's internal state, but it's not enough to achieve complete visibility. The following section will introduce the concept of metrics and how to configure them in Spring Boot.

13.3 Metrics and monitoring with Spring Boot Actuator, Prometheus, and Grafana

To properly monitor, manage, and troubleshoot an application running in production, we need to be able to answer questions like "How much CPU and RAM is the application consuming?", "How many threads are used over time?", "What's the rate of failing requests?". Event logs and health probes can't help us answer those questions. We need something more. We need more data.

Metrics are numeric data about the application measured and aggregated in regular time intervals. We use metrics to track the occurrence of an event (an HTTP request has been received), count items (the number of allocated JVM threads), measure the time taken to perform a task (latency of a database query), or get the current value of a resource (current CPU and RAM consumption). They're all valuable information to understand why the application behaves a certain way. You can monitor metrics and set alerts or notifications for them.

Spring Boot Actuator collects application metrics out-of-the-box leveraging the Micrometer library (<https://micrometer.io>). Micrometer contains instrumentation code to collect valuable metrics from common components in a JVM-based application. It provides a vendor-neutral facade so that you can export the metrics collected from Micrometer using different formats, such as Prometheus/Open Metrics, Humio, Datadog, and VMware Tanzu Observability. Just like SLF4J provides a vendor-neutral facade to logging libraries, Micrometer does the same with metrics exporters.

On top of the default Micrometer instrumentation libraries configured by Spring Boot, you can import additional instrumentation to collect metrics from specific libraries like Resilience4J or even define your own without vendor lock-in.

The most common format for exporting metrics is the one used by Prometheus, which is "an open-source system monitoring and alerting toolkit" (<https://prometheus.io>). Like Loki aggregates and stores event logs, Prometheus does the same with metrics.

In this section, you'll see how to configure metrics in Spring Boot. Then, you'll use Prometheus to aggregate metrics and Grafana to visualize them in dashboards.

13.3.1 Configuring metrics with Spring Boot Actuator and Micrometer

Spring Boot Actuator autoconfigures Micrometer out-of-the-box to collect metrics about the application. One way of exposing such metrics is by enabling the `/actuator/metrics` HTTP endpoint implemented by Actuator. Let's see how to do that.

In your Catalog Service project (`catalog-service`), update the `application.yml` file to expose the metrics endpoint via HTTP.

Listing 13.12 Expose the "metrics" Actuator endpoint (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: health, metrics ①
```

- ① Exposes both health and metrics endpoints.

Ensure the backing services required by Catalog Service are up and running with the following command.

```
$ docker-compose up -d polar-keycloak polar-postgres
```

Then, run the application (`./gradlew bootRun`) and call the `/actuator/metrics` endpoint.

```
$ http :9001/actuator/metrics
```

The result is a collection of metrics you can further explore by adding the name of a metric to the endpoint (e.g., `/actuator/metrics/jvm.memory.used`).

Micrometer provides the instrumentation to generate those metrics, but you might want to export them in a different format. After deciding which monitoring solution you'd like to use to collect and store metrics, you need to add a specific dependency on that tool. In the Grafana observability stack, that tool is Prometheus.

In the Catalog Service project (`catalog-service`), update the `build.gradle` file with a dependency on the Micrometer library that provides the integration with Prometheus. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 13.13 Add dependency for Micrometer Prometheus (build.gradle)

```
dependencies {
    ...
    runtimeOnly 'io.micrometer:micrometer-registry-prometheus'
}
```

Then, update the `application.yml` file to expose the Prometheus Actuator endpoint via HTTP. You can also remove the more generic `metrics` endpoint since we're not going to use it anymore.

Listing 13.14 Expose the "prometheus" Actuator endpoint (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: health, prometheus ①
```

- ① Exposes both health and Prometheus endpoints.

The default strategy used by Prometheus is pull-based, meaning there's a Prometheus instance that scrapes (*pulls*) metrics in regular time intervals from the application via a dedicated endpoint, which is `/actuator/prometheus` in the Spring Boot scenario. Rerun the application (`./gradlew bootRun`) and call the Prometheus endpoint to check the result.

```
$ http :9001/actuator/prometheus
```

The result is the same collection of metrics you got from the `metrics` endpoint, but this time they are exported using a format understood by Prometheus. The following snippet shows an extract of the complete response, highlighting metrics related to the current number of threads.

```
# HELP jvm_threads_states_threads The current number of threads
# TYPE jvm_threads_states_threads gauge
jvm_threads_states_threads{state="terminated",} 0.0
jvm_threads_states_threads{state="blocked",} 0.0
jvm_threads_states_threads{state="waiting",} 13.0
jvm_threads_states_threads{state="timed-waiting",} 7.0
jvm_threads_states_threads{state="new",} 0.0
jvm_threads_states_threads{state="runnable",} 11.0
```

Such a format is based on plain text and is called *Prometheus exposition format*. Given the wide adoption of Prometheus for generating and exporting metrics, its format has been polished and standardized in OpenMetrics (<https://openmetrics.io>), a CNCF-incubating project. Spring Boot

supports both the original Prometheus format (default behavior) and OpenMetrics, depending on the `Accept` header of the HTTP request. If you'd like to get metrics according to the OpenMetrics format, then you need to ask for it explicitly.

```
$ http :9001/actuator/prometheus \
'Accept:application/openmetrics-text; version=1.0.0; charset=utf-8'
```

When you're done analyzing the Prometheus metrics, stop the application (`Ctrl+C`) and all the containers (`docker-compose down`).

NOTE

You might encounter scenarios where you need to collect metrics from ephemeral applications or batch jobs which don't run long enough to be pulled. In that case, Spring Boot lets you adopt a push-based strategy so that the application itself sends metrics to the Prometheus server. The official documentation explains how to configure such behavior (<http://spring.io/projects/spring-boot>).

Spring Boot Actuator relies on the Micrometer instrumentation and provides auto-configuration to generate metrics for various technologies you might use in your applications: JVM, loggers, Spring MVC, Spring WebFlux, RestTemplate, WebClient, data sources, Hibernate, Spring Data, RabbitMQ, and more.

When Spring Cloud Gateway is in the classpath, like in the case of Edge Service, additional metrics are exported regarding the gateway routes. Some libraries like Resilience4J contribute dedicated Micrometer instrumentation through specific dependencies to register additional metrics.

Open the `build.gradle` file in the Edge Service project (`edge-service`) and add the following dependency to include Micrometer instrumentation for Resilience4J. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 13.15 Add dependency for Micrometer Resilience4J (build.gradle)

```
dependencies {
    ...
    runtimeOnly 'io.github.resilience4j:resilience4j-micrometer'
}
```

After configuring Spring Boot to expose metrics, let's see how to configure Prometheus to scrape them and Grafana to visualize them.

13.3.2 Monitoring metrics with Prometheus and Grafana

Similar to Loki, Prometheus collects and stores metrics. It even provides a GUI to visualize them and define alarms, but we'll use Grafana for that since it's a more comprehensive tool.

Metrics are stored as time-series data, containing the timestamp they were registered and, optionally, labels. In Prometheus, labels are key/value pairs that add more information to the metric being recorded. For example, a metric registering the number of threads used by the application could be enhanced with labels qualifying the state of the threads (e.g., blocked, waiting, or idle). Labels help aggregate and query metrics.

Micrometer provides the concept of *tags*, equivalent to Prometheus *labels*. In Spring Boot, you can leverage configuration properties to define common labels for all the metrics produced by the application. For example, it's useful to add an `application` label that tags each metric with the name of the application from which it's produced.

Open the Catalog Service project (`catalog-service`), go to the `application.yml` file, and define a Micrometer tag with the application name, which will result in a label being applied to all metrics. Since the application name is already defined in the `spring.application.name` property, let's reuse that instead of duplicating the value.

Listing 13.16 Tag all metrics with the application name (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: health, prometheus
    endpoint:
      health:
        show-details: always
        show-components: always
        probes:
          enabled: true
  metrics:
    tags:
      application: ${spring.application.name} ①
```

- ① Adds a Micrometer common tag with the application name. It results in a Prometheus label applied to all metrics.

With this change, all metrics will have an `application` label with the application name, which is very useful when querying metrics and building dashboards to visualize them in Grafana.

```
jvm_threads_states_threads{application="catalog-service",
  ➔state="waiting",} 13.0
```

You have already encountered Grafana when working with logs. Similar to how you browsed logs using Loki as a data source for Grafana, you can query metrics using Prometheus as a data source. Furthermore, you can use the metrics stored by Prometheus to define dashboards, graphically visualize data, and set alarms or notifications when certain metrics return known critical values. For example, when the rate of failing HTTP requests per minute goes above a certain threshold, you might want to get an alarm or a notification and act on it. Figure 13.5 illustrates the monitoring architecture.

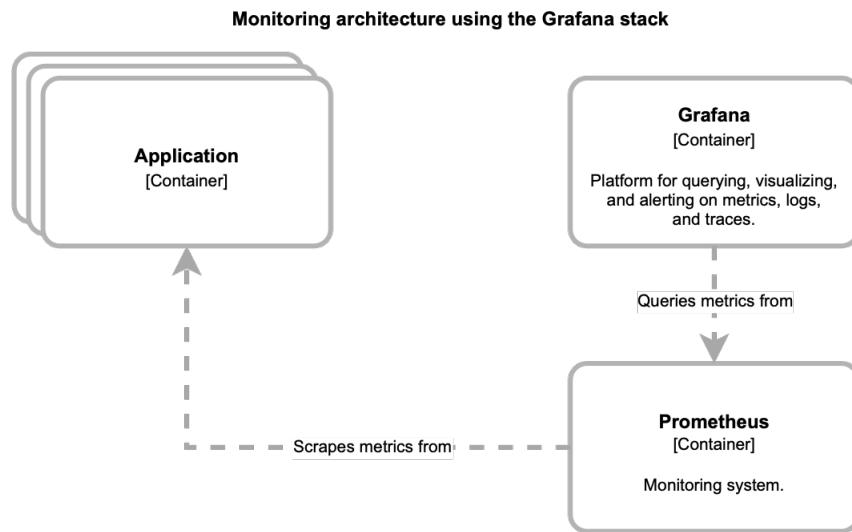


Figure 13.5 Monitoring architecture for cloud native applications based on the Grafana stack.

In your Polar Deployment project (`polar-deployment`), update the Docker Compose configuration (`docker/docker-compose.yml`) to include Prometheus. Grafana is already configured to use Prometheus as a data source using the configuration files you imported earlier into your project from `Chapter13/13-end/polar-deployment/docker/observability`.

Listing 13.17 Define Prometheus container for collecting metrics (`docker-compose.yml`)

```

version: "3.8"
services:
  ...
  grafana:
    image: grafana/grafana:8.4.4
    container_name: grafana
    depends_on:
      - loki
      - prometheus ①
    ...
  ...
  prometheus:
    image: prom/prometheus:v2.34.0
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      ②
      - ./observability/prometheus/prometheus.yml:/etc/prometheus/
      prometheus.yml

```

- ① Ensures Prometheus is started before Grafana.
- ② Volumes are used to load configuration for Prometheus scraping.

Unlike Loki, we don't need a dedicated component to collect metrics from the applications. The Prometheus Server container can both collect and store metrics.

Next, open a Terminal window, navigate to the folder where you keep your Docker Compose file

(polar-deployment/docker), and run the complete monitoring stack with the following command.

```
$ docker-compose up -d grafana
```

The Prometheus container is configured to poll metrics every 2 seconds from all the Spring Boot applications in Polar Bookshop when they run as containers. Package Catalog Service as a container image (`./gradlew bootBuildImage`), then run it from Docker Compose.

```
$ docker-compose up -d catalog-service
```

Send a few requests to Catalog Service (`http :9001/books`), then open a browser window and go to Grafana at <http://localhost:3000> (user/password). In the "Explore" section, you can query metrics similarly to how you browsed logs. Choose "Prometheus" as the data source, select "Last 5 minutes" from the time drop-down menu, and query the metrics related to the JVM memory used by the application as follows (figure 13.6).

```
jvm_memory_used_bytes{application="catalog-service"}
```

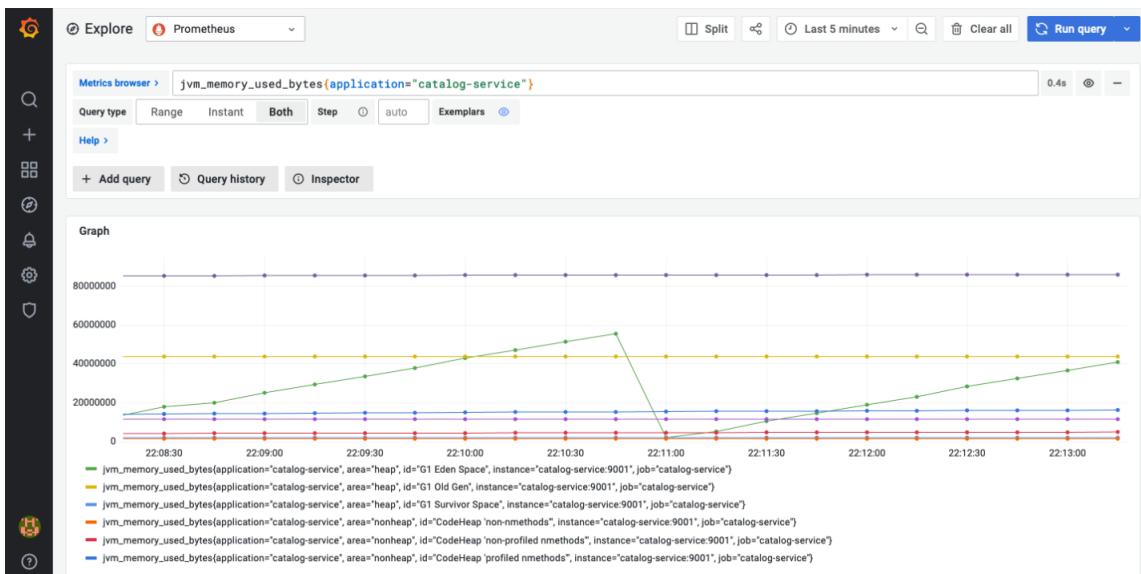


Figure 13.6 In Grafana, you can browse and query metrics aggregated and stored by Prometheus.

The metrics data can be used to draw dashboards for monitoring different application aspects. Select *Dashboards > Manage* from the left menu and explore the dashboards I have included in Grafana, grouped within the "Application" folder. For example, open the "JVM Dashboard" (figure 13.7). The dashboard visualizes different metrics regarding the JVM where Spring Boot applications run, such as CPU usage, heap memory, non-heap memory, garbage collections, and threads.

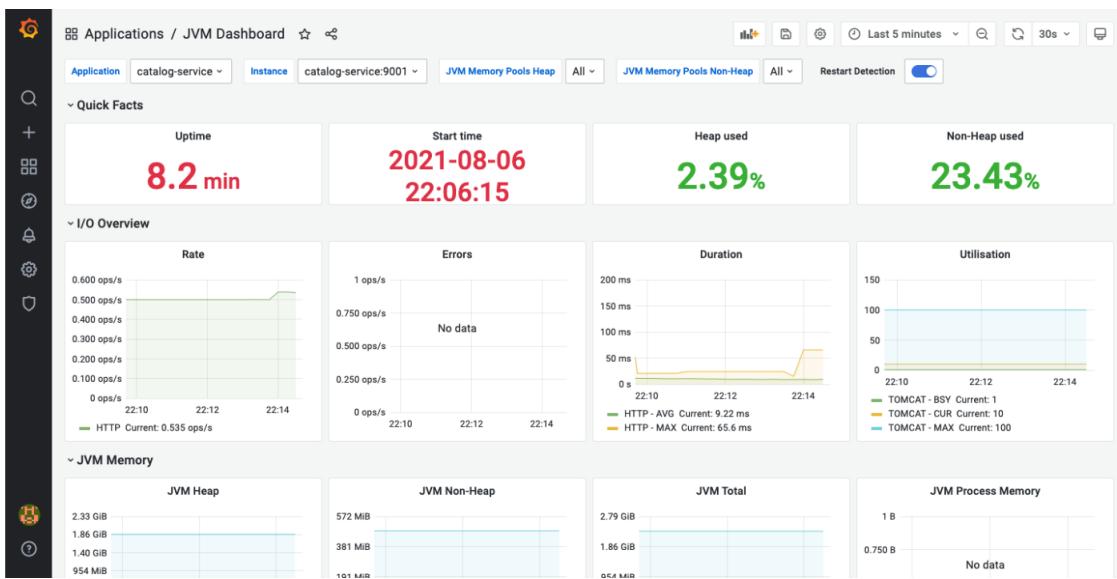


Figure 13.7 In Grafana, dashboards can be used to visualize Prometheus metrics.

On the "Dashboards" page, go ahead and explore the other dashboards I have configured for you to get more visibility into the Polar Bookshop applications. Each dashboard is enhanced with additional information on its goal and how to use it.

When you're done checking the application metrics in Grafana, stop all containers (`docker-compose down`).

13.3.3 Configuring Prometheus metrics in Kubernetes

When running applications in Kubernetes, we can use dedicated annotations to mark which containers the Prometheus server should scrape and inform it about the HTTP endpoint and port number to call.

You'll have the chance to test this setup later in the book, where we'll deploy the full Grafana observability stack in a production Kubernetes cluster. For now, let's prepare the Deployment manifests for all the Spring Boot applications in Polar Bookshop. For example, the following listing shows how to change the Catalog Service manifest (`catalog-service/k8s/deployment.yaml`).

Listing 13.18 Annotate Catalog Service for Prometheus metrics scraping in Kubernetes (deployment.yml)

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  labels:
    app: catalog-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: catalog-service
  template:
    metadata:
      labels:
        app: catalog-service
    annotations:
      prometheus.io/scrape : "true" ①
      prometheus.io/path : /actuator/prometheus ②
      prometheus.io/port : "9001" ③
...

```

- ① Signals that Prometheus should scrape containers in this Pod.
- ② Signals the HTTP endpoint exposing Prometheus metrics.
- ③ Signals the port number where the metrics endpoint is available.

Annotations in Kubernetes manifests should be of type `String`, which is why quotes are needed in case of values that could be mistakenly parsed as numbers or boolean.

Go ahead and configure metrics and Prometheus for all the remaining applications in the Polar Bookshop system, including the configuration for the Kubernetes manifests. As a reference, you can look at the source code repository accompanying this book ([Chapter13/13-end](#)).

The next section will cover another type of telemetry we need to monitor applications and make them observable: traces.

13.4 Distributed tracing with OpenTelemetry and Tempo

Event logs, health probes, and metrics provide a wide variety of valuable data to infer the internal state of an application. However, none of them consider that cloud native applications are distributed systems. A user request is likely processed by multiple applications, but so far, we have no way to correlate data across a single application boundary.

A simple way to solve that problem could be to generate an identifier for each request at the edge of the system (a *correlation id*), use it in event logs, and pass it over to the other services involved. Then, by using that correlation id, we could fetch all log messages related to the same transaction from multiple applications.

Expanding that idea further, we get to *distributed tracing*, a technique to track requests as they flow through a distributed system, let us localize where errors occur, and troubleshoot performance issues.

- A **trace** represents the activities associated with a request or a transaction, identified uniquely by a *trace id*. It's composed of one or more spans across one or more services.
- Each step of the request processing is called a **span**, characterized by a start and end timestamp and identified uniquely by the pair *trace id* and *span id*.
- **Tags** are metadata that provide additional information regarding the span context, such as the request URI, the username for the currently logged-in user, or the tenant identifier.

Let's consider an example. In Polar Bookshop, you can fetch books through the gateway (Edge Service), and the request is then forwarded to Catalog Service. The trace related to handling such a request would involve the two applications and at least three spans:

- The first span is the step performed by Edge Service to accept the initial HTTP request.
- The second span is the step performed by Edge Service to route the request to Catalog Service.
- The third span is the step performed by Catalog Service to handle the routed request.

There are multiple choices for a distributed tracing system. First, we must choose the format and protocol to generate and propagate traces. We'll use OpenTelemetry (also called *OTel*), a CNCF-incubating project that is quickly becoming the de-facto standard for distributed tracing and aims at unifying the collection of telemetry data (<https://opentelemetry.io>).

Next, we need to choose whether to use OpenTelemetry directly (with the OpenTelemetry Java instrumentation) or rely on a facade that instruments the code in a vendor-neutral way and integrates with different distributed tracing systems (for example, Spring Cloud Sleuth). We'll go with the first option.

Once the applications are instrumented for distributed tracing, we need a tool to collect and store traces. In the Grafana observability stack, the distributed tracing backend of choice is Tempo (<https://grafana.com/oss/tempo>), a project that "lets you scale tracing as far as possible with minimal operational cost and less complexity than ever before". Unlike the way we used Prometheus, Tempo follows a push-based strategy where the application itself pushes data to the distributed tracing backend.

This section will show you how to complete the Grafana observability setup with Tempo and use it to collect and store traces. Then, I'll show you how to use the OpenTelemetry Java instrumentation in your Spring Boot applications to generate and send traces to Tempo. Finally, you'll see how to query traces from Grafana.

SIDE BAR**OpenTelemetry, Spring Cloud Sleuth, and Micrometer Tracing**

A few standards have emerged to implement distributed tracing and define guidelines for how to generate and propagate traces and spans. OpenZipkin is the more mature project (<https://zipkin.io>). OpenTracing and OpenCensus are more recent projects that tried to standardize how to instrument application code to support distributed tracing. They are both deprecated now since they joined forces to work on the ultimate framework to "instrument, generate, collect, and export telemetry data (metrics, logs, and traces)": OpenTelemetry. Tempo supports all those options.

Spring Cloud Sleuth (<https://spring.io/projects/spring-cloud-sleuth>) is a project that provides auto-configuration for distributed tracing in Spring Boot applications. It takes care of instrumenting commonly used libraries in Spring applications and provides an abstraction layer on top of specific distributed tracing libraries. OpenZipkin is the default choice.

In this book, I decided to show you how to use the OpenTelemetry Java instrumentation directly for two main reasons. First, support for OpenTelemetry in Spring Cloud Sleuth is still experimental and not ready for production at the time of writing (<https://github.com/spring-projects-experimental/spring-cloud-sleuth-otel>).

Second, Spring Cloud Sleuth will not be developed further once Spring Framework 6 and Spring Boot 3 are released. The Spring project donated the Sleuth core framework to Micrometer and created a new Micrometer Tracing subproject aiming to provide a vendor-neutral facade for traces similar to what Micrometer already does for metrics. Micrometer Tracing will provide support for OpenZipkin and OpenTelemetry. Based on Micrometer Tracing, code instrumentation will become a core aspect of all Spring libraries as part of the Spring Observability initiative.

13.4.1 Managing traces with Tempo and Grafana

A distributed tracing backend is responsible for aggregating, storing, and making traces searchable. Tempo is the solution in the Grafana observability stack. Figure 13.8 illustrates the tracing architecture.

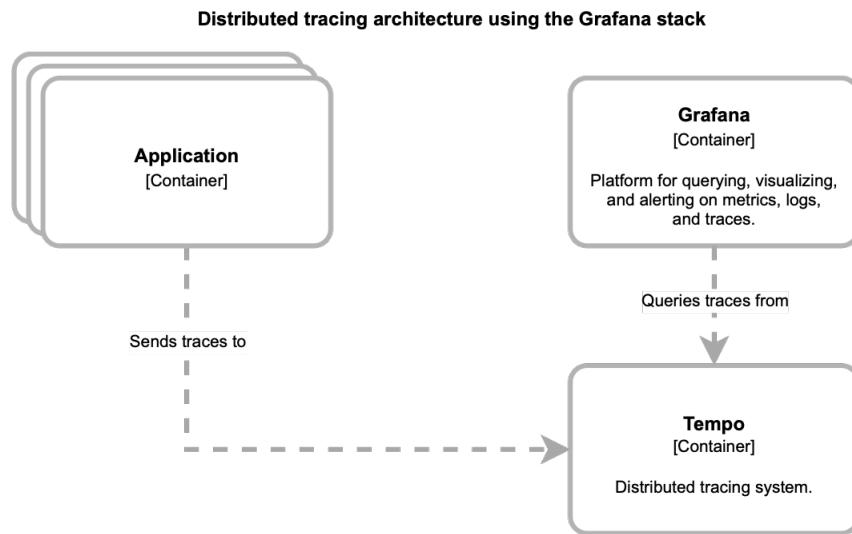


Figure 13.8 Distributed tracing architecture for cloud native applications based on the Grafana stack.

NOTE

Most vendors support OpenTelemetry, so you can easily swap your distributed tracing backend without changing anything in your applications. For example, instead of Tempo, you could send traces to other platforms like Honeycomb, Lightstep, or VMware Tanzu Observability.

First, let's update the Docker Compose file for Polar Bookshop to include Tempo (`polar-deployment/docker/docker-compose.yml`). Grafana is already configured to use Tempo as a data source using the configuration files you imported earlier into your project from `Chapter13/13-end/polar-deployment/docker/observability`.

Listing 13.19 Define a Tempo container for collecting and storing traces (docker-compose.yml)

```

version: "3.8"
services:
  ...
  grafana:
    image: grafana/grafana:8.4.4
    container_name: grafana
    depends_on:
      - loki
      - prometheus
      - tempo ①
    ...
  tempo:
    image: grafana/tempo:1.3.2
    container_name: tempo
    command: -config.file /etc/tempo-config.yml ②
    ports:
      - "4317:4317" ③
    volumes:
      - ./observability/tempo/tempo.yml:/etc/tempo-config.yml ④

```

- ① Ensures Tempo is started before Grafana.
- ② Loads the custom configuration during the startup phase.
- ③ Port to accept traces using the OpenTelemetry protocol over gRPC.
- ④ Volumes are used to load configuration for Tempo.

Next, let's run the full Grafana observability stack on Docker. Open a Terminal window, navigate to the folder where you keep your Docker Compose file, and run the following command.

```
$ docker-compose up -d grafana
```

Tempo is now ready to accept OpenTelemetry traces over gRPC on port 4317. In the next section, we'll see how to update a Spring Boot application to generate traces and send them over to Tempo.

13.4.2 Configuring tracing in Spring Boot with OpenTelemetry

The OpenTelemetry project includes instrumentation taking care of generating traces and spans for the most common Java libraries, including Spring, Tomcat, Netty, Reactor, JDBC, Hibernate, and Logback. The OpenTelemetry Java Agent is a JAR artifact provided by the project that can be attached to any Java application. It injects the necessary bytecode dynamically to capture traces and spans from all those libraries and export them in different formats without having to change your Java source code explicitly.

Java agents are often provided to the application at runtime from the outside. For better dependency management capabilities, in this case, I prefer using Gradle (or Maven) to include the agent JAR file in the final application artifact. Let's see how.

Open your Catalog Service project (`catalog-service`). Then, add a dependency on the OpenTelemetry Java Agent in your `build.gradle` file. Remember to refresh/reimport the Gradle dependencies after the new addition.

Listing 13.20 Add dependency for OpenTelemetry Java Agent in Catalog Service (build.gradle)

```
ext {
    ...
    set('otelVersion', "1.12.1")      ①
}

dependencies {
    ...
    runtimeOnly "io.opentelemetry.javaagent:opentelemetry-javaagent:${otelVersion}" ②
}
```

- ① The OpenTelemetry version
- ② The OpenTelemetry agent instrumenting the Java code dynamically via bytecode.

Besides instrumenting the Java code to capture traces, the OpenTelemetry Java Agent also integrates with SLF4J (and its implementation). It provides trace and span identifiers as contextual information that can be injected into log messages through the MDC abstraction provided by SLF4J. That makes it extremely simple to navigate from log messages to traces and vice versa, achieving better visibility into the application than querying telemetry in isolation.

Let's expand the default log format used by Spring Boot and add the following contextual information.

- application name (value from the `spring.application.name` property we configured for all applications);
- trace identifier (value from the `trace_id` field populated by the OpenTelemetry agent, when enabled);
- span identifier (value from the `span_id` field populated by the OpenTelemetry agent, when enabled).

In your Catalog Service project, open the `application.yml` file, and add the three new pieces of information next to the log level (represented by `%5p`) following the Logback syntax. That is the same format also used by Spring Cloud Sleuth.

Listing 13.21 Add contextual information to log messages, next to the log level field (application.yml)

```
logging:
  pattern:
    level: "%5p [%${spring.application.name},%X{trace_id},%X{span_id}]" ①
```

- ① Includes application name, trace ID, and span ID next to the log level (`%5p`).

Next, open a Terminal window, navigate to the Catalog Service root folder, and run `./gradlew bootBuildImage` to package the application as a container image.

The final step is configuring and enabling the OpenTelemetry Java Agent. For simplicity, we'll enable OpenTelemetry only when running applications in containers and rely on environment variables to configure it.

We need three pieces of configuration to enable tracing successfully.

- Instruct the JVM to load the OpenTelemetry Java agent. We can do that via the `JAVA_TOOL_OPTIONS` standard environment variable supported by OpenJDK to provide additional configuration to the JVM.
- Use the application name to tag and categorize traces. We'll use the

OTEL_SERVICE_NAME environment variable supported by the OpenTelemetry Java agent.

- Define the URL of the distributed tracing backend. In our case, it's Tempo on port 4317 and can be configured via the OTEL_EXPORTER_OTLP_ENDPOINT environment variable supported by the OpenTelemetry Java agent. By default, traces are sent over gRPC.

Go to your Polar Deployment project (`polar-deployment`) and open the Docker Compose file (`docker/docker-compose.yml`). Then, add the necessary configuration to Catalog Service to support tracing.

Listing 13.22 Define OpenTelemetry for the Catalog Service container (`docker-compose.yml`)

```
version: "3.8"
services:
  ...
  catalog-service:
    depends_on:
      - fluent-bit
      - polar-keycloak
      - polar-postgres
      - tempo ①
    image: "catalog-service"
    container_name: "catalog-service"
    ports:
      - 9001:9001
      - 8001:8001
    environment:
      - JAVA_TOOL_OPTIONS=-javaagent:/workspace/BOOT-INF/lib/
        opentelemetry-javaagent-1.12.1.jar ②
      - OTEL_SERVICE_NAME=catalog-service ③
      - OTEL_EXPORTER_OTLP_ENDPOINT=http://tempo:4317 ④
    ...
  ...
```

- ① Ensures Tempo is started before Catalog Service.
- ② Instructs the JVM to run the OpenTelemetry Java agent from the path where Cloud Native Buildpacks placed the application dependencies.
- ③ The name of the application, used to tag the traces produced by Catalog Service.
- ④ The URL of the distributed tracing backend supporting the OpenTelemetry protocol (OTLP).

Finally, from the same folder, run Catalog Service as a container.

```
$ docker-compose up -d catalog-service
```

Once the application is up and running, send a few requests to trigger the generation of some logs and traces about your HTTP requests.

```
$ http :9001/books
```

Then, check the logs from the container (`docker logs catalog-service`). You can see that each log message has now a new section containing the application name and, when available,

the trace and span identifiers.

```
[catalog-service,d9e61c8cf853fe7fdf953422c5ff567a,eef9e08caeae9e32a]
```

Distributed tracing helps follow a request through multiple services, so we need another application to test if it works correctly. Go ahead and make the same changes to Edge Service to support OpenTelemetry. Then, run the application as a container from your Docker Compose file.

```
$ docker-compose up -d edge-service
```

Once again, send a few requests to trigger the generation of some logs and traces about your HTTP requests. This time, you should go through the gateway.

```
$ http :9000/books
```

Using the trace id logged by Catalog Service, we can retrieve (*correlate*) all the steps involved in processing the HTTP request to the `/books` endpoint started in Edge Service. Being able to navigate from logs to traces (and the other way around) is extremely useful to get more visibility into all the steps involved in processing a request throughout a distributed system. Let's see how it works in the Grafana stack.

Open a browser window, go to Grafana (<http://localhost:3000>), and log in with the credentials configured in Docker Compose (user/password). On the "Explore" page, check the logs for Catalog Service (`{container_name="/catalog-service"}`), similar to what we did earlier. Next, click on the most recent log message to get more details. You'll see a "Tempo" button next to the trace identifier associated with that log message. If you click that, Grafana redirects you to the related trace using data from Tempo, all in the same view (figure 13.9).

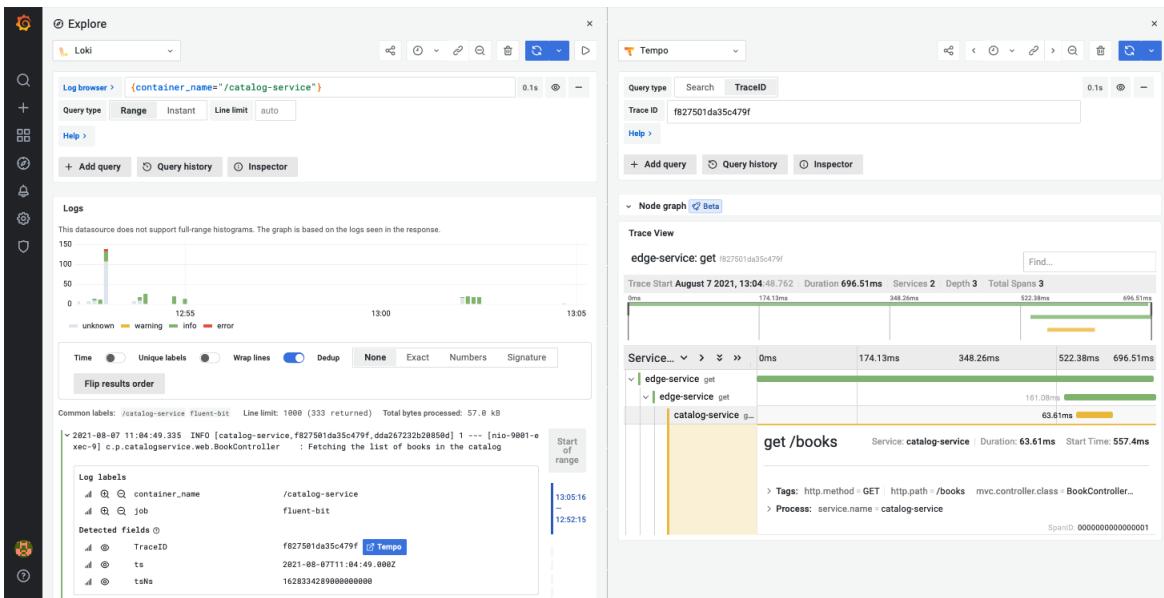


Figure 13.9 In Grafana, you can navigate from logs (Loki) to traces (Tempo) using the trace id included in logs.

When you’re done inspecting logs and traces, stop all containers (`docker-compose down`). Before moving on, go ahead and configure OpenTelemetry for all the remaining applications in the Polar Bookshop system. As a reference, you can look at the source code repository accompanying this book (`chapter13/13-end`).

So far, we have worked with the three main types of telemetry data: logs, metrics, and traces. We also enabled health endpoints to provide additional information regarding the application status. The following section will cover how to retrieve even more information from the applications and achieve better visibility into their operations.

13.5 Application management and monitoring with Spring Boot Actuator

In the previous sections, I’ve shown you the primary telemetry data all cloud native applications should provide to achieve better observability. This final section will be dedicated to some specific information you can retrieve from applications to enhance further the knowledge inferred about their operations.

Spring Boot Actuator provides many features to make your applications production-ready. You have already learned about health and metrics endpoints, but there are more. Table 13.1 lists some of the most useful management and monitoring endpoints implemented by Actuator.

Table 13.1 Some of the most useful management/monitoring endpoints exposed by Spring Boot Actuator.

Endpoint	Description
/beans	Shows a list of all the Spring beans managed by the application.
/configprops	Shows a list of all the @ConfigurationProperties-annotated beans.
/env	Shows a list of all the properties available to the Spring Environment.
/flyway	Shows a list of all the migrations run by Flyway and their status.
/health	Shows information about the application's health.
/heapdump	Returns a heap dump file.
/info	Shows arbitrary application information.
/loggers	Shows the configuration of all the loggers in the application and allows you to modify them.
/metrics	Returns metrics about the application.
/mappings	Shows a list of all the paths defined in web controllers.
/prometheus	Returns metrics about the application either in Prometheus or OpenMetrics format.
/sessions	Shows a list of all the active sessions managed by Spring Session and allows you to delete them.
/threaddump	Returns a thread dump in JSON format.

This section will show you how to use some of them.

13.5.1 Monitoring Flyway migrations in Spring Boot

In chapters 5 and 8, you saw how to version-control your database schemas using Flyway migrations and integrate them with Spring Boot, both in an imperative and reactive stack. Flyway keeps the history of all the migrations run on the application in a dedicated table in the database. It would be convenient to extract such information and monitor it to be alerted if any migration should fail.

Spring Boot Actuator provides a dedicated endpoint (`/actuator/flyway`) to display information about all the migrations run by Flyway, including status, date, type, and version. As you learned in the previous sections, you can enable new HTTP endpoints implemented by Actuator through the property `management.endpoints.web.exposure.include`. Let's see that in action.

NOTE

If you use Liquibase instead of Flyway, Spring Boot Actuator provides a `/actuator/liquibase` endpoint.

Open the Catalog Service project (`catalog-service`), go to the `application.yml` file, and configure the Flyway endpoint to be exposed over HTTP by Spring Boot Actuator.

Listing 13.23 Expose the "flyway" Actuator endpoint (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: flyway, health, prometheus ①
```

- ① Adds Flyway to the list of Actuator endpoints exposed over HTTP.

Then, run the backing services required by Catalog Service as a container. From your Docker Compose file, execute the following command.

```
$ docker-compose up -d polar-keycloak polar-postgres
```

Next, run Catalog Service (`./gradlew bootRun`) and call the Flyway endpoint.

```
$ http :9001/actuator/flyway
```

The result is a JSON file containing the list of all migrations run by Flyway and their details. The following snippet shows an extract of the complete response.

```
{
  "contexts": {
    "catalog-service": {
      "flywayBeans": {
        "flyway": {
          "migrations": [
            {
              "checksum": -567578088, ①
              "description": "Initial schema", ②
              "executionTime": 66,
              "installedBy": "user",
              "installedOn": "2022-03-19T17:06:54Z", ③
              "installedRank": 1,
              "script": "V1__Initial_schema.sql", ④
              "state": "SUCCESS", ⑤
              "type": "SQL", ⑥
              "version": "1" ⑦
            },
            ...
          ]
        }
      }
    }
}
```

- ① The checksum of the migration script, used to ensure the file has not been changed.
- ② Description of the migration.
- ③ When the migration was performed.
- ④ The name of the script containing the migration code.
- ⑤ The state of the migration execution.

- ⑥ The type of migration (SQL or Java).
- ⑦ The migration version (as defined in the script file name).

13.5.2 Exposing application information

Among all the endpoints implemented by Spring Boot Actuator, `/actuator/info` is the most peculiar one since it doesn't return any data. Instead, it's up to you to define what data you reckon is useful.

One way to contribute data for the endpoint is through configuration properties. For example, go to your Catalog Service project (`catalog-service`), open the `application.yml` file, and add the following property to include the name of the system of which Catalog Service is part. You also need to enable the `info` endpoint to be exposed through HTTP (similar to what we did with the other endpoints) and enable the `env` contributor responsible for parsing all the properties with the `info.` prefix.

Listing 13.24 Expose and configure the "info" Actuator endpoint (application.yml)

```
info:
  system: Polar Bookshop ①

management:
  endpoints:
    web:
      exposure:
        include: flyway, health, info, prometheus ②
  info:
    env:
      enabled: true ③
```

- ① Any property starting with the "info" prefix will be returned by the info endpoint.
- ② Adds "info" to the list of Actuator endpoints to be exposed over HTTP.
- ③ Enables environmental info fetched from "info." properties.

You can also include information generated automatically by Gradle or Maven regarding the application build or the last git commit. Let's see how to add details about the application build configuration. In your Catalog Service project, go to the `build.gradle` file and configure the `springBoot` task to generate build information that will be parsed into a `BuildProperties` object and included in the result from the `info` endpoint.

Listing 13.25 Configure Spring Boot to generate build information and include it in the application (build.gradle)

```
springBoot {
  buildInfo() ①
}
```

- ① Stores build information in a "META-INF/build-info.properties" file parsed by a `BuildProperties` object.

Let's test it out. Rerun Catalog Service (`./gradlew bootRun`). Then, invoke the info endpoint.

```
$ http :9001/actuator/info
```

The result is a JSON object containing build information and the custom `info.system` property we defined explicitly.

```
{
  "build": {
    "artifact": "catalog-service",
    "group": "com.polarbookshop",
    "name": "catalog-service",
    "time": "2021-08-06T12:56:25.035Z",
    "version": "0.0.1-SNAPSHOT"
  },
  "system": "Polar Bookshop"
}
```

Additional information you can expose is about the operating system and the Java version in use. Both of them can be enabled via configuration properties. Let's update the `application.yml` file for the Catalog Service project as follows.

Listing 13.26 Add Java and OS details to the "info" Actuator endpoint (application.yml)

```
management:
  ...
info:
  env:
    enabled: true
  java:
    enabled: true ①
  os:
    enabled: true ②
```

- ① Enables Java information in the "info" endpoint.
- ② Enables OS information in the "info" endpoint.

Let's test it out. Rerun Catalog Service (`./gradlew bootRun`). Then, invoke the info endpoint.

```
$ http :9001/actuator/info
```

The result now includes additional information about the Java version and operating system in use, which will be different depending on where you run the application.

```
{
  ...
  "java": {
    "version": "17.0.3",
    "vendor": {
      "name": "Eclipse Adoptium",
      "version": "Temurin-17.0.3+7"
    },
    "runtime": {
      "name": "OpenJDK Runtime Environment",
      "version": "17.0.3+7"
    },
    "jvm": {
      "name": "OpenJDK 64-Bit Server VM",
      "vendor": "Eclipse Adoptium",
      "version": "17.0.3+7"
    }
  },
  "os": {
    "name": "Mac OS X",
    "version": "12.3.1",
    "arch": "aarch64"
  }
}
```

13.5.3 Generating and analyzing heap dumps

Among the most annoying errors to debug in Java applications, memory leaks are probably the first ones that come to mind. Monitoring tools should alert you when a memory leak pattern is detected, usually inferred if the JVM heap usage metric keeps increasing over time. If we don't catch the memory leak in advance, the application will throw the dreaded `OutOfMemoryError` and crash.

Once we suspect an application might suffer from a memory leak, we must find which objects are held in memory and block the garbage collection. There are different ways to proceed with finding problematic objects. For example, you could enable the Java Flight Recorder or attach a profiler like jProfiler to the running application. Another way is to take a snapshot of all the Java objects in the JVM heap memory (a *heap dump*) and analyze it with a specialized tool to find the root cause for the memory leak.

Spring Boot Actuator provides a convenient endpoint (`/actuator/heapdump`) we can call to generate a heap dump. Let's see that in action. Go to your Catalog Service project (`catalog-service`), open the `application.yml` file, and configure Actuator to expose the heap dump endpoint.

Listing 13.27 Expose the "heapdump" Actuator endpoint (application.yml)

```
management:
  endpoints:
    web:
      exposure:
        include: flyway, health, heapdump, info, prometheus ①
```

- ① Adds "heapdump" to the list of Actuator endpoints to be exposed over HTTP.

Next, build and run Catalog Service (`./gradlew bootRun`). Finally, invoke the heap dump endpoint.

```
$ http --download :9001/actuator/heapdump
```

The command will save a "heapdump.bin" file in the current directory. You can then open it in a dedicated tool for heap analysis like VisualVM (<https://visualvm.github.io>) or JDK Mission Control (<https://adoptopenjdk.net/jmc.html>). Figure 13.10 shows an example of heap analysis in VisualVM.

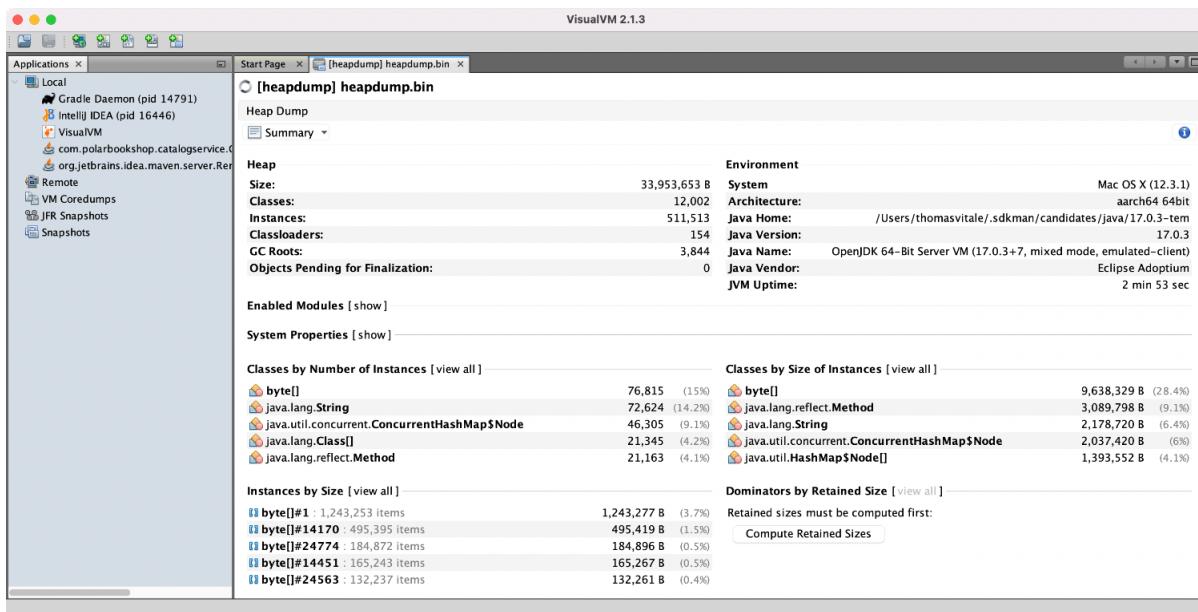


Figure 13.10 VisualVM provides tools to analyse a Java application heap dump.

Finally, stop the application process (`Ctrl+C`) and all containers (`docker-compose down`).

I encourage you to check the Spring Boot Actuator official documentation, try out all the supported endpoints, and make the applications of the Polar Bookshop system more observable. For inspiration, refer to the source code repository accompanying the book to see which endpoints I have enabled on each application (Chapter13/13-end). They're powerful tools you'll likely find helpful and convenient in real-world applications running in production.

13.6 Summary

- Observability is a property of cloud native applications that measures how well we can infer the internal state of an application from its outputs.
- Monitoring is about controlling known faulty states. Observability goes beyond that and permits us to ask questions about the unknown.
- Logs (or event logs) are discrete records of something that happened over time in a software application.
- Spring Boot supports logging through SLF4J, which provides a facade over the most common logging libraries.
- By default, logs are printed through the standard output as recommended by the 15-Factors methodology.
- Using the Grafana observability stack, Fluent Bit collects logs produced by all applications and forwards them to Loki, which stores them and makes them searchable. Then, you can use Grafana to navigate logs.
- Applications should expose health endpoints to check their status.
- Spring Boot Actuator exposes an overall health endpoint showing the status of the application and all the components or services it might use. It also provides specialized endpoints to be used as liveness and readiness probes by Kubernetes.
- When the liveness probe is down, it means the application entered an unrecoverable faulty state, so Kubernetes will try to restart it.
- When the readiness probe is down, the application is not ready to handle requests, so Kubernetes will stop any traffic directed to that instance.
- Metrics are numeric data about the application measured in regular time intervals.
- Spring Boot Actuator leverages the Micrometer facade to instrument the Java code, generate metrics, and expose them through a dedicated endpoint.
- When the Prometheus client is on the classpath, Spring Boot can expose metrics also in the Prometheus or OpenMetrics format.
- Using the Grafana observability stack, Prometheus aggregates and stores metrics from all applications. Then, you can use Grafana to query metrics, design dashboards, and set alerts.
- Distributed tracing, a technique to track requests as they flow through a distributed system, lets us localize where errors occur in a distributed system and troubleshoot performance issues.
- Traces are characterized by a trace id and are composed of multiple spans, representing steps in a transaction.
- The OpenTelemetry project includes APIs and instrumentation, taking care of generating traces and spans for the most common Java libraries.
- The OpenTelemetry Java Agent is a JAR artifact provided by the project that can be attached to any Java application. It injects the necessary bytecode dynamically to capture traces and spans from all those libraries and export them in different formats without having to change your Java source code explicitly.
- Using the Grafana observability stack, Tempo aggregates and stores metrics from all applications. Then, you can use Grafana to query traces and correlate them with logs.
- Spring Boot Actuator provides management and monitoring endpoints to fulfill any requirement you might have to make your applications production-ready.

14

Configuration and secrets management

This chapter covers

- Configuring applications on Kubernetes
- Using ConfigMaps and Secrets in Kubernetes
- Managing deployments and configuration with Kustomize

Releasing applications to production involves two important aspects: an executable artifact and its configuration. The executable artifact could be a JAR file or a container image. The previous chapters covered several principles, patterns, and tools to build applications that are loosely coupled, resilient, scalable, secure, and observable. You saw how to package applications as executable JAR artifacts or container images. And I guided you through the implementation of the commit stage of a deployment pipeline, which ultimately produces a release candidate.

The other aspect of being ready for production is configuration. Chapter 4 introduced the importance of externalized configuration for cloud native applications and covered several techniques for configuring Spring Boot applications. This chapter will continue that discussion in preparation for deploying an entire cloud native system to a Kubernetes production environment.

First, I'll describe a few options for configuring Spring Boot applications on Kubernetes and describe what's missing for using Spring Cloud Config in production. Then, you'll learn how to use ConfigMaps and Secrets, a native mechanism for handling configuration on Kubernetes. As part of the discussion, you'll get to know Spring Cloud Kubernetes and its primary use cases. Finally, I'll expand on configuration and secrets management for production workloads on Kubernetes, and you'll learn how to implement that using Kustomize.

NOTE

The source code for the examples in this chapter is available in the Chapter14/14-begin and Chapter14/14-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

14.1 Configuring applications on Kubernetes

According to the 15-Factor methodology, configuration is anything that changes between deployment environments. We started working with configuration in chapter 4 and since then used different configuration strategies.

- **Property files packaged with the application.** They can act as specifications of what configuration data the application supports and are useful to define sensible default values, mainly oriented to the development environment.
- **Environment variables.** Supported by any operating system, so they are great for portability. Useful to define configuration data depending on the infrastructure/platform where the application is deployed, such as active profiles, hostnames, service names, and port numbers. We used them in Docker and Kubernetes.
- **Configuration service.** It provides configuration data persistence, auditing, and accountability. Useful to define configuration data specific to the application, such as feature flags, thread pools, connection pools, timeouts, and URLs to third-party services. We adopted this strategy with Spring Cloud Config.

Those three strategies are generic enough that we can use them to configure applications in any cloud environment and service model (CaaS, PaaS, FaaS). When it comes to Kubernetes, there's an additional configuration strategy that is provided natively by the platform: ConfigMaps and Secrets.

They are a very convenient way to define configuration data depending on the infrastructure and platform where the application is deployed: service names (defined by Kubernetes Service objects), credentials/certificates to access other services running on the platform, graceful shutdown, logging, and monitoring. You could use ConfigMaps and Secrets to complement or completely replace what a configuration service does. What to choose depends on the context. In any case, Spring Boot provides native support for all those options.

For the Polar Bookshop system, we'll use ConfigMaps and Secrets instead of the Config Service to configure applications in Kubernetes environments. Still, all the work we've done so far on Config Service would make it straightforward to include it in the overall deployment of Polar Bookshop on Kubernetes. In this section, I'll share with you some final considerations to make Config Service production-ready, in case you'd like to expand the examples and include it in the final deployment in production.

14.1.1 Securing the configuration server with Spring Security

In previous chapters, we spent quite some time ensuring a high-security level for the Spring Boot applications in Polar Bookshop. However, Config Service was not one of them, and it's still unprotected. Even if it's a config server, it's still a Spring Boot application at its heart. As such, we can secure it using any of the strategies provided by Spring Security.

Config Service is accessed over HTTP by the other Spring Boot applications in the architecture. Before using it in production, we must ensure that only authenticated and authorized parties can retrieve configuration data. One option would be to use the OAuth2 Client Credentials flow to secure the interactions between Config Service and applications based on an Access Token. It's an OAuth2 flow specific for protecting service-to-service interactions.

Assuming that applications will communicate over HTTPS, the *HTTP Basic* authentication strategy would be another viable option. When using this strategy, applications can be configured with the username and password via the properties exposed by Spring Cloud Config Client: `spring.cloud.config.username` and `spring.cloud.config.password`. For more information, refer to the official documentation for Spring Security (<https://spring.io/projects/spring-security>) and Spring Cloud Config (<https://spring.io/projects/spring-cloud-config>).

14.1.2 Refreshing configuration at runtime with Spring Cloud Bus

Imagine you have deployed your Spring Boot applications in a cloud environment like Kubernetes. During the startup phase, each application loaded its configuration from an external config server. At some point, you decide to make changes in the config repo. How can you make the applications aware of the configuration change and have them reload it?

In chapter 4, you learned that you could trigger a configuration refresh operation by sending a POST request to the `/actuator/refresh` endpoint provided by Spring Boot Actuator. A request to that endpoint results in a `RefreshScopeRefreshedEvent` inside the application context. All beans marked with `@ConfigurationProperties` or `@RefreshScope` listen to that event and get reloaded when it happens.

You tried the refresh mechanism on Catalog Service, and it worked fine since it was just one application, not even replicated. How about production? Considering the distribution and scale of cloud native applications, sending an HTTP request to all the instances of each application might be a problem. Automation is a crucial part of any cloud native strategy. We need a way to trigger a `RefreshScopeRefreshedEvent` in all of them in one shot. There are a few viable solutions. Using Spring Cloud Bus is one of them.

Spring Cloud Bus (<https://spring.io/projects/spring-cloud-bus>) establishes a convenient communication channel to broadcast events among all the application instances linked to it. It

provides an implementation for AMQP brokers (like RabbitMQ) and Kafka, relying on the Spring Cloud Stream project you learned in chapter 10.

Any configuration change consists of pushing a commit to the config repo. It would be convenient to set up some automation to make Config Service refresh the configuration when a new commit is pushed to the repository, completely removing the need for manual intervention. Spring Cloud Config provides a Monitor library that makes that possible. It exposes a `/monitor` endpoint that can trigger a configuration change event in Config Service, which then would send it over the Bus to all the listening applications. It also accepts arguments describing which files have been changed and supports receiving push notifications from the most common code repository providers like GitHub, GitLab, or Bitbucket. You can set up a webhook in those services to automatically send a POST request to Config Service after each new push to the config repo.

Spring Cloud Bus solves the problem of broadcasting a configuration change event to all the connected applications. With Spring Cloud Config Monitor, we can further automate the refresh and make it happen after a configuration change is pushed to the repository backing the config server. This solution is illustrated in figure 14.1.

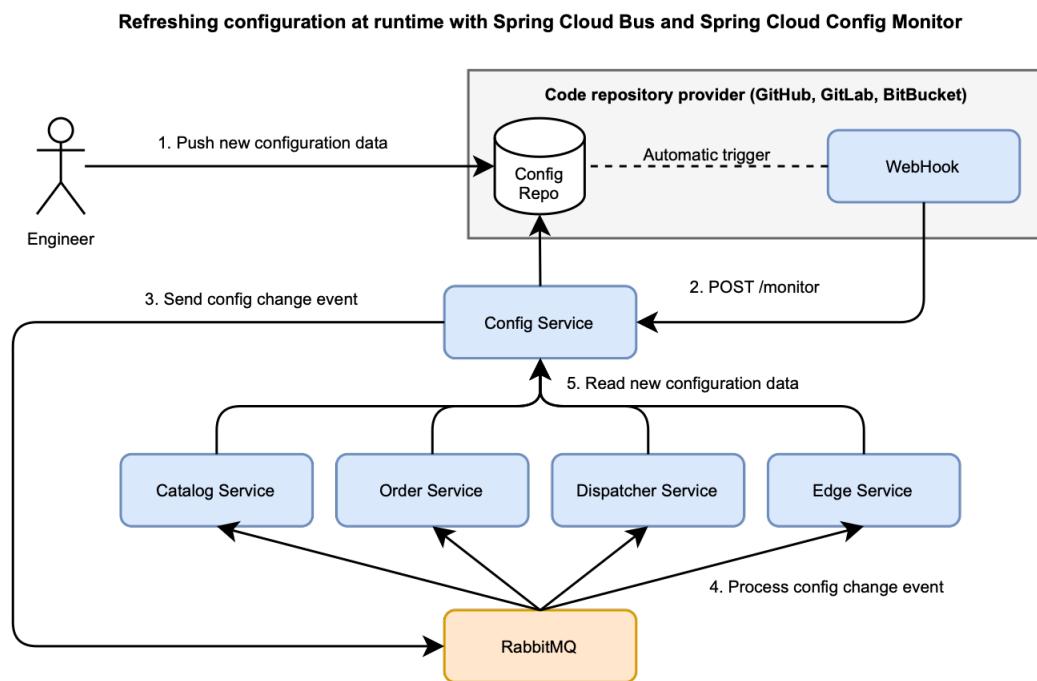


Figure 14.1 Broadcasting configuration changes through Spring Cloud Bus after the Config Service receives push notifications on every config repo change.

NOTE

You can rely on Spring Cloud Bus to broadcast configuration changes even when you use other options like Consul (with Spring Cloud Consul), Azure Key Vault (Spring Cloud Azure), AWS Parameter Store/AWS Secrets Manager (Spring Cloud AWS), or Google Cloud Secret Manager (Spring Cloud GCP). Unlike Spring Cloud Config, they don't have built-in push notifications capabilities, so you need to trigger a configuration change or implement your monitor functionality manually.

14.1.3 Managing secrets with Spring Cloud Config

Managing secrets is a critical task of any software system and is too dangerous when mistakes are made. So far, we have included passwords either in property files or environment variables, but they were unencrypted in both cases. One of the consequences of not being encrypted is that we can't version-control them safely. We would like to keep everything under version control and use Git repositories as the single sources of truth, which is one of the principles behind the GitOps strategy I'll cover in chapter 15.

The Spring Cloud Config project is well-equipped with features to handle any configuration aspect for cloud native applications, including secrets management. The main goal is to include secrets in the property files and put them under version control, which can only be done if they are encrypted.

Spring Cloud Config Server supports encryption and decryption and exposes two dedicated endpoints: `/encrypt` and `/decrypt`. Encryption can be based on a symmetric key or asymmetric key pair.

When using a symmetric key, Spring Cloud Config Server decrypts secrets locally and sends them decrypted to the client applications. In production, all communications between applications would happen over HTTPS. Therefore, the response sent from Config Service would be encrypted even if the configuration property is not, making this approach secure enough for real-world usage.

You also have the option to send property values encrypted and let the applications themselves decrypt them, but it would require you to configure the symmetric key for all applications. And you should consider that decryption is not a cheap operation to perform.

Spring Cloud Config also supports encryption and decryption through asymmetric keys. Such an option provides more robust security than the symmetric alternative but also increases complexity and maintenance costs due to key management tasks. In that case, you might want to consider relying on a dedicated secret management solution. For example, you can use one of

those offered by cloud providers and rely on the Spring Boot integration implemented by Spring Cloud, such as Azure Key Vault (Spring Cloud Azure), AWS Parameter Store/AWS Secrets Manager (Spring Cloud AWS), or Google Cloud Secret Manager (Spring Cloud GCP).

Should you prefer an open-source solution, Hashicorp Vault (<https://www.vaultproject.io>) might be a good fit for you. It's a tool you can use to manage all your credentials, tokens, and certificates, both from a CLI and from a convenient GUI. You can integrate it directly with your Spring Boot applications using the Spring Vault project or add it as an additional backend for Spring Cloud Config Server.

For more information about secrets management in Spring, check out the official documentation for Spring Vault (<https://spring.io/projects/spring-vault>) and Spring Cloud Config (<https://spring.io/projects/spring-cloud-config>).

14.1.4 Disabling Spring Cloud Config

The next section will introduce a different way of configuring Spring Boot applications based on the native functionality provided by Kubernetes through ConfigMaps and Secrets. That's what we're going to use in production.

Even if we're not going to use it anymore for the rest of the book, we'll keep all the work we have done with Config Service so far. However, to make things easier, we'll turn the Spring Cloud Config Client integration off by default.

Open your Catalog Service project (`catalog-service`) and update the `application.yml` file to stop importing configuration data from Config Service and disable the Spring Cloud Config Client integration. Everything else will stay the same. Whenever you want to use Spring Cloud Config again, you can enable it again with ease (for example, when running the applications on Docker).

Listing 14.1 Disable Spring Cloud Config in Catalog Service (application.yml)

```
spring:
  config:
    import: "" ①
  cloud:
    config:
      enabled: false ②
      uri: http://localhost:8888
      request-connect-timeout: 5000
      request-read-timeout: 5000
      fail-fast: false
      retry:
        max-attempts: 6
        initial-interval: 1000
        max-interval: 2000
        multiplier: 1.1
```

- ① Stops importing configuration data from Config Service.

- ② Disables the Spring Cloud Config Client integration.

In the next section, you'll use ConfigMaps and Secrets to configure Spring Boot applications instead of the Config Service.

14.2 Using ConfigMaps and Secrets in Kubernetes

The 15-Factor methodology recommends keeping code, configuration, and credentials always separate. Kubernetes fully embraces that principle and defines two APIs to handle configuration and credentials independently: ConfigMaps and Secrets. This section will introduce this new configuration strategy provided by Kubernetes natively.

Spring Boot provides native and flexible support for both ConfigMaps and Secrets. I'll show you how to work with ConfigMaps and their relationships with environment variables, which are still a valid configuration option in Kubernetes. You'll learn that Secrets are not really secret and what to do to make them really so. Finally, I'll go through a few options for dealing with configuration changes and how to propagate them to the applications.

Before moving further, let's set the scene and start a local Kubernetes cluster. Go to your Polar Deployment project (`polar-deployment`), navigate to the `kubernetes/platform/development` folder, and run the following command to start a minikube cluster and deploy the backing services used by Polar Bookshop.

```
$ ./create-cluster.sh
```

NOTE

If you haven't followed along with the examples implemented in the previous chapters, you can refer to the repository accompanying the book (<https://github.com/ThomasVitale/cloud-native-spring-in-action>) and use the projects in Chapter14/14-begin as a starting point.

The command will take a few minutes to complete. When it's over, you can verify that all backing services are ready and available with the following command.

```
$ kubectl get deploy
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
polar-keycloak	1/1	1	1	3m94s
polar-postgres	1/1	1	1	3m94s
polar-rabbitmq	1/1	1	1	3m94s
polar-redis	1/1	1	1	3m94s
polar-ui	1/1	1	1	3m94s

Let's start by introducing ConfigMaps.

14.2.1 Configuring Spring Boot with ConfigMaps

In chapter 7, we used environment variables to pass hard-coded configuration to containers running in Kubernetes, but they lack maintainability and structure. ConfigMaps let you store configuration data in a structured, maintainable way. They can be version-controlled together with the rest of your Kubernetes deployment manifests and have the same nice properties of a dedicated configuration repository, including data persistence, auditing, and accountability.

SIDE BAR
DEFINITION

A ConfigMap is an API object that stores non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or configuration files in a volume. (<https://kubernetes.io/docs>)

You can build a ConfigMap starting from a literal key-value pair string, from a file (for example, `.properties` or `.yml`), or even from a binary object. When working with Spring Boot applications, the most straightforward way to build a ConfigMap is starting from a property file.

Let's see an example. In the previous chapters, we configured Catalog Service via environment variables. For better maintainability and structure, let's store some of those values in a ConfigMap.

Open the Catalog Service project (`catalog-service`) and create a new `configmap.yml` file in the `k8s` folder. We'll use it to apply the following configuration that will overwrite the default values included in the `application.yml` file packaged with the application.

- Configure a custom greeting.
- Configure the URL for the PostgreSQL data source.
- Configure the URL for Keycloak.

Listing 14.2 Define a ConfigMap to configure Catalog Service (k8s/configmap.yml)

```
apiVersion: v1 ①
kind: ConfigMap ②
metadata:
  name: catalog-config ③
  labels: ④
    app: catalog-service
data: ⑤
  application.yml: | ⑥
    polar:
      greeting: Welcome to the book catalog from Kubernetes!
    spring:
      datasource:
        url: jdbc:postgresql://polar-postgres/polardb_catalog
      security:
        oauth2:
          resourceserver:
            jwt:
              issuer-uri: http://polar-keycloak/realm/PolarBookshop
```

- ① The API version for ConfigMap objects.
- ② The type of object to create.
- ③ The name of the ConfigMap.
- ④ A set of labels attached to the ConfigMap.
- ⑤ Section containing the configuration data.
- ⑥ A key/value pair where the key is the name of a YAML configuration file and the value is its content.

Like the other Kubernetes objects we have worked with so far, manifests for ConfigMaps can be applied to a cluster using the Kubernetes CLI. Open a Terminal window, navigate to your Catalog Service project (`catalog-service`), and run the following command.

```
$ kubectl apply -f k8s/configmap.yaml
```

You can verify the ConfigMap has been created correctly with this command.

```
$ kubectl get cm -l app=catalog-service
NAME          DATA   AGE
catalog-config    1      7s
```

The values stored in a ConfigMap can be used to configure containers running in a few different ways.

- Use a ConfigMap as a configuration data source to pass command-line arguments to the container.
- Use a ConfigMap as a configuration data source to populate environment variables for the container.
- Mount a ConfigMap as a volume in the container.

As you learned in chapter 4 and practiced since then, Spring Boot supports externalized configuration in many ways, including via command-line arguments and environment variables. Passing configuration data as command-line arguments or environment variables to containers has its drawbacks, even if stored in a ConfigMap. For example, whenever you add a property to a ConfigMap, you must update the Deployment manifest. When a ConfigMap is changed, the Pod is not informed about it and must be re-created to read the new configuration. Both those issues are solved by mounting ConfigMaps as volumes.

When a ConfigMap is mounted as a volume to a container, it generates two possible outcomes (figure 14.2).

- If it includes an **embedded property file**, mounting the ConfigMap as a volume results in the property file being created in the mounted path. Spring Boot automatically finds and includes any property files located in a `/config` folder either in the same root as the application executable or in a subdirectory, so it's the perfect path to mount a

ConfigMap. You can also specify additional locations where to look for property files via the `spring.config.additional-location=<path>` configuration property.

- If it includes **key/value pairs**, mounting the ConfigMap as a volume results in a *config tree* being created in the mounted path. For each key/value pair, a file is created named like the key and containing the value. Spring Boot supports reading configuration properties from a config tree. You can specify where to load the config tree from via the `spring.config.import=configtree:<path>` property.

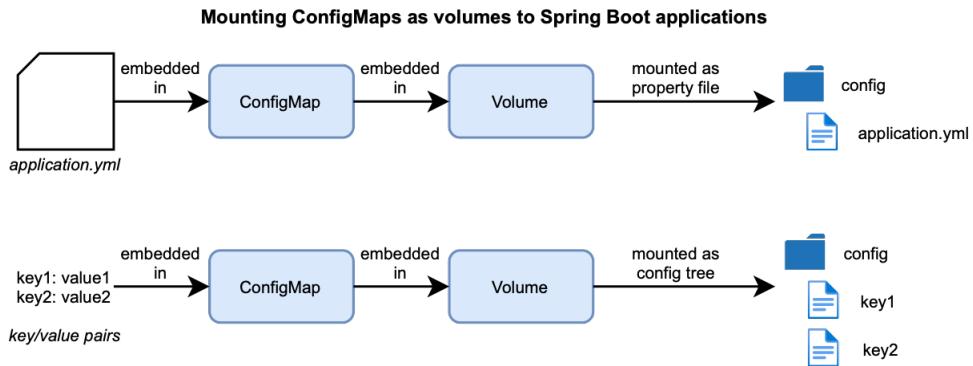


Figure 14.2 ConfigMaps mounted as volumes can be consumed by Spring Boot as property files or as config trees.

When configuring Spring Boot applications, the first option is the most convenient since it uses the same property file format used for the default configuration inside the application. Let's see how to mount the ConfigMap created earlier into the Catalog Service container.

Open the Catalog Service project (`catalog-service`) and go to the `deployment.yaml` file in the `k8s` folder. We need to apply three changes:

- Remove the environment variables for the values we declared in the ConfigMap.
- Declare a volume generated from the `catalog-config` ConfigMap.
- Specify a volume mount for the `catalog-service` container to load the ConfigMap as an `application.yaml` file from `/workspace/config`. The `/workspace` folder is created and used by Cloud Native Buildpacks to host the application executables, so Spring Boot will automatically look for a `/config` folder in the same path and load any property files contained within. There's no need to configure additional locations.

Listing 14.3 Mount a ConfigMap as a volume to the Catalog Service container (k8s/deployment.yml)

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  labels:
    app: catalog-service
spec:
  ...
  template:
    ...
    spec:
      containers:
        - name: catalog-service
          image: catalog-service
          imagePullPolicy: IfNotPresent
        ...
        env: ①
          - name: BPL_JVM_THREAD_COUNT
            value: "50"
          - name: SPRING_PROFILES_ACTIVE
            value: testdata
        ...
        volumeMounts: ②
          - name: catalog-config-volume
            mountPath: /workspace/config ③
      volumes: ④
        - name: catalog-config-volume ⑤
          configMap: ⑥
            name: catalog-config

```

- ① JVM threads and Spring profile are still configured via environment variables.
- ② Mounts the ConfigMap in the container as a volume.
- ③ Spring Boot will automatically find and include property files from this folder.
- ④ Defines volumes for the Pod.
- ⑤ The name of the volume.
- ⑥ The ConfigMap from which to create a volume.

We have previously applied the ConfigMap to the cluster. Let's do the same for the Deployment and Service manifests so that we can verify if Catalog Service is correctly reading configuration data from the ConfigMap.

First, we must package the application as a container image and load it into the cluster. Open a Terminal window, navigate to the root folder of your Catalog Service project (`catalog-service`), and run the following commands.

```
$ ./gradlew bootBuildImage
$ minikube image load catalog-service --profile polar
```

Then, we're ready to deploy the application in the local cluster by applying the Deployment and Service manifests.

```
$ kubectl apply -f k8s/deployment.yml -f k8s/service.yml
```

You can verify when Catalog Service is available and ready to accept requests with this command. Internally, Kubernetes uses the liveness and readiness probes we configured in the previous chapter to infer the application's health.

```
$ kubectl get deploy -l app=catalog-service
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
catalog-service	1/1	1	1	21s

Next, forward traffic from your local machine to the Kubernetes cluster by running the following command.

```
$ kubectl port-forward service/catalog-service 9001:80
Forwarding from 127.0.0.1:9001 -> 9001
Forwarding from [::1]:9001 -> 9001
```

NOTE The process started by the `kubectl port-forward` command will keep running until you explicitly stop it with `Ctrl+C`.

Now, you can call Catalog Service from your local machine on port 9001, and the request will be forwarded to the Service object inside the Kubernetes cluster. Open a new Terminal window and call the root endpoint exposed by the application to verify that the `polar.greeting` value specified in the ConfigMap is used instead of the default one.

```
$ http :9001/
Welcome to the book catalog from Kubernetes!
```

Try also retrieving the books from the catalog to verify that the PostgreSQL URL specified in the ConfigMap is used correctly.

```
$ http :9001/books
```

When you're done testing the application, stop the port-forward process (`Ctrl+C`) and delete the Kubernetes objects created so far. Open a Terminal window, navigate to your Catalog Service project (`catalog-service`), and run the following command. Keep the cluster running since we're going to use it again soon.

```
$ kubectl delete -f k8s
```

ConfigMaps are convenient for providing configuration data to applications running on Kubernetes. What if we had to pass sensitive data? In the next section, you'll see how to use Secrets in Kubernetes.

14.2.2 Storing sensitive information with Secrets (or not)

The most critical part of configuring applications is managing secret information like passwords, certificates, tokens, and keys. Kubernetes provides a Secret object to hold such data and pass them to containers.

SIDE BAR

DEFINITION

A Secret is an API object used to store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Pods can consume Secrets as environment variables or configuration files in a volume. (<https://kubernetes.io/docs>)

What makes this object *secret* is the process used to manage it. By themselves, Secrets are just like ConfigMaps. The only difference is that data in a Secret is usually Base64-encoded, a technical choice made to support binary files. Any Base64-encoded object can be decoded in a very straightforward way. It's a common mistake to think that Base64 is a kind of encryption. If you remember only one thing about Secrets, make it the following: **Secrets are not secret!**

The configuration we have been using to run Polar Bookshop on a local Kubernetes cluster relies on the same default credentials used in development, so we won't need Secrets yet. We'll start using them in the next chapter when deploying applications in production. For now, I want to show you how to create Secrets. Then, I'll go through some options you have for ensuring they are adequately protected.

One way of creating a Secret is using the Kubernetes CLI with an imperative approach. Open a Terminal window and generate a `test-credentials` Secret object for some fictitious test credentials (user/password).

```
$ kubectl create secret generic \ ①
  test-credentials \ ②
    --from-literal=username=user \ ③
    --from-literal=password=password ④
```

- ① Creates a generic secret with Base64-encoded values.
- ② The name of the Secret.
- ③ Adds a secret value for the test username.
- ④ Adds a secret value for the test password.

We can verify that the Secret has been created successfully with the following command.

```
$ kubectl get secret test-credentials

NAME          TYPE      DATA   AGE
test-credentials  Opaque    2      73s
```

We can also retrieve the internal representation of the Secret in the familiar YAML format with the following command. I rearranged the resulting YAML to increase its readability and omitted additional fields that are not relevant to our discussion.

```
$ kubectl get secret test-credentials -o yaml

apiVersion: v1 ①
kind: Secret ②
metadata:
  name: test-credentials ③
type: Opaque
data:
  test.username: dXNlcg==
  test.password: cGFzc3dvcmQ=
```

- ① The API version for Secret objects.
- ② The type of object to create.
- ③ The name of the Secret.
- ④ Section containing the secret data with Base64-encoded values.

I want to repeat it: Secrets are not secret! I can decode the value stored in the `test-credentials` Secret with a simple command.

```
$ echo 'cGFzc3dvcmQ=' | base64 --decode
password
```

Like ConfigMaps, Secrets can be passed to a container as environment variables or through a volume mount. In this second case, you can mount them as property files or config trees. For example, the `test-credentials` Secret would be mounted as a config tree because it's composed of key-value pairs rather than a file.

Since Secrets are not encrypted, we can't include them in a version control system. It's up to the platform engineers to ensure that Secrets are adequately protected. For example, Kubernetes could be configured to store Secrets in its internal `etcd` storage encrypted. That would help ensure security at rest, but it doesn't solve the problem of managing them in a version control system.

Bitnami introduced a project called **Sealed Secrets** (<https://github.com/bitnami-labs/sealed-secrets>) aimed at encrypting Secrets and putting them under version control. First, you would generate an encrypted `SealedSecret` object starting from literal values, similar to what we did for the plain Secret. Then, you would include that in your repository and safely put it under version control. When the `SealedSecret` manifest is applied to a Kubernetes cluster, the Sealed Secrets controller decrypts its content and generates a standard Secret object that can be used within a Pod.

What if your secrets are stored in a dedicated backend like Hashicorp Vault or Azure Key Vault?

In that case, you can use a project like **External Secrets** (<https://github.com/external-secrets/kubernetes-external-secrets>). As you can guess from its name, this project lets you generate a Secret from an external source. The `ExternalSecret` object would be safe to store in your repository and put under version control. When the `ExternalSecret` manifest is applied to a Kubernetes cluster, the External Secrets controller fetches the value from the configured external source and generates a standard Secret object that can be used within a Pod.

NOTE

If you're interested in learning more about how to secure Kubernetes Secrets, you can check out chapter 7 of "GitOps and Kubernetes" by Billy Yuen, Alexander Matyushentsev, Todd Ekenstam, and Jesse Suen (<https://livebook.manning.com/book/gitops-and-kubernetes/chapter-7>) and "Securing Kubernetes Secrets" by Alex Soto Bueno and Andrew Block (<https://www.manning.com/books/securing-kubernetes-secrets?>). I'm not providing more information here since this is usually a task for the platform team, not developers.

When we start using ConfigMaps and Secrets, we must decide which policy to use to update configuration data and how to make applications use the new values. That's the topic of the next section.

14.2.3 Refreshing configuration at runtime with Spring Cloud Kubernetes

When using an external configuration service, you probably want a mechanism to reload the applications when configuration changes. For example, when using Spring Cloud Config, we can implement such a mechanism with Spring Cloud Bus.

In Kubernetes, we need a different approach. When you update a ConfigMap or a Secret, Kubernetes takes care of providing containers with the new versions when mounted as volumes. If you use environment variables, they will not be replaced with the new values. That's why we usually prefer the volume solution.

The updated ConfigMaps or Secrets are provided to the Pod when mounted as volumes, but it's up to the specific application to refresh the configuration. By default, Spring Boot applications read configuration data only at startup time. There are three main options to refresh configuration when provided through ConfigMaps and Secrets.

- **Rolling restart.** Changing a ConfigMap or a Secret can be followed by a rolling restart of all the Pods affected, making the applications reload all the configuration data. With this option, Kubernetes Pods would remain immutable.
- **Spring Cloud Kubernetes Configuration Watcher.** Spring Cloud Kubernetes provides a Kubernetes controller called Configuration Watcher that monitors ConfigMaps and Secrets mounted as volumes to Spring Boot applications. Leveraging the Spring Boot Actuator `/actuator/refresh` endpoint or Spring Cloud Bus, when any of the

ConfigMaps or Secrets is updated, the Configuration Watcher will trigger a configuration refresh for the affected applications.

- **Spring Cloud Kubernetes Config Server.** Spring Cloud Kubernetes provides a configuration server with support for using ConfigMaps and Secrets as one of the configuration data source options for Spring Cloud Config. You could use such a server to load configuration from both a Git repository and Kubernetes objects, with the possibility to use the same configuration refresh mechanism for both.

For Polar Bookshop, we'll use the first option and rely on Kustomize to trigger a restart of the applications whenever a new change is applied to a ConfigMap or a Secret. I'll describe that strategy further in the next part of the chapter. Instead, this section will focus on the features offered by Spring Cloud Kubernetes and its subprojects.

Spring Cloud Kubernetes (<https://spring.io/projects/spring-cloud-kubernetes>) is an exciting project that provides Spring Boot integration with the Kubernetes API. Its original goal was to make it easier to transition from a microservices architecture based on Spring Cloud to Kubernetes. It provides an implementation for standard Spring Cloud interfaces used for service discovery and load balancing to integrate with Kubernetes and adds support for loading configuration from ConfigMaps and Secrets.

If you work on a greenfield project, you don't need Spring Cloud Kubernetes. Kubernetes provides service discovery and load balancing natively, as you experienced in chapter 7. Furthermore, Spring Boot supports configuration via ConfigMaps and Secrets natively, so there's no need for Spring Cloud Kubernetes, even in this case.

When migrating a brownfield project to Kubernetes, if it uses libraries like Spring Cloud Netflix Eureka for service discovery and Spring Cloud Netflix Ribbon/Spring Cloud Load Balancer for load balancing, you might use Spring Cloud Kubernetes for a smoother transition. However, I would recommend refactoring your code to leverage the native service discovery and load balancing features from Kubernetes rather than adding Spring Cloud Kubernetes to your project.

The main reason why I recommend not using Spring Cloud Kubernetes in standard applications is that it requires access to the Kubernetes API Server to manage Pods, Services, ConfigMaps, and Secrets. Besides the security concerns related to granting applications access to the Kubernetes internal objects, it would also couple the applications to Kubernetes unnecessarily and affect the maintainability of the solution.

When does it make sense to use Spring Cloud Kubernetes? For example, Spring Cloud Gateway could be enhanced with Spring Cloud Kubernetes to get more control over service discovery and load balancing, including automatic registration of new routes based on Services metadata and choice of the load balancing strategy. In this case, you could rely on the Spring Cloud Kubernetes Discovery Server component, limiting the need for Kubernetes API access to the discovery server.

Spring Cloud Kubernetes really shines when it comes to implementing Kubernetes controller applications to accomplish administrative tasks within the cluster. For example, you could implement a controller which monitors when ConfigMaps or Secrets change and then triggers a configuration refresh on the application using them. As a matter of fact, the Spring team used Spring Cloud Kubernetes to build a controller doing precisely that: the Configuration Watcher.

NOTE

Spring Cloud Kubernetes Configuration Watcher is available as a container image on Docker Hub. If you'd like to know more about how it works and how to deploy it, you can refer to the official documentation (<https://spring.io/projects/spring-cloud-kubernetes>).

Besides the Configuration Watcher, Spring Cloud Kubernetes provides other convenient off-the-shelf applications for addressing common concerns of distributed systems in Kubernetes. One of them is a configuration server built on top of Spring Cloud Config and extending its functionality to support reading configuration data from ConfigMaps and Secrets. It's called Spring Cloud Kubernetes Config Server.

You can use such an application directly (the container image is published on Docker Hub) and deploy it on Kubernetes following the instructions provided in the official documentation (<https://spring.io/projects/spring-cloud-kubernetes>).

As an alternative, you can even use its source code on GitHub as a foundation to build your own Kubernetes-aware configuration server. For example, as I explained earlier in this chapter, you might want to protect it via HTTP Basic authentication. In that case, you could use your experience working with Spring Cloud Config and build an enhanced version of Config Service for Polar Bookshop on top of Spring Cloud Kubernetes Config Server.

In the next section, I will introduce Kustomize for managing deployment configurations in Kubernetes.

14.3 Configuration management with Kustomize

Kubernetes provides many useful features for running cloud native applications. Still, it requires writing several YAML manifests, which are sometimes redundant and not easy to manage in a real-world scenario. After collecting the multiple manifests needed to deploy an application, we are faced with additional challenges. How can we change the values in a ConfigMap depending on the environment? How can we change the container image version? What about Secrets and volumes? Is it possible to update the health probes configuration?

Many tools have been introduced in the last few years to improve how we configure and deploy workloads in Kubernetes. For the Polar Bookshop system, we would like a tool that lets us handle multiple Kubernetes manifests as a single entity and customize part of the configuration

depending on the environment where the application is deployed.

Kustomize (<https://kustomize.io>) is a declarative tool that helps configure deployments for different environments via a layering approach. It produces standard Kubernetes manifests, and it's built natively in the Kubernetes CLI (`kubectl`), so you don't need to install anything else.

NOTE

Other popular options to manage deployment configuration in Kubernetes are `ytt` from the Carvel suite (<https://carvel.dev/ytt>) and Helm (<https://helm.sh>).

This section will show you the key features offered by Kustomize. First, you'll see how to *compose* related Kubernetes manifests and handle them as a single unit. Then, I'll show you how Kustomize can generate a ConfigMap for you from a property file. Finally, I'll guide you through a series of customizations we'll apply to the base manifests before deploying workloads in a staging environment. The next chapter will expand on that and cover the production scenario.

Before moving on, make sure you still have your local `minikube` cluster up and running and that the Polar Bookshop backing services have been deployed correctly. If you don't, run `./create-cluster.sh` from `polar-deployment/kubernetes/platform/development`.

NOTE

The platform services are exposed only within the cluster. If you want to access any of them from your local machine, you can use the port-forwarding feature you learned in chapter 7. You can either leverage the GUI provided by Octant or use the CLI (`kubectl port-forward service/polar-postgres 5432:5432`).

Now that we have all the backing services available, let's see how to manage and configure a Spring Boot application using Kustomize.

14.3.1 Using Kustomize to manage and configure Spring Boot applications

So far, we've been deploying applications to Kubernetes by applying multiple Kubernetes manifests. For example, deploying Catalog Service requires applying the ConfigMap, Deployment, and Service manifests to the cluster. When using Kustomize, the first step is composing related manifests together so that we can handle them as a single unit. Kustomize does that via a `Kustomization` resource. In the end, we want to let Kustomize manage, process, and generate Kubernetes manifests for us.

Let's see how it works. Open your Catalog Service project (`catalog-service`) and create a `kustomization.yaml` file inside the `k8s` folder. It will be the entry point for Kustomize.

We'll first instruct Kustomize on which Kubernetes manifests it should use as a foundation for

future customizations. For now, we'll use the existing Deployment and Service manifests.

Listing 14.4 Define the base Kubernetes manifests that Kustomize should use as a foundation (`k8s/kustomization.yml`)

```
apiVersion: kustomize.config.k8s.io/v1beta1 ①
kind: Kustomization ②

resources: ③
- deployment.yaml
- service.yaml
```

- ① The API version for Kustomize.
- ② The kind of resource defined by the manifest.
- ③ Kubernetes manifests that Kustomize should manage and process.

You might be wondering why we didn't include the ConfigMap. I'm glad you asked! We might have included the `configmap.yaml` file we created earlier in the chapter, but Kustomize offers a better way. Instead of referencing a ConfigMap directly, we can provide a property file and let Kustomize use it to generate a ConfigMap. Let's see how it works.

For starters, let's move the body of the ConfigMap we created previously (`configmap.yaml`) to a new `application.yaml` file within the `k8s` folder.

Listing 14.5 Configuration properties that will be provided via a ConfigMap (`k8s/application.yaml`)

```
polar:
  greeting: Welcome to the book catalog from Kubernetes!
spring:
  datasource:
    url: jdbc:postgresql://polar-postgres/polardb_catalog
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://polar-keycloak/realm/PolarBookshop
```

Then, delete the `configmap.yaml` file. We'll not need it anymore. Finally, update the `kustomization.yaml` file to generate a `catalog-config` ConfigMap starting from the `application.yaml` file we have just created.

Listing 14.6 Instruct Kustomize to generate a ConfigMap from a property file (k8s/kustomization.yml)

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - deployment.yaml
  - service.yaml

configMapGenerator: ①
  - name: catalog-config
    files: ②
      - application.yaml
    options:
      labels: ③
        app: catalog-service
```

- ① The section containing information to generate ConfigMaps.
- ② Uses a property file as the source for a ConfigMap.
- ③ Defines the labels to assign the generated ConfigMap.

NOTE

In a similar way, Kustomize can also generate Secrets starting from literal values or files.

Let's pause for a moment and verify that what we have done so far works correctly. Your local cluster should already have your Catalog Service container image from before. If that's not the case, build the container image (`./gradlew bootBuildImage`) and load it into `minikube` (`minikube image load catalog-service --profile polar`).

Next, open a Terminal window, navigate to your Catalog Service project (`catalog-service`), and deploy the application using the familiar Kubernetes CLI. When applying standard Kubernetes manifests, we use the `-f` flag. When applying a Kustomization, we use the `-k` flag.

```
$ kubectl apply -k k8s
```

The final result should be the same as we got earlier when applying the Kubernetes manifests directly, but this time Kustomize handled everything via a Kustomization resource.

To complete the verification, use the port-forward strategy to expose the Catalog Service application to your local machine (`kubectl port-forward service/catalog-service 9001:80`). Then, open a new Terminal window and ensure that the root endpoint returns the message configured via the ConfigMap generated by Kustomize.

```
$ http :9001/
Welcome to the book catalog from Kubernetes!
```

ConfigMaps and Secrets generated by Kustomize are named with a unique suffix (a *hash*) when

deployed. You can verify the actual name assigned to the `catalog-config` ConfigMap with the following command.

```
$ kubectl get cm -l app=catalog-service
NAME          DATA   AGE
catalog-config-btcmff5d78   1      7m58s
```

Every time you update the input to the generators, Kustomize creates a new manifest with a different hash which triggers a **rolling restart** of the containers where the updated ConfigMaps or Secrets are mounted as volumes. That is a highly convenient way to achieve an automated configuration refresh without implementing or configuring any additional component.

Let's verify that it's true. First, update the value for the `polar.greeting` property in the `application.yml` used by Kustomize to generate the ConfigMap.

Listing 14.7 Update the configuration used as input to the ConfigMap generator (k8s/application.yml)

```
polar:
  greeting: Welcome to the book catalog from a development
  Kubernetes environment!
  ...
```

Then, apply the Kustomization again (`kubectl apply -k k8s`). Kustomize will generate a new ConfigMap with a different suffix hash, triggering a rolling restart of all the Catalog Service instances. In this case, there's only one instance running. In production, there will be more. The fact that instances are restarted one at a time means that the update happens with zero downtime, which is what we aim for in the cloud. The Catalog Service root endpoint should now return the new message.

```
$ http :9001/
Welcome to the book catalog from a development Kubernetes environment!
```

If you're curious, you could try to compare this result with what would happen when updating a ConfigMap without Kustomize. Kubernetes would update the volume mounted to the Catalog Service container, but the application would not be restarted and still return the old value.

NOTE

Depending on your requirements, you might need to avoid a rolling restart and have the applications reloading configuration at runtime. In that case, you can disable the hash suffix strategy with the `disableNameSuffixHash: true` generator option and, perhaps, rely on something like Spring Cloud Kubernetes Configuration Watcher to notify the applications whenever a ConfigMap or Secret is changed.

When you're done experimenting with the Kustomize setup, you can stop the port-forward

process (Ctrl+C) and undeploy Catalog Service (`kubectl delete -k k8s`).

Since we moved from plain Kubernetes manifests to Kustomize, we still need to update a couple of things. In chapter 7, we used Tilt to achieve a better development workflow when working locally on Kubernetes. Tilt supports Kustomize, so we can configure it to deploy applications via a `Kustomization` resource rather than plain Kubernetes manifests. Go ahead and update the `Tiltfile` in your Catalog Service project as follows.

Listing 14.8 Configure Tilt to deploy Catalog Service using Kustomize (Tiltfile)

```
custom_build(
    ref = 'catalog-service',
    command = './gradlew bootBuildImage --imageName $EXPECTED_REF',
    deps = ['build.gradle', 'src']
)

k8s_yaml(kustomize('k8s')) ①

k8s_resource('catalog-service', port_forwards=['9001'])
```

- ① Run the application from the Kustomization located in the "k8s" folder.

Finally, we need to update the manifest validation step in the commit stage workflow for Catalog Service, or else it will fail the next time we push changes to GitHub. In your Catalog Service project, open the `commit-stage.yml` file (`.github/workflows`) and update it as follows.

Listing 14.9 Use Kubeval to validate the Kubernetes manifests generated by Kustomize (commit-stage.yml)

```
name: Commit Stage
on: push
...
jobs:
  build:
    name: Build and Test
    ...
    steps:
      ...
      - name: Validate Kubernetes manifests
        uses: stefanprodan/kube-tools@v1
        with:
          kubectl: 1.24.1
          kubeval: 0.16.1
          command: |
            kustomize build k8s | kubeval --strict - ①
```

- ① Uses Kustomize to generate the manifests and then validates them with Kubeval.

So far, the most significant benefit we got from Kustomize is the automatic rolling restart of applications when a ConfigMap or Secret is updated. In the next section, you'll learn more about Kustomize and explore its powerful features to manage different Kubernetes configurations depending on the deployment environment.

14.3.2 Managing Kubernetes configuration for multiple environments with Kustomize

During development, we followed the 15-Factors methodology and externalized the configuration for each aspect of an application that could change between deployments in different environments. You saw how to use property files, environment variables, configuration services, and ConfigMaps. I also showed you how to use Spring profiles to customize the application configuration based on the deployment environment. Now, we need to take a step further and define a strategy to customize the entire deployment configuration depending on where we deploy an application.

In the previous section, you learned how to compose and process Kubernetes manifests together via a `Kustomization` resource. For each environment, we can specify **patches** to apply changes or additional configurations on top of those basic manifests. All the customization steps you'll see in this section will be applied without changing anything in the application source code but using the same release artifacts produced earlier. That's quite a powerful concept and one of the main features of cloud native applications.

The Kustomize approach to configuration customization is based on the concepts of **bases** and **overlays**. The `k8s` folder we created in the Catalog Service project can be considered a *base*: a directory with a `kustomization.yaml` file combining Kubernetes manifests and customizations. An *overlay* is another directory with a `kustomization.yaml` file. What makes it special is that it defines customizations in relation to one or more bases and combines them. Starting from the same base, you can specify an overlay for each deployment environment (such as development, test, staging, and production).

As shown in figure 14.3, each `Kustomization` includes a `kustomization.yaml` file. The one acting as the *base* composes together several Kubernetes resources like Deployments, Services, and ConfigMaps. Also, it's not aware of overlays; therefore, it's completely independent of them. The *overlays* use one or more *bases* as a foundation and provide additional configuration via patches.

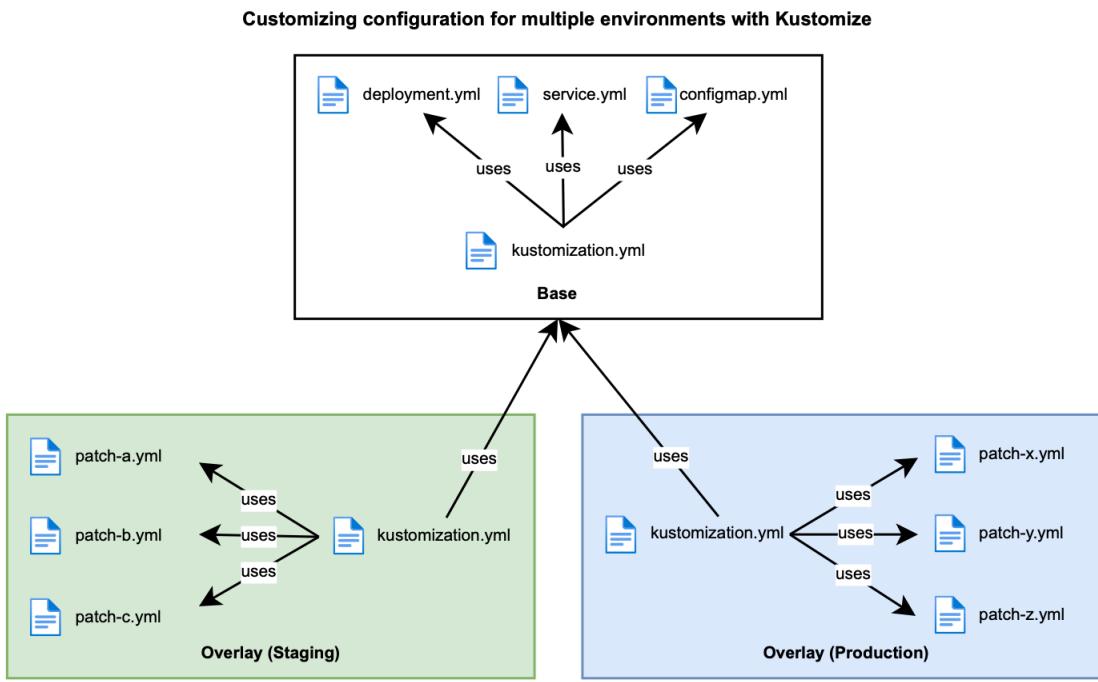


Figure 14.3 Kustomize bases can be used as the foundation for further customizations (overlays) depending on the deployment environment.

Bases and overlays could be defined either in the same repository or different ones. For the Polar Bookshop system, we'll use the `k8s` folder in each application project as a base and define overlays in the `polar-deployment` repository. Similar to what was explained in chapter 3 about application codebases, you can decide whether to keep your deployment configuration in the same repository as your application. I decided to go for a separate repository for a few reasons:

- it makes it possible to control the deployment of all the system components from a single place;
- it allows focused version-control, auditing, and compliance checks before deploying anything to production;
- it fits the GitOps approach where delivery and deployment tasks are decoupled.

As an example, figure 14.4 shows how the Kustomize manifests could be structured in the case of Catalog Service, having bases and overlays in two separate repositories.

Customizing Catalog Service for multiple environments with Kustomize

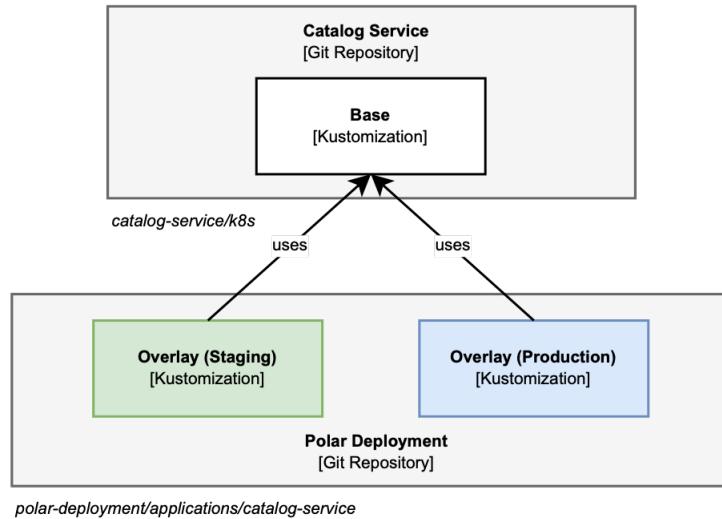


Figure 14.4 Kustomize bases and overlays can be stored in the same repository or two separate ones. Overlays can be used to customize deployments for different environments.

Another decision to make is whether to keep the base Kubernetes manifests together with the application source code or move them to the deployment repository. I decided to go with the first approach for the Polar Bookshop example, similar to how we did with the default configuration properties. One of the benefits is that it makes it simple to run each application on a local Kubernetes cluster during development, either directly or using Tilt. Depending on your requirements, you might decide to use one approach or the other. Both are valid and used in real-world scenarios.

SIDE BAR**Patches vs. templates**

The approach to configuration customization adopted by Kustomize is based on applying patches. It's quite the opposite of how Helm works (<https://helm.sh>). Helm requires you to template every part of a manifest you would like to change (resulting in non-valid YAML). After that, you can provide different values for those templates in each environment. If a field is not templated, you can't customize its value. For that reason, it's not rare to use Helm and Kustomize in sequence, overcoming each other's shortcomings. Each approach has pros and cons.

In this book, I decided to use Kustomize because it's natively available in the Kubernetes CLI, works with valid YAML files, and it's purely declarative. Helm is more powerful and can also handle complex application rollouts and upgrades that Kubernetes doesn't support natively. On the other hand, it has a steep learning curve, its templating solution has a few drawbacks, and it's not declarative.

Another option is `ytt` from the Carvel suite (<https://carvel.dev/ytt>). It provides a superior experience, with support for both patches and templates. It works with valid YAML files, and its templating strategy is more robust. It takes a bit more than Kustomize to get familiar with `ytt`, but it's worth the effort. Because it treats YAML as a first-class citizen, `ytt` can be used to configure and customize any YAML file, even outside Kubernetes. GitHub Actions workflows? Ansible playbooks? Jenkins pipelines? You can use `ytt` in all those scenarios.

Let's consider Catalog Service. We already have the base deployment configuration composed with Kustomize. It's located within the project repository in a dedicated folder (`catalog-service/k8s`). Now, let's see how to define an overlay to customize the deployment for production.

14.3.3 Defining a configuration overlay for staging

In the previous sections, we used Kustomize to manage the configuration of Catalog Service in a local development environment. Those manifests will represent the *base* for multiple customizations applied for each environment as an *overlay*. Since we'll define overlays in the `polar-deployment` repository while the base is in the `catalog-service` repository, all the Catalog Service manifests must be available in the main remote branch. If you haven't done it yet, push all the changes applied to your Catalog Service project so far to the remote repository on GitHub.

NOTE

As explained in chapter 2, I expect you have created a different repository on GitHub for each project in the Polar Bookshop system. In this chapter, we’re working only with the `polar-deployment` and `catalog-service` repositories, but you should have also created repositories for `edge-service`, `order-service`, and `dispatcher-service`.

As anticipated, we’ll store any configuration overlay in the `polar-deployment` repository. In this section and the following ones, we’ll define an overlay for the staging environment. The next chapter will cover production.

Go ahead and create a new `kubernetes/applications` folder in your `polar-deployment` repository. We’ll use it to keep the customizations for all the applications in the Polar Bookshop system. In the newly created path, add a `catalog-service` folder that will contain any overlay for customizing the deployment of Catalog Service in different environments. In particular, we want to prepare the deployment in staging, so create a `staging` folder for Catalog Service.

Any customization (base or overlay) requires a `kustomization.yml` file. Let’s create one for the staging overlay of Catalog Service (`polar-deployment/kubernetes/applications/catalog-service/staging`). The first thing to configure is a reference to the base manifests.

If you followed along, you should have your Catalog Service source code tracked in a `catalog-service` repository on GitHub. A reference to a remote base needs to point to the folder containing the `kustomization.yml` file, which is `k8s` in our case. Also, we should refer to a specific tag or digest for the version we want to deploy. We’ll talk about release strategies and versioning in the next chapter, so we’ll simply point to the `main` branch for now. The final URL should be something like `github.com/<your_github_username>/catalog-service/k8s?ref=main`. For example, in my case, it would be `github.com/polarbookshop/catalog-service/k8s?ref=main`.

Listing 14.10 Define an overlay for staging on top of a remote base (staging/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources: ①
- github.com/<your_github_username>/catalog-service/k8s?ref=main
```

- ① Uses the manifests in your Catalog Service repo on GitHub as the base for further customizations.

NOTE

I'll assume that all the GitHub repositories we created for Polar Bookshop are publicly accessible. If that's not the case, you can go to the specific repository page on GitHub and access the "Settings" section for that repository. Then, scroll to the bottom of the settings page, and make the package public by clicking the "Change visibility" button.

We could now deploy Catalog Service from the staging overlay using the Kubernetes CLI, but the result wouldn't be different than using the base directly. Let's start applying some customizations specifically for staging deployments.

14.3.4 Customizing environment variables

The first customization we could apply is an environment variable to activate the `staging` Spring profile for Catalog Service. Most customizations can be applied via patches following a merge strategy. Similar to how Git merges changes coming from different branches, Kustomize adopts a strategy to produce final Kubernetes manifests with changes coming from different Kustomizations (one or more bases and an overlay).

A best practice when defining Kustomize patches is to keep them small and focused. For customizing environment variables, create a `patch-env.yml` file within the staging overlay for Catalog Service (`kubernetes/applications/catalog-service/staging`). We need to specify some contextual information for Kustomize to figure out where to apply the patch and how to merge the changes. When the patch is for customizing a container, Kustomize requires us to specify the kind and name of the Kubernetes resource (that is, `Deployment`) and the name of the container. This customization option is called "Strategic Merge Patch".

Listing 14.11 A patch for customizing environment variables in a container (staging/patch-env.yml).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  template:
    spec:
      containers:
        - name: catalog-service
          env:
            - name: SPRING_PROFILES_ACTIVE ①
              value: prod
```

- ① Defines which Spring profiles should be activated.

Next, we need to instruct Kustomize to apply the patch. In the `kustomization.yml` file for the staging overlay of Catalog Service, list the `patch-env.yml` file as follows.

Listing 14.12 Configure Kustomize to apply the patch for environment variables (staging/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- github.com/<your_github_username>/catalog-service/k8s?ref=main

patchesStrategicMerge: ①
- patch-env.yml ②
```

- ① Section containing the list of patches to apply to the base manifests according to the Strategic Merge strategy.
- ② The patch for customizing the environment variables passed to the Catalog Service container.

You can use this same approach to customize many aspects of a Deployment, such as the number of replicas, liveness probe, readiness probe, graceful shutdown timeout, environment variables, volumes, and more. In the next section, I'll show you how to customize ConfigMaps.

14.3.5 Customizing ConfigMaps

The base Kustomization for Catalog Service instructs Kustomize to generate a catalog-config ConfigMap starting from an application.yml file. For customizing the values in that ConfigMap, we have two main options: replace the entire ConfigMap or overwrite only the values that should be different in staging. In this second case, we could generally rely on some Kustomize advanced patching strategy to overwrite specific values in the ConfigMap.

When working with Spring Boot, we can take advantage of the power of Spring profiles. Instead of updating values in the existing ConfigMap, we could add an application-staging.yml file which we know takes precedence over application.yml when the staging profile is active. The final result will be a ConfigMap containing both files.

First, let's create an application-staging.yml file within the staging overlay for Catalog Service. We'll use this property file to define a different value for the polar.greeting property. Since we'll use the same minikube cluster from earlier as the staging environment, URLs to backing services and credentials are the same as in the development environment. In a real-world scenario, this stage would involve more customizations.

Listing 14.13 Staging-specific configuration for Catalog Service (staging/application-staging.yml).

```
polar:
  greeting: Welcome to the book catalog from a staging
  ↪Kubernetes environment!
```

Next, we can rely on the ConfigMap Generator provided by Kustomize to combine the `application-staging.yml` file (defined in the staging overlay) with the `application.yml` file (defined in the base Kustomization) within the same `catalog-config` ConfigMap. Go ahead and update the `kustomization.yml` file for the staging overlay as follows.

Listing 14.14 Merge property files within the same ConfigMap (staging/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- github.com/<your_github_username>/catalog-service/k8s?ref=main

patchesStrategicMerge:
- patch-env.yml

configMapGenerator:
- behavior: merge ①
  files:
  - application-staging.yml ②
  name: catalog-config ③
```

- ① Merges this ConfigMap with the one defined in the base Kustomization.
- ② The additional property file added to the ConfigMap.
- ③ The same ConfigMap name used in the base Kustomization.

That's it for ConfigMaps. The following section will cover how to configure which image name and version to deploy.

14.3.6 Customizing image name and version

The base Deployment manifest defined in the Catalog Service repository (`catalog-service/k8s/deployment.yml`) is configured to use a local container image and doesn't specify a version number (which means the `latest` tag is used). That's convenient in the development phase, but it doesn't work for other deployment environments.

If you followed along, you should have your Catalog Service source code tracked in a `catalog-service` repository on GitHub and a `ghcr.io/<your_github_username>/catalog-service:latest` container image published to GitHub Container Registry (as per the Commit Stage workflow). The next chapter will cover release strategies and versioning. Until then, we'll still use the `latest` tag. Regarding the image name, it's time to start pulling container images from the registry rather than using the local ones.

NOTE

Images published to GitHub Container Registry will have the same visibility as the related GitHub code repository. I'll assume that all the images we build for Polar Bookshop are publicly accessible via the GitHub Container Registry. If that's not the case, you can go to the specific repository page on GitHub and access the "Packages" section for that repository. Then, select "Package settings" from the sidebar menu, scroll to the bottom of the settings page, and make the package public by clicking the "Change visibility" button.

Similar to what we've done for environment variables, we could use a patch to change the image used by the Catalog Service Deployment resource. Since it's a very common customization and needs to be changed every time we deliver a new version of our applications, Kustomize provides a more convenient way to declare what image name and version we want to use for each container. Furthermore, we can either update the `kustomization.yml` file directly or rely on the Kustomize CLI (installed as part of the Kubernetes CLI). Let's try the latter.

Open a Terminal window, navigate to the staging overlay for Catalog Service (`kubernetes/applications/catalog-service/staging`), and run this command to define what image and version to use for the `catalog-service` container. Remember to replace `<your_github_username>` with your GitHub username in lowercase.

```
$ kustomize edit set image \
  catalog-service=ghcr.io/<your_github_username>/catalog-service:latest
```

The command will automatically update the `kustomization.yml` file with the new configuration, as you can see in the following listing.

Listing 14.15 Configure the image name and version for the Catalog Service container (staging/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - github.com/<your_github_username>/catalog-service/k8s?ref=main

patchesStrategicMerge:
  - patch-env.yml

configMapGenerator:
  - behavior: merge
    files:
      - application-staging.yml
    name: catalog-config

images:
  - name: catalog-service ①
    newName: ghcr.io/<your_github_username>/catalog-service ②
    newTag: latest ③
```

- ① The name of the container as defined in the Deployment manifest.

- ② The new image name for the container (with your GitHub username in lowercase).
- ③ The new tag for the container.

In the next section, I'll show you how to configure the number of replicas to deploy.

14.3.7 Customizing the number of replicas

Cloud native applications should be highly available. Catalog Service is not. So far, we've always been deploying one single application instance. What happens if it crashes or becomes momentarily unavailable due to a high workload? We would not be able to use the application anymore. Not very resilient, isn't it? Among other things, a staging environment is a good target for performance and availability tests. At a minimum, we should have two instances running. Kustomize provides a convenient way to update the number of replicas for a given Pod.

Open the `kustomization.yml` file in the staging overlay for Catalog Service (`kubernetes/applications/catalog-service/staging`) and configure two replicas for the application.

Listing 14.16 Configure the number of replicas for the Catalog Service container (staging/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- github.com/<your_github_username>/catalog-service/k8s?ref=main

patchesStrategicMerge:
- patch-env.yml

configMapGenerator:
- behavior: merge
  files:
  - application-staging.yml
  name: catalog-config

images:
- name: catalog-service
  newName: ghcr.io/<your_github_username>/catalog-service
  newTag: latest

replicas:
- name: catalog-service ①
  count: 2 ②
```

- ① The name of the Deployment for which to define the number of replicas.
- ② The number of replicas.

It's finally time to deploy Catalog Service and test the configuration provided by the staging overlay. For simplicity, we'll use the same `minikube` local cluster we have been using so far as the staging environment. If you still have your `minikube` cluster up and running from before,

you're good to go. Otherwise, you can start it by running `./create-cluster.sh` from `polar-deployment/kubernetes/platform/development`. The script will spin up a Kubernetes cluster and deploy the backing services required by Polar Bookshop.

Then, open a Terminal window, navigate to the staging overlay folder for Catalog Service (`applications/catalog-service/staging`) and run the following command to deploy the application via Kustomize.

```
$ kubectl apply -k .
```

You can monitor the operation result via the Kubernetes CLI (`kubectl get pod -l app=catalog-service`) or the Octant GUI (refer to chapter 7 for more information). Once the applications are available and ready, let's check the application logs using the CLI. One of the first Spring Boot log events will tell you that the `staging` profile is enabled, just like we configured in the staging overlay via a patch.

```
$ kubectl logs deployment/catalog-service
```

The application is not exposed outside the cluster, but you can use the port-forward functionality to forward traffic from your local environment on port 9001 to the Service running in the cluster on port 80.

```
$ kubectl port-forward service/catalog-service 9001:80
```

Next, open a new Terminal window and call the application root endpoint.

```
$ http :9001
Welcome to the book catalog from a staging Kubernetes environment!
```

The result is the customized message we defined in the `application-staging.yml` file for the `polar.greeting` property. That's exactly what we were expecting.

NOTE

It's worth noticing that if you send a GET request to :9001/books, you'll get an empty list. In staging, we haven't enabled the test-data profile controlling the generation of books at startup time. We want that only in a development or test environment.

The last customization we applied to the staging overlay was the number of replicas to deploy. Let's verify that with the following command.

```
$ kubectl get pod -l app=catalog-service
NAME                               READY   STATUS    RESTARTS   AGE
catalog-service-6c5fc7b955-9kvgf   1/1     Running   0          3m94s
catalog-service-6c5fc7b955-n7rgl   1/1     Running   0          3m94s
```

Kubernetes is designed to ensure the availability of each application. If enough resources are available, it will try to deploy the two replicas on two different nodes. If one node crashes, the application would still be available on the other one. At the same time, Kubernetes takes care of deploying the second instance somewhere else to ensure there are always two replicas up and running. You can check which node each Pod has been allocated on with `kubectl get pod -o wide`. In our case, the `minikube` cluster has only one node, so both instances will be deployed together.

If you're curious, you can also try to update the `application-staging.yml` file, apply the Kustomization to the cluster again (`kubectl apply -k .`), and see how the Catalog Service Pods are restarted one after the other (*rolling restart*) to load the new ConfigMap with zero downtime. To visualize the sequence of events, you can either use Octant or launch this command on a separate Terminal window before applying the Kustomization: `kubectl get pods -l app=catalog-service --watch`.

When you're done testing the application, you can terminate the port-forward process with `Ctrl+C` and delete the cluster with `./destroy-cluster.sh` from `polar-deployment/kubernetes/platform/development`.

After learning the fundamentals of configuring and deploying Spring Boot applications in Kustomize, it's time to go to production. That's what the next chapter is all about.

SIDE BAR

Polar Labs

Feel free to apply what you learned in this chapter to all the applications in the Polar Bookshop system. You'll need it in the next chapter, where we'll deploy everything in production.

1. Disable the Spring Cloud Config client.
2. Define a base Kustomization manifest, and update Tilt and commit stage workflow.
3. Use Kustomize to generate a ConfigMap.
4. Configure a staging overlay.

You can refer to the `Chapter14/14-end` folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

14.4 Summary

- A configuration server built with Spring Cloud Config Server can be protected with any of the features offered by Spring Security. For example, you can require a client to use HTTP Basic authentication to access the configuration endpoints exposed by the server.
- Configuration data in a Spring Boot application can be reloaded by calling the `/actuator/refresh` endpoint exposed by Spring Boot Actuator.
- To propagate the config refresh operation to other applications in the system, you can use Spring Cloud Bus.
- Spring Cloud Config Server offers a Monitor module that exposes a `/monitor` endpoint for code repository providers to call through a webhook whenever a new change is pushed to the configuration repository. The result is that all the applications affected by the configuration change will be triggered by Spring Cloud Bus to reload the configuration. The whole process happens automatically.
- Managing secrets is a critical task of any software system and is too dangerous when mistakes are made.
- Spring Cloud Config offers encryption and decryption features to handle secrets safely in the configuration repository, using either symmetric or asymmetric keys.
- You can also use secrets management solutions offered by cloud providers like Azure, AWS, and Google Cloud and leverage the integration with Spring Boot provided by Spring Cloud Azure, Spring Cloud AWS, and Spring Cloud GCP.
- Hashicorp Vault is another option. You can either use it to configure all Spring Boot applications directly through the Spring Vault project or make it a backend for Spring Cloud Config Server.
- When Spring Boot applications are deployed to a Kubernetes cluster, you can configure them also through ConfigMaps (for non-sensitive configuration data) and Secrets (for sensitive configuration data).
- You can use ConfigMaps and Secrets as a source of values for environment variables or mount them as volumes to the container. The latter approach is the preferred one and is supported by Spring Boot natively.
- Secrets are not secret. The data contained within them are not encrypted by default, so you shouldn't put them under version control and include them in your repository.
- The platform team is responsible for protecting Secrets, for example, using the Sealed Secrets project to encrypt Secrets and make it possible to put them under version control.
- Managing several Kubernetes manifests to deploy an application is not very intuitive. Kustomize provides a convenient way to manage, deploy, configure, and upgrade an application in Kubernetes.
- Among the other things, Kustomize provides generators to build ConfigMaps and Secrets, and a way to trigger a rolling restart whenever they are updated.
- The Kustomize approach to configuration customization is based on the concepts of bases and overlays.
- Overlays are built on top of base manifests, and any customization is applied via patches. You saw how to define patches for customizing environment variables, ConfigMaps, container images, and replicas.

15

Continuous delivery and GitOps

This chapter covers

- Understanding continuous delivery and release management
- Configuring Spring Boot for production with Kustomize
- Deploying in production with GitOps and Kubernetes

Chapter after chapter, we have gone through patterns, principles, and best practices to work with cloud native applications and built a bookshop system using Spring Boot and Kubernetes. It's time to deploy Polar Bookshop to production.

I expect you have the projects part of the Polar Bookshop system placed in separate Git repositories and stored on GitHub. If you haven't followed along through the previous chapters, you can refer to the `Chapter15/15-begin` folder in the source code accompanying the book, and use it as a foundation to define those repositories.

This chapter will guide you through some final aspects of preparing applications for production. First, I'll discuss versioning strategies for release candidates and how to design the acceptance stage of a deployment pipeline. Then, you'll see how to configure Spring Boot applications for production and deploy them on a Kubernetes cluster in a public cloud. Next, I'll show you how to complete the deployment pipeline by implementing the production stage. Finally, you'll get to use Argo CD to implement continuous deployment based on the GitOps principles.

NOTE

The source code for the examples in this chapter is available in the Chapter15/15-begin and Chapter15/15-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

15.1 Deployment pipeline: Acceptance stage

Continuous delivery is one of the fundamental practices we have identified to support us in our journey to achieve the cloud native goals: speed, resilience, scale, and cost optimization. It's a holistic approach for delivering high-quality software quickly, reliably, and safely. The main idea behind continuous delivery is that an application is always in a releasable state. The primary pattern for adopting continuous delivery is the deployment pipeline, which goes from code commit to releasable software. It should be automated as much as possible and represent the only path to production.

Chapter 3 explained that a deployment pipeline can be composed of three key stages: commit stage, acceptance stage, and production stage. Throughout the book, we have automated the commit stage as a workflow in GitHub Actions. After a developer commits new code to the mainline, this stage goes through build, unit tests, integration tests, static code analysis, and packaging. At the end of this stage, an executable application artifact is published to an artifact repository. That is a **release candidate**.

This section will cover how to version release candidates for continuous delivery. Then, you'll learn more about the acceptance stage, its purpose, and its outcome. Finally, I'll show you how to implement the skeleton for a minimal workflow in GitHub Actions for the acceptance stage. At the end of this stage, the release candidate is ready to be deployed to production.

15.1.1 Versioning release candidates for continuous delivery

The output of the commit stage of the deployment pipeline is a release candidate. That's the deployable artifact for an application. In our case, it's a container image. All the subsequent steps in the pipeline will evaluate the quality of that container image through multiple different tests. If no issue is found, the release candidate is ultimately deployed to production and released to users.

A release candidate is stored in an artifact repository. If it's a JAR, it would be stored in a Maven repository. In our case, it's a container image and will be stored in a container registry. In particular, we'll use GitHub Container Registry.

Each release candidate must be uniquely identified. So far, we have used the implicit `latest` tag for all container image versions. Also, we ignored the `0.0.1-SNAPSHOT` version configured in

each Spring Boot project by default in Gradle. How should we version release candidates?

A popular strategy is *semantic versioning* (<https://semver.org>). It consists of identifiers in the form of <major>. <minor>. <patch>. Optionally, you can also add a hyphen at the end and a string marking a pre-release. By default, a Spring Boot project generated from <https://start.spring.io> is initialized with version 0.0.1-SNAPSHOT, which identifies a snapshot release. A variation of this strategy is *calendar versioning* (<https://calver.org>), which combines the concepts of semantic versioning with date and time.

Both those strategies are broadly used for open source projects and software released as products to customers, because they provide implicit information about what a new release contains. For example, we expect a new major version to contain new functionality and API changes incompatible with the previous one. On the other hand, we would expect a patch to have a limited scope and guarantee backward compatibility.

NOTE

If you're working on software projects for which semantic versioning makes sense, I recommend checking out JReleaser, a release automation tool. "Its goal is to simplify creating releases and publishing artifacts to multiple package managers while providing customizable options." (<https://jreleaser.org>)

Semantic versioning would require some form of manual step to assign a version number based on the content of the release artifact. Does it contain breaking changes? Does it only contain bug fixes? Once we have a number, it's not clear what it's included in the new release artifact, so we need to use Git tags and define a mapping between Git commit identifiers and version numbers.

Things get even more challenging for snapshot artifacts. Let's consider a Spring Boot project as an example. By default, we start with version 0.0.1-SNAPSHOT. Until we're ready to cut the 0.0.1 release, every time we push new changes to the main branch, the commit stage would be triggered, and a new release candidate would be published with the number 0.0.1-SNAPSHOT. All release candidates will have the same number until version 0.0.1 is released. This approach doesn't ensure traceability of the changes. Which commits are included in a release candidate 0.0.1-SNAPSHOT?. We can't tell. Furthermore, it's affected by the same unreliability as using latest. Every time we retrieve the artifact, it might be different from the last time.

When it comes to continuous delivery, using an approach like semantic versioning is not ideal to uniquely identify release candidates. When we work following the principles of continuous integration, there are many release candidates built daily. And every release candidate can potentially be promoted to production. Would we have to update explicitly the semantic version for each new code commit, with a different approach based on its content (major, minor, patch)?

The path from code commit to production should be automated as much as possible, trying to eliminate any manual intervention. If we go with continuous deployment, even the promotion to production will happen automatically. What should we do?

A solution would be using the Git commit hash to version release candidates, which would be automated, traceable, and reliable. And you wouldn't need Git tags. You could use the commit hash as-is (for example, 486105e261cb346b87920aaa4ea6dce6eebd6223) or use it as the base for generating a more human-friendly number. For example, you could prefix it with a timestamp or with an increasing sequence number, with the goal of making it possible to tell which release candidate is the newest (e.g., 20220731210356-486105e261cb346b87920aaa4ea6dce6eebd6223).

Still, semantic versioning or similar strategies have their place in continuous delivery. They can be used as *display names* in addition to the unique identifier, as Dave Farley suggests in his book "Continuous Delivery Pipelines". That would be a way to provide users with information about the release candidate while still making it possible to benefit from continuous delivery.

For Polar Bookshop, we'll adopt a simple solution and use the Git commit hash directly to identify our release candidates. Therefore, we'll ignore the version number configured in the Gradle project (which could be used as the display version name, instead). For example, a release candidate for Catalog Service would be `ghcr.io/<your_github_username>/catalog-service:<commit-hash>`. Now that we have a strategy, let's see how to implement it for Catalog Service.

Go to your Catalog Service project (`catalog-service`) and open the `commit-stage.yml` file within the `.github/workflows` folder. We have previously defined a `VERSION` environment variable to hold the release candidate unique identifier. At the moment, it's statically set to `latest`. Let's replace that with `$(github.sha)`, which will be dynamically resolved to the current Git commit hash by GitHub Actions. For convenience, we'll also add the `latest` tag to the newest release candidate, useful for local development scenarios.

Listing 15.1 Use the Git commit hash to version release candidates (commit-stage.yml).

```

name: Commit Stage
on: push

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: polarbookshop/catalog-service
  VERSION: ${{ github.sha }} ①

build:
  name: Build and Test
  ...

package:
  name: Package and Publish
  ...
steps:
  ...
  - name: Publish container image ①
    run: docker push \
      ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:${{ env.VERSION }}
  - name: Publish container image (latest) ②
    run:
      docker tag \
        ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:${{ env.VERSION }} \
        ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:latest
      docker push ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}:latest

```

- ① Publishes a release candidate with a version equal to the Git commit hash.
- ② Adds the "latest" tag to the newest release candidate.

After updating the workflow, commit your changes and push them to GitHub. That will trigger the execution of the commit stage workflow (figure 15.1). The outcome will be a container image published to GitHub Container Registry, versioned with the current Git commit hash and the additional `latest` tag.

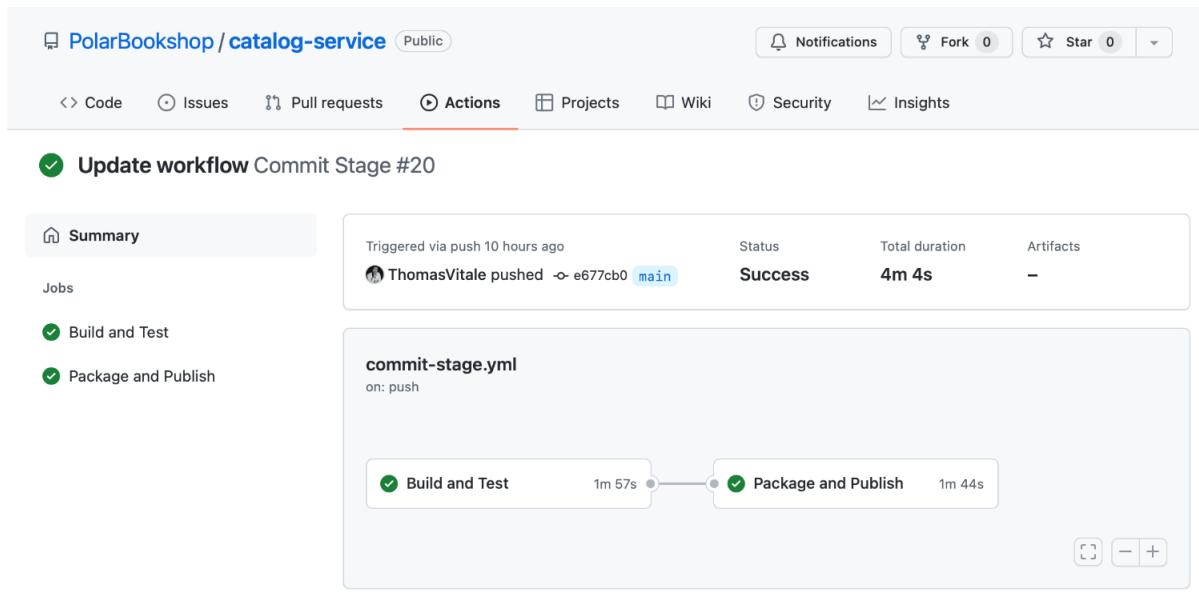


Figure 15.1 The commit stage goes from code commit to release candidate published to an artifact repository.

Once the pipeline is executed successfully, you can see the newly-published container image from your `catalog-service` repository main page on GitHub. In the sidebar, you'll find a "Packages" section with a "catalog-service" item. Click that, and you'll be directed to the container repository for Catalog Service (figure 15.2). When using the GitHub Container Registry, container images are stored next to the source code, which is very convenient.

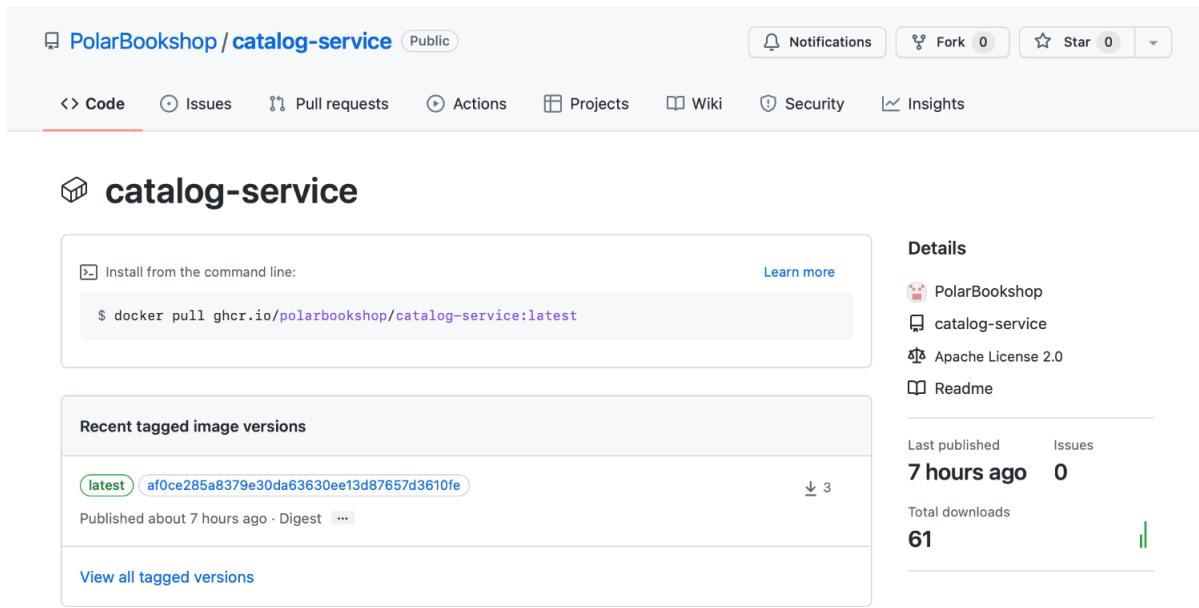


Figure 15.2 In our case, release candidates are container images published to GitHub Container Registry.

At this point, the container image (our release candidate) is uniquely identified and ready to go

through the acceptance stage. That's the topic of the next section.

15.1.2 Understanding the acceptance stage of the deployment pipeline

The acceptance stage of the deployment pipeline is triggered whenever a new release candidate is published to the artifact repository at the end of the commit stage. It consists of deploying the application to a production-like environment and running additional tests to increase the confidence in its releasability. The tests running in the acceptance stage are usually slow, but we should strive to keep the whole deployment pipeline execution under one hour.

In chapter 3, you learned about the software test classification provided by the **Agile Testing Quadrants**. The *Quadrants* (figure 15.3) classify software tests based on whether they are technology or business-facing, and whether they support development teams or are used to critique the project.

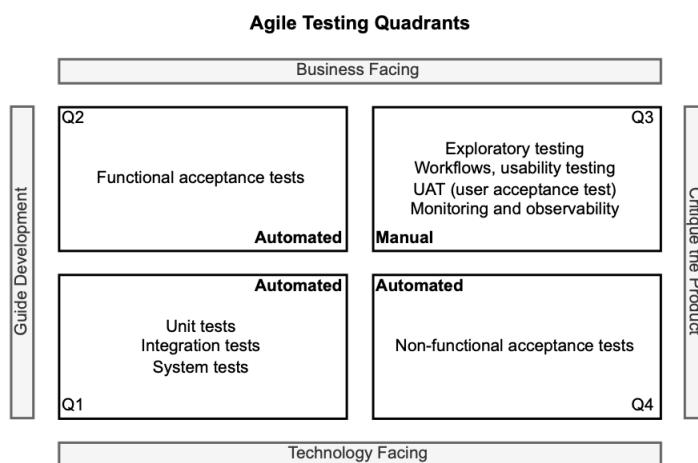


Figure 15.3 The Agile Testing Quadrants are a taxonomy helpful to plan a software testing strategy.

In the commit stage, we mainly focus on the first quadrant, including unit and integration tests. They are technology-facing tests that support the team, ensuring they build the *software right*. On the other hand, the acceptance stage focuses on the second and fourth quadrants and tries to eliminate the need for manual regression testing. This stage includes functional and non-functional acceptance tests.

The **functional acceptance tests** are business-facing and support development teams ensuring they are building the *right software*. They take on the user perspective and are usually implemented via *executable specifications* using a high-level domain-specific language (DSL), then translated into a lower-level programming language. For example, you could use Cucumber (<https://cucumber.io>) to write scenarios like "browse the book catalog" or "place a book order" in human-friendly plain text. Those scenarios can then be executed and verified using a programming language like Java.

In the acceptance stage, we can also verify the *quality attributes* of a release candidate via **non-functional acceptance tests**. For example, we could run performance and load tests using a tool like Gatling (<https://gatling.io>), security and compliance tests, and resilience tests. In this last case, we could embrace *chaos engineering*, a discipline made popular by Netflix and consisting of making certain parts of the system fail to verify how the rest will react and how resilient the system is to failures. For Java applications, you can have a look at Chaos Monkey for Spring Boot (<https://codecentric.github.io/chaos-monkey-spring-boot>).

NOTE

How about the third quadrant? Following the continuous delivery principles, we strive not to include manual tests in the deployment pipeline. Yet, we usually need them. They are particularly important for software products aimed at end-users like web and mobile applications. Therefore, we run them on the side in the form of exploratory testing and usability testing, so that we ensure more freedom for testers and fewer constraints on the pace and timing required by continuous integration and the deployment pipeline.

An essential feature of the acceptance stage is that all tests are run against a **production-like environment** to ensure the best reliability. The deployment would follow the same procedure and scripts as production and could be tested via dedicated system tests (first quadrant).

If a release candidate passes all the tests in the acceptance stage, it means that it's in a *releasable* state and can be delivered and deployed to production. Figure 15.4 illustrates inputs and outputs for the commit and acceptance stages in a deployment pipeline.

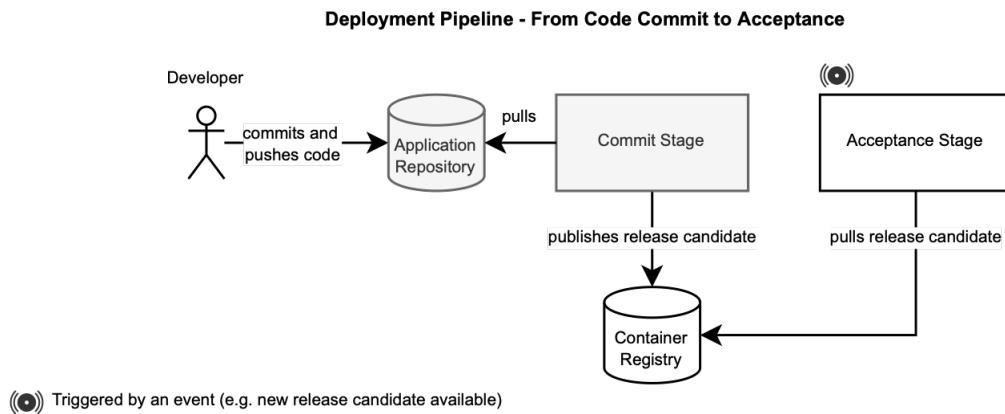


Figure 15.4 The commit stage goes from code commit to release candidate, which goes through the acceptance stage. If it passes all the tests, it's ready for production.

15.1.3 Implementing the acceptance stage with GitHub Actions

In this section, you'll see how to implement a skeleton of workflow for the acceptance stage using GitHub Actions. Throughout the book, we focused on unit and integration tests, which we run in the commit stage. For the acceptance stage, we would need to write functional and non-functional acceptance tests. That's out of the scope for this book, but I still want to show you some principles for designing the workflow using Catalog Service as an example.

Open your Catalog Service project (`catalog-service`) and create a new `acceptance-stage.yml` file within the `.github/workflows` folder. The acceptance stage is triggered whenever a new release candidate is published to the artifact repository. One option for defining such a trigger is listening for the events published by GitHub whenever the commit stage workflow is completed a run.

Listing 15.2 Trigger the acceptance stage after the commit stage completes (acceptance-stage.yml).

```
name: Acceptance Stage ①
on:
  workflow_run: ②
    workflows: ['Commit Stage']
    types: [completed]
    branches: main ③
```

- ① The name of the workflow.
- ② This workflow is triggered when the "Commit Stage" workflow completes a run.
- ③ This workflow runs only on the main branch.

However, that's not enough. Following the continuous integration principles, developers commit often during the day and trigger the commit stage. Since the commit stage is much faster than the acceptance stage, we risk creating a bottleneck. Every time an acceptance stage run is completed, we are not interested in verifying all the release candidates that queued up in the meantime. We are only interested in the newest one whereas the others can be discarded. GitHub Actions provides a mechanism to handle this scenario via concurrency controls.

Listing 15.3 Configures concurrency for the workflow execution (acceptance-stage.yml).

```
name: Acceptance Stage
on:
  workflow_run:
    workflows: ['Commit Stage']
    types: [completed]
    branches: main
  concurrency: acceptance ①
```

- ① Ensures that only one workflow runs at a time.

Next, you would define several jobs to run in parallel against a production-like environment,

accomplishing functional and non-functional acceptance tests. For our example, we'll simply print a message since we haven't implemented the autotests for this stage.

Listing 15.4 Run functional and non-functional acceptance tests (acceptance-stage.yml).

```
name: Acceptance Stage
on:
  workflow_run:
    workflows: ['Commit Stage']
    types: [completed]
    branches: main
    concurrency: acceptance

jobs:
  functional: ①
    name: Functional Acceptance Tests
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    runs-on: ubuntu-22.04
    steps:
      - run: echo "Running functional acceptance tests"
  performance: ①
    name: Performance Tests
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    runs-on: ubuntu-22.04
    steps:
      - run: echo "Running performance tests"
  security: ①
    name: Security Tests
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    runs-on: ubuntu-22.04
    steps:
      - run: echo "Running security tests"
```

- ① The job runs only if the commit stage completed successfully.

NOTE

The acceptance tests could be run against a staging environment that closely resembles production. And the application could be deployed using the staging overlay we configured in the previous chapter.

At this point, push your changes to your GitHub catalog-service repository and have a look at how GitHub will first run the commit stage workflow (triggered by your code commit) and then the acceptance stage workflow (triggered by the commit stage workflow completing successfully). Figure 15.5 shows the result of the acceptance stage workflow execution.

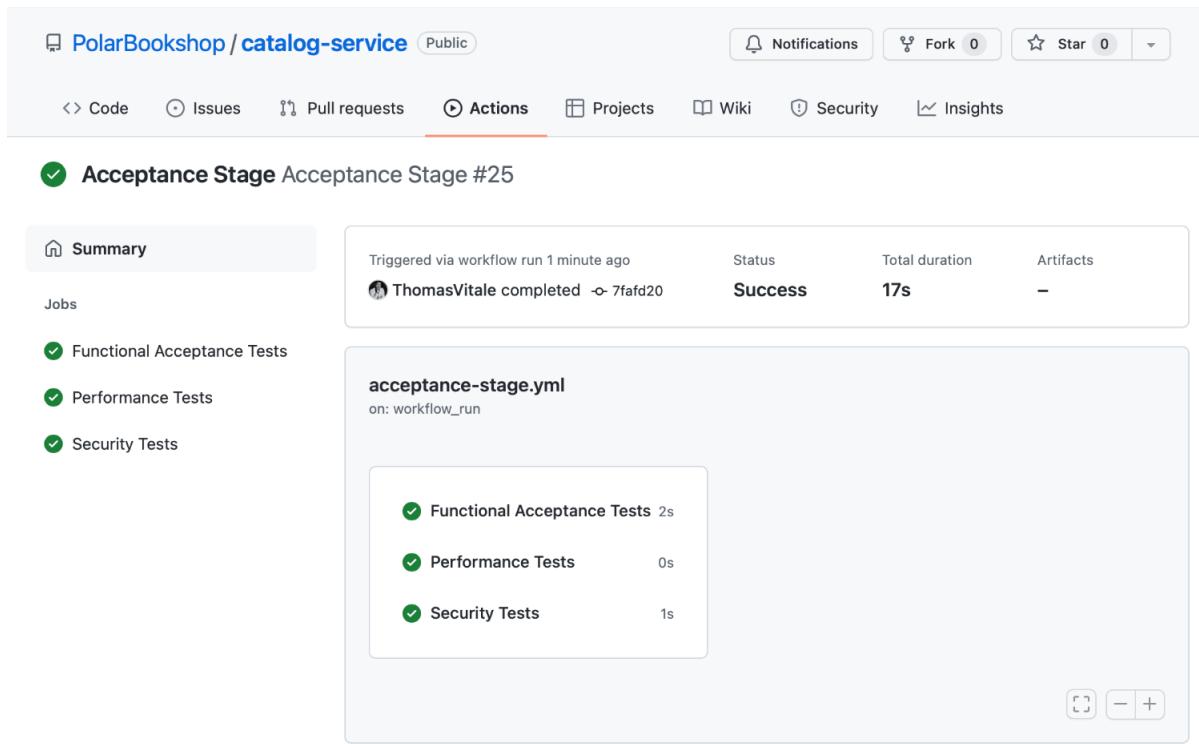


Figure 15.5 The commit stage goes from code commit to release candidate, which then goes through the acceptance stage. If it passes all the tests, it's ready for production.

SIDEBAR

Polar Labs

It's time to apply what you learned in this section to Edge Service, Dispatcher Service, and Order Service.

1. Update the commit stage workflow so that each release candidate is uniquely identified.
2. Push your changes to GitHub, ensure the workflow is completed successfully, and a container image is published to GitHub Container Registry.
3. Create an acceptance stage workflow, push your changes to GitHub, and verify that it is triggered correctly once the commit stage workflow is completed.

In the source code repository accompanying the book, you can check the final result in the Chapter15/15-end folder (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

Deploying to production requires a combination of a release candidate and its configuration. After validating that a release candidate is ready for production, let's see how to customize its configuration.

15.2 Configuring Spring Boot for production

We're getting closer and closer to deploying cloud native applications to a Kubernetes cluster in production. So far, we have worked with local clusters using `minikube`. We now need a full-fledged Kubernetes cluster for our production environment. Before you continue reading this section, follow the instructions in Appendix B (sections B.1 through B.6) to initialize one on the DigitalOcean public cloud. You'll also find some tips if you want to use a different cloud provider.

Once you have a Kubernetes cluster up and running in the cloud, you can continue reading this section that will cover what additional configuration we need to provide our Spring Boot applications before deploying them to the production environment.

In the previous chapter, you learned about Kustomize and the overlay technique to manage customizations for different deployment environments on top of a common base. You also tried your hand at customizing the Catalog Service deployment for a staging environment. In this section, we'll do something similar for production. Extending what you saw in chapter 14, I'll show you how to customize volume mounts for ConfigMaps and Secrets. Also, you'll see how to configure the CPU and memory for containers running in Kubernetes and learn more about how the Paketo Buildpacks manage resources for the JVM within each container.

15.2.1 Defining a configuration overlay for production

First, we need to define a new overlay to customize the deployment of Catalog Service for a production environment. As you probably remember from the previous chapter, the Kustomization base for Catalog Service is stored in the `catalog-service` repository. On the other hand, we keep the overlays in the `polar-deployment` repository.

Go ahead and create a new production folder within `kubernetes/applications/catalog-service` (in the `polar-deployment` repository). We'll use it to store all customizations related to the production environment. Any base or overlay requires a `kustomization.yaml` file. Let's create one for the production overlay. Remember to replace `<your_github_username>` with your GitHub username in lowercase. Also, replace `<release_sha>` with the unique identifier associated with your latest release candidate for Catalog Service. You can retrieve that version from the "Packages" section of your `catalog-service` GitHub repository main page.

Listing 15.5 Define an overlay for production on top of a remote base (production/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources: ①
- github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>
```

- ① The git commit hash (sha) identifying your latest release candidate.

NOTE

I'll assume that all the GitHub repositories we created for Polar Bookshop are publicly accessible. If that's not the case, you can go to the specific repository page on GitHub and access the "Settings" section for that repository. Then, scroll to the bottom of the settings page, and make the package public by clicking the "Change visibility" button.

CUSTOMIZING ENVIRONMENT VARIABLES

The first customization we could apply is an environment variable to activate the `prod` Spring profile for Catalog Service. Following the same approach from the previous chapter, create a `patch-env.yml` file within the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`).

Listing 15.6 A patch for customizing environment variables in a container (production/patch-env.yml).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  template:
    spec:
      containers:
        - name: catalog-service
          env:
            - name: SPRING_PROFILES_ACTIVE ①
              value: prod
```

- ① Defines which Spring profiles should be activated.

Next, we need to instruct Kustomize to apply the patch. In the `kustomization.yml` file for the production overlay of Catalog Service, list the `patch-env.yml` file as follows.

Listing 15.7 Configure Kustomize to apply the patch for environment variables (production/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>

patchesStrategicMerge: ①
- patch-env.yml ②
```

- ① Section containing the list of patches to apply according to the Strategic Merge strategy.
- ② The patch for customizing the environment variables passed to the Catalog Service container.

CUSTOMIZING SECRETS AND VOLUMES

In the previous chapter, you learned how to define ConfigMaps and Secrets, and you saw how to mount them as volumes to Spring Boot containers. In the base Kustomization, we have no Secrets configured since we rely on the same default values from development. In production, we need to pass different URLs and credentials to make it possible for Catalog Service to access the PostgreSQL database and Keycloak.

When we set up the production environment on DigitalOcean earlier, we also created a Secret with the credentials to access the PostgreSQL database (`polar-postgres-catalog-credentials`) and one for Keycloak (`keycloak-issuer-resource-server-secret`). Now, we can mount them as volumes to the Catalog Service container, similar to what we did with ConfigMaps in chapter 14. We'll do that in a dedicated patch.

Create a `patch-volumes.yml` file within the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`) and configure the patch as follows. When Kustomize applies this patch to the base deployment manifests, it will merge the ConfigMap volume defined in the base with the Secret volumes defined in the patch.

Listing 15.8 Mount Secrets as volumes to the Catalog Service container (production/patch-volumes.yml).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  template:
    spec:
      containers:
        - name: catalog-service
          volumeMounts:
            - name: postgres-credentials-volume
              mountPath: /workspace/secrets/postgres ①
            - name: keycloak-issuer-resourceserver-secret-volume
              mountPath: /workspace/secrets/keycloak ②
      volumes:
        - name: postgres-credentials-volume
          secret:
            ③
            secretName: polar-postgres-catalog-credentials
        - name: keycloak-issuer-resourceserver-secret-volume
          secret:
            ④
            secretName: keycloak-issuer-resourceserver-secret
```

- ① Mounts the volume with the Secret containing the PostgreSQL credentials.
- ② Mounts the volume with the Secret containing the Keycloak issuer URL.
- ③ Defines a volume from the Secret containing the PostgreSQL credentials.
- ④ Defines a volume from the Secret containing the Keycloak issuer URL.

Then, just like you learned in the previous section, reference the patch in the `kustomization.yml` file for the production overlay.

Listing 15.9 Configure Kustomize to apply the patch for mounting Secrets as volumes (production/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>

patchesStrategicMerge:
  - patch-env.yml
  - patch-volumes.yml ①
```

- ① Defines a patch for mounting Secrets as volumes.

Currently, the Secrets are configured to be provided to the container, but Spring Boot is not aware of them yet. In the next section, I'll show you how to instruct Spring Boot to load those Secrets as config trees.

CUSTOMIZING CONFIGMAPS

The base Kustomization for Catalog Service instructs Kustomize to generate a catalog-config ConfigMap starting from an `application.yml` file. As you learned in the previous chapter, we can ask Kustomize to add an additional file to that same ConfigMap, `application-prod.yml`, which we know takes precedence over the base `application.yml` file. That's how we're going to customize the application configuration for production.

First, create an `application-prod.yml` file within the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`). We'll use this property file to configure a custom greeting. We also need to instruct Spring Boot to load the Secrets as config trees, using the `spring.config.import` property. For more information on config trees, you can refer to the previous chapter.

Listing 15.10 Production-specific configuration for Catalog Service (production/application-prod.yml).

```
polar:
  greeting: Welcome to our book catalog from a production
  ➔ Kubernetes environment!
spring:
  config:
    import: configtree:/workspace/secrets/* ①
```

- ① Imports configuration from the path where volumes with Secrets are mounted. Make sure you include the final slash, or else the import will fail.

Next, we can rely on the ConfigMap Generator provided by Kustomize to combine the `application-prod.yml` file (defined in the production overlay) with the `application.yml` file (defined in the base kustomization) within the same catalog-config ConfigMap. Go ahead and update the `kustomization.yml` file for the production overlay as follows.

Listing 15.11 Merge property files within the same ConfigMap (production/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>

patchesStrategicMerge:
  - patch-env.yml
  - patch-volumes.yml

configMapGenerator:
  - behavior: merge ①
    files:
      - application-prod.yml ②
    name: catalog-config ③
```

- ① Merges this ConfigMap with the one defined in the base Kustomization.

- ② The additional property file added to the ConfigMap.
- ③ The same ConfigMap name used in the base Kustomization.

CUSTOMIZING IMAGE NAME AND VERSION

The next step is updating the image name and version following the same procedure we used in the previous chapter. This time, we'll be able to use a proper version number for the container image (our release candidate).

First, make sure you have the `kustomize` CLI installed on your computer. You can refer to the instructions at <https://kustomize.io>. If you're on macOS or Linux, you can install `kustomize` with the following command: `brew install kustomize`.

Then, open a Terminal window, navigate to the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`), and run the following command to define which image and version to use for the `catalog-service` container. Remember to replace `<your_github_username>` with your GitHub username in lowercase. Also, replace `<sha>` with the unique identifier associated with your latest release candidate for Catalog Service. You can retrieve that version from the "Packages" section of your `catalog-service` GitHub repository main page.

```
$ kustomize edit set image \
  catalog-service=ghcr.io/<your_github_username>/catalog-service:<sha>
```

The command will automatically update the `kustomization.yml` file with the new configuration, as you can see in the following listing.

Listing 15.12 Configure the image name and version for the Catalog Service container (production/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>

patchesStrategicMerge:
  - patch-env.yml
  - patch-volumes.yml

configMapGenerator:
  - behavior: merge
    files:
      - application-prod.yml
    name: catalog-config

images:
  - name: catalog-service ①
    newName:
      ghcr.io/<your_github_username>/catalog-service ②
      newTag: <release_sha> ③
```

- ① The name of the container as defined in the Deployment manifest.
- ② The new image name for the container (with your GitHub username in lowercase).
- ③ The new tag for the container (with your release candidate unique identifier).

NOTE

Images published to GitHub Container Registry will have the same visibility as the related GitHub code repository. I'll assume that all the images we build for Polar Bookshop are publicly accessible via the GitHub Container Registry. If that's not the case, you can go to the specific repository page on GitHub and access the "Packages" section for that repository. Then, select "Package settings" from the sidebar menu, scroll to the bottom of the settings page, and make the package public by clicking the "Change visibility" button.

Currently, we use the release candidate unique identifier in two places: the URL for the remote base and the image tag. Whenever a new release candidate is promoted to production, we need to remember to update both of them. Even better, we should automate the update. That's what I'll describe later when we'll implement the production stage of the deployment pipeline.

CUSTOMIZING THE NUMBER OF REPLICAS

Cloud native applications are supposed to be highly available, but only one instance of Catalog Service is deployed by default. Similar to what we did for the staging environment, let's customize the number of replicas for the application.

Open the `kustomization.yaml` file within the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`) and define two replicas for the `catalog-service` container.

Listing 15.13 Configure the number of replicas for the Catalog Service container (production/kustomization.yml).

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>

patchesStrategicMerge:
- patch-env.yml
- patch-volumes.yml

configMapGenerator:
- behavior: merge
  files:
  - application-prod.yml
  name: catalog-config

images:
- name: catalog-service
  newName: ghcr.io/<your_github_username>/catalog-service
  newTag: <release_sha>

replicas:
- name: catalog-service ①
  count: 2 ②
```

- ① The name of the Deployment for which to define the number of replicas.
- ② The number of replicas.

NOTE

In a real scenario, you would probably want Kubernetes to dynamically scale applications in and out depending on the current workload rather than providing a fixed number. Dynamic scaling is a pivotal feature of any cloud platform. In Kubernetes, it's implemented by a dedicated component called Horizontal Pod Autoscaler based on well-defined metrics, such as the CPU consumption per container. For more information, refer to the Kubernetes documentation (<https://kubernetes.io/docs>).

The next section will cover how to configure CPU and memory for Spring Boot containers running in Kubernetes.

15.2.2 Configuring CPU and memory for Spring Boot containers

When dealing with containerized applications, it's best to assign **resource limits** explicitly. In chapter 1, you learned that containers are isolated contexts leveraging Linux features like namespaces and cgroups to partition and limit resources among processes. However, suppose you don't specify any resource limit. In that case, each container will have access to the whole CPU set and memory available on the host machine with the risk of some of them taking up more resources than they should and causing other containers to crash due to a lack of resources.

For JVM-based applications like Spring Boot, defining CPU and memory limits is even more critical because they will be used to properly size items like JVM thread pools, heap memory, and non-heap memory. Configuring those values has always been a challenge for Java developers and a critical matter since they directly affect the application performance. Fortunately, if you use the Paketo implementation of Cloud Native Buildpacks included in Spring Boot, you don't need to worry about that. When you packaged the Catalog Service application with Paketo in chapter 6, a **Java Memory Calculator** component got included automatically. When you run the containerized application, that component will configure the JVM memory based on the resource limits assigned to the container. If you don't specify any, the result is unpredictable, which is not what we want.

There's also an economic aspect to consider. If you run your applications in a public cloud, you're usually charged based on how many resources you consume. Consequently, you probably want to be in control of how much CPU and memory each of your containers can use to avoid nasty surprises when the bill arrives.

When it comes to orchestrators like Kubernetes, there's another critical issue related to resources you should consider. Kubernetes schedules Pods to be deployed in any of the cluster nodes. What if a Pod is assigned to a node with insufficient resources to run the container correctly? The solution is to declare the minimum CPU and memory a container needs to operate (**resource requests**). Kubernetes will use that information to deploy a Pod to a specific node only if it can guarantee the container to get at least the requested resources.

Resource requests and limits are defined per container. You can specify both requests and limits in a Deployment manifest. In the base manifests for Catalog Service, we haven't defined any resources to run things more smoothly in a local environment without strict resource requirements. However, production workloads should always contain resource configuration. Let's see how we can do that for the production deployment of Catalog Service.

ASSIGNING RESOURCE REQUESTS AND LIMITS TO A CONTAINER

It shouldn't be a surprise that we'll use a patch to apply CPU and memory configuration to Catalog Service. Create a `patch-resources.yml` file within the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`), and define both requests and limits for the container resources. Even if we're considering a production scenario, I'll use low values to optimize the resource usage in your cluster and avoid incurring additional costs. In a real-world scenario, you might want to analyze more carefully which requests and limits are more appropriate for your use case.

**Listing 15.14 Configure resource requests and limits to the Catalog Service container
(production/patch-resources.yml)**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  template:
    spec:
      containers:
        - name: catalog-service
          resources:
            requests: ①
              memory: 756Mi ②
              cpu: 0.1 ③
            limits: ④
              memory: 756Mi ⑤
              cpu: 2 ⑥
```

- ① Minimum amount of resources required by the container to operate.
- ② The container is guaranteed 756 MiB.
- ③ The container is guaranteed CPU cycles equivalent to 0.1 CPU.
- ④ Maximum amount of resources the container is allowed to consume.
- ⑤ The container can consume 756 MiB at most.
- ⑥ The container can consume CPU cycles equivalent to 2 CPUs at most.

Next, open the `kustomization.yml` file in the production overlay for Catalog Service and configure Kustomize to apply the patch.

Listing 15.15 Apply the patch for configuring resource requests and limits for Catalog Service (production/kustomization.yml).

```

apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - github.com/<your_github_username>/catalog-service/k8s?ref=<release_sha>

patchesStrategicMerge:
  - patch-env.yml
  - patch-resources.yml ①
  - patch-volumes.yml

configMapGenerator:
  - behavior: merge
    files:
      - application-prod.yml
    name: catalog-config

images:
  - name: catalog-service
    newName: ghcr.io/<your_github_username>/catalog-service
    newTag: <release_sha>

replicas:
  - name: catalog-service
    count: 2

```

- ① Configures resource requests and limits.

In listing [15.14](#), memory request and limit are the same, but that's not true for the CPU. The following section will explain the reasoning behind those choices.

OPTIMIZING CPU AND MEMORY FOR SPRING BOOT APPLICATIONS

The amount of CPU available to a container directly affects the startup time of a JVM-based application like Spring Boot. In fact, it leverages as much CPU as available to run the initialization tasks concurrently and reduce the startup time. After the startup phase, the application will use much lower CPU resources.

A common strategy is to define the CPU request (`resources.requests.cpu`) with the amount the application will use under normal conditions so that it's always guaranteed to have the resources required to operate correctly. Then, depending on the system, you could decide to specify a higher CPU limit or omit it entirely (`resources.limits.cpu`) to optimize performance at startup so that the application can use as much CPU as available on the node at that moment.

CPU is a **compressible resource**, meaning that a container can consume as much of it as available. When hitting the limit (either because of `resources.limits.cpu` or because there's no more CPU available on the node), the operating system starts throttling the container process, which keeps running but with possibly lower performance. Since it's compressible, not

specifying the CPU limit can be a valid option sometimes to gain a performance boost. Still, you probably want to consider the specific scenario and evaluate the consequences of such a decision.

Unlike CPU, **memory** is a **non-compressible resource**. If a container hits the limit (either because of `resources.limits.memory` or because there's no more memory available on the node), a JVM-based application will throw the dreadful `OutOfMemoryError`, and the operating system will terminate the container process with an *OOMKilled* (OutOfMemory killed) status. There is no throttling. Setting the correct memory value is, therefore, particularly important. There's no shortcut to inferring the proper configuration: you must monitor the application running under normal conditions. That's true for both CPU and memory.

Once you find a suitable value for how much memory your application needs, I recommend you use it both as a request (`resources.requests.memory`) and as a limit (`resources.limits.memory`). The reason for that is deeply connected to how the JVM works, particularly how the JVM heap memory behaves. Growing and shrinking the container memory dynamically would affect the application performance since the heap memory is dynamically allocated based on the memory available to the container. Using the same value for the request and the limit ensures a fixed amount of memory which is always guaranteed, resulting in better JVM performance. Furthermore, it allows the Java Memory Calculator provided by the Paketo Buildpacks to configure the JVM memory in the most efficient way.

I mentioned the Java Memory Calculator a few times now. The following section will expand on the subject.

CONFIGURING RESOURCES FOR THE JAVA VIRTUAL MACHINE

The Paketo Buildpacks used by the Spring Boot plugin for Gradle/Maven provide a Java Memory Calculator component when building container images for Java applications. Such a component implements an algorithm that has been refined and improved over the years thanks to the Pivotal (now VMware Tanzu) experience with running containerized Java workloads in the cloud.

In a production scenario, the default configuration is a good starting point for most applications. However, it can be too resource-demanding for local development or demos. One way to make the JVM consume fewer resources is by lowering the default 250 JVM thread count for imperative applications. For that reason, we've been using the `BPL_JVM_THREAD_COUNT` environment variable to configure a low number of threads for the two Servlet-based applications in Polar Bookshop: Catalog Service and Config Service. Reactive applications are configured already with fewer threads since they are much more resource-efficient than their imperative counterparts. For that reason, we haven't customized the thread count for Edge Service, Order Service, and Dispatcher Service.

NOTE The Paketo team is working on extending the Java Memory Calculator to provide a low-profile mode which will be helpful when working locally or on low-volume applications. In the future, it will be possible to control the memory configuration mode via a flag rather than tweaking the single parameters. You can find more information about this feature on the GitHub project for Paketo Buildpacks (<http://mng.bz/5Q87>).

The Java Virtual Machine has two main memory areas: heap and non-heap. The Calculator focuses on computing values for the different non-heap memory parts according to a specific formula. The remaining memory resources are assigned to the heap. If the default configuration is not good enough, you can customize it as you prefer. For example, I experienced some memory issues with an imperative application handling session management with Redis. It required more direct memory than what was configured by default. In that case, I used the standard `-XX:MaxDirectMemorySize=50M` JVM setting via the `JAVA_TOOL_OPTIONS` environment variable and increased the maximum size for the direct memory from 10MB to 50MB. If you customize the size of a specific memory region, the Calculator will adjust the allocation of the remaining areas accordingly.

NOTE Memory handling in the JVM is a fascinating topic that would require its own book to be fully understood. Therefore, I won't go into detail regarding how to configure it.

Since we are configuring deployments for production, let's update the thread count for Catalog Service using a more suitable number like 100. In a real-world scenario, I would recommend starting with the default value of 250 as a baseline. For Polar Bookshop, I'm trying to compromise between showing how an actual production deployment would look and minimizing the resources you need to consume (and perhaps pay for) on a public cloud platform.

We can update the thread count for Catalog Service in the patch we defined earlier to customize environment variables. Open the `patch-env.yml` file in the production overlay for Catalog Service (`kubernetes/applications/catalog-service/production`) and update the JVM thread count as follows.

Listing 15.16 Configure the number of JVM threads used by the Java Memory Calculator (production/patch-env.yml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  template:
    spec:
      containers:
        - name: catalog-service
          env:
            - name: BPL_JVM_THREAD_COUNT ①
              value: "100"
            - name: SPRING_PROFILES_ACTIVE
              value: prod
```

- ① The number of threads considered in the memory calculation.

That was the last configuration we needed before being ready to deploy the application in production. We'll do that next.

15.2.3 Deploying Spring Boot in production

Our end goal is to automate the full process from code commit to production. Before looking into the production stage of the deployment pipeline, let's verify that the customizations we defined so far are correct and deploy Catalog Service in production manually.

As you learned in the previous chapter, we can use the Kubernetes CLI to deploy applications on Kubernetes from a Kustomization overlay. Open a Terminal window, navigate to the production overlay folder for Catalog Service (polar-deployment/kubernetes/applications/catalog-service/production), and run the following command to deploy the application via Kustomize.

```
$ kubectl apply -k .
```

You can follow when the two application instances are ready to accept requests by running this command.

```
$ kubectl get pods -l app=catalog-service --watch
```

For additional information on the deployment, you can keep using the Kubernetes CLI or rely on Octant, a tool that lets you visualize your Kubernetes workloads via a convenient GUI. As explained in chapter 7, you can start Octant with the command `octant`. Furthermore, the application logs might be interesting to verify that Catalog Service is running correctly.

```
$ kubectl logs deployment/catalog-service
```

The application is not exposed outside the cluster yet (for that, we need Edge Service), but you can use the port-forward functionality to forward traffic from your local environment on port 9001 to the Service running in the cluster on port 80.

```
$ kubectl port-forward service/catalog-service 9001:80
```

NOTE **The process started by the `kubectl port-forward` command will keep running until you explicitly stop it with `Ctrl+C`.**

Now, you can call Catalog Service from your local machine on port 9001, and the request will be forwarded to the Service object inside the Kubernetes cluster. Open a new Terminal window and call the root endpoint exposed by the application to verify that the `polar.greeting` value specified in the ConfigMap for the `prod` Spring profile is used instead of the default one.

```
$ http :9001/
Welcome to our book catalog from a production Kubernetes environment!
```

That's great! You are officially in production, congratulations! When you're done, you can terminate the port-forwarding with `Ctrl+C`. Finally, delete the deployment by running the following command from the production overlay folder for Catalog Service.

```
$ kubectl delete -k .
```

Kubernetes provides the infrastructure to implement different types of deployment strategies. When we update our application manifests with a new release version and apply them to the cluster, Kubernetes performs a **rolling update**. This strategy consists in incrementally updating Pods instances with new ones and guarantees zero downtime for the user. You saw that in action in the previous chapter.

By default, Kubernetes adopts the rolling update strategy, but there are other techniques that you can employ based on the standard Kubernetes resources or relying on a tool like Knative. For example, you might want to use **blue/green deployments** (<http://mng.bz/WxOI>), consisting of deploying the new version of the software in a second production environment. By doing that, you have the possibility to test one last time that everything runs correctly. When the environment is ready, you move the traffic from the first (*blue*) to the second (*green*) production environment.

Another deployment technique is the **canary release** (<http://mng.bz/8Mz5>). It's similar to the blue/green deployment, but the traffic from the blue to the green environment is moved gradually over time. The goal is to roll out the change to a small subset of users first, perform some verifications, and then do the same for more and more users until everyone is using the new version. Both blue/green deployments and canary releases provide a straightforward way to roll back changes.

NOTE

If you're interested in learning more about deployment and release strategies on Kubernetes, I recommend reading chapter 5 from the "Continuous Delivery for Kubernetes" book by Mauricio Salatino (<https://livebook.manning.com/book/continuous-delivery-for-kubernetes/chapter-5>), published by Manning.

Currently, every time you commit changes, a new release candidate is ultimately published and approved if it passes successfully through the commit and acceptance stage. Then, you need to copy the version number of the new release candidate and paste it into the Kubernetes manifests before being able to update the application in production manually. In the next section, you'll see how to automate that process by implementing the final part of the deployment pipeline: the production stage.

15.3 Deployment pipeline: Production stage

We started implementing a deployment pipeline back in chapter 3, and we have gone a long way since then. So far, we have automated all the steps from code commit up to having a release candidate ready for production. There are still two operations that we have performed manually so far: updating the production scripts with the new application version and deploying it to Kubernetes.

In this section, we'll start looking at the final part of a deployment pipeline, the production stage, and I'll show you how to implement it as a workflow in GitHub Actions.

15.3.1 Understanding the production stage of the deployment pipeline

After a release candidate has gone through the commit and acceptance stages, we are confident enough to deploy it to production. The production stage can be triggered manually or automatically, depending on whether you'd like to achieve **continuous deployment**.

Continuous delivery is "a software development discipline where you build software in such a way that the software can be released to production at any time" (<http://mng.bz/7yXV>). The key part is understanding that the software *can* be released to production, but it doesn't *have to*. That's a common source of confusion between continuous delivery and continuous deployment. If you also want to take the newest release candidate and deploy it to production automatically, then you would have *continuous deployment*.

This stage consists of two main steps:

1. Update the deployment scripts (in our case, the Kubernetes manifests) with the new release version.
2. Deploy the application to the production environment.

NOTE

An optional third step would be to run some final automated tests to verify that the deployment was successful. Perhaps, you could reuse the same system tests that you might have included in the acceptance stage to verify the deployment in a staging environment.

The next section will show you how to implement the first step of the production stage using GitHub Actions, and will discuss some implementation strategies for the second step. We'll aim to automate the whole path from code commit to production and achieve continuous deployment.

15.3.2 Implementing the production stage with GitHub Actions

Compared to the previous stages, implementing the production stage of a deployment pipeline can differ a lot depending on several factors. Let's start by focusing on the first step of the production stage.

At the end of the acceptance stage, we get a release candidate proven to be ready for production. After that, we need to update the Kubernetes manifests in our production overlay with the new release version. When keeping both application source code and deployment scripts in the same repository, the production stage could be listening to a specific event published by GitHub whenever the acceptance stage completes successfully, similar to what we did to configure the flow between the commit stage and acceptance stage.

In our case, we keep the deployment scripts in a separate repository, which means that whenever the acceptance stage workflow completes its execution in the application repository, we need to notify the production stage workflow in the deployment repository. GitHub Actions provide the option of implementing this notification process via a custom event. Let's see how it works.

Open your Catalog Service project (`catalog-service`) and go to the `acceptance-stage.yml` file within the `.github/workflows` folder. After all the acceptance tests run successfully, we have to define a final step to send a notification to the `polar-deployment` repository and ask it to update the Catalog Service production manifests with the new release version. That will be the trigger for the production stage that we'll implement in a moment.

Listing 15.17 Trigger the production stage in the deployment repository upon a new release (acceptance-stage.yml).

```

name: Acceptance Stage
on:
  workflow_run:
    workflows: ['Commit Stage']
    types: [completed]
    branches: main
  concurrency: acceptance

env: ①
  OWNER: <your_github_username>
  REGISTRY: ghcr.io
  APP_REPO: catalog-service
  DEPLOY_REPO: polar-deployment
  VERSION: ${{ github.sha }}

jobs:
  functional:
    ...
  performance:
    ...
  security:
    ...
  deliver:
    name: Deliver release candidate to production
    needs: [ functional, performance, security ] ②
    runs-on: ubuntu-22.04
    steps:
      - name: Deliver application to production
        uses: peter-evans/repository-dispatch@v2 ③
        with:
          token: ${{ secrets.DISPATCH_TOKEN }} ④
          repository:
${{ env.OWNER }}/{{{ env.DEPLOY_REPO }}} ⑤
            event-type: app_delivery ⑥
            client-payload: '{ ⑦
              "app_image":
                "${{ env.REGISTRY }}/{{{ env.OWNER }}}/{{{ env.APP_REPO }}}",
              "app_name": "${{ env.APP_REPO }}",
              "app_version": "${{ env.VERSION }}"
            }'

```

- ① Defines relevant data as environment variables.
- ② Runs only when all functional and non-functional acceptance tests are completed successfully.
- ③ An action to send an event to another repository and trigger a workflow.
- ④ A token to grant the action permission to send events to another repository.
- ⑤ The repository to notify.
- ⑥ A name to identify the event. It's up to you.
- ⑦ The payload of the message sent to the other repository. Add here any information the other repository might need to perform its operations.

With this new step, if no error is found during the execution of the acceptance tests, then a notification is sent to the polar-deployment repository to trigger an update for Catalog Service.

By default, GitHub Actions doesn't allow triggering workflows located in other repositories, even if they both belong to you or your organization. Therefore, we need to provide the `repository-dispatch` action with an access token that grants it such permissions. The token can be a Personal Access Token (PAT), a GitHub tool that we have already used in chapter 6.

Go to your GitHub account, navigate to *Settings > Developer Settings > Personal access token*, and choose "Generate new token". Input a meaningful name, and assign it the workflow scope to give the token permissions to trigger workflows in other repositories (figure 15.6). Finally, generate the token and copy its value. GitHub will show you the token value only once. Make sure you save it since you'll need it soon.

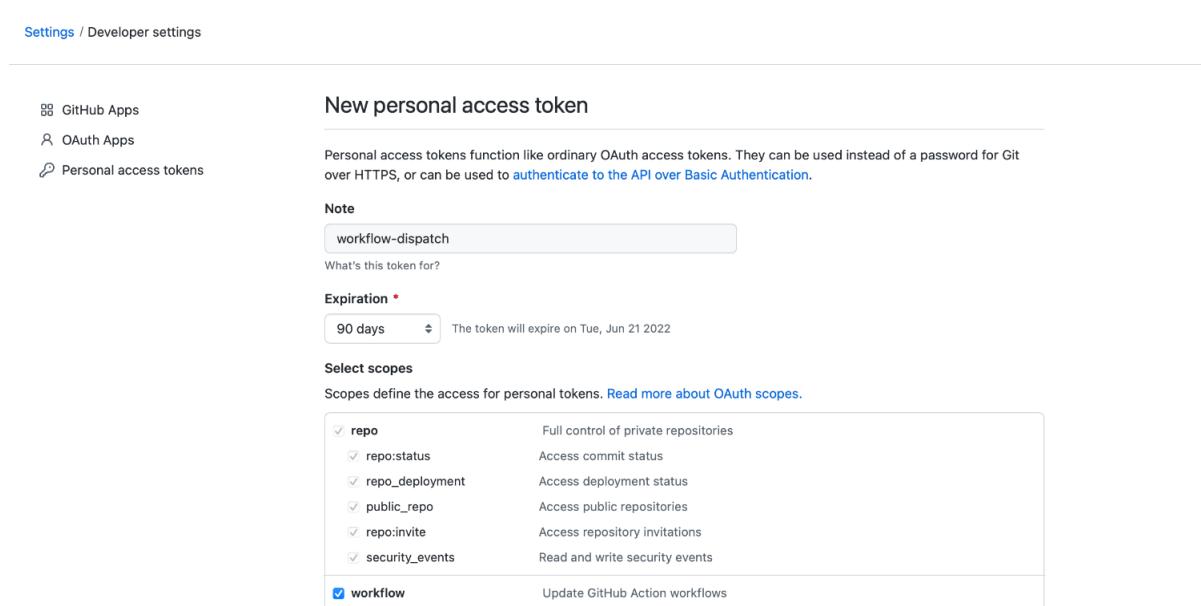


Figure 15.6 A Personal Access Token (PAT) granting permissions to trigger workflows in other repositories.

Next, go to your Catalog Service repository on GitHub, navigate to the "Settings" tab, and then *Secrets > Actions*. On that page, choose "New repository secret", name it `DISPATCH_TOKEN` (the same name we used in listing 15.17), and input the value of the PAT you generated earlier. Using the Secrets feature provided by GitHub, we can provide the PAT securely to the acceptance stage workflow.

WARNING

As explained in chapter 3, when using actions from the GitHub marketplace, you should handle them like any other third-party application and manage the security risks accordingly. In the acceptance stage, we provide an access token to a third-party action with permissions to manipulate repositories and workflows. You shouldn't do that light-heartedly. In this case, I trusted the author of the action and decided to trust the action with the token.

Don't commit your changes to the `catalog-service` repository yet. We'll do that later. At this point, we have implemented the trigger for the production stage, but we haven't initialized the final stage yet. Let's move to the Polar Deployment repository and do that.

Open your Polar Deployment project (`polar-deployment`) and create a `production-stage.yml` file within a new `.github/workflows` folder. The production stage is triggered whenever the acceptance stage from an application repository dispatches an `app_delivery` event. The event itself contains contextual information about the application name, image, and version for the newest release candidate. Since the application-specific information is parametrized, we can use this workflow for all the applications of the Polar Bookshop system, not only Catalog Service.

The first job of the production stage is updating the production Kubernetes manifests with the new release. This job will consist of three steps:

1. Check out the `polar-deployment` source code.
2. Update the production Kustomization with the new version for the given application.
3. Commit the changes to the `polar-deployment` repository.

We can implement those three steps as follows.

Listing 15.18 Update the image version upon a new application delivery (production-stage.yml).

```

name: Production Stage

on:
  repository_dispatch: ①
    types: [app_delivery]

jobs:
  update:
    name: Update application version
    runs-on: ubuntu-22.04
    permissions:
      contents: write
    env: ②
      APP_IMAGE: ${{ github.event.client_payload.app_image }}
      APP_NAME: ${{ github.event.client_payload.app_name }}
      APP_VERSION: ${{ github.event.client_payload.app_version }}
    steps:
      - name: Checkout source code
        uses: actions/checkout@v3 ③
      - name: Update image version
        run: |
          cd \ ④
          kubernetes/applications/${{ env.APP_NAME }}/production
          kustomize edit set image \ ⑤
            ${{ env.APP_NAME }}=${{ env.APP_IMAGE }}:${{ env.APP_VERSION }}
          sed -i 's/ref=[\w+]/${{ env.APP_VERSION }}/' \
            kustomization.yaml ⑥
      - name: Commit updated manifests
        uses: stefanzweifel/git-auto-commit-action@v4 ⑦
        with:
          commit_message: "Update ${{ env.APP_NAME }}"
    ➔ to version ${{ env.APP_VERSION }}"
        branch: main

```

- ① Executes the workflow only when a new "app_delivery" event is received, dispatched from another repository.
- ② Saves the event payload data as environment variables for convenience.
- ③ Checks out the repository.
- ④ Navigates to the production overlay for the given application.
- ⑤ Updates the image name and version via Kustomize for the given application.
- ⑥ Updates the tag used by Kustomize to access the correct base manifests stored in the application repository.
- ⑦ An action to commit and push the changes applied to the current repository from the previous step.
- ⑧ Details about the commit operation.

That's all we need for now. Commit and push the changes to your remote `polar-deployment` on GitHub. Then, move again to your Catalog Service project, commit your previous changes to the acceptance stage, and push them to your remote `catalog-service` on GitHub.

The new commit to the `catalog-service` repository will trigger the deployment pipeline. First, the commit stage will produce a container image (our release candidate) and publish it to GitHub Container Registry. Then, the acceptance stage will fictitiously run further tests on the application and finally send a notification (a custom `app_delivery` event) to the `polar-deployment` repository. The event triggers the production stage that will update the production Kubernetes manifests for Catalog Service and commit the changes to the `polar-deployment` repository. Figure 15.7 illustrates inputs and outputs for the three stages of the deployment pipeline.

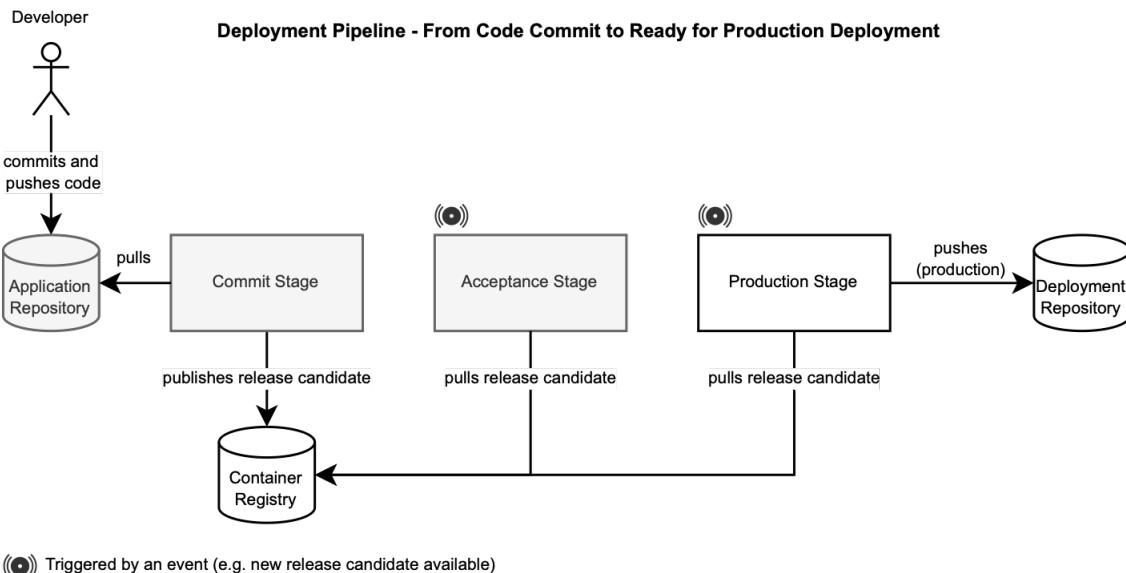


Figure 15.7 The commit stage goes from code commit to release candidate, which goes through the acceptance stage. If it passes all the tests, the production stage updates the deployment manifests.

Go to your GitHub projects and follow the execution of the three stages. In the end, you'll find a new commit in your `polar-deployment` repository, which was submitted by GitHub Actions and contains a change to the Catalog Service production overlay to use the newest release version.

Perfect! We just got rid of the first of the two remaining manual steps: updating the deployment scripts with the newest release version. We still have to apply the Kubernetes manifests to the cluster manually, using the Kubernetes CLI. The second step of the production stage will take care of automating the application deployment whenever a new version is promoted to production. That's the topic of the next section, which will cover continuous deployment with GitOps.

SIDE BAR**Polar Labs**

It's time to apply what you learned in this section to Edge Service, Dispatcher Service, and Order Service.

1. Generate a PAT with a workflow scope for each application. It's a security best practice not to reuse tokens for multiple purposes.
2. For each application, save the PAT as a Secret from the GitHub repository page.
3. Update the acceptance stage workflow with a final step sending a notification to the production stage with information about the newest release candidate.
4. Push your changes to GitHub, ensure the workflow is completed successfully and that the production stage workflow in the `polar-deployment` repository is triggered correctly.

Edge Service is the only application available through the public Internet and requires an additional patch to configure the Ingress to block requests to the Actuator endpoints from outside the cluster. You can get the additional patch from the `applications/edge-service/production` folder, within Chapter15/15-end/polar-deployment.

For simplicity, we accept that the Actuator endpoints are available without authentication from within the cluster. Internal applications like Catalog Service are not affected since their Actuator endpoints are not accessible through Spring Cloud Gateway. On the other hand, the Edge Service ones are currently accessible via the public Internet. That's not safe in a production environment. A simple way of fixing that is configuring the Ingress to block any request to the `/actuator/**` endpoints from outside the cluster. They will all still be available from within the cluster so that the health probes can work. We are using an NGINX-based Ingress Controller, so we can use its configuration language to express a deny rule for the Actuator endpoints.

In the source code repository accompanying the book, you can check the final result in the Chapter15/15-end folder (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

15.4 Continuous deployment with GitOps

Traditionally, continuous deployment is implemented by adding a further step to the production stage of the deployment pipeline. This additional step would authenticate with the target platform (for example, a virtual machine or a Kubernetes cluster) and deploy the new version of the application. In recent years, a different approach has become more and more popular: GitOps. The term was coined by Alexis Richardson, CEO and founder of Weaveworks (<https://www.weave.works>).

GitOps is a set of practices for operating and managing software systems, enabling continuous delivery and deployment while ensuring agility and reliability. Compared to the traditional approach, GitOps favors decoupling between delivery and deployment. Instead of having the pipeline *pushing* deployments to the platform, it's the platform itself *pulling* the desired state from a source repository and performing deployments. In the first case, the deployment step is implemented within the same production stage workflow. In the second case, which will be our focus, we still consider the step part of the production stage from a theoretical point of view, but the implementation differs. Even if GitOps doesn't enforce specific technologies, it's best implemented with Git and Kubernetes. That will be our focus.

The GitOps Working Group, part of the CNCF, defines GitOps in terms of four principles (<https://opengitops.dev>):

1. **Declarative.** "A system managed by GitOps must have its desired state expressed declaratively."
 - Working with Kubernetes, we can express the desired state via YAML files (manifests).
 - Kubernetes manifests declare what we want to achieve, not how. The platform is responsible for finding a way to achieve the desired state.
2. **Versioned and Immutable.** "Desired state is stored in a way that enforces immutability, versioning and retains a complete version history."
 - Git is the preferred choice for ensuring the desired state is versioned and the whole history retained. That makes it possible, among other things, to roll back to a previous state with ease.
 - The desired state stored in Git is immutable and represents the single source of truth.
3. **Pulled Automatically.** "Software agents automatically pull the desired state declarations from the source."
 - Examples of software agents (*GitOps agents*) are Flux (<https://fluxcd.io>), Argo CD (<https://argoproj.github.io/cd>), and kapp-controller (<https://carvel.dev/kapp-controller>).
 - Rather than granting CI/CD tools like GitHub Actions full access to the cluster or running commands manually, we grant the GitOps agent access to a source like Git so that it pulls changes automatically.
4. **Continuously Reconciled.** "Software agents continuously observe actual system state and attempt to apply the desired state."
 - Kubernetes is composed of controllers that keep observing the system and ensuring the actual state of the cluster matches the desired state.
 - On top of that, GitOps ensures that it's the right desired state to be considered in the cluster. Whenever a new change is detected in the Git source, the agent steps up and reconciles the desired state with the cluster.

Figure 15.8 illustrates the result of applying the GitOps principles.

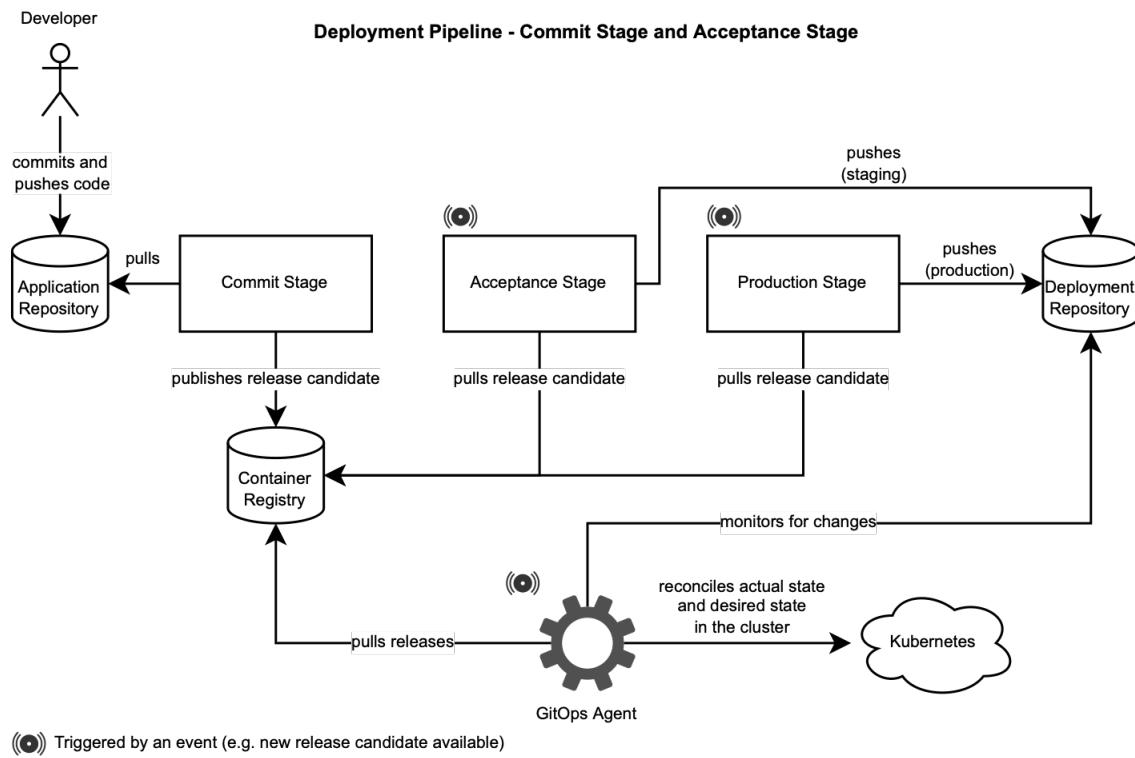


Figure 15.8 Every time the production stage workflow updates the deployment repository, the GitOps controller reconciles desired and actual state.

If we consider the four principles, we can notice that we applied the first two already. We expressed the desired state for our applications declaratively using Kubernetes manifests and Kustomize. And we stored the desired state in a Git repository on GitHub (`polar-deployment`), making it versioned and immutable. We are still missing a software agent that automatically pulls the desired state declarations from the Git source and continuously reconciles them inside the Kubernetes cluster, therefore achieving continuous deployment.

We'll start by installing Argo CD (<https://argo-cd.readthedocs.io>), a GitOps software agent. Then, we'll configure it to complete the final step of the deployment pipeline and let it monitor our `polar-deployment` repository. Whenever there's a change in the application manifests, Argo CD will apply the changes to our production Kubernetes cluster.

15.4.1 Implementing GitOps with Argo CD

Let's start by installing the Argo CD CLI. Refer to the project website for installation instructions (<https://argo-cd.readthedocs.io>). If you are on macOS or Linux, you can use Homebrew as follows.

```
$ brew install argocd
```

We'll use the CLI to instruct Argo CD about the Git repository to monitor and configure it to apply changes to the cluster to achieve continuous deployment automatically. But first, we need

to deploy Argo CD to the production Kubernetes cluster.

NOTE

I assume your Kubernetes CLI is still configured to access the production cluster on DigitalOcean. You can check that with `kubectl config current-context`. If you need to change the context, you can run `kubectl config use-context <context-name>`. A list of all the contexts available can be retrieved from `kubectl config get-contexts`.

Open a Terminal window, go to your Polar Deployment project (`polar-deployment`), and navigate to the folder `kubernetes/platform/production/argocd`. You should have copied that folder over to your repository when you set up the production cluster. If that's not the case, please do so now from the source code repository accompanying this book (Chapter15/15-end/polar-deployment/platform/production/argocd).

Then, run the following script to install Argo CD into the production cluster. Feel free to open the file and look at the instructions before running it.

```
$ ./deploy.sh
```

TIP

You might need to make the script executable first with the command `chmod +x deploy.sh`.

The deployment of Argo CD consists of several components, including a convenient web interface where you can visualize and control all the deployments controlled by Argo CD. For now, we'll use the CLI. During the installation, Argo CD has autogenerated a password for the admin account (the username is `admin`). Run the following command to fetch the password value (it takes a few seconds before the value is available).

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
-o jsonpath="{.data.password}" | base64 -d; echo
```

Next, let's find out the external IP address assigned to Argo CD server. The platform might take a few minutes to provision a load balancer for Argo CD. During the provisioning, the `EXTERNAL-IP` column will show a `<pending>` status. Wait and try again until an IP address is shown. Note it down since we're going to use it soon.

```
$ kubectl -n argocd get service argocd-server
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
argocd-server	LoadBalancer	10.245.16.74	<external-ip>

Since the Argo CD server is now exposed via a public load balancer, we can use the external IP address to access its services. For this example, we'll use the CLI, but you can achieve the same by opening `<argocd-external-ip>` in a browser window (where `<argocd-external-ip>` is

the IP address assigned to your Argo CD server). Either way, you'll have to log in with the auto-generated admin account. The username is `admin`. The password is the one you fetched earlier. Be aware that you might get a warning since we are not using HTTPS.

```
$ argocd login <argocd-external-ip>
```

It's time to see continuous deployment in action with GitOps. I assume you have been through all the previous sections of this chapter. At this point, the commit stage of your Catalog Service repository on GitHub (`catalog-service`) should have built a container image, the acceptance stage should have triggered the Polar Deployment repository on GitHub (`polar-deployment`), and the production stage should have updated the production layer for Catalog Service with the newest release version (`polar-deployment/kubernetes/applications/catalog-service/production`). Now, we'll configure Argo CD to monitor the production layer for Catalog Service and synchronize it with the production cluster whenever it detects a change in the repository. In other words, Argo CD will continuously deploy new versions of Catalog Service as made available by the deployment pipeline.

```
$ argocd app create catalog-service \ ①
--repo \ ②
https://github.com/<your_github_username>/polar-deployment.git \
--path kubernetes/applications/catalog-service/production \ ③
--dest-server https://kubernetes.default.svc \ ④
--dest-namespace default \ ⑤
--sync-policy auto \ ⑥
--auto-prune ⑦
```

- ① Creates a "catalog-service" application in Argo CD.
- ② The Git repository to monitor for changes. Insert your GitHub username.
- ③ The folder to monitor for changes within the configured repository.
- ④ The Kubernetes cluster where to deploy the application. We use the default cluster configured in the "kubectl" context.
- ⑤ The namespace where to deploy the application. We use the "default" namespace.
- ⑥ Configures Argo CD to automatically reconcile the desired state in the Git repo with the actual state in the cluster.
- ⑦ Configures Argo CD to delete old resources after a synchronization automatically.

You can verify the status of the continuous deployment of Catalog Service with the following command (I have filtered the result for the sake of clarity). Argo CD has automatically applied the production overlay for Catalog Service (`polar-deployment/kubernetes/applications/catalog-service/production`) to the cluster.

```
$ argocd app get catalog-service
```

GROUP	KIND	NAMESPACE	NAME	STATUS	HEALTH
	ConfigMap	default	catalog-config-6d5dkt7577	Synced	
	Service	default	catalog-service	Synced	Healthy
apps	Deployment	default	catalog-service	Synced	Healthy

Once all the resources listed by the previous command have the `Synced` status, let's verify that the application is running correctly. The application is not exposed outside the cluster yet, but you can use the port-forward functionality to forward traffic from your local environment on port 9001 to the Service running in the cluster on port 80.

```
$ kubectl port-forward service/catalog-service 9001:80
```

Next, call the root point exposed by the application. We expect to get the value we configured for the `polar.greeting` property in the Catalog Service production overlay.

```
$ http :9001/
Welcome to our book catalog from a production Kubernetes environment!
```

Perfect! In one step, we automated not only the first deployment but also any future update. Argo CD will detect any change in the production overlay for Catalog Service and apply the new manifests to the cluster immediately. There could be a new release version to deploy, but it could also be a change to the production overlay. For example, let's try configuring a different value for the `polar.greeting` property.

Open your Polar Deployment project (`polar-deployment`), go to the production overlay for Catalog service (`kubernetes/applications/catalog-service/production`), and update the value of the `polar.greeting` property in the `application-prod.yml` file.

Listing 15.19 Update the production-specific configuration for Catalog Service (production/application-prod.yml).

```
polar:
  greeting: Welcome to our production book catalog
  synchronized with Argo CD!
spring:
  config:
    import: configtree:/workspace/secrets/*
```

Then, commit and push the changes to your remote `polar-deployment` repository on GitHub. By default, Argo CD checks the Git repository for changes every 3 minutes. It will notice the change and apply the Kustomization again, resulting in a new ConfigMap being generated by Kustomize and in a rolling restart of the Pods to refresh the configuration. Once the deployment in the cluster is in sync with the desired state in the Git repo (you can check with `argocd app get catalog-service`), call again the root endpoint exposed by Catalog Service. We expect to

get the value we have just updated. If you get a network error, it might be that the port-forward process got interrupted. Run `kubectl port-forward service/catalog-service 9001:80` again to fix it.

```
$ http :9001/
Welcome to our production book catalog synchronized with Argo CD!
```

Great! We finally achieved continuous deployment! Pause for a minute and celebrate with a beverage of your choice, you deserve it!

SIDE BAR

Polar Labs

It's time to apply what you learned in this section to Edge Service, Dispatcher Service, and Order Service.

1. Using the Argo CD CLI, register each of the remaining applications similar to what we did for Catalog Service. Remember to authenticate to Argo CD first, as explained earlier.
2. For each application, verify that Argo CD has synchronized the desired state from the `polar-deployment` repository with the actual state in the cluster.

In case of problems with Argo CD, you can use the `argocd app get catalog-service` command to verify the synchronization status or directly use the web interface available at `<argocd-external-ip>`. For troubleshooting Kubernetes resources, you can take advantage of Octant or use one of the techniques explained in the last section of chapter 7.

15.4.2 Putting it all together

If you followed along and completed all the "Polar Labs", at this point, you have the whole Polar Bookshop system up and running in a production Kubernetes cluster in the public cloud. That's a huge accomplishment! In this section, we'll give it a try and refine a few last points. Figure 15.9 shows the status of the applications from the Argo CD GUI accessible via the `<argocd-external-ip>` discovered earlier.

Figure 15.9 The Argo CD GUI shows an overview of all the applications managed via a GitOps flow.

So far, we have worked with Catalog Service, an internal application that is not exposed outside the cluster. For that reason, we relied on the port-forward functionality to test it out. Now that the whole system is deployed, we can access the applications as intended: via the Edge Service. The platform automatically configures a load balancer with an external IP address whenever we deploy an Ingress resource. Let's discover the external IP address for the Ingress sitting in front of Edge Service.

```
$ kubectl get ingress
NAME      CLASS   HOSTS    ADDRESS          PORTS   AGE
polar-ingress  nginx * <ip-address>  80      31m
```

Using the Ingress external IP address, you can use Polar Bookshop from the public Internet. Open a browser window and navigate to `<ip-address>`.

Try logging in as Isabelle. Feel free to add some books and browse the catalog. Then, log out and log in again, this time as Bjorn. Verify you can't create nor edit books, but you can place orders.

When you are done testing the application using the two accounts, log out and ensure you can't access the Actuator endpoints by visiting, for example, `<ip-address>/actuator/health`. NGINX, the technology that powers the Ingress Controller, will reply with a 403 response.

NOTE If you'd like to provisioning the Grafana observability stack, refer to the instructions in the source code repository accompanying the book.

Great job! When you're done using the production cluster, follow the last section of Appendix B to delete all the cloud resources from DigitalOcean. That's fundamental to avoid incurring

unexpected costs.

15.5 Summary

- The idea behind continuous delivery is that an application is always in a releasable state.
- When the delivery pipeline completes its execution, we obtain an artifact (the container image) we can use to deploy the application in production.
- When it comes to continuous delivery, each release candidate should be uniquely identifiable.
- Using the Git commit hash, we ensure uniqueness, traceability, and automation. Semantic versioning can be used as the *display name* communicated to users and customers.
- At the end of the commit stage, a release candidate is delivered to the artifact repository. Next, the acceptance stage deploys the application in a production-like environment and runs functional and non-functional tests. If they all succeed, the release candidate is ready for production.
- The Kustomize approach to configuration customization is based on the concepts of bases and overlays. Overlays are built on top of base manifests and customized via patches.
- You saw how to define patches for customizing environment variables, Secrets mounted as volumes, CPU and memory resources, ConfigMaps, and Ingress.
- The final part of a deployment pipeline is the production stage, where the deployment manifests are updated with the newest release version and ultimately deployed.
- Deployment can be push-based or pull-based.
- GitOps is a set of practices for operating and managing software systems.
- GitOps is based on four principles according to which a system deployment should be declarative, versioned and immutable, pulled automatically, and continuously reconciled.
- Argo CD is a software agent running in a cluster that automatically pulls the desired state from a source repository and applies it to the cluster whenever the two states diverge. That's how we implemented continuous deployment.

16

Serverless, GraalVM and Knative

This chapter covers

- Producing native images with Spring Native and GraalVM
- Building serverless applications with Spring Cloud Function
- Deploying serverless applications with Knative and Kubernetes

In the previous chapter, you completed a long journey from development to production. You built cloud native applications using Spring and deployed them on a Kubernetes cluster in a public cloud. This final chapter aims to provide you with some additional tools to get even more out of your cloud native applications.

One significant benefit of cloud infrastructures is increasing or reducing resources on-demand and paying only for what you use. Java applications have traditionally been very resource-intensive, resulting in higher CPU and memory consumption than other stacks like Go. Not anymore. Using GraalVM and Spring Native, you can compile your Spring Boot applications to native executables, which are more performant and efficient than their JVM counterpart. The first part of this chapter will guide you through how to take advantage of this new technology.

The second part of the chapter will expand on serverless architectures. Compared to CaaS and PaaS infrastructures, serverless architectures move most operational tasks to the platform and let developers focus on the applications. Some applications are naturally event-driven and aren't always busy processing requests. Or they might have sudden peaks which require more computational resources. Serverless platforms provide fully-managed auto-scaling features and can scale application instances to zero so that you don't have to pay anything if there's nothing

to process. You'll learn more about the serverless model. Then, you'll build a serverless application using Spring Native and Spring Cloud Function. Finally, you'll see how to deploy applications using Knative, a Kubernetes-based serverless platform.

NOTE

The source code for the examples in this chapter is available in the Chapter16/16-begin and Chapter16/16-end folders, containing the initial and final state of the project. (<https://github.com/ThomasVitale/cloud-native-spring-in-action>)

16.1 Native images with Spring Native and GraalVM

One of the reasons why Java applications became widely popular was the common platform (the Java Runtime Environment), allowing developers to "write them once, run them everywhere", no matter the operating system. That comes from the way applications are compiled. Rather than compiling the application code directly into machine code (the one understood by operating systems), the Java compiler produces byte code that a dedicated component (the Java Virtual Machine) runs. During the execution, the JRE interprets the byte code into machine code dynamically, allowing the same application executable to run on any machine/OS where a JVM is available. That is called a **Just-In-Time (JIT) compilation**.

Applications running on the JVM are subjected to startup and footprint costs. The startup phase used to be quite long for traditional applications, for which it could even take several minutes. Standard cloud native applications have a much faster startup phase: a few seconds rather than a few minutes. It's good enough for most scenarios, but it can become a serious issue for serverless workloads that are required to start almost instantaneously.

Standard Java applications also have a higher footprint cost than other stacks like Go. Cloud services are usually based on a pay-per-use model. Reducing CPU and memory footprint would mean cutting the costs down. This section will show you how to address this issue using GraalVM and Spring Native.

16.1.1 Understanding GraalVM and native images

So far, you have used the JVM and tools provided by OpenJDK, which comes in many distributions, such as Eclipse Adoptium (previously known as AdoptOpenJDK), BellSoft Liberica JDK, and Microsoft OpenJDK. GraalVM is a newer distribution from Oracle based on OpenJDK and is "designed to accelerate the execution of applications written in Java and other JVM languages" (<https://www.graalvm.org>).

By replacing a standard OpenJDK distribution with GraalVM as the runtime environment for your Java applications, you can increase their performance and efficiency thanks to a new optimized technology to perform JIT compilation (the GraalVM compiler). GraalVM also

provides runtimes to execute code written in other languages like JavaScript, Python, and R. You can even write polyglot applications, for example, including Python scripts in your Java code.

GraalVM offers two primary operational modes. The **JVM Runtime** mode lets you run your Java applications like any other OpenJDK distribution while improving performance and efficiency thanks to the GraalVM compiler. What makes GraalVM so innovative and popular in the serverless context is the **Native Image** mode. Rather than compiling your Java code into byte code and relying on a JVM to interpret it and convert it to machine code, GraalVM offers a new technology (the Native Image builder) to compile Java applications directly into machine code, obtaining a *native executable* or *native image* that contains the whole machine code necessary for its execution.

Java applications compiled as native images have faster startup time, optimized memory consumption, and instant peak performance compared to the JVM option. GraalVM builds them by changing the way applications are compiled. Instead of a JIT-compiler optimizing and producing machine code at runtime, the *Native Image* mode is based on **Ahead-Of-Time (AOT) compilation**. Starting from the `main()` method, all classes and methods that are reachable during the application execution are statically analyzed at build-time and compiled into a standalone binary executable, including any dependency and library. Such an executable doesn't run on a JVM but directly on the machine, just like C or C++ applications.

When using native images, much of the work that used to be performed at runtime by the JVM is now done at build time. As a result, building an application into a native executable takes longer and requires more computational resources than the JVM option. The GraalVM AOT compiler does not support some Java features out of the box. For example, reflection, dynamic proxies, serialization, and dynamic class loading require extra configuration to help the AOT compiler understand how to analyze them statically.

How can we adapt existing Java applications to run as native images? How much configuration is required to support frameworks and libraries? How can we provide the necessary configuration for the AOT compiler? That's where Spring Native enters the scene.

16.1.2 Introducing GraalVM support for Spring Boot with Spring Native

Spring Native is a new project introduced to support compiling Spring Boot applications with GraalVM. The main goal of Spring Native is to make it possible to compile any Spring application into a native executable using GraalVM without any code change. To achieve that goal, the project provides an AOT infrastructure (invoked from a dedicated Gradle/Maven plugin) for contributing all the required configurations for GraalVM to AOT-compile Spring classes. The project is one of the latest additions to the Spring portfolio and is currently in beta. At the time of writing, most Spring libraries are supported, as well as common libraries like Hibernate, Lombok, and gRPC.

For Spring libraries not yet supported or for your own code, Spring Native offers helpful tools to configure the GraalVM compiler. For example, if you use reflection or dynamic proxies in your code, GraalVM would require a dedicated configuration to know how to AOT-compile it. Spring Native offers convenient annotations like `@NativeHints` and `@TypedHint` to instruct the GraalVM compiler directly from your Java code, taking advantage of the IDE auto-completion features and type checks.

NOTE

Spring Native will go out of the beta phase and become part of the core Spring libraries starting with Spring Framework 6 and Spring Boot 3, which are expected to be released in December 2022.

In this section, we'll explore the features of Spring Native by building Quote Service, a web application exposing an API to fetch quotes from books.

BOOTSTRAPPING A NEW PROJECT WITH SPRING NATIVE AND SPRING REACTIVE WEB

You can initialize the Quote Service project from Spring Initializr (<https://start.spring.io>), store the result in a new quote-service Git repository, and push it to GitHub. The parameters for the initialization are shown in figure 16.1.

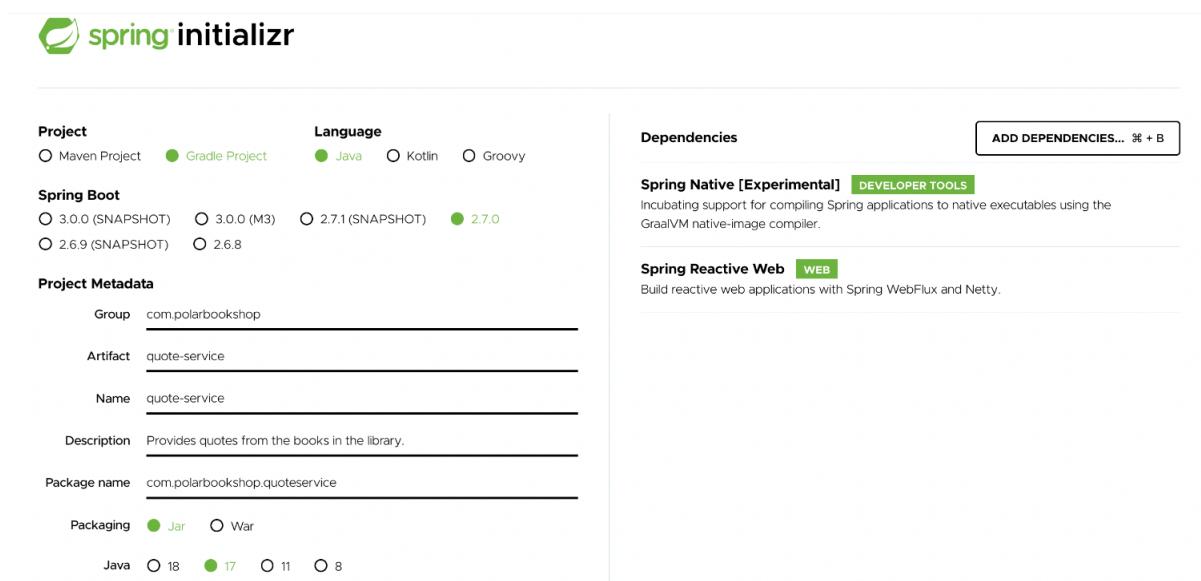


Figure 16.1 The parameters for initializing the Quote Service project.

The project contains the following main dependencies:

- **Spring Reactive Web** provides the necessary libraries for building reactive web applications with Spring WebFlux and includes Netty as the default embedded server.
- **Spring Native** supports compiling Spring applications to native executables using the GraalVM native-image compiler.

The resulting dependencies section of the `build.gradle` file looks as follows.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
}
```

At this point, you might ask: where is the Spring Native dependency? There isn't one. Where is Spring Native? The answer can be found in the `plugins` sections of the `build.gradle` file.

```
plugins {
    id 'org.springframework.boot' version '2.7.0'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
    id 'org.springframework.experimental.aot' version '0.12.0' ①
}
```

- ① Spring AOT plugin provided by Spring Native.

When you add Spring Native to a project, you'll get the Spring AOT plugin which provides the required configuration for GraalVM to compile the Spring classes as well as convenient functionality to build native executables from Gradle (or Maven).

If you bootstrap a new project from Spring Initializr, you'll also get additional information in the `HELP.md` file about how to use Spring Native. Should you select any unsupported dependency, you'll find a message warning you about it. For example, Spring Cloud Stream is not fully supported at the moment of writing. If you initialize a project with Spring Native and Spring Cloud Stream, the `HELP.md` file will show you a message like the following:

The following dependency is not known to work with Spring Native: 'Cloud Stream'. As a result, your application may not work as expected.

NOTE

You can follow the status of which Spring libraries are supported in the Spring Native official documentation (<https://docs.spring.io/spring-native/docs/current/reference/htmlsingle>).

Next, let's implement the business logic for Quote Service.

IMPLEMENTING THE BUSINESS LOGIC

Quote Service will return random book quotes through a REST API. First, create a new `com.polarbookshop.quoteservice.domain` package and define a `Quote` record to model the domain entity.

Listing 16.1 Define a domain entity for representing book quotes (Quote.java)

```
public record Quote (
    String content,
    String author,
    Genre genre
){}
```

Quotes are categorized by the genre of the book from which they are extracted. Add a `Genre` enum to model this classification.

Listing 16.2 Define an enumeration to represent book genres (Genre.java)

```
public enum Genre {
    ADVENTURE,
    FANTASY,
    SCIENCE_FICTION
}
```

Finally, implement the business logic to retrieve book quotes in a new `QuoteService` class. Quotes will be defined and stored in a static in-memory list.

Listing 16.3 The business logic to query book quotes (QuoteService.java)

```
@Service
public class QuoteService {
    private static final Random random = new Random();
    private static final List<Quote> quotes = List.of( ①
        new Quote("Content A", "Abigail", Genre.ADVENTURE),
        new Quote("Content B", "Beatrix", Genre.ADVENTURE),
        new Quote("Content C", "Casper", Genre.FANTASY),
        new Quote("Content D", "Dobby", Genre.FANTASY),
        new Quote("Content E", "Eileen", Genre.SCIENCE_FICTION),
        new Quote("Content F", "Flora", Genre.SCIENCE_FICTION)
    );

    public Flux<Quote> getAllQuotes() { ②
        return Flux.fromIterable(quotes);
    }

    public Mono<Quote> getRandomQuote() {
        return Mono.just(quotes.get(random.nextInt(quotes.size() - 1)));
    }

    public Mono<Quote> getRandomQuoteByGenre(Genre genre) {
        var quotesForGenre = quotes.stream()
            .filter(q -> q.genre().equals(genre))
            .toList();
        return Mono.just(quotesForGenre.get(
            random.nextInt(quotesForGenre.size() - 1)));
    }
}
```

- ① Stores a list of quotes in memory.
- ② Returns all quotes as a reactive data stream.

NOTE Since the focus of this example is the native-image compilation with GraalVM and Spring Native, we'll keep it simple and skip the persistence layer. Feel free to expand it on your own. For example, you could add Spring Data R2DBC and Spring Security, both supported by Spring Native.

That's it for the business logic. Next, we'll expose the functionality through an HTTP API.

IMPLEMENTING THE WEB CONTROLLER

Create a new `com.polarbookshop.quoteservice.web` package and add a `QuoteController` class to expose three endpoints for the following:

- return all the quotes;
- return a random quote;
- return a random quote for a given genre.

Listing 16.4 Define handlers for HTTP endpoints (QuoteController.java)

```
@RestController
public class QuoteController {
    private final QuoteService quoteService;

    public QuoteController(QuoteService quoteService) {
        this.quoteService = quoteService;
    }

    @GetMapping("/quotes")
    public Flux<Quote> getAllQuotes() {
        return quoteService.getAllQuotes();
    }

    @GetMapping("/quotes/random")
    public Mono<Quote> getRandomQuote() {
        return quoteService.getRandomQuote();
    }

    @GetMapping("/quotes/random/{genre}")
    public Mono<Quote> getRandomQuote(@PathVariable Genre genre) {
        return quoteService.getRandomQuoteByGenre(genre);
    }
}
```

Then, configure the embedded Netty server to listen to port 9101 and define the application name. Open the `application.yml` file and add the following configuration.

Listing 16.5 Configure the Netty server port and the application name (application.yml)

```
server:
  port: 9101

spring:
  application:
    name: quote-service
```

Finally, let's write some integration tests using the same techniques you learned in chapter 8.

WRITING INTEGRATION TESTS

When we bootstrapped the project from Spring Initializr, we got an autogenerated `QuoteServiceApplicationTests` class. Let's update it with a few integration tests checking the REST API exposed by Quote Service.

Listing 16.6 Integration tests for Quote Service (`QuoteServiceApplicationTests.java`)

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class QuoteServiceApplicationTests {

    @Autowired
    WebTestClient webTestClient;

    @Test
    void whenAllQuotesThenReturn() {
        webTestClient.get().uri("/quotes")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBodyList(Quote.class);
    }

    @Test
    void whenRandomQuoteThenReturn() {
        webTestClient.get().uri("/quotes/random")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(Quote.class);
    }

    @Test
    void whenRandomQuoteByGenreThenReturn() {
        webTestClient.get().uri("/quotes/random/FANTASY")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(Quote.class)
            .value(quote -> assertThat(quote.genre()).isEqualTo(Genre.FANTASY));
    }
}
```

That's it for the implementation. Next, we'll execute the auto-tests and run the application on the JVM.

RUNNING AND TESTING ON THE JAVA VIRTUAL MACHINE

So far, Quote Service is a standard Spring Boot application, not different from any other application we built in the previous chapters. For example, we can run the autotests with Gradle and ensure it behaves correctly. Open a Terminal window, navigate to the project root folder, and execute the following command.

```
$ ./gradlew test
```

We can also run it on the Java Virtual Machine or package it as a JAR artifact. From the same Terminal window, execute the following command to run the application.

```
$ ./gradlew bootRun
```

Feel free to verify the application works correctly by calling the endpoints exposed by Quote Service.

```
$ http :9101/quotes
$ http :9101/quotes/random
$ http :9101/quotes/random/FANTASY
```

When you're done testing the application, stop the process with `Ctrl+C`.

How can we compile it to a native executable and take advantage of instant startup time, instant peak performance, and reduced memory consumption? That's the topic of the next section.

16.1.3 Compiling Spring Boot applications as native images

There are two ways to compile your Spring Boot applications into native executables. The first option uses GraalVM explicitly and produces an OS-specific executable that runs directly on a machine. The second option relies on Cloud Native Buildpacks to containerize the native executable and run it on a container runtime like Docker. We'll use both.

COMPILING NATIVE EXECUTABLES WITH GRAALVM

The first option requires the GraalVM runtime to be available on your machine. You can install it directly from the website (<https://www.graalvm.org>) or use a tool like `sdkman`. You can find instructions on how to install `sdkman` in section A.1 of Appendix A.

For the examples in this chapter, I'll be using the latest GraalVM 22.1 distribution available at the time of writing, based on OpenJDK 17. Using `sdkman`, you can install GraalVM as follows.

```
$ sdk install java 22.1.0.r17-grl
```

At the end of the installation procedure, `sdkman` will ask whether you want to make that distribution the default one. I recommend you say no since we're going to be explicit whenever we need to use GraalVM instead of the standard OpenJDK.

Then, open a Terminal window, navigate to your Quote Service project (`quote-service`), configure the shell to use GraalVM, and install the `native-image` GraalVM component as follows.

```
$ sdk use java 22.1.0.r17-grl ①
$ gu install native-image ②
```

- ① Configures the current shell to use the specified Java runtime.
- ② Uses the "gu" utility provided by GraalVM to install the native-image component.

When you initialized the Quote Service project, the GraalVM Gradle/Maven official plugin got included automatically. That's the one providing the functionality to compile applications using the GraalVM Native Image mode.

NOTE

The following Gradle tasks require that GraalVM is the current Java runtime.

When using sdkman, you can do that by running `sdk use java 22.1.0.r17-grl` in the Terminal window where you want to use GraalVM.

Take into account that the compilation step for GraalVM apps is more prolonged, taking several minutes depending on the computational resources available on your machine. That is one of the drawbacks of working with native images. Also, since Spring Native is still in an experimental phase, you might get several debug logs and warnings, but that should be fine if the process is completed successfully.

From the same Terminal window where you switched to GraalVM as the current Java runtime, run the following command to compile the application to a native image.

```
$ ./gradlew nativeCompile
```

A standalone binary is the result of the command. Since it's a native executable, it will be different on macOS, Linux, and Windows. You can run it on your machine natively, without the need for a Java Virtual Machine. In the case of Gradle, the native executable is generated in the folder `build/native/nativeCompile`. Go ahead and run it.

```
$ build/native/nativeCompile/quote-service
```

The first thing to notice is the startup time, usually less than 100ms with Spring Native. It's an impressive improvement considering the JVM option, which takes a few seconds. The best part of it is that we didn't have to write any code to make that happen! Let's send a request to ensure the application is running correctly.

```
$ http :9101/quotes/random
```

When you're done testing the application, stop the process with `Ctrl+C`.

You can even run the autotests as native executables to make them even more reliable since they would use the actual runtime environment used in production. Keep in mind that the compilation step still takes longer than when running on the JVM.

```
$ ./gradlew nativeTest
```

Finally, you can run a Spring Boot application as a native image directly from Gradle/Maven.

```
$ ./gradlew nativeRun
```

Remember to stop the application process with `Ctrl+C` before moving to the next section that will show you another option for compiling your Spring Boot applications to native executables. It won't require having GraalVM installed on your computer and will produce a containerized native executable using Cloud Native Buildpacks.

CONTAINERIZING NATIVE IMAGES WITH BUILDPACKS

The second option for compiling Spring Boot applications to native executables relies on Cloud Native Buildpacks. Similar to how we packaged Spring Boot applications as container images in chapter 6, we can use Buildpacks to build a container image from the application native executable compiled by GraalVM. This approach benefits from not requiring GraalVM to be installed on your machine.

Spring Initializr didn't just include the Spring AOT plugin when you bootstrapped the Quote Service project, but it also provided additional configuration for the Buildpacks integration available in Spring Boot. If you check the `build.gradle` file again, you can see that the `bootBuildImage` task is configured to produce a containerized native image through the `BP_NATIVE_IMAGE` environment variable. While you're there, configure the image name and the container registry authentication as we did for the other Polar Bookshop applications.

Listing 16.7 Configuration for containerizing Quote Service (build.gradle)

```
tasks.named('bootBuildImage') {
    builder = 'paketobuildpacks/builder:tiny'      ①
    environment = ['BP_NATIVE_IMAGE': 'true']          ②
    imageName = "${project.name}"

    docker {
        publishRegistry {
            username = project.findProperty("registryUsername")
            password = project.findProperty("registryToken")
            url = project.findProperty("registryUrl")
        }
    }
}
```

- ① Uses the "tiny" version of Paketo Buildpacks to minimize the container image size.
- ② Enables the GraalVM support and produces a containerized native image.

NOTE

As you probably noticed, when running the native image compilation process on your machine, it takes not only time but also more computational resources than usual. When using Buildpacks, make sure you have at least 16GB of RAM on your computer. If you use Docker Desktop, configure the Docker virtual machine with at least 8GB of RAM. On Windows, it's recommended to use Docker Desktop on WSL2 rather than Hyper-V. For more recommendations about the setup, refer to the Spring Native documentation (<https://docs.spring.io/spring-native/docs/current/reference/htmlsingle>).

The command to use Buildpacks and produce a containerized native image is the same you'd use for JVM images. Open a Terminal window, navigate to your Quote Service project (`quote-service`), and run the following command.

```
$ ./gradlew bootBuildImage
```

When it's done, try running the resulting container image.

```
$ docker run --rm -p 9101:9101 quote-service
```

Also in this case, the startup time should be less than 100ms. Go ahead and send a few requests to test if the application is working correctly.

```
$ http :9101/quotes/random
```

When you're done testing the application, stop the container process with `Ctrl+C`.

16.2 Serverless applications with Spring Cloud Function

As introduced in chapter 1, serverless is a further abstraction layer on top of virtual machines and containers, moving even more responsibilities from product teams to the platform. Following the serverless computing model, developers focus on implementing the business logic for their applications. Using an orchestrator like Kubernetes would still require considering infrastructure provisioning, capacity planning, and scaling. On the other hand, a serverless platform takes care of setting up the underlying infrastructure needed by the applications to run, including virtual machines, containers, and dynamic scaling.

Serverless applications typically only run when there is an event to handle, such as an HTTP request (*request-driven*) or a message (*event-driven*). The event can be external or produced by another function. For example, a function might be triggered whenever a message is added to a queue, process it, and then exit the execution. When there is nothing to process, the platform shuts down all the resources involved with the function. So you can really pay for the actual usage.

In the other cloud native topologies like CaaS or PaaS, there is always a server involved that runs 24/7. Compared to traditional systems, you get the advantage of dynamic scalability to reduce the number of resources provisioned at any given time. Still, there is always something up and running that has a cost. In the serverless model, instead, resources are provisioned only when necessary. If there is nothing to process, everything is shut down. That's what we call **scaling to zero**, one of the main features offered by serverless platforms.

A consequence of scaling applications to zero is that when eventually there's a request to handle, a new application instance is started and must be ready to process it very quickly. Standard JVM applications are not suitable for serverless applications since it's hard to achieve a startup time

lower than a few seconds. That's why GraalVM native images became popular. Their instant startup time and reduced memory consumption make them perfect for the serverless model. The **instant startup time** is required for scaling. The **reduced memory consumption** helps reduce costs, which is one of the goals of serverless and cloud native in general.

Besides cost optimization, serverless technologies also move some extra responsibility from the application to the platform. That might be an advantage since it allows developers to focus exclusively on the business logic. But it's also essential to consider which degree of control you would like to have and how to deal with vendor lock-in. Each serverless platform has its own features and APIs. Once you start writing functions for a specific platform, you can't move them easily to another, as you would do with containers. You might compromise responsibility and scope with control and portability more than any other approach. That's why **Knative** became popular quickly: being built on Kubernetes means that you can easily move your serverless workloads between platforms and vendors.

This section will guide you through developing and deploying a serverless application. You'll use Spring Native to compile it to a GraalVM native image, and Spring Cloud Function to implement the business logic as functions, which is an excellent choice since serverless applications are event-driven.

16.2.1 Building serverless applications with Spring Cloud Function

You have already worked with Spring Cloud Function in chapter 10. As you learned there, it's a project aimed at promoting business logic implementation via functions based on the standard interfaces introduced by Java 8: `Supplier`, `Function`, and `Consumer`.

Spring Cloud Function is very flexible. You have already seen how it integrates transparently with external messaging systems like RabbitMQ and Kafka, a handy feature for building serverless applications triggered by messages. In this section, I'd like to show you one more feature offered by Spring Cloud Function, which lets you expose functions as endpoints triggered by HTTP requests and CloudEvents, a specification standardizing format and distribution of events in cloud architectures.

We'll use the same requirements of the Quote Service application we built earlier, but this time we'll implement the business logic as functions and let the framework deal with exposing them as HTTP endpoints.

BOOTSTRAPPING A NEW PROJECT WITH SPRING NATIVE AND SPRING CLOUD FUNCTION

You can initialize the Quote Function project from Spring Initializr (<https://start.spring.io>), store the result in a new `quote-function` Git repository, and push it to GitHub. The parameters for the initialization are shown in figure 16.2.

The screenshot shows the Spring Initializr web application. On the left, there are sections for 'Project' (Maven Project or Gradle Project), 'Language' (Java, Kotlin, Groovy), and 'Spring Boot' (version 3.0.0, 3.0.0(M3), 2.7.1(SNAPSHOT), 2.6.9(SNAPSHOT), 2.6.8). Below these are 'Project Metadata' fields: Group (com.polarbookshop), Artifact (quote-function), Name (quote-function), Description (Provides quotes from the books in the library.), Package name (com.polarbookshop.quotefunction), and Packaging (Jar, selected). At the bottom, Java version options (18, 17, 11, 8) are shown with 17 selected. On the right, there's a 'Dependencies' section with a 'ADD DEPENDENCIES...' button, followed by sections for 'Spring Native [Experimental]' (incubating support for native executables using GraalVM), 'Spring Reactive Web' (WEB, build reactive web applications with Spring WebFlux and Netty), and 'Function' (SPRING CLOUD, promotes business logic via functions and supports a uniform programming model across serverless providers).

Figure 16.2 The parameters for initializing the Quote Function project.

The project contains the following dependencies:

- **Spring Reactive Web** provides the necessary libraries for building reactive web applications with Spring WebFlux and includes Netty as the default embedded server.
- **Spring Cloud Function** provides the necessary libraries to support business logic implementation via functions, export them via several communication channels, and integrate them with serverless platforms.
- **Spring Native** supports compiling Spring applications to native executables using the GraalVM native-image compiler.

The resulting dependencies section of the `build.gradle` file looks as follows.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    implementation
    org.springframework.cloud:spring-cloud-starter-function-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
}
```

Then, update the Cloud Native Buildpacks configuration in `build.gradle` similar to what we did for Quote Service.

Listing 16.8 Configuration for containerizing Quote Function (build.gradle)

```

tasks.named('bootBuildImage') {
    builder = 'paketobuildpacks/builder:tiny'      ①
    environment = ['BP_NATIVE_IMAGE': 'true']        ②
    imageName = "${project.name}"

    docker {
        publishRegistry {
            username = project.findProperty("registryUsername")
            password = project.findProperty("registryToken")
            url = project.findProperty("registryUrl")
        }
    }
}

```

- ① Uses the "tiny" version of Paketo Buildpacks to minimize the container image size.
- ② Enables the GraalVM support and produces a containerized native image.

Next, copy all the classes from the `com.polarbookshop.quoteservice.domain` in Quote Service to a new `com.polarbookshop.quotefunction.domain` package in Quote Function. In the next section, we'll implement the business logic as functions.

IMPLEMENTING THE BUSINESS LOGIC AS FUNCTIONS

As you learned in chapter 10, Spring Cloud Function enhances standard Java functions when they are registered as beans. Let's start by adding a `QuoteFunctions` class in a new `com.polarbookshop.quotefunction.functions` package for the Quote Function project.

The application should expose similar functionality to Quote Service:

- returning all the quotes can be expressed as a `Supplier` since it takes no input;
- returning a random quote can also be expressed as a `Supplier` since it takes no input;
- returning a random quote for a given genre can be expressed as a `Function` since it has both input and output;
- logging a quote to standard output can be expressed as a `Consumer` since it has input but no output.

Listing 16.9 Implement the business logic as functions (QuoteFunctions.java)

```

@Configuration ①
public class QuoteFunctions {
    private static final Logger log =
        LoggerFactory.getLogger(QuoteFunctions.class); ②

    @Bean ③
    Supplier<Flux<Quote>> allQuotes(QuoteService quoteService) {
        return () -> {
            log.info("Getting all quotes");
            return Flux.fromIterable(quoteService.getAllQuotes())
                .delaySequence(Duration.ofSeconds(1)); ④
        };
    }

    @Bean ⑤
    Supplier<Quote> randomQuote(QuoteService quoteService) {
        return () -> {
            log.info("Getting random quote");
            return quoteService.getRandomQuote();
        };
    }

    @Bean ⑥
    Consumer<Quote> logQuote() {
        return quote -> log.info("Quote: '{}' by {}",
            quote.content(), quote.author());
    }
}

```

- ① Functions are declared as beans in a Spring configuration class.
- ② A logger used by the functions.
- ③ A supplier producing all the quotes.
- ④ Quotes are streamed one at a time with a 1-second pause between each other.
- ⑤ A supplier producing a random quote.
- ⑥ A function logging the quote received as the input.

Spring Cloud Function will automatically expose all the registered functions as HTTP endpoints when the Spring web dependencies are on the classpath. Each endpoint uses the same name as the function. In general, suppliers can be invoked through GET requests while functions and consumers as POST requests.

Quote Function contains the Spring Reactive Web dependency, so Netty will be the server to handle HTTP requests. Let's make it listen to port 9102 and configure the application name. Open the `application.yml` file and add the following configuration.

Listing 16.10 Configure the Netty server port and application name (application.yml)

```
server:
  port: 9102

spring:
  application:
    name: quote-function
```

Then, run the Quote Function application (`./gradlew bootRun`) and open a Terminal window. For starters, you can test the two suppliers by sending GET requests.

```
$ http :9102/allQuotes
$ http :9102/randomQuote
```

To get a random quote by genre, you need to provide a genre string in the body of a POST request.

```
$ echo 'FANTASY' | http :9102/genreQuote
```

When only one function is registered as a bean, Spring Cloud Function will automatically expose it through the root endpoint. In case of multiple functions, you can choose the function through the `spring.cloud.function.definition` configuration property.

For example, we could expose the `allQuotes` function through the root endpoint. In the Quote Function project, open the `application.yml` and update it as follows.

Listing 16.11 Configure the main function managed by Spring Cloud Function (application.yml)

```
server:
  port: 9102

spring:
  application:
    name: quote-function
  cloud:
    function:
      definition: allQuotes
```

Re-run the application and send a GET request to the root endpoint. Since the `allQuotes` function is a `Supplier` returning a `Flux` of `Quote`, you can leverage the streaming capabilities of Project Reactor and ask the application to return the quotes as they become available. That is done automatically when the `Accept:text/event-stream` header is used (e.g., `curl -H 'Accept:text/event-stream' localhost:9102`). When using the `httpie` utility, you also need to use the `--stream` argument to enable data streaming.

```
$ http :9102 Accept:text/event-stream --stream
```

Similar to what you did in chapter 10, you can build a pipeline by combining functions. When

exposed as HTTP endpoints, you can use the , (comma) character to compose functions on the fly. For example, you could combine the `genreQuote` function with `logQuote` as follows.

```
$ echo 'FANTASY' | http :9102/genreQuote,logQuote
```

Since `logQuote` is a consumer, the HTTP response has a 202 status with no body. If you check the application logs, you can see that the random quote by genre has been printed out instead.

Spring Cloud Function integrates with several communication channels. You have seen how to expose functions through exchanges and queues leveraging Spring Cloud Stream and how to expose them as HTTP endpoints. The framework also supports RSocket (a binary reactive protocol) and CloudEvents, a specification standardizing format and distribution of events in cloud architectures (<https://cloudevents.io>).

CloudEvents can be consumed over HTTP, messaging channels like AMPQ (RabbitMQ), and RSocket. They ensure a standard way of describing events, making them portable across a wide variety of technologies, including applications, messaging systems, build tools, and platforms.

Since Quote Function is already configured to expose functions as HTTP endpoints, you can make it consume CloudEvents without changing any code. Ensure the application is up and running, then send an HTTP request with the additional headers defined by the CloudEvents specification.

```
$ echo 'FANTASY' | http :9102/genreQuote \
  ce-specversion:1.0 \ ❶
  ce-type:quote \ ❷
  ce-id:394 ❸
```

- ❶ The CloudEvents specification version.
- ❷ The type of event (domain-specific).
- ❸ The id of the event.

When you're done testing the application, stop the process with `Ctrl+C`.

NOTE

You can refer to the Spring Cloud Function official documentation for more details on how HTTP, CloudEvents, and RSocket are supported (<https://spring.io/projects/spring-cloud-function>).

16.2.2 Deployment pipeline: Build and publish

Following the continuous delivery principles and techniques explained throughout the book, we can implement a deployment pipeline also for Quote Service and Quote Function. Since the release candidate for those projects is a container image, most of the operations will be the same as for standard JVM applications.

When working locally, it's convenient to run and test serverless applications on the JVM rather than using GraalVM due to the shorter build time and the less resource-demanding process. However, to achieve better quality and catch errors earlier, we should run and verify the applications in native mode as early in the delivery process as possible. The commit stage is where we compile and test our applications, so it might be a good place for adding those additional steps.

In your Quote Function project (`quote-function`), add a new `.github/workflows` folder, and create a `commit-stage.yml` file. As a starting point, you can copy the implementation of the commit stage from one of the other applications we built in the previous chapters, such as Catalog Service. The commit stage workflow we have used so far is composed of two jobs: "Build & Test" and "Package and Publish". We'll reuse the same exact implementation from the other applications, but we'll add an intermediate job responsible for testing the native mode.

Listing 16.12 Configure a job for building and testing the application in native mode (`commit-stage.yml`).

```
name: Commit Stage
on: push

env:
  REGISTRY: ghcr.io ①
  IMAGE_NAME: <your_github_username>/quote-function ②
  VERSION: ${{ github.sha }} ③

jobs:
  build:
    name: Build and Test
    ...

    native: ④
      name: Build and Test (Native) ⑤
      runs-on: ubuntu-22.04 ⑥
      permissions: ⑦
        contents: read ⑧
      steps:
        - name: Checkout source code
          uses: actions/checkout@v3 ⑨
        - name: Set up GraalVM
          uses: graalvm/setup-graalvm@v1 ⑩
          with:
            version: '22.1.0'
            java-version: '17'
            components: 'native-image'
            github-token: ${{ secrets.GITHUB_TOKEN }}
        - name: Build, unit tests and integration tests (native)
          run:
            chmod +x gradlew
            ./gradlew nativeBuild ⑪

  package:
    name: Package and Publish
    if: ${{ github.ref == 'refs/heads/main' }}
    needs: [ build, native ] ⑫
    ...

① Uses the GitHub Container Registry.
② Sets the image name to the user's GitHub username followed by /quote-function.
③ Sets the image version to the current GitHub commit hash.
④ Configures a job named 'native' for building and testing in native mode.
⑤ Sets the job name to 'Build and Test (Native)'.
⑥ Runs the job on an Ubuntu 22.04 host.
⑦ Grants the job permission to read repository contents.
⑧ Specifies that the job has read access to repository contents.
⑨ Checks out the repository code.
⑩ Sets up the GraalVM environment.
⑪ Runs the 'nativeBuild' Gradle task.
⑫ Specifies that the 'native' job depends on both the 'build' and 'native' jobs.
```

① Uses the GitHub Container Registry.

- ② The name of the image. Remember to add your GitHub username all in lowercase.
- ③ For simplicity, any new image will be tagged as "latest".
- ④ The job unique identifier.
- ⑤ A human-friendly name for the job.
- ⑥ The type of machine where to run the job.
- ⑦ The permissions granted to the job.
- ⑧ Permission to check out the current Git repository.
- ⑨ Checks out the current Git repository ("quote-function").
- ⑩ Installs and configures GraalVM with Java 17 and native image component.
- ⑪ Compiles the application as a native executable and runs unit and integration tests.
- ⑫ The "Package and Publish" job runs only if both the previous jobs complete successfully.

NOTE

In the source code repository accompanying the book, you can check the final result in the Chapter16/16-end/quote-function folder.

When you're done, commit all your changes and push them to your GitHub quote-function repository to trigger the commit stage workflow. We're going to use the container image published by that workflow later in the chapter, so make sure it runs successfully.

You'll notice that the commit stage execution for Quote Function is taking quite longer than what we experienced with the other applications throughout the book. In chapter 3, I wrote that the commit stage is supposed to be fast, possibly under five minutes, to provide developers with fast feedback about their changes and allow them to move on to the next task in the spirit of continuous integration. The additional steps using GraalVM that we have just added might slow down the workflow too much. In that case, you might consider moving this check to the acceptance stage, where we allow the overall process to take longer.

The following section will cover some options for deploying serverless applications implemented with Spring Cloud Function.

16.2.3 Deploying serverless applications on the cloud

Applications using Spring Cloud Function can be deployed in a few different ways. First of all, since they're still Spring Boot applications, you can package them as JAR artifacts or container images and deploy them on servers or container runtimes like Docker/Kubernetes, respectively, just like you did in the previous chapters.

Then, when Spring Native is included, you also have the option to compile them to native images

and run them on servers or container runtimes. Thanks to instant startup time and reduced memory consumption, you can also seamlessly deploy such applications on serverless platforms. The next section covers how to use Knative to run your serverless workloads on Kubernetes.

Spring Cloud Function also supports deploying applications on vendor-specific FaaS platforms like AWS Lambda, Azure Functions, and Google Cloud Functions. Once you choose a platform, you can add the related adapter provided by the framework to accomplish the integration. Each adapter works in a slightly different way depending on the specific platform and the configuration required to integrate the functions with the underlying infrastructure. The adapters provided by Spring Cloud Function don't require any change to your business logic, but they might need some additional code to configure the integration.

When you use one of those adapters, you must choose which function to integrate with the platform. If there's only one function registered as a bean, that's the one used. If there are more (like in Quote Function), you need to use the `spring.cloud.function.definition` property to declare the function the FaaS platform will manage.

NOTE

You can refer to the [Spring Cloud Function official documentation](#) for more details on the Spring Cloud Function adapters for AWS Lambda, Azure Functions, and Google Cloud Functions (<https://spring.io/projects/spring-cloud-function>).

The following section will cover how to deploy a serverless application like Quote Function on a Kubernetes-based platform using Knative.

16.3 Deploying serverless applications with Knative

In the previous sections, you learned about Spring Native and how to use it with Spring Cloud Function to build serverless applications. This section will guide you through deploying Quote Function to a serverless platform on top of Kubernetes using the Knative project.

Knative is a "Kubernetes-based platform to deploy and manage modern serverless workloads" (<https://knative.dev>). It's a CNCF project that you can use to deploy standard containerized workload and event-driven applications. The project offers a superior user experience to developers and higher abstractions that make it simpler to deploy applications on Kubernetes.

You can decide to run your own Knative platform on top of a Kubernetes cluster or choose a managed service offered by a cloud provider, such as VMware Tanzu Platform, Google Cloud Run, or RedHat OpenShift Serverless. Since they are all based on open-source software and standards, you could migrate from Google Cloud Run to VMware Tanzu Platform without changing your application code and with minimal changes to your deployment pipeline.

The Knative project consists of two main components: Serving and Eventing.

- **Knative Serving** is for running serverless workloads on Kubernetes. It takes care of autoscaling, networking, revisions, and deployment strategies while letting engineers focus on the application business logic.
- **Knative Eventing** provides management for integrating applications with event sources and sinks based on the CloudEvents specification, abstracting backends like RabbitMQ or Kafka.

Our focus will be on Knative Serving for running serverless workloads while avoiding vendor lock-in.

NOTE

Originally, Knative consisted of a third component called Build that subsequently became a standalone product, renamed Tekton (<https://tekton.dev>), and donated to the Continuous Delivery Foundation (<https://cd.foundation>). Tekton is a Kubernetes-native framework for building deployment pipelines that support continuous delivery. For example, you could use Tekton instead of GitHub Actions.

This section will show you how to set up a local development environment comprising both Kubernetes and Knative. Then, I'll introduce Knative manifests to declare the desired state for serverless applications and show you how to apply them to a Kubernetes cluster.

16.3.1 Setting up a local Knative platform

Since Knative runs on top of Kubernetes, we first need a cluster. Let's create one with `minikube` following the same approach we used throughout the book. Open a Terminal window and run the following command.

```
$ minikube start --profile knative
```

Next, we can install Knative. For simplicity, I have collected the necessary commands in a script you can find in the source code repository accompanying the book. From the `Chapter16/16-end/polar-deployment/kubernetes/development` folder, copy the `install-knative.sh` file into the same path in your Polar Deployment repository (`polar-deployment`).

Then, open a Terminal window, navigate to the folder where you have just copied the script, and run the following command to install Knative on your local Kubernetes cluster. Feel free to open the file and look at the instructions before running it. You can find more information about installing Knative on the project website (<https://knative.dev/docs/install>).

```
$ ./install-knative.sh
```

NOTE

On macOS and Linux, you might need to make the script executable via the `chmod +x install-knative.sh` command.

The Knative project provides a convenient CLI tool that you can use to interact with Knative resources in a Kubernetes cluster. You can find instructions on how to install it in section A.4 of Appendix A. In the next section, I'll show you how to deploy Quote Function using the Knative CLI.

16.3.2 Deploying applications with the Knative CLI

Knative provides a few different options for deploying applications. In production, we want to stick to a declarative configuration as we did for standard Kubernetes deployments and rely on a GitOps flow to reconcile desired state (in a Git repository) and actual state (in the Kubernetes cluster).

When experimenting or working locally, we can also take advantage of the Knative CLI to deploy applications in an imperative way. From a Terminal window, run the following command to deploy Quote Function. The container image is the one published by the commit stage workflow we defined before. Remember to replace `<your_github_username>` with your GitHub username in lowercase.

```
$ kn service create quote-function \
  --image ghcr.io/<your_github_username>/quote-function \
  --port 9102
```

You can refer to figure 16.3 for a description of the command.

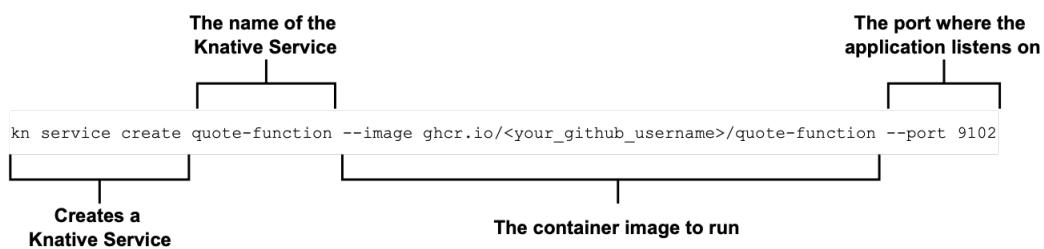


Figure 16.3 The Knative command to create a Service from a container image. Knative will take care of creating all the resources necessary to deploy the applications on Kubernetes.

The command will initialize a new `quote-function` service in the default namespace on Kubernetes and return the public URL through which the application is exposed, similar to the following message.

```
Creating service 'quote-function' in namespace 'default':
  0.045s The Route is still working to reflect the latest desired
  ↵specification.
  0.096s Configuration "quote-function" is waiting for a Revision
  ↵to become ready.
  3.337s ...
  3.377s Ingress has not yet been reconciled.
  3.480s Waiting for load balancer to be ready
  3.660s Ready to serve.

Service 'quote-function' created to latest revision 'quote-function-00001'
  ↵is available at URL:
  http://quote-function.default.127.0.0.1.sslip.io
```

Let's test it out! First, we need to open a tunnel to the cluster with `minikube`. The first time you run this command, you might be asked to input your machine password to authorize the tunneling to the cluster.

```
$ minikube tunnel --profile knative
```

Then, open a new Terminal window and call the application at the root endpoint to fetch the complete list of quotes. The URL to call is the same returned by the previous command (`http://quote-function.default.127.0.0.1.sslip.io`), which is in the format `<service-name>.<namespace>.<domain>`.

```
$ http http://quote-function.default.127.0.0.1.sslip.io
```

Since we are working locally, I configured Knative to use `sslip.io`, a DNS service that "when queried with a hostname with an embedded IP address, returns that IP address". For example, the `127.0.0.1.sslip.io` hostname would be resolved to the `127.0.0.1` IP address. Since we opened a tunnel to the cluster, requests to `127.0.0.1` will be handled by the cluster where Knative will route them to the right service.

Knative takes care of scaling the application without any further configuration. For each request, it evaluates whether more instances are required. When an instance stays idle for a specific time period (30 seconds, by default), Knative will shut it down. If no request is received for more than 30 seconds, Knative will scale the application to zero, meaning there will be no instance of Quote Function running.

When a new request is eventually received, Knative starts a new instance and uses it to handle the request. Thanks to Spring Native, the startup time of Quote Function is almost instantaneous, so users or clients won't have to deal with long wait times, as would be the case with standard JVM applications. This powerful feature lets you optimize costs and pay only for what you use and need.

Using an open-source platform like Knative has the advantage of letting you migrate your applications to another cloud provider without any code change. But that's not all! You can even

use the same deployment pipeline as-is or with minor modifications. The next section will show you how to define Knative Services in a declarative way via YAML manifests, the recommended approach for production scenarios.

Before moving on, make sure you have deleted the Quote Function instance you created previously.

```
$ kn service delete quote-function
```

16.3.3 Deploying applications with the Knative manifests

Kubernetes is an extensible system. Besides using built-in objects like Deployments and Pods, we can define our own objects via Custom Resource Definitions (CRDs). That is the strategy used by many tools built on top of Kubernetes, including Knative.

One of the benefits of using Knative is a better developer experience and the possibility to declare the desired state for our applications in a more straightforward and less verbose way. Rather than dealing with Deployments, Services, and Ingresses, we can work with one single type of resource: the Knative Service.

NOTE

Throughout the book, I talked about applications as services. Knative offers a way to model an application in one single resource declaration: the Knative Service. At first, it might not be very clear since there is already a Kubernetes built-in Service type. In reality, the Knative choice is very intuitive because it maps one-to-one the architectural concept with the deployment concept.

Let's see how Knative Services look. Open your Quote Function project (`quote-function`), and create a new `knative` folder. Then, define a new `kservice.yaml` file inside to declare the desired state of the Knative Service for Quote Function. Remember to replace `<your_github_username>` with your GitHub username in lowercase.

Listing 16.13 Knative Service manifest for Quote Function (service.yml)

```

apiVersion: serving.knative.dev/v1 ①
kind: Service ②
metadata:
  name: quote-function ③
spec:
  template:
    spec:
      containers:
        - name: quote-function ④
          image: ⑤
        ↵ghcr.io/<your_github_username>/quote-function
      ports:
        - containerPort: 9102 ⑥
      resources: ⑦
        requests:
          cpu: '0.1'
          memory: '128Mi'
        limits:
          cpu: '2'
          memory: '512Mi'

```

- ① The API version for Knative Serving objects.
- ② The type of object to create.
- ③ The name of the Service.
- ④ the name of the container.
- ⑤ The image used to run the container. Remember to insert your GitHub username.
- ⑥ The port exposed by the container.
- ⑦ CPU and memory configuration for the container.

Like any other Kubernetes resource, you can apply a Knative Service manifest to a cluster with `kubectl apply -f <manifest-file>` or through some automated flow like we did with ArgoCD in the previous chapter. For this example, we'll use the Kubernetes CLI.

Open a Terminal window, navigate to your Quote Function project (`quote-function`), and run the following command to deploy Quote Function from the Knative Service manifest.

```
$ kubectl apply -f knative/kbservice.yml
```

Using the Kubernetes CLI, you can get information about all the created Knative Services and their URLs by running the following command (the shown result is partial to fit on the page).

```

$ kubectl get ksvc
NAME           URL                                     READY
quote-function http://quote-function.default.127.0.0.1.sslip.io   True

```

Let's verify that the application is correctly deployed by sending an HTTP request to its root endpoint. If the tunnel you opened earlier is not active anymore, run `minikube tunnel`

--profile knative before calling the application.

```
$ http http://quote-function.default.127.0.0.1.sslip.io
```

Knative provides an abstraction on top of Kubernetes. However, it still runs Deployments, ReplicaSets, Pods, Services, and Ingresses under the hood. It means you can use all the techniques you learned in the previous chapters. For example, you can configure Quote Function through ConfigMaps and Secrets.

```
$ kubectl get pod
```

NAME	READY	STATUS
pod/quote-function-00001-deployment-c6978b588-llf9w	2/2	Running

If you wait for 30 seconds and then check for the running Pods in your local Kubernetes cluster, you'll see there are none because Knative scaled the application to zero due to inactivity.

```
$ kubectl get pod
No resources found in default namespace.
```

Now, try sending a new request to the application on <http://quote-function.default.127.0.0.1.sslip.io>. Knative will immediately spin up a new Pod for Quote Function to answer the request.

```
$ kubectl get pod
```

NAME	READY	STATUS
quote-function-00001-deployment-c6978b588-f49x8	2/2	Running

When you're done testing the application, you can remove it with `kubectl delete -f knative/service.yml`. Finally, you can stop and delete the local cluster with the following command.

```
$ minikube stop --profile knative
$ minikube delete --profile knative
```

The Knative Service resource represents an application service in its entirety. Thanks to this abstraction, we no longer need to deal directly with Deployments, Services, and Ingresses. Knative takes care of all that. It creates and manages them under the hood while freeing us from dealing with those lower-level resources provided by Kubernetes. By default, Knative can even expose an application outside the cluster without the need to configure an Ingress resource, providing you directly with a URL to call the application.

Thanks to its features focused on developer experience and productivity, Knative can be used to run and manage any kind of workload on Kubernetes, limiting the scale-to-zero functionality only to the applications that provides support for it (for example, using Spring Native). We could

easily run the entire Polar Bookshop system on Knative. We would use the `autoscaling.knative.dev/minScale` annotation to mark the applications we don't want to be scaled to zero.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: catalog-service
  annotations:
    autoscaling.knative.dev/minScale: "1" ①
...
```

- ① Ensures this Service is never scaled to zero.

Knative offers such a great developer experience that it's becoming the de-facto abstraction when deploying workloads on Kubernetes, not only for serverless but also for more standard containerized applications. Whenever I provision a new Kubernetes cluster, Knative is the first thing I install. It's also a foundational part of platforms like Tanzu Community Edition, Tanzu Application Platform, RedHat OpenShift, and Google Cloud Run.

NOTE

Tanzu Community Edition (<https://tanzucommunityedition.io>) is a Kubernetes platform providing a great developer experience on top of Knative. It's open source and free to use.

Another great feature offered by Knative is an intuitive and developer-friendly option to adopt deployment strategies like blue/green deployments, canary deployments, A/B deployments, all via the same Knative Service resource. Implementing those strategies in plain Kubernetes would require a lot of manual work. Instead, Knative supports them out of the box.

NOTE

To get more information about serverless applications and Knative, you can refer to the official documentation (<https://knative.dev>). Also, I recommend checking out a couple of books from the Manning catalog on this subject: "Knative in Action" by Jacques Chester (<https://www.manning.com/books/knative-in-action>) and "Continuous Delivery for Kubernetes" by Mauricio Salatino (<https://www.manning.com/books/continuous-delivery-for-kubernetes>).

SIDE BAR**Polar Labs**

Feel free to apply what you learned in the last sections to Quote Service.

1. Define a commit stage workflow, including the steps for compiling and testing the application as a native executable.
2. Push your changes to GitHub, ensure the workflow completes successfully and publishes a container image for your application.
3. Deploy Quote Service on Kubernetes via the Knative CLI.
4. Deploy Quote Service on Kubernetes via the Kubernetes CLI from a Knative Service manifest.

You can refer to the `Chapter16/16-end` folder in the code repository accompanying the book to check the final result (<https://github.com/ThomasVitale/cloud-native-spring-in-action>).

16.4 Summary

- By replacing a standard OpenJDK distribution with GraalVM as the runtime environment for your Java applications, you can increase their performance and efficiency thanks to a new optimized technology to perform JIT compilation (the GraalVM compiler).
- What makes GraalVM so innovative and popular in the serverless context is the Native Image mode.
- Rather than compiling your Java code into byte code and relying on a JVM to interpret it and convert it to machine code at run time, GraalVM offers a new technology (the Native Image builder) to compile Java applications directly into machine code, obtaining a native executable or native image.
- Java applications compiled as native images have faster startup time, optimized memory consumption, and instant peak performance than the JVM option.
- The main goal of Spring Native is to make it possible to compile any Spring application into a native executable using GraalVM without any code change.
- Spring Native provides an AOT infrastructure (invoked from a dedicated Gradle/Maven plugin) for contributing all the required configurations for GraalVM to AOT-compile Spring classes.
- There are two ways to compile your Spring Boot applications into native executables. The first option produces an OS-specific executable and runs the application directly on a machine. The second option relies on Buildpacks to containerize the native executable and run it on a container runtime like Docker.
- Serverless is a further abstraction layer on top of virtual machines and containers, which moves even more responsibilities from product teams to the platform.
- Following the serverless computing model, developers focus on implementing the business logic for their applications.
- Serverless applications are triggered by an incoming request or a specific event. We call such applications request-driven or event-driven.
- Applications using Spring Cloud Function can be deployed in a few different ways.
- When Spring Native is included, you also can compile applications to native images and run them on servers or container runtimes. Thanks to instant startup time and reduced memory consumption, you can seamlessly deploy such applications on Knative.
- Knative is a "Kubernetes-based platform to deploy and manage modern serverless workloads" (<https://knative.dev>). You can use it to deploy standard containerized workload and event-driven applications.
- The Knative project offers a superior user experience to developers and higher abstractions that make it simpler to deploy applications on Kubernetes.
- Knative offers such a great developer experience that it's becoming the de-facto abstraction when deploying workloads on Kubernetes, not only for serverless but also for more standard containerized applications.

Setting up your development environment



This appendix covers

- Setting up Java
- Setting up Docker
- Setting up Kubernetes
- Setting up other tools

This appendix will go over the instructions to set up your development environment and install the tools we use throughout the book to build, manage, and deploy cloud native applications.

A.1 Java

All the examples in this book are based on Java 17, the latest long-term release of Java. You can install any of the OpenJDK 17 distributions. I'll be using Eclipse Temurin from the Adoptium project (adoptium.net), previously known as AdoptOpenJDK, but feel free to choose another one.

Managing different Java versions and distributions on your machine might be painful. I recommend using a tool like `sdkman` (sdkman.io) to easily install, update, and switch between different JDKs. On macOS and Linux, you can install `sdkman` as follows. Refer to the official documentation for installation instructions on Windows.

```
$ curl -s "https://get.sdkman.io" | bash
```

First, check all the available OpenJDK distributions and versions by running the following command.

```
$ sdk list java
```

Then, choose a distribution and install it. For example, I can install the latest 17 version of Eclipse Temurin available at the moment of writing as follows. By the time you read this paragraph, newer versions might be available, so please check the list returned from the previous command to identify the latest one.

```
$ sdk install java 17.0.2-tem
```

At the end of the installation procedure, `sdkman` will ask whether you want to make that distribution the default one. I recommend you say `yes` to ensure you have access to Java 17 from all the projects we build throughout the book. You can always change the default version with the following command.

```
$ sdk default java 17.0.2-tem
```

Let's now verify the OpenJDK installation.

```
$ java --version
openjdk 17.0.2 2022-01-18
OpenJDK Runtime Environment Temurin-17.0.2+8 (build 17.0.2+8)
OpenJDK 64-Bit Server VM Temurin-17.0.2+8 (build 17.0.2+8, mixed mode, sharing)
```

You also have the option to change the Java version only within the context of the current shell.

```
$ sdk use java 17.0.2-tem
```

Finally, if you want to check which version of Java is configured as the default, you can do that as follows.

```
$ sdk current java
Using java version 17.0.2-tem
```

A.2 Docker

The Open Container Initiative (OCI), a Linux Foundation project, defines industry standards for working with containers (opencontainers.org). In particular, the OCI Image Specification defines how to build container images, the OCI Runtime Specification defines how to run those container images, and the OCI Distribution Specification defines how to distribute them. The tool we use throughout the book to work with containers is Docker, compliant with the OCI specifications.

On the Docker website (www.docker.com), you can find instructions on setting up Docker in your local environment. I'll be using the latest versions available at the time of writing: Docker 20.10 and Docker Desktop 4.4.

- On Linux, you can install the Docker open-source platform directly, also known as Docker Community Edition (Docker CE).
- On macOS and Windows, you have the option to use Docker Desktop, a commercial

product built on top of Docker that makes it possible to run Linux containers from those operating systems. At the moment of writing, Docker Desktop is free for personal use, education, non-commercial open-source projects, and small businesses. Please, read the Docker Subscription Service Agreement carefully before installing the software and make sure you are compliant with it (www.docker.com/legal).

Docker Desktop provides support for both ARM64 and AMD64 architectures, meaning that you can run all the examples in this book also on the new Apple computers with M1 processors.

If you work on Windows, Docker Desktop provides two types of setup: Hyper-V or WSL2. I recommend you choose the latter since it offers better performance, and it's more stable.

Docker comes pre-configured with downloading OCI images from Docker Hub, a container registry hosting images for many popular open-source projects, like Ubuntu, PostgreSQL, and Redis. It's free to use, but it's subjected to strict rate-limiting policies if you use it anonymously. Therefore, I recommend you create a free account on the Docker website (www.docker.com).

After creating an account, open a Terminal window and authenticate with Docker Hub (make sure your Docker Engine is running). Since it's the default container registry, you don't need to specify its URL. When asked, insert your username and password.

```
$ docker login
```

Using the Docker CLI, you can now interact with Docker Hub to download images (*pull*) or upload your own (*push*). For example, try pulling the Ubuntu image from Docker Hub.

```
$ docker pull ubuntu:20.04
```

Throughout the book, you'll learn more about using Docker. Until then, if you would like to experiment with containers, I'll leave you a list of useful commands to control the container lifecycle (table A.1).

Table A.1 Useful Docker CLI commands to manage images and containers.

Docker CLI command	What it does
<code>docker images</code>	Shows all images.
<code>docker ps</code>	Shows the running containers.
<code>docker ps -a</code>	Shows all containers created, started, and stopped.
<code>docker run <image></code>	Run a container from the given image.
<code>docker start <name></code>	Starts an existing container.
<code>docker stop <name></code>	Stops a running container.
<code>docker logs <name></code>	Shows the logs from a given container.
<code>docker rm <name></code>	Removes a stopped container.
<code>docker rmi <name></code>	Removes an image.

All the containers we build throughout the book are OCI-compliant and will work with any other

OCI container runtime, such as Podman ([podman.io](#)). Should you decide to use a platform other than Docker, be aware that some tools we use for local development and integration testing might require additional configurations to work correctly.

A.3 Kubernetes

There are a few ways to install Kubernetes in your local environment. Some of the most used options are the following.

- **minikube** ([minikube.sigs.k8s.io](#)) lets you run a local Kubernetes cluster on any operating system. It's maintained by the Kubernetes community.
- **kind** ([kind.sigs.k8s.io](#)) lets you run local Kubernetes clusters as Docker containers. It was primarily developed to test Kubernetes itself, but you can also use it for local development with Kubernetes. It's maintained by the Kubernetes community.
- **k3d** ([k3d.io](#)) lets you run local Kubernetes clusters based on k3s, a minimal distribution of Kubernetes implemented by Rancher Lab. It's maintained by the Rancher community.

Feel free to choose the tool that best fits your needs. I'll be using `minikube` throughout the book because of its stability and compatibility with any operating system and architecture, including the new Apple M1 computers. You should have at least 2 CPUs and 4GB of free memory to use `minikube` for running all the examples in the book.

You can find the installation guide on the project website ([minikube.sigs.k8s.io](#)). I'll be using the latest versions available at the time of writing: Kubernetes 1.23 and `minikube` 1.25. On macOS, you can install `minikube` with Homebrew as follows.

```
$ brew install minikube
```

Running a local Kubernetes cluster with `minikube` requires a container runtime or a virtual machine manager. Since we are already using Docker, that's what we're going to use. Under the hood, any `minikube` cluster will run as a Docker container.

After installing `minikube`, let's start a new local Kubernetes cluster using the Docker driver. The first time you run this command, it will take a few minutes to download all the components needed to run the cluster.

```
$ minikube start --driver=docker
```

I recommend making Docker the default driver for `minikube` by running the following command.

```
$ minikube config set driver docker
```

To interact with the newly created Kubernetes cluster, you need to install `kubectl`, the Kubernetes CLI. Installation instructions are available on the official website ([kubernetes.io/docs/tasks/tools/install-kubectl/](#)).

kubernetes.io/docs/tasks/tools). On macOS, you can install it with Homebrew as follows.

```
$ brew install kubectl
```

Then, you can verify the `minikube` cluster is started correctly and check that a node is running in your local cluster.

```
$ kubectl get nodes
NAME     STATUS   ROLES      AGE      VERSION
minikube Ready    control-plane,master 2m20s  v1.23.1
```

I recommend stopping `minikube` whenever you don't need it to free up resources in your local environment.

```
$ minikube stop
```

Throughout the book, you'll learn more about using Kubernetes and `minikube`. Until then, if you would like to experiment with Kubernetes resources, I'll leave you with some useful commands (table [A.2](#)).

Table A.2 Useful Kubernetes CLI commands to manage Pods, Deployments, and Services.

Kubernetes CLI command	What it does
<code>kubectl get deployment</code>	Show all Deployments.
<code>kubectl get pod</code>	Show all Pods.
<code>kubectl get svc</code>	Show all Services.
<code>kubectl logs <pod_id></code>	Show the logs for the given Pod.
<code>kubectl delete deployment <name></code>	Delete the given Deployment.
<code>kubectl delete pod <name></code>	Delete the given Pod.
<code>kubectl delete svc <service></code>	Delete the given Service.
<code>kubectl port-forward svc <service> <host-port>:<cluster-port></code>	Forwards traffic from your local machine to within the cluster.

A.4 Other Tools

This section will present a series of helpful tools used throughout the book for performing specific tasks, such as security vulnerability scanning or HTTP interactions.

A.4.1 HTTPie

HTTPie is a convenient "command-line HTTP and API testing client" (httpie.org). It's designed for humans and offers a superior user experience. Refer to the official documentation for installation instructions and more information on the tool.

On macOS and Linux, you can install it with Homebrew as follows.

```
$ brew install httpie
```

As part of the installation, you'll get two tools you can use from your Terminal window: `http` and `https`. For example, you can send a GET request as follows.

```
$ http pie.dev/get
```

A.4.2 grype

In the context of supply chain security, we use `grype` to scan Java codebases and container images for vulnerabilities (github.com/anchore/grype). The scanning happens locally on the machine where you run it, meaning that none of your files or artifacts is sent to an external service. That makes it a good fit for more regulated environments or air-gapped scenarios. Refer to the official documentation for more information.

On macOS and Linux, you can install it with Homebrew as follows.

```
$ brew tap anchore/grype
$ brew install grype
```

The tool is not available for Windows yet. If you are a Windows user, I recommend taking advantage of the Windows Subsystem for Linux 2 (WSL2) and install `grype` there. For more information on WSL2, you can refer to the official documentation (docs.microsoft.com/en-us/windows/wsl/).

A.4.3 Tilt

Tilt (tilt.dev) aims at providing a good developer experience when working on Kubernetes. It's an open-source tool that offers features to build, deploy, and manage containerized workloads on your local environment. Refer to the official documentation for installation instructions (docs.tilt.dev/install.html).

On macOS and Linux, you can install it with Homebrew as follows.

```
$ brew install tilt-dev/tap/tilt
```

A.4.4 Octant

Octant (octant.dev) is an "open source developer-centric web interface for Kubernetes that lets you inspect a Kubernetes cluster and its applications". Refer to the official documentation for installation instructions (reference.octant.dev).

On macOS and Linux, you can install it with Homebrew as follows.

```
$ brew install octant
```

A.4.5 Kubeval

Kubeval (www.kubeval.com) is a convenient tool to "validate one or more Kubernetes configuration files". We use it in a deployment pipeline to ensure all our Kubernetes manifests are properly formatted and compliant with the Kubernetes API. Refer to the official documentation for installation instructions (www.kubeval.com/installation/).

On macOS and Linux, you can install it with Homebrew as follows.

```
$ brew tap instrumenta/instrumenta
$ brew install kubeval
```

A.4.6 Knative CLI

Knative (knative.dev) is a "Kubernetes-based platform to deploy and manage modern serverless workloads" (knative.dev). The project provides a convenient CLI tool you can use to interact with Knative resources in a Kubernetes cluster. Refer to the official documentation for installation instructions (knative.dev/docs/install/quickstart-install).

On macOS and Linux, you can install it with Homebrew as follows.

```
$ brew install kn
```



Kubernetes in production with DigitalOcean

This appendix covers

- Running a Kubernetes cluster on Digital Ocean
- Running a PostgreSQL database on DigitalOcean
- Running Redis on DigitalOcean
- Running RabbitMQ using a Kubernetes Operator
- Running Keycloak using a Helm chart

Kubernetes is the de-facto standard for deploying and managing containerized workloads. We've been relying on a local Kubernetes cluster to deploy applications and services in the Polar Bookshop system throughout the book. For production, we need something else.

All major cloud providers offer a managed Kubernetes service. In this appendix, you'll see how to use DigitalOcean to spin up a Kubernetes cluster. We'll also rely on other managed services provided by the platform, including PostgreSQL and Redis. Finally, this appendix will guide you through the deployment of RabbitMQ and Keycloak directly in Kubernetes.

Before moving on, you need to ensure you have a DigitalOcean account. When you sign up, DigitalOcean offers a 60-days free trial with a 100\$ credit that is more than enough to go through the examples in chapter 15. Follow the instructions on the official website to create an account and start a free trial (try.digitalocean.com/freetrialoffer).

There are two main options for interacting with the DigitalOcean platform. The first one is through the web portal (cloud.digitalocean.com), which is very convenient to explore the available services and their features. The second option is via `doctl`, the DigitalOcean CLI. That's what we're going to use in the following.

You can find instructions on installing `doctl` on the official website (docs.digitalocean.com/reference/doctl/how-to/install). If you're on macOS or Linux, you can easily install it using Homebrew.

```
$ brew install doctl
```

Then, follow the subsequent instructions on the same page to generate an API token and grant `doctl` access to your DigitalOcean account.

NOTE

In a real production scenario, you would automate the platform management tasks using a tool like Terraform or Crossplane. That responsibility is usually on the platform team, not application developers, so I won't add extra complexity here by introducing yet another tool. Instead, we'll use the DigitalOcean CLI directly. If you're interested in Terraform, Manning has a book in its catalog on the subject: "Terraform in Action" by Scott Winkler (www.manning.com/books/terraform-in-action). About Crossplane, I recommend reading chapter 4 of "Continuous Delivery with Kubernetes" by Mauricio Salatino (add link).

B.1 Running a Kubernetes cluster on Digital Ocean

The first resource we need to create on DigitalOcean is a Kubernetes cluster. You could rely on the IaaS capabilities offered by the platform and install a Kubernetes cluster manually on top of virtual machines. Instead, we'll move up the abstraction staircase and go for a solution managed by the platform. Using DigitalOcean Kubernetes (docs.digitalocean.com/products/kubernetes), the platform will take care of many infrastructural concerns so that we developers can focus more on application development.

You can straightforwardly create a new Kubernetes cluster using `doctl`. I promised that we would deploy Polar Bookshop in a real production environment, and that's what we'll do. Although, I won't ask you to size and configure the cluster as I would in a real scenario.

For starters, setting up a Kubernetes cluster is not a developer's responsibility, but it's a job for the platform team. Second, it would require a more in-depth coverage of Kubernetes than what is provided by this book to understand the configuration fully. Third, I don't want you to incur extra costs on DigitalOcean for using a lot of computational resources and services. Cost optimization is a cloud property that is true for real applications. It might become expensive if you're trying things out or running demo applications. Please, keep an eye on your DigitalOcean account to monitor when your free trial and 100\$ credit expire.

Each cloud resource can be created in a data center hosted in a specific geographical region. For better performance, I recommend you choose one near to you. I'll use "Amsterdam 3" (`ams3`). You can get the complete list of regions with the following command.

```
$ doctl k8s options regions
```

Let's go ahead and initialize a Kubernetes cluster using DigitalOcean Kubernetes (DOKS). It will be composed of 3 worker nodes, for which you can decide the technical specifications. You can choose between different options in terms of CPU, memory, and architecture. I'll use nodes with 2 vCPU and 4GB of memory.

```
$ doctl k8s cluster create polar-cluster \①  
--node-pool "name=basiccnp;size=s-2vcpu-4gb;count=3;label=type=basic;" \②  
--region <your_region> ③
```

- ① Defines the name of the cluster to create.
- ② Provides the requested specifications for the worker nodes.
- ③ The data center region of your choice. For example, "ams3".

NOTE

If you'd like to know more about the different compute options and their prices, you can use the `doctl compute size list` command.

The cluster provision will take a few minutes. At the end, it will print out the unique ID assigned to the cluster. Take note since you'll need it later. You can fetch the cluster ID at any time by running the following command (I have filtered the result for the sake of clarity).

```
$ doctl k8s cluster list
```

ID	Name	Region	Status	Node Pools
<cluster-id>	polar-cluster	ams3	running	basiccnp

At the end of the cluster provisioning, `doctl` will also configure the context for your Kubernetes CLI so that you can interact with the cluster running on DigitalOcean from your computer, similar to what you did so far with your local cluster. You can verify the current context for `kubectl` by running the following command.

```
$ kubectl config current-context
```

NOTE

If you want to change context, you can run `kubectl config use-context <context-name>`.

Once the cluster is provisioned, you can get information about the worker nodes as follows.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
<node-1>	Ready	<none>	2m34s	v1.22.7
<node-2>	Ready	<none>	2m36s	v1.22.7
<node-3>	Ready	<none>	2m26s	v1.22.7

Do you remember the Octant dashboard you used to visualize the workloads on your local Kubernetes cluster? You can now use it to get information about the cluster on DigitalOcean as well. Open a Terminal window and start Octant with the following command.

```
$ octant
```

Octant will open in your browser and show data from your current Kubernetes context, which should be the cluster on DigitalOcean. From the upper right menu, you can switch between contexts from the drop-down box, as shown in figure [B.1](#).

The screenshot shows the Octant interface. On the left is a sidebar with icons for Applications, Namespace Overview (which is selected), Cluster Overview, and Plugins. The main area is titled 'Overview' and contains two tables: 'Services' and 'ConfigMaps'. The 'Services' table has columns for Name, Labels, and Type. It shows one entry for 'kubernetes' with labels 'component:apiserver' and 'provider:kubernetes' and type 'ClusterIP'. The 'ConfigMaps' table has columns for Name, Labels, and Data. It shows one entry for 'kube-root-ca.crt'. On the right, there is a vertical dropdown menu titled 'Contexts' with a list of clusters. The 'polar-cluster' option is highlighted. At the bottom of the dropdown, there are buttons for 'Items per page' (set to 10) and '1 - 1 of 1 items'.

Figure B.1 Octant lets you visualize workloads from different Kubernetes clusters by switching contexts.

As I mentioned in chapter 9, Kubernetes doesn't come packaged with an Ingress Controller, but it's up to you to install one. Since we'll rely on an Ingress resource to allow traffic from the public Internet to inside the cluster, we need to install an Ingress Controller. Let's install the same we used locally: ingress-nginx.

In your `polar-deployment` repository, create a new `kubernetes/platform/production` folder and copy over the content from the `Chapter15/15-end/polar-deployment/kubernetes/platform/production` folder in the source code repository accompanying the book.

Then, open a Terminal window, navigate to the `kubernetes/platform/production/ingress-nginx` folder in your `polar-deployment` project, and run the following command to deploy ingress-nginx to your production Kubernetes cluster. Feel free to open the file and look at the instructions before running it.

```
$ ./deploy.sh
```

NOTE You might need to make the script executable first with the command `chmod +x deploy.sh`.

In the next section, you'll see how to initialize a PostgreSQL database on DigitalOcean.

B.2 Running a PostgreSQL database on DigitalOcean

In most of the book, you've been running PostgreSQL database instances as containers, both in Docker and in your local Kubernetes cluster. In production, we'd like to take advantage of the platform and use a managed PostgreSQL service provided by DigitalOcean (docs.digitalocean.com/products/databases/postgresql).

The applications we developed throughout the book are cloud native and follow the 15-Factors methodology. As such, they treat backing services as attached resources that can be swapped without changing anything in the application code. Furthermore, we followed the environment parity principle and used a real PostgreSQL database both for development and testing, the same we want to use in production.

Moving from a PostgreSQL container running in your local environment to a managed service with high availability, scalability, and resilience is a matter of changing the values of a few configuration properties for Spring Boot. How great is that?

First, create a new PostgreSQL server named `polar-postgres`. We'll use PostgreSQL 14, which is the same version we used for development and testing. Remember to replace `<your_region>` with the geographical region you'd like to use. It should be the same as you used for the Kubernetes cluster. In my case, it's `ams3`.

```
$ doctl databases create polar-db --engine pg --region <your_region> --version 14
```

The database server provisioning will take several minutes. You can verify the installation status with the following command (I have filtered the result for the sake of clarity). When it's `online`, then your database server is ready. Take note of the database server ID. You'll need it later.

```
$ doctl databases list
```

ID	Name	Engine	Version	Region	Status
<polar-db-id>	polar-db	pg	14	ams3	online

To mitigate unnecessary attack vectors, we can configure a firewall so that the PostgreSQL server is only accessible from the Kubernetes cluster created previously. Remember that I asked you to take notes of the resource IDs for PostgreSQL and Kubernetes? Use them in the following command to configure the firewall and secure access to the database server.

```
$ doctl databases firewalls append <postgres_id> --rule k8s:<cluster_id>
```

Next, let's create two databases to be used by Catalog Service (`polardb_catalog`) and Order Service (`polardb_order`). Remember to replace `<postgres_id>` with your PostgreSQL resource ID.

```
$ doctl databases db create <postgres_id> polardb_catalog
$ doctl databases db create <postgres_id> polardb_order
```

Finally, let's retrieve the details for connecting to PostgreSQL. Remember to replace `<postgres_id>` with your PostgreSQL resource ID.

```
$ doctl databases connection <postgres_id> --format Host,Port,User,Password
Host          Port          User          Password
<database-host> <database-port> <database-user> <database-password>
```

Before concluding this section, let's create some Secrets in the Kubernetes cluster with the PostgreSQL credentials required by the two applications. In a real-world scenario, we should create dedicated users for two applications and grant limited privileges. For simplicity, we'll use the admin account for both.

First, create a Secret for Catalog Service using the information returned by the previous `doctl` command.

```
$ kubectl create secret generic polar-postgres-catalog-credentials \
--from-literal=spring.datasource.url=jdbc:postgresql://<postgres_host>:<postgres_port>/polardb_catalog \
--from-literal=spring.datasource.username=<postgres_username> \
--from-literal=spring.datasource.password=<postgres_password>
```

Similarly, create a Secret for Order Service. Pay attention to the slightly different syntax required by Spring Data R2DBC for the URL.

```
$ kubectl create secret generic polar-postgres-order-credentials \
--from-literal="spring.flyway.url=jdbc:postgresql://<postgres_host>:<postgres_port>/polardb_order" \
--from-literal="spring.r2dbc.url=r2dbc:postgresql://<postgres_host>:<postgres_port>/polardb_order?ssl=true&sslMode=require" \
--from-literal=spring.r2dbc.username=<postgres_username> \
--from-literal=spring.r2dbc.password=<postgres_password>
```

That's it for PostgreSQL. In the next section, you'll see how to initialize Redis using DigitalOcean.

B.3 Running Redis on DigitalOcean

In most of the book, you've been running Redis instances as containers, both in Docker and in your local Kubernetes cluster. In production, we'd like to take advantage of the platform and use a managed Redis service provided by DigitalOcean (docs.digitalocean.com/products/databases/redis/).

Once again, since we followed the 15-Factors methodology, we can swap the Redis backing service used by Edge Service without changing anything in the application code. We'll only need to change a few configuration properties for Spring Boot.

First, create a new Redis server named `polar-redis`. We'll use Redis 6, which is the same version we used for development and testing. Remember to replace `<your_region>` with the geographical region you'd like to use. It should be the same as you used for the Kubernetes cluster. In my case, it's `ams3`.

```
$ doctl databases create polar-redis --engine redis --region <your_region> --version 6
```

The Redis server provisioning will take several minutes. You can verify the installation status with the following command (I have filtered the result for the sake of clarity). When it's `online`, then your Redis server is ready. Take note of the Redis resource ID. You'll need it later.

```
$ doctl databases list

ID          Name      Engine  Version  Region   Status
<redis-db-id>  polar-redis  redis    6        ams3     creating
```

To mitigate unnecessary attack vectors, we can configure a firewall so that the Redis server is only accessible from the Kubernetes cluster created previously. Remember that I asked you to take notes of the resource IDs for Redis and Kubernetes? Use them in the following command to configure the firewall and secure access to the Redis server.

```
$ doctl databases firewalls append <redis_id> --rule k8s:<cluster_id>
```

Finally, let's retrieve the details for connecting to PostgreSQL. Remember to replace `<redis_id>` with your Redis resource ID.

```
$ doctl databases connection <redis_id> --format Host,Port,User,Password

Host          Port          User          Password
<redis-host>  <redis-port>  <redis-user>  <redis-password>
```

Before concluding this section, let's create a Secret in the Kubernetes cluster with the Redis credentials required by Edge Service. In a real-world scenario, we should create a dedicated user for the application and grant limited privileges. For simplicity, we'll use the default account. Populate the Secret with the information returned by the previous `doctl` command.

```
$ kubectl create secret generic polar-redis-credentials \
--from-literal=spring.redis.host=<redis_host> \
--from-literal=spring.redis.port=<redis_port> \
--from-literal=spring.redis.username=<redis_username> \
--from-literal=spring.redis.password=<redis_password> \
--from-literal=spring.redis.ssl=true
```

That's it for Redis. The following section will cover how to deploy RabbitMQ using a Kubernetes Operator.

B.4 Running RabbitMQ using a Kubernetes Operator

In the previous sections, we initialized and configured PostgreSQL and Redis servers offered and managed by the platform. We can't do the same for RabbitMQ because DigitalOcean doesn't have a RabbitMQ offering, similar to other cloud providers like Azure or GCP.

A popular and convenient way of deploying and managing services like RabbitMQ in a Kubernetes cluster is the *Operator* pattern.

SIDE BAR

DEFINITION

Operators are "software extensions to Kubernetes that make use of custom resources to manage applications and their components". ([kubernetes.io/docs](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/))

Think about RabbitMQ. To use it in production, we need to configure it for high availability and resilience. Depending on the workload, we might want to scale it dynamically. When a new version of the software is available, we would need a reliable way of upgrading the service and, in case, migrating existing constructs and data. We could perform all those tasks manually. Or we could use an Operator to capture all those operational requirements and instruct Kubernetes to take care of them automatically. In practice, an Operator is an application that runs on Kubernetes and interacts with its API to accomplish its functionality.

The RabbitMQ project provides an official Operator to run the event broker on a Kubernetes cluster (www.rabbitmq.com). I have already configured all the necessary resources to use the RabbitMQ Kubernetes Operator and prepared a script to deploy it.

Open a Terminal window, go to your Polar Deployment project (`polar-deployment`), and navigate to the folder `kubernetes/platform/production/rabbitmq`. You should have copied that folder over to your repository when configuring the Kubernetes cluster. If that's not the case, please do so now from the source code repository accompanying this book (`Chapter15/15-end/polar-deployment/platform/production/rabbitmq`).

Then, run the following command to deploy RabbitMQ to your production Kubernetes cluster. Feel free to open the file and look at the instructions before running it.

```
$ ./deploy.sh
```

NOTE

You might need to make the script executable first with the command `chmod +x deploy.sh`.

The script will output details about all the operations performed to deploy RabbitMQ. Finally, it will create a `polar-rabbitmq-credentials` Secret with the credentials that Order Service and Dispatcher Service will need to access RabbitMQ. You can verify the Secret has been successfully created as follows.

```
$ kubectl get secrets polar-rabbitmq-credentials
```

The RabbitMQ broker is deployed in a dedicated `rabbitmq-system` namespace. Applications can interact with it at `polar-rabbitmq.rabbitmq-system.svc.cluster.local` on port 5672.

That's it for RabbitMQ. In the next section, you'll see how to deploy a Keycloak server to a production Kubernetes cluster.

B.5 Running Keycloak using a Helm chart

Similar to RabbitMQ, DigitalOcean doesn't provide a managed Keycloak service. The Keycloak project is working on an Operator, but it's still in beta at the moment of writing, so we'll deploy it using a different approach: Helm charts.

Think of Helm as a package manager. To install software on your computer, you would use one of the operating system package managers, like `apt` (Ubuntu), `Homebrew` (macOS), or `Chocolatey` (Windows). In Kubernetes, you can similarly use Helm, but they're called *charts* instead of *packages*.

Go ahead and install Helm on your computer. You can find the instructions on the official website ([helm.sh](#)). If you are on macOS or Linux, you can install Helm with Homebrew.

```
$ brew install helm
```

I have already configured all the necessary resources to use the Keycloak Helm chart provided by Bitnami ([bitnami.com](#)) and prepared a script to deploy it. Open a Terminal window, go to your Polar Deployment project (`polar-deployment`), and navigate to the folder `kubernetes/platform/production/keycloak`. You should have copied that folder over to your repository when configuring the Kubernetes cluster. If that's not the case, please do so now from the source code repository accompanying this book (`Chapter15/15-end/polar-deployment/platform/production/keycloak`).

Then, run the following command to deploy Keycloak to your production Kubernetes cluster.

Feel free to open the file and look at the instructions before running it. You need to provide three arguments:

```
$ ./deploy.sh
```

NOTE

You might need to make the script executable first with the command `chmod +x deploy.sh`.

The script will output details about all the operations performed to deploy Keycloak and print the admin username and password you can use to access the Keycloak Admin Console. Feel free to change the password after your first login. Note the credentials down since you might need them later. The deployment can take several minutes to complete, so it's a good time to take a break and drink a beverage of your choice as a reward for everything you have accomplished so far. Good job!

Finally, it will create a `polar-keycloak-client-credentials` Secret with the Client secret that Edge Service will need to authenticate with Keycloak. You can verify the Secret has been successfully created as follows. The value is generated randomly by the script.

```
$ kubectl get secrets polar-keycloak-client-credentials
```

NOTE

The Keycloak Helm chart spins up a PostgreSQL instance inside the cluster and uses it to persist the data used by Keycloak. We could have integrated it with the PostgreSQL service managed by DigitalOcean, but the configuration on the Keycloak side would have been quite complicated. If you'd like to use an external PostgreSQL database, you can refer to the Keycloak Helm chart documentation (bitnami.com/stack/keycloak/helm).

The Keycloak server is deployed in a dedicated `keycloak-system` namespace. Applications can interact with it at `polar-keycloak.keycloak-system.svc.cluster.local` on port 8080 from within the cluster. It's also exposed outside the cluster via a public IP address. Let's find out the external IP address with the following command. The platform might take a few minutes to provision a load balancer. During the provisioning, the `EXTERNAL-IP` column will show a `<pending>` status. Wait and try again until an IP address is shown. Note it down since we're going to use it in multiple scenarios.

```
$ kubectl get service polar-keycloak -n keycloak-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
polar-keycloak	LoadBalancer	10.245.191.181	<external-ip>	80:31268/TCP,443:31892/TCP
7m32s				

Since Keycloak is exposed via a public load balancer, we can use the external IP address to

access the Admin Console. Open a browser window, navigate to `<external-ip>/auth/admin`, and log in with the credentials you defined earlier.

Now that we have a public DNS name for Keycloak, we can define a couple of Secrets to configure the Keycloak integration in Edge Service (OAuth2 Client), Catalog Service and Order Service (OAuth2 Resource Servers). Open a Terminal window, navigate to the `kubernetes/platform/production/keycloak` folder in your `polar-deployment` project, and run the following command to create the Secrets that the applications will use to integrate with Keycloak. Feel free to open the file and look at the instructions before running it. Remember to replace `<external-ip>` with the external IP address assigned to your Keycloak server.

```
$ ./create-secrets.sh http://<external-ip>/auth/realms/PolarBookshop
```

That's it for Keycloak. The next section will show you how to deploy Polar UI to the production cluster.

B.6 Running Polar UI

Polar UI is a single-page application built with Angular and served by NGINX. As you saw in chapter 11, I have already prepared a container image you can use to deploy this application since frontend development is out of scope for this book.

Open a Terminal window, go to your Polar Deployment project (`polar-deployment`), and navigate to the folder `kubernetes/platform/production/polar-ui`. You should have copied that folder over to your repository when configuring the Kubernetes cluster. If that's not the case, please do so now from the source code repository accompanying this book (`Chapter15/15-end/polar-deployment/platform/production/polar-ui`).

Then, run the following command to deploy Polar UI to your production Kubernetes cluster. Feel free to open the file and look at the instructions before running it.

```
$ ./deploy.sh
```

NOTE You might need to make the script executable first with the command `chmod +x deploy.sh`.

Now that we have Polar UI and all the main platform services up and running, you can proceed with reading chapter 15 and complete the configuration of all the Spring Boot applications in Polar Bookshop for production deployment.

B.7 Deleting all cloud resources

When you’re done experimenting with the Polar Bookshop project, follow this section to delete all the cloud resources created on DigitalOcean. That’s fundamental to avoid incurring in unexpected costs.

First, delete the Kubernetes cluster.

```
$ doctl k8s cluster delete polar-cluster
```

Next, delete the PostgreSQL and Redis databases. You need to know their IDs first, so run this command to extract that information.

```
$ doctl databases list
```

ID	Name	Engine	Version	Region	Status
<polar-db-id>	polar-db	pg	14	ams3	online
<redis-db-id>	polar-redis	redis	6	ams3	creating

Then, go ahead and delete both of them using the resource identifiers returned by the previous command.

```
$ doctl databases delete <polar-db-id>
$ doctl databases delete <redis-db-id>
```

Finally, open a browser window, navigate to the DigitalOcean web interface (cloud.digitalocean.com), and go through the different categories of cloud resources in your account to verify there’s no outstanding service. If there is, delete it. There could be load balancers or persistent volumes created as a side effect of creating a cluster or a database, but that have not been deleted by the previous commands.

Notes

1. P. Fremantle, "Cloud Native", <http://pzf.fremantle.org/2010/05/cloud-native.html>
2. Cloud Native Computing Foundation, "CNCF Cloud Native Definition v1.0", <https://github.com/cncf/toc/blob/master/DEFINITION.md>
3. Cloud Native Computing Foundation, "CNCF Cloud Native Interactive Landscape", <https://landscape.cncf.io/>
4. NIST, "The NIST Definition of Cloud Computing", SP 800-145
- N.R. Herbst, S. Kounev, R. Reussner, "Elasticity in Cloud Computing: What it is, and What it is Not" in *Proceedings of the 10th International Conference on Autonomic Computing* (ICAC 2013), San Jose, CA, pages 23–27. USENIX.
- C. Davis, "Realizing Software Reliability in the Face of Infrastructure Instability" in *IEEE Cloud Computing*, vol. 4, no. 5, pp. 34–40, September/October 2017.
- D. L. Parnas, "On the criteria to be used in decomposing systems into modules", Commun. ACM 15, 12 (December 1972), 1053–1058
- J. E. Blyler, "Heuristics for resilience — A richer metric than reliability," 2016 IEEE International Symposium on Systems Engineering (ISSE), Edinburgh, 2016, pp. 1-4.
- A. Avizienis, J. Laprie, B. Randell, "Fundamental Concepts of Dependability", 2001
- Twitter, "Observability at Twitter: technical overview, part I", https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i
- M. Fowler, "Infrastructure As Code", <https://martinfowler.com/bliki/InfrastructureAsCode.html>
- M. Fowler, "Continuous Delivery", <https://martinfowler.com/bliki/ContinuousDelivery.html>
- M. Fowler, "Continuous Integration Certification", <https://martinfowler.com/bliki/ContinuousIntegrationCertification.html>
- D. Farley, "Continuous Delivery Pipelines", 2021
- K. Mugrage, "DevOps and DevOpsDays - Where it started, where it is, where it's going", <https://www.youtube.com/watch?v=VEg9VaHzqC8>

16. K. Mugrage, "My definition of DevOps", <https://dev.to/kmugrage/my-definition-of-devops-2baj>
17. ACM Queue: Interview with Amazon's Werner Vogels, https://aws.amazon.com/blogs/aws/acm_queue_inter
18. N. Kratzke and R. Peinl, "ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects," 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), Vienna, 2016, pp. 1-10, doi: 10.1109/EDOCW.2016.7584353.
19. J. Lewis, M. Fowler, "Microservices", <https://martinfowler.com/articles/microservices.html>
20. C i n d y S r i d h a r a n ,
<https://twitter.com/copyconstruct/status/1525860086685478912?s=20&t=Nqy10tGzLnjjLkXiCmrWPw>
21. Cornelia Davis, "Cloud Native Patterns", Manning, 2019
22. Adam Wiggins et al. from Heroku, *The Twelve-Factor App*, <https://12factor.net>
23. Kevin Hoffman, *Beyond the Twelve-Factor App*, O'Reilly, 2016
24. C4 Model, <https://c4model.com>